# CHAPTER 2
# Designing Beautiful APIs

As developers, we experience the difference between good and bad programmatic interfaces every single day. Some modules we work with seem to speak right to us, clearly expressing their role in our project in a loud, confident voice. Others mumble nonsense and occasionally freak out on us in ways we'd never expect. In many ways, our code is only as good as its application programming interface (API). Good APIs provide exactly what we need, in the way we need it. Bad APIs confuse us and make us jump through hoops to get things working, pushing us one step closer to the dreaded "Big Rewrite." Ruby provides the tools to build beautifully clear interfaces, but these same tools can produce chaos and disarray when placed in misguided hands.

In this chapter, we'll take a look at the infrastructure Ruby provides to help you design solid interfaces. We'll examine both the technical details and the motivation behind the various different approaches, allowing you to see both the how and why that's involved in designing "Rubyish" APIs. We'll begin by looking through a practical example of API design from a real project, and then move on to more specific tips and tricks. Along the way, you'll gain both conceptual and technical understanding of how to design suitable APIs for your Ruby projects.

## Designing for Convenience: Ruport's Table( ) feature

In Ruby Reports (Ruport), virtually all of our users will need to work with tabular data at some point. This data can come from any number of sources, ranging from a simple CSV file to a complex data analysis procedure that produces tables as its output. Providing a simple, clear API that works well for all of these cases is a bit of a challenge. Although the ultimate goal of each of the different tasks that we can imagine is fundamentally the same, the means for accomplishing them are quite different.

Our first instinct was to reflect this in the API, resulting in several different constructors for the different ways of building tables:

```
table1 = Ruport::Data::Table.new(
  :column_names => %w[first_name last_name],
   :data         => [["Gregory","Brown"],["Deborah","Orlando"]] )

table2 = Ruport::Data::Table.load("people.csv")

csv = "first_name,last_name\nGregory,Brown\nDeborah\nOrlando\n"
table3 = Ruport::Data::Table.parse(csv)
```

Although it is clear enough what is going on here, it is necessary to remember the names and signatures of several different methods. We were convinced we could do better, and eventually came up with this:

```
table1 = Table(%w[first_name last_name],
   :data => [["Gregory","Brown"],["Deborah","Orlando"]])

table2 = Table("people.csv")

csv = "first_name,last_name\nGregory,Brown\nDeborah,Orlando\n"
table3 = Table(:string => csv)
```

Though the difference here is somewhat subtle, there is definitely something more natural about the way this code reads. Rather than feeling like we're initializing a class, it feels more as if we're converting our data into a `Table` object. This is similar to the way a number of Ruby's built-in methods work, including `Integer()`, `Array()`, and friends.

Introducing the `Table()` method was a big win, as it simplified things for the most common cases in Ruport. However, rather than trashing the old interface, we instead opted to wrap it. This approach gives curmudgeons who dislike magic an alternative avenue while providing some additional benefits. The most obvious one is that the `Table()` method itself is fairly simple code:

```
def Table(*args,&block)
  table = case args[0]
  when Array
    opts = args[1] || {}
    Ruport::Data::Table.new({:column_names => args[0]}.merge(opts),&block)
  when /\.csv$/i
    Ruport::Data::Table.load(*args,&block)
  when Hash
    if file = args[0].delete(:file)
      Ruport::Data::Table.load(file,args[0],&block)
    elsif string = args[0].delete(:string)
      Ruport::Data::Table.parse(string,args[0],&block)
    else
      Ruport::Data::Table.new(args[0],&block)
    end
  else
     Ruport::Data::Table.new(:data => [], :column_names => args,&block)
  end

  return table
end
```

Though it may be a little bit busy with so much going on, the entire purpose of this method is simply to recognize the kind of table loading you are doing, then delegate to the appropriate constructor method. We do this by treating the arguments passed to the method as an array. We also capture any block passed to the method so that we can pass it down into the constructors, but we'll get back to that later. For now, let's simply focus on the array of arguments passed in.

We can walk through the examples already shown to get a sense of what's going on. In our first example, we called the method with an array of column names, and a hash of options:

```
table1 = Table(%w[first_name last_name],
  :data => [["Gregory","Brown"],["Deborah","Orlando"]])
```

In this case, `args` contains two elements: an array of column names and a hash of options. The structure looks like this:

```
[ %w[first_name last_name],
  { :data => [["Gregory","Brown"],["Deborah","Orlando"]] } ]
```

Our case statement matches `args[0]` as an `Array`, and pulls the `Hash` out from `args[1]`. After processing, our final call is equivalent to this:

```
Ruport::Data::Table.new(
  :column_names => %w[first_name last_name],
  :data         => [["Gregory","Brown"],["Deborah","Orlando"]] )
```

As you can see, this is exactly equivalent to the very first call shown in this chapter. If we left out the data parameter in our `Hash`, we'd see the following translation:

```
headings = %w[first_name last_name]
Ruport::Data::Table.new(:column_names => headings) == Table(headings) #=> true
```

Working with our second example is even easier. We're passing a CSV filename like so:

```
table2 = Table("people.csv")
```

In this situation, `args` is a single element array with `args[0]` as `"people.csv"`. Our case statement immediately matches this and then uses the array splat operator (*) to pass the arguments directly to `Ruport::Data::Table.load()`. In this case, our `Table()` method is just being lazy, ultimately just acting identically to `Ruport::Data::Table.load()` as long as your file ends in *.csv*.

What if your file has a nonstandard extension or no extension at all? Both you and `Table()` need to do a bit more work. The most simple form would look like this:

```
Table(:file => "people.txt")
```

In this case, `args` is a single element array containing a `Hash`. However, we know this is not ambiguous, because our third example was also of this form:

```
csv = "first_name,last_name\nGregory,Brown\nDeborah,Orlando\n"
table3 = Table(:string => csv)
```

Also, although it has not been shown yet, `Table()` also provides a compatible interface with `Ruport::Data::Table.new()`. This means that the following code is also valid:

```
table4 = Table( :column_names => %w[first_name last_name],
                :data         => [["Gregory","Brown"],["Deborah","Orlando"]] )
```

These three different cases are distinguished fairly easily, as you can see here:

```
if file = args[0].delete(:file)
  Ruport::Data::Table.load(file,args[0],&block)
elsif string = args[0].delete(:string)
  Ruport::Data::Table.parse(string,args[0],&block)
else
  Ruport::Data::Table.new(args[0],&block)
end
```

This code attempts to be polite by removing the `:file` or `:string` option from the `options` hash before delegating to the relevant constructor. This is a good practice when forwarding options hashes downstream, so as to avoid providing the underlying methods with options that it may not handle properly.

Putting this all together, we find that it is possible to be extremely flexible with Ruby interfaces. In fact, the following examples show even more advanced behavior that is possible with `Table()`:

```
# removes records that don't have a first name of Gregory
table = Table("foo.csv") do |t,r|
  t.filter { |r| r.first_name == "Gregory" }
  t << r
end

# doubles the first column's values
table = Table("a","b","c") do |t|
  t.transform { |r| r["a"] *= 2 }
  t << [1,2,3]
  t << [4,5,6]
end
```

Both examples build up a table within a block, but there is a key difference between the two. The first example is an iterator, walking over the rows of a CSV file as they are read in and parsed. The second example starts with an empty table and no data source to feed from, and allows users to build up a table from whatever sources they wish. The power here is that we can allow our blocks to act differently based on the expected number of arguments. This is only the tip of the iceberg when it comes to the kinds of things you can do with blocks, and in just a little bit, we'll look at some fresh examples to really get a sense of one of Ruby's most powerful API building tools.

For now, let's take a step back and look at some kinds of argument processing that Ruby can do. The methods that Ruport's `Table()` method delegates to are a little too domain-specific to show this stuff off usefully, so instead we'll use a few more basic examples to demonstrate the various different kinds of interfaces you can provide for methods in Ruby.

## Ruby's Secret Power: Flexible Argument Processing

When you think of a method, how do you envision its parameters? Depending on whom you ask, you might get a lot of different answers. Some folks might think of something vaguely mathematical, such as `f(x,y,z)`, where each argument is necessary, and the order in which they are provided is significant. Others might think of methods as manipulating configuration data, where keyword-like parameters seem natural, e.g., `create_person( first_name: "Joe", last_name: "Frasier")`. You might also contemplate mixing the two together, or dreaming up something else entirely.

Ruby provides a great deal of flexibility in how it handles method arguments, which might lead to some confusion. However, this is also a key part of building beautiful APIs in Ruby. The following examples give just a small taste of the kind of diversity you can expect in Ruby:

```
# Standard ordinal arguments
def distance(x1,y1,x2,y2)
  Math.hypot(x2 - x1, y2 - y1)
end

# Ordinal arguments, with an optional argument
def load_file(name,mode="rb")
  File.open(name,mode)
end

# Pseudo-keyword arguments
def story(options)
  "#{options[:person]} went to town, riding on a #{options[:animal]}"
end

# Treating arguments as an Array
def distance2(*points)
  distance(*points.flatten)
end
```

Invoking these methods shows how they look in action:

```
>> distance(3,3,4,5)
=> 2.23606797749979

>> load_file "foo.jpg"
=> #<File:foo.jpg>
>> load_file "foo.jpg", "r"
=> #<File:foo.jpg>
>> load_file "foo.jpg", "kitten"
ArgumentError: illegal access mode kitten ...

>> story(animal: "Tiger", person: "Yankee Doodle")
=> "Yankee Doodle went to town, riding on a Tiger"
>> story(person: "Yankee Doodle", animal: "Tiger")
=> "Yankee Doodle went to town, riding on a Tiger"
```

```
>> distance2 [3,3], [4,5]
=> 2.23606797749979
```

Each approach shown here has its merits, and how to choose the right one depends mainly on what your method needs to do. Let's look at each of these trivial methods one by one to get a sense of their pros and cons.

## Standard Ordinal Arguments

```
def distance(x1,y1,x2,y2)
  Math.hypot(x2 - x1, y2 - y1)
end

>> distance(3,3,4,5)
=> 2.23606797749979
```

Ordinal argument processing is the most simple, but also the most restrictive way to pass arguments into your methods in Ruby. In this case, it may be all we need. All four arguments are necessary to do the calculation, and there is a logical way to order them. In situations like these, the no-frills approach is frequently good enough.

## Ordinal Arguments with Optional Parameters

```
def load_file(name,mode="rb")
  File.open(name,mode)
end

>> load_file "foo.jpg"
=> #<File:foo.jpg>
>> load_file "foo.jpg", "r"
=> #<File:foo.jpg>
>> load_file "foo.jpg", "kitten"
ArgumentError: illegal access mode kitten ...
```

The ability to set default values makes ordinal arguments a bit more pleasant to work with in some cases. The example code provides a trivial wrapper on `File.open()` that causes it to default to reading a binary file. For this default use, the method appears to accept a single argument, keeping the call clean and straightforward. However, a second argument can be provided if needed, due to this more flexible way of defining the interface.

However, what happens when we have more than one optional argument, such as in the following example code?

```
def load_file2(name="foo.jpg",mode="rb")
  File.open(name,mode)
end
```

For the obvious cases, this works as we might expect:

```
>> load_file2
=> #<File:foo.jpg>
```

```
>> load_file2 "bar.jpg"
=> #<File:bar.jpg>

>> load_file2 "bar.jpg", "r"
=> #<File:bar.jpg>

>> load_file2 "bar.jpg", "kitten"
ArgumentError: invalid access mode kitten ...
```

However, the trouble arises when we want to use the default value for the first argument and override the second one. Sadly enough, we simply cannot do it using this approach. We'd need to supply both values, like this:

```
>> load_file2 "foo.jpg", "r"
=> #<File:foo.jpg>
```

For this reason, it's relatively rare to find a good use for multiple optional parameters in the ordinal format. When you do find yourself needing this sort of thing, there are most likely better options available, anyway.

## Pseudo-Keyword Arguments

```
def story(options)
  "#{options[:person]} went to town, riding on a #{options[:animal]}"
end

>> story(animal: "Tiger", person: "Yankee Doodle")
=> "Yankee Doodle went to town, riding on a Tiger"
>> story(person: "Yankee Doodle", animal: "Tiger")
=> "Yankee Doodle went to town, riding on a Tiger"
```

Although Ruby doesn't have support for real keyword arguments, it does a pretty good job of imitating them. If we peek behind the curtain though, we find that we're really dealing with something much more basic:

```
>> argument_hash = { :animal => "Tiger", :person => "Yankee Doodle" }
>> story(argument_hash)
=> "Yankee Doodle went to town, riding on a Tiger"
```

Seen in this form, all the magic disappears. The method is really only processing a single argument in the form of a basic `Hash` of key/value pairs.

Utilizing some syntactic sugar, we can rearrange things a bit:

```
>> argument_hash = { animal: "Tiger", person: "Yankee Doodle" }
>> story(argument_hash)
=> "Yankee Doodle went to town, riding on a Tiger"
```

If our hash happens to be the last (or the only) argument to the method, we can leave off the brackets when passing a `Hash` literal. This gets us full circle back to the original:

```
>> story(animal: "Tiger", person: "Yankee Doodle")
=> "Yankee Doodle went to town, riding on a Tiger"
```

With a basic understanding of the underlying mechanics, you can begin to see the benefits of this style of API. Perhaps the most significant is that the order in which you specify the arguments doesn't matter at all, as you've seen in the example code. If we combine that feature with a basic idiom for setting default values passed in the hash, we come up with something pretty interesting:

```
def story2(options={})
  options = { person: "Yankee Doodle", animal: "Tiger" }.merge(options)
  "#{options[:person]} went to town, riding on a #{options[:animal]}"
end

>> story2
=> "Yankee Doodle went to town, riding on a Tiger"
>> story2(person: "Joe Frasier")
=> "Joe Frasier went to town, riding on a Tiger"
>> story2(animal: "Kitteh")
=> "Yankee Doodle went to town, riding on a Kitteh"
>> story2(animal: "Kitteh", person: "Joe Frasier")
=> "Joe Frasier went to town, riding on a Kitteh"
```

As you can see here, it is possible to handle multiple default values fairly elegantly. This avoids the problems we encountered when attempting to do something similar via simple default values for ordinal arguments. Though this example is a bit contrived, it's worth mentioning that if one or more of your arguments are really mandatory, it's worth it to break them out, like so:

```
def write_story_to_file(file,options={})
  File.open(file,"w") { |f| f << story2(options) }
end
```

which enables fun stuff like:

```
>> write_story_to_file "output.txt"
>> write_story_to_file "output.txt", animal: "Kitteh"
>> write_story_to_file "output.txt", person: "Joe Frasier"
>> write_story_to_file "output.txt", animal: "Slug", person: "Joe Frasier"
```

Though you could write code to ensure that certain options are present in a hash, generally it is most natural to just let Ruby do the hard work for you by placing your mandatory arguments before your options hash in your method definition.

## Treating Arguments As an Array

```
def distance2(*points)
  distance(*points.flatten)
end

>> distance2 [3,3], [4,5]
=> 2.23606797749979
```

A powerful feature involves treating the passed arguments as an array. In the case of a distance method, we might want to provide the arguments in one of several different ways, depending on our situation. The previous example shows the primary use of this approach, which is to allow the data to be passed as two points. However, if you play around, you'll find that a lot of other possibilities exist, some more sane than others. Here are a few to give you a sense of what's going on here:

```
>> distance2 [3],[3],[4],[5]
=> 2.23606797749979
>> distance2 3,3,4,5
=> 2.23606797749979
>> distance2 [3,3,4],5
=> 2.23606797749979
```

It should be clear at this point that Ruby is treating the arguments as one big array, flattening it down to eliminate any nesting, and then passing the four values as arguments to the original `distance()` method. With this in mind, the following issues shouldn't be too surprising:

```
>> distance2 [3,3,4]
ArgumentError: wrong number of arguments (3 for 4) ...

>> distance2 [3,3,4],5,6
ArgumentError: wrong number of arguments (5 for 4) ...
```

Still, our error messages are a bit obscure, and the fact that we're wrapping another method might make things seem a bit tricky. Considering this issue, we can try to make something a little more solid using the same general approach:

```
def distance3(*points)
  case(points.length)
  when 2
    x1,y1,x2,y2 = points.flatten
  when 4
    x1,y1,x2,y2 = points
  else
     raise ArgumentError,
        "Points may be specified as [x1,y1], [x2,y2] or x1,y1,x2,y2"
  end
  Math.hypot(x2 - x1, y2 - y1)
end
```

In this case, our method behaves much more strictly:

```
>> distance3 [3,3,4]
ArgumentError: Points may be specified as [x1,y1], [x2,y2] or x1,y1,x2,y2 ...

>> distance3 3,3,3,4,5
ArgumentError: Points may be specified as [x1,y1], [x2,y2] or x1,y1,x2,y2 ...

>> distance3 [3,3,3,4]
ArgumentError: Points may be specified as [x1,y1], [x2,y2] or x1,y1,x2,y2 ...

>> distance3 [3,3],[3,4]
=> 1.0

>> distance3 3,3,3,4
=> 1.0
```

Though this may be a more robust way to write the method, it is more complicated. Playing fast and loose might not be a bad thing if you can expect your users to provide sane input. The following version is the compromise I might use in production code:

```
def distance4(*points)
  x1,y1,x2,y2 = points.flatten
  raise ArgumentError unless [x1,y1,x2,y2].all? { |e| Numeric === e }
  Math.hypot(x2 - x1, y2 - y1)
end
```

This code checks the first four arguments after flattening any nesting to make sure that they are `Numeric` values, and then assigns them to the variables `x1`, `y1`, `x2`, and `y2`. Though it does not cover all edge cases, it does some sanity checks and makes sure that you've provided all the necessary data in a form that is usable by the method. In many cases, this will be good enough.

Although we've covered only some of the most basic argument forms here, these techniques form the basis for building solid Ruby APIs. The following short list of guidelines will help you in designing your methods:

- Try to keep the number of ordinal arguments in your methods to a minimum.
- If your method has multiple parameters with default values, consider using pseudo-keyword arguments via an options hash.
- Use the array splat operator (*) when you want to slurp up your arguments and pass them to another method.
- The `*args` idiom is also useful for supporting multiple simultaneous argument processing styles, as in `Table()`, but can lead to complicated code.
- Don't use `*args` when a normal combination of mandatory ordinal arguments and an options hash will do.
- If some parameters are mandatory, avoid putting them in an options hash, and instead write a signature like `foo(mandatory1, mandatory2, options={})`, unless there is a good reason not to.

Although having a decent understanding of how argument processing works in Ruby will take you far, there are many situations that need a little more firepower. Ruby's ability to utilize code blocks in association with method calls is often the answer. We'll now dive into the various ways that working with blocks can simplify your interface as well as the internals of your methods.

## Ruby's Other Secret Power: Code Blocks

In Ruby, code blocks are everywhere. If you've ever used `Enumerable`, you've worked with blocks. But what are they? Are they simply iterators, working to abstract away our need for the `for` loop? They certainly do a good job of that:

```
>> ["Blocks","are","really","neat"].map { |e| e.upcase }
=> ["BLOCKS", "ARE", "REALLY", "NEAT"]
```

But other blocks don't really iterate over things—they just do helpful things for us. For example, they allow us to write something like:

```
File.open("foo.txt","w") { |f| f << "This is sexy" }
```

instead of forcing us to write this:

```
file = File.open("foo.txt","w")
file << "This is tedious"
file.close
```

So blocks are useful for iteration, and also useful for injecting some code between preprocessing and postprocessing operations in methods. But is that all they're good for? Sticking with Ruby built-ins, we find that isn't the case. Blocks can also shift our scope temporarily, giving us easier access to places we want to be:

```
"This is a string".instance_eval do
   "O hai, can has reverse? #{reverse}. kthxbye"
 end

 #=> "O hai, can has reverse? gnirts a si sihT. kthxbye"
```

But blocks aren't necessarily limited to code that gets run right away and then disappears. They can also form templates for what to do down the line, springing to action when ready:

```
>> foo = Hash.new { |h,k| h[k] = [] }
=> {}
>> foo[:bar]
=> []
>> foo[:bar] << 1 << 2 << 3
=> [1, 2, 3]
>> foo[:baz]
=> []
```

So even if we label all methods that accept a block as iterators, we know the story runs deeper than that. With this in mind, we can leverage some basic techniques to utilize any of the approaches shown here, as well as some more advanced tricks. By doing things in a way that is consistent with Ruby itself, we can make life easier for our users. Rather than piling on new concepts, we can allow them to reuse their previous knowledge. Let's take a look at a few examples of how to do that now.

## Working with Enumerable

The most common use of blocks in Ruby might be the most trivial. The following class implements a basic sorted list, and then mixes in the `Enumerable` module. The block magic happens in `each()`:

```
class SortedList

  include Enumerable
  def initialize
    @data = []
  end

  def <<(element)
    (@data << element).sort!
  end

  def each
    @data.each { |e| yield(e) }
  end

end
```

Our `each()` method simply walks over each element in our `@data` array and passes it through the block provided to the method by calling `yield`. The resulting iterator works exactly the same as `Array#each` and `Hash#each` and all the Ruby built-ins, and indeed simply wraps `Array#each` in this case:

```
>> a = SortedList.new
=> #<SortedList:0x5f0e74 @data=[]>
>> a << 4
=> [4]
>> a << 5
=> [4, 5]
>> a << 1
=> [1, 4, 5]
>> a << 7
=> [1, 4, 5, 7]
>> a << 3
=> [1, 3, 4, 5, 7]

>> x = 0
=> 0
>> a.each { |e| x += e }
=> [1, 3, 4, 5, 7]
>> x
=> 20
```

This shouldn't be surprising. What is really the interesting bit is that by including the module `Enumerable`, we gain access to most of the other features we're used to working with when processing Ruby's built-in collections. Here are just a few examples:

```
>> a.map { |e| "Element #{e}" }
=> ["Element 1", "Element 3", "Element 4", "Element 5", "Element 7"]
>> a.inject(0) { |s,e| s + e }
=> 20
>> a.to_a
=> [1, 3, 4, 5, 7]
>> a.select { |e| e > 3 }
=> [4, 5, 7]
```

In a lot of cases, the features provided by `Enumerable` will be more than enough for traversing your data. However, it's often useful to add other features that build on top of the `Enumerable` methods. We can show this by adding a simple reporting method to `SortedList`:

```
class SortedList
  def report(head)
    header = "#{head}\n#{'-'*head.length}"
    body = map{|e| yield(e)}.join("\n") + "\n"
    footer = "This report was generated at #{Time.now}\n"

    [header, body, footer].join("\n")
  end
end
```

which, when run, produces output like this:

```
>> puts a.report("So many fish") { |e| "#{e} fish" }
So many fish
------------
1 fish
3 fish
4 fish
5 fish
7 fish

This report was generated at 2008-07-22 22:47:20 -0400
```

Building custom iterators is really that simple. This provides a great deal of flexibility, given that the code block can execute arbitrary expressions and do manipulations of its own as it walks across the elements. But as mentioned before, blocks can be used for more than just iteration.

## Using Blocks to Abstract Pre- and Postprocessing

We looked at the block form of `File.open()` as an example of how blocks can provide an elegant way to avoid repeating tedious setup and teardown steps. However, files are surely not the only resources that need to be properly managed. Network I/O via sockets is another place where this technique can come in handy.

On the client side, we'd like to be able to create a method that allows us to send a message to a server, return its response, then cleanly close the connection. The first thing that comes to mind is something simple like this:

```
require "socket"

class Client

  def initialize(ip="127.0.0.1",port=3333)
    @ip, @port = ip, port
  end

  def send_message(msg)
    socket = TCPSocket.new(@ip,@port)
    socket.puts(msg)
    response = socket.gets
  ensure
    socket.close
  end

end
```

This is reasonably straightforward, but what happens when we want to add another method that waits to receive a message back from the server?

```
require "socket"

class Client

  def initialize(ip="127.0.0.1",port=3333)
    @ip, @port = ip, port
  end

  def send_message(msg)
    socket = TCPSocket.new(@ip,@port)
    socket.puts(msg)
    response = socket.gets
  ensure
    socket.close
  end

  def receive_message
    socket = TCPSocket.new(@ip,@port)
    response = socket.gets
  ensure
    socket.close
  end

end
```

This is starting to look messy, as we have repeated most of the code between `send_message` and `receive_message`. Ordinarily, we'd break off the shared code into a private method that the two could share, but the trouble here is that the difference between these two methods is in the middle, not in a single extractable chunk. This is where blocks come to the rescue:

```
require "socket"

class Client
  def initialize(ip="127.0.0.1",port=3333)
    @ip, @port = ip, port
  end

  def send_message(msg)
    connection do |socket|
      socket.puts(msg)
      socket.gets
    end
  end
  def receive_message
    connection { |socket| socket.gets }
  end

  private

  def connection
    socket = TCPSocket.new(@ip,@port)
    yield(socket)
  ensure
    socket.close
  end

end
```

As you can see, the resulting code is a lot cleaner. As long as we use our `connection()` method with a block, we won't need to worry about opening and closing the `TCPSocket`—it'll handle that for us. This means we've captured that logic in one place, and can reuse it however we'd like.

To make things a bit more interesting, let's take a look at a simple server with which this code can interact, which gives us a chance to look at yet another way that blocks can be useful in interface design.

## Blocks As Dynamic Callbacks

There is a lot of power in being able to pass around code blocks just like they were any other object. This allows for the capability of creating and storing dynamic callbacks, which can later be looked up and executed as needed.

In order to play with our `Client` code from the previous example, we're going to create a trivial `TCPServer` that attempts to match incoming messages against patterns to determine how it should respond. Rather than hardcoding behaviors into the server itself or relying on inheritance to handle responses, we will instead allow responses to be defined through ordinary method calls accompanied by a block. Our goal is to get an interface that looks like this:

```
server = Server.new

server.handle(/hello/i) { "Hello from server at #{Time.now}" }
server.handle(/goodbye/i) { "Goodbye from server at #{Time.now}" }
server.handle(/name is (\w+)/) { |m| "Nice to meet you #{m[1]}!" }

server.run
```

The first two examples are fairly simple, matching a single word and then responding with a generic message and timestamp. The third example is a bit more interesting, repeating the client's name back in the response message. This will be accomplished by querying a simple `MatchData` object, which is yielded to the block.

Though making this work might seem like black magic to the uninitiated, a look at its implementation reveals that it is actually a fairly pedestrian task:

```
class Server

  def initialize(port=3333)
    @server   = TCPServer.new('127.0.0.1',port)
    @handlers = {}
  end

  def handle(pattern, &block)
    @handlers[pattern] = block
  end

  def run
    while session = @server.accept
      msg = session.gets
      match = nil

      @handlers.each do |pattern,block|
        if match = msg.match(pattern)
          break session.puts(block.call(match))
        end
      end

      unless match
        session.puts "Server received unknown message: #{msg}"
      end
    end
  end

end
```

The `handle()` method slurps up the provided block using the `&block` syntax, and stores it in a hash keyed by the given pattern. When `Server#run` is called, an endless loop is started that waits for and handles client connections. Each time a message is received, the hash of handlers is iterated over. If a pattern is found that matches the message, the associated block is called, providing the match data object so that the callback can respond accordingly.

If you'd like to try this out, use the following code to spin up a server:

```
server = Server.new

server.handle(/hello/i) { "Hello from server at #{Time.now}" }
server.handle(/goodbye/i) { "Goodbye from server at #{Time.now}" }
server.handle(/name is (\w+)/) { |m| "Nice to meet you #{m[1]}!" }

server.run
```

Once you have that running and listening for connections, execute the following client code:

```
client = Client.new
```

```
["Hello", "My name is Greg", "Goodbye"].each do |msg|
  response = client.send_message(msg)
  puts response
end
```

You will get back something like this:

```
Hello from server at Wed Jul 23 16:15:37 -0400 2008
Nice to meet you Greg!
Goodbye from server at Wed Jul 23 16:15:37 -0400 2008
```

It would be easy to extend both the client and server to do more interesting things that build on this very simple foundation. Feel free to take a few minutes to play around with that,[*] and then we'll look at one more block trick that's fairly common in Ruby.

## Blocks for Interface Simplification

Does it feel like the word "server" is written too many times in this code?

```
server = Server.new

server.handle(/hello/i) { "Hello from server at #{Time.now}" }
server.handle(/goodbye/i) { "Goodbye from server at #{Time.now}" }
server.handle(/name is (\w+)/) { |m| "Nice to meet you #{m[1]}!" }

server.run
```

When you see code like this, it might be a sign that you could do better. Although there are merits to this somewhat standard approach, we can cheat a little bit with blocks (of course) and make things prettier. It would be nice to be able to write this instead:

```
Server.run do
  handle(/hello/i) { "Hello from server at #{Time.now}" }
  handle(/goodbye/i) { "Goodbye from server at #{Time.now}" }
  handle(/name is (\w+)/) { |m| "Nice to meet you #{m[1]}!" }
end
```

As you may recall from an earlier example, it is possible to execute a block within the scope of an instantiated object in Ruby. Using this knowledge, we can implement this handy shortcut interface as a simple class method:

```
class Server

  # other methods same as before

  def self.run(port=3333,&block)
    server = Server.new(port)
    server.instance_eval(&block)
    server.run
  end

end
```

[*] Of course, you might want to look at `GServer` in the standard library for a real generic server implementation.

This is all you need to get the new interface running, and rounds off our quick exploration of the different ways that you can use blocks to improve your API design while simplifying your method implementations.

Keep the following things in mind when using blocks as part of your interface:

- If you create a collection class that you need to traverse, build on top of `Enumerable` rather than reinventing the wheel.
- If you have shared code that differs only in the middle, create a helper method that yields a block in between the pre/postprocessing code to avoid duplication of effort.
- If you use the `&block` syntax, you can capture the code block provided to a method inside a variable. You can then store this and use it later, which is very useful for creating dynamic callbacks.
- Using a combination of `&block` and `instance_eval`, you can execute blocks within the context of arbitrary objects, which opens up a lot of doors for highly customized interfaces.
- The return value of `yield` (and `block.call`) is the same as the return value of the provided block.

Between clever use of code blocks and powerful argument processing, Ruby makes designing beautiful interfaces a joy. However, it takes a little more than this to really complete the picture. Before we wrap things up, let's take a quick look at some common conventions for naming your methods as well as what to do when things go wrong.

## Avoiding Surprises

Though Ruby is a language that embraces the TIMTOWTDI[†] concept, it is also one that seeks the "Ruby Way" of doing things. In this section are a number of miscellaneous tips to help you move your API in that direction.

### Use attr_reader, attr_writer, and attr_accessor

In Ruby, there is no direct external access to the internal state of objects. This means that it is necessary for you to provide public accessors for your internal objects.

Technically, the following code does that just fine:

```
class Message

  def initialize(m)
    @message = m
  end
```

† "There Is More Than One Way To Do It."

```
  def get_message
    @message
  end

  def set_message(m)
    @message = m
  end

end

>> m = Message.new('foo')
=> #<Message:0x603bf0 @message="foo">
>> m.get_message
=> "foo"
>> m.set_message('bar')
=> "bar"
>> m.get_message
=> "bar"
```

However, this approach is almost never seen in code written by practicing Rubyists. Instead, you'll see the preceding code example implemented like this:

```
class Message

  attr_accessor :message

  def initialize(m)
    @message = m
  end

end

>> m = Message.new('foo')
=> #<Message:0x5f3c50 @message="foo">
>> m.message
=> "foo"
>> m.message = "bar"
=> "bar"
>> m.message
=> "bar"
```

Aside from requiring less typing overall, this code is very clear and expressive, because it doesn't include the unnecessary `get` and `set` verbs. However, you might wonder how to do data verification/protection with this approach.

If you need to add some special logic on write, you can still use `attr_reader` to provide the reading side of things and then use a custom method to handle the writing:

```
class Message
  attr_reader :message

  def message=(m)
    @message = m.dup
  end
end
```

On the other hand, if you need to do some handling on read but can afford to use the default writer, `attr_writer` is what you want:

```
class Message
  attr_writer :message

  def message
    @message.encode!("UTF-8")
  end
end
```

Of course, if you need both custom readers and writers, there is no need for the `attr_*` helpers. However, in this case, remember that unless there is a good reason to name things otherwise, use the methods `something()` and `something=()` instead of `get_something()` and `set_something()`.

## Understand What method? and method! Mean

In Ruby, question marks and exclamation points are allowed at the end of method names. Although there is no doubt that Matz wants us to be able to express ourselves freely, these special characters have conventional baggage that comes along with them, and it is useful to honor these conventions when developing your own interfaces.

### Question marks

The purpose of the question mark is pretty straightforward. It allows us to query our object about things and make use of the response in conditionals. In essence, it allows things like this:

```
unless some_string.empty?
  puts some_string.reverse
end
```

In practice, the exact way that this sort of method is implemented varies, but the return value is always some sort of logical boolean. If you write a method that looks like `foo.is_dumb?` that returns `:no`, most Rubyists will disagree with you. If the condition described by the method is not satisfied, be sure that it returns either `false` or `nil`.

Purists might say that when you use this convention, the result should return boolean objects, meaning `true` and `false` only. In this case, a hack for converting Ruby objects to their boolean values is often used:

```
>> !!(:blah)
=> true
>> !!(false)
=> false
>> !!(nil)
=> false
>> !!(123)
=> true
```

This hack is somewhat controversial, but will negate the negation of the boolean status of your object, giving you back a boolean. So one might write a method like this:

```
def person?
   !! @person
end
```

However, there is something to be said for the other side of this argument. Sometimes it is useful to return the actual object, as in the following case:

```
if user = foo.person?
  user.say_hello
end
```

This is ultimately a matter of personal taste, but people on both sides of the fence agree that the use of a question mark in a Ruby method should return some logically boolean value that can be used meaningfully in conditional statements. If that is not the purpose you had in mind, consider avoiding the question mark in your method names.

### Exclamation points

Most people tend to conceptually grasp the question mark convention fairly quickly, but the use of exclamation points is sometimes a little less intuitive. However, the convention itself is not that complicated and can be quite useful.

A common misconception is that we use the exclamation point when we want to let people know we are modifying the receiving object. This is probably due to the fact that, in many cases, this is what the exclamation point (also known as a bang) is warning us of. Here are just a few examples from Ruby's built-in classes:

```
>> a = "foo"
=> "foo"
>> a.delete!("f")
=> "oo"
>> a
=> "oo"

>> a = [1,2,3]
=> [1, 2, 3]
>> a.map! { |e| e + 1 }
=> [2, 3, 4]
>> a
=> [2, 3, 4]

>> a = { foo: "bar" }
=> {:foo=>"bar"}
>> a.merge!(baz: "foobar")
=> {:foo=>"bar", :baz=>"foobar"}
>> a
=> {:foo=>"bar", :baz=>"foobar"}
```

However, what about `Hash#update`?

```
>> a = { foo: "bar" }
=> {:foo=>"bar"}
>> a.update(baz: "foobar")
=> {:foo=>"bar", :baz=>"foobar"}
>> a
=> {:foo=>"bar", :baz=>"foobar"}
```

This does the same thing as `Hash#merge!`, but no bang is present. I can think of tons of other examples where this is true. `String#replace` doesn't have a bang, and neither does `Array#push` or `Array#pop`. If the convention was really to slap an exclamation point at the end of every method that changed something about its receiver, we'd have more exclamation points in Ruby's method list than a teenager could use in an IM session.

Truthfully, the purpose of this convention is to mark a method as special. It doesn't necessarily mean that it will be destructive or dangerous, but it means that it will require more attention than its alternative. This is why it doesn't make much sense to have some method `foo!()` without a corresponding `foo()` method that does something similar. So essentially, if you have only one way of doing something destructive, write this:

```
class Message
  def destroy
     #...
  end
end
```

instead of this:

```
class Message
  def destroy!
    #...
  end
end
```

Following this idea that an exclamation point doesn't necessarily mean that a method is doing a *destructive* operation, we can find more varied uses for it. For example, if you look at the way the ActiveRecord object-relational mapping (ORM) works, you can see a good example of a proper use for this convention.

Creating a user that doesn't pass validations does not raise an exception, but rather stores issues in an errors array, and allows you to check whether a record is valid:

```
>> a = User.create(:login => "joe")
=> #<User id: nil, login: "joe", ... >
>> a.valid?
=> false
```

By calling `User.create!`, we can cause ActiveRecord to raise an error:

```
>> a = User.create!(:login => "joe")
ActiveRecord::RecordInvalid: Validation failed: Password confirmation can't be
blank, Password can't be blank,
Password is too short (minimum is 7 characters), Password Must include at
least three of the following character types: upper case, lower case, numeric,
non alphanumeric, Email can't be blank, Email is too short (minimum is 3
characters)
```

These two methods are functionally equivalent otherwise, but the latter has a more severe response, which exactly fits the conditions under which this convention is useful. Essentially, when you see a ! at the end of a Ruby method, think "Pay attention!" rather than "You are on your way to destruction!" and you'll be fine.

## Make Use of Custom Operators

Ruby allows you to define custom operators for your classes. This is especially easy in Ruby because most operators are actually just syntactic sugar for ordinary methods:

```
>> 1.+(3)
=> 4

>> [1,2,3].<<(4)
=> [1, 2, 3, 4]
```

We can thus define our operators as if they were ordinary methods. Here's a quick example of one of the most common operators to implement, the append operator (`<<`):

```
class Inbox

  attr_reader :unread_count

  def initialize
    @messages     = []
    @unread_count = 0
  end

  def <<(msg)
    @unread_count += 1
    @messages << msg
    return self
  end

end

>> i = Inbox.new
=> #<Inbox:0x603290 @messages=[], @unread_count=0>
>> i << "foo" << "bar" << "baz"
=> #<Inbox:0x603290 @messages=["foo", "bar", "baz"], @unread_count=3>
>> i.unread_count
=> 3
```

A good habit to get into is to have your `<<` method return the object itself, so the calls can be chained, as just shown.

Another good operator to know about is the *spaceship operator* (`<=>`), mainly because it allows you to make use of `Comparable`, which gives you a host of comparison methods: `<`, `<=`, `==`, `!=`, `>=`, `>`, and `between?()`.

The spaceship operator should return `-1` if the current object is less than the object it is being compared to, `0` if it is equal, and `1` if it is greater. Most of Ruby's core objects that can be meaningfully compared already have `<=>` implemented, so it's often simply a matter of delegating to them, as shown here:

```
class Tree

  include Comparable

  attr_reader :age

  def initialize(age)
    @age = age
  end

  def <=>(other_tree)
    age <=> other_tree.age
  end

end

>> a = Tree.new(2)
=> #<Tree:0x5c9ba8 @age=2>
>> b = Tree.new(3)
=> #<Tree:0x5c7fb0 @age=3>
>> c = Tree.new(3)
=> #<Tree:0x5c63b8 @age=3>

>> a < b
=> true
>> b == c
=> true
>> c > a
=> true
>> c != a
=> true
```

You can, of course, override some of the individual operators that `Comparable` provides, but its defaults are often exactly what you need.

Most operators you use in Ruby can be customized within your objects. Whenever you find yourself writing `append()` when you really want `<<`, or `add()` when you really want `+`, consider using your own custom operators.

None of the conventions mentioned here are set laws that need to be followed; in fact, you'll certainly run into situations where it'll make sense to violate some of them from time to time. However, generally these practices have become popular because they make your code better, and make it easier for someone who has never used your code before to get up and running. Here's a quick recap of some of the tips we've covered:

- Use `attr_reader`, `attr_writer`, and `attr_accessor` whenever possible, and avoid writing your own accessors unless it is necessary.
- Consider ending methods that are designed to be used in conditional statements with a question mark.

- If you have a method `foo()`, and a similar method that does nearly the same thing but requires the user to pay more attention to what's going on, consider calling it `foo!()`.
- Don't bother creating a method `foo!()` if there is not already a method called `foo()` that does the same thing with less severe consequences.
- If it makes sense to do so, define custom operators for your objects.

## Conclusions

The difficulty of designing a solid API for a given problem depends largely on the problem itself. However, we've seen in this chapter that Ruby is pretty much happy to get out of your way and provide you with an enormous amount of flexibility so that you can more easily design what you had in mind, rather than what Matz thinks you should do.

When developing your interfaces, be sure to actually use them in order to drive them along. In this way, you are forced to eat your own dog food, and this ensures that the API ends up satisfying the goal of working nicely rather than simply looking nice from a distance. You can gain a lot of inspiration by looking at the way in which the core Ruby objects are designed, API-wise. The best way to make your code more Rubyish is to make it work like core Ruby objects do whenever you can. Of course, like anything else, trying to stretch this idea too far can be disastrous. In moderation, however, this general approach combined with the technical details in this chapter should put you on your way to writing solid Ruby libraries in no time, or, failing that, should clean up your existing code a bit and make it easier to work with.