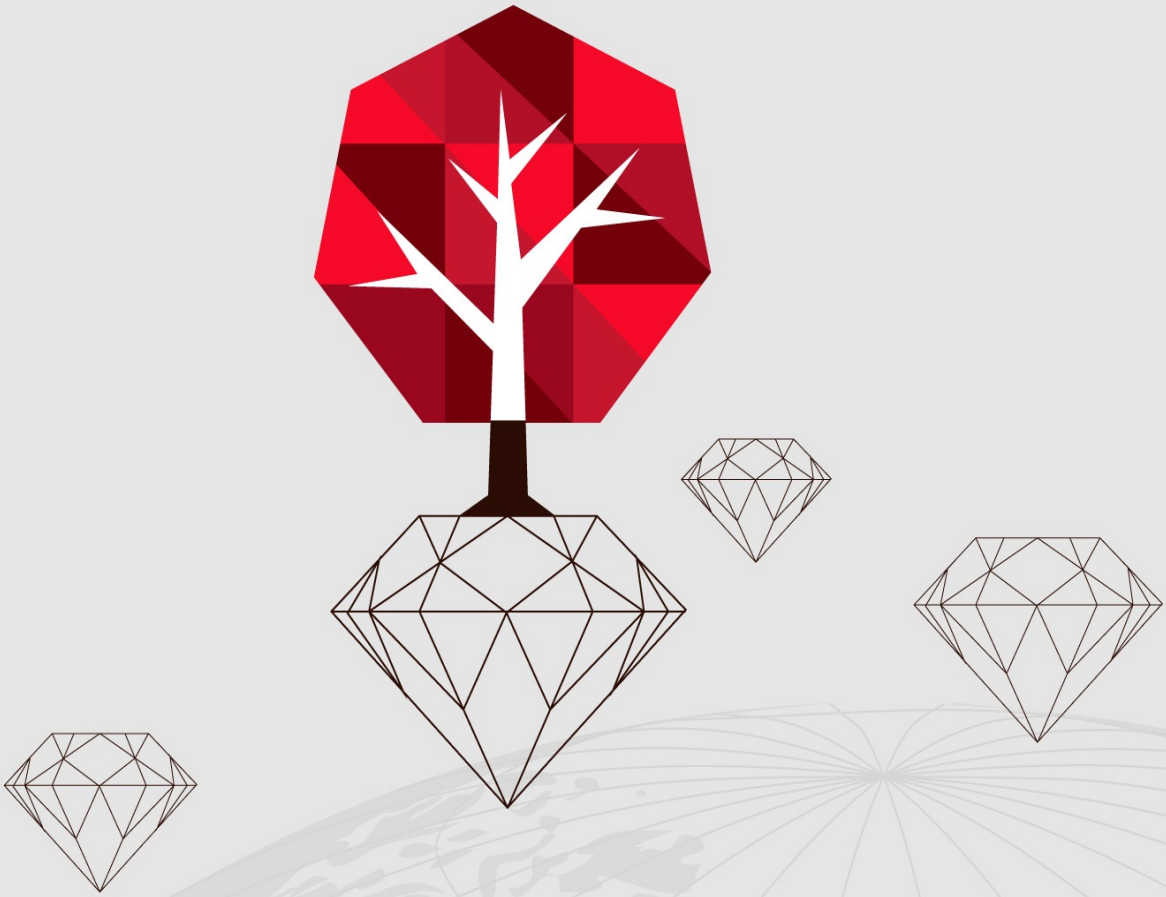
asian tech *inc.*



Ruby Training Book

Version 1.0

Table of Contents

Introduction

Table of Content

Who we are

Asian Tech Co., Ltd.

> We at Asian Tech want to be at the forefront of the IT revolution in Vietnam. Our engineers work not only for professional and technological development, but also aspire to be part of an internationally expanding economy that is shaping the future of their country.
>
> At our current speed of growth, we see ourselves to be the biggest and most successful company in Vietnam by 2020. But for us, it is about more than just numbers. We want to create a dynamic environment where all our employees have the opportunity to expand their horizons globally and to use their experience at Asian Tech to shape their career path, wherever it may lead.

Why we create this book

This book is made public as part of the Asian Tech Ruby and Front End team's effort to:

- Deliver an organized and free guideline for everyone in the community who is currently and will be doing web development in the future. As the web-development space can easily outgrowth developer's ability to grasp, content in this book has been carefully placed in the most logical and systematic that we could think of.
- Deliver Study material for our training regime here at Asian Tech. If you find this book helpful, feel free to pass it around however please note that what you see here does not include our Teaching material.

More about Asian Tech's Ruby and Front End Training Program

Targeted Audience

Our training courses target to: **Interns** and **Associate Software Engineer**

Definitions for Interns and Associate Software Engineer at AT:

Audience	Definition
Interns	Interns are usually university students, or university graduates who have not yet found employment. Interns are less frequently college students (under 18) or older "career changers".
Associate Software Engineers	Just graduated engineer with major in IT who has less or no working experience. The employee is able to understand and perform work assignments with limited guidance and support from supervisor or manager. The engineer will participate in coding, scripting and executing unit test, and fix bugs for assigned project.

After reading this book

After finishing the content in this book, if you feel curious about us and want to find out about opportunities to work here at Asian Tech feel free to drop by our Career page at http://asiantech.vn/en/careers. The back-end language in this book does target Ruby. However if you are not familiar with Ruby, you can still benefit from this book because the Front-end chapter has been compiled very extensively.

What is a Web Developer?

> A web developer is a programmer who specializes in, or is specifically engaged in, the development of World Wide Web applications, or distributed network applications that are run over HTTP from a web server to a web browser.

Learning Web Development

> Whether you are a complete beginner or not, web development can be challenging — we will hold your hand and provide enough detail for you to feel comfortable and learn the topics properly.

Web development can be devided into three parts:

- *Client-side scripting*, which is code that executes in a web browser and determines what your customers or clients will see when they land on your website.
- *Server-side scripting*, which is code that executes on a web server and powers the behind-the-scenes mechanics of how a website works.
- *Database technology*, which also helps keep a website running smoothly.

Throughout the hard learning time, you may find yourself fit into:

- Front End Development
- Back End Development

But at the beginning of the journey, we encourage and challenge you to become **One Web Developer** by all means!

Into Ruby Development

Table of Content

What is a Ruby Developer?

> A Ruby developer architects and develops websites and applications.

Section 1: Evironment Devloper

Table of Content

Install Developer Environment

Table of Content

Install Ruby

The first step is to install some dependencies for Ruby.

```
sudo apt-get install git-core curl zlib1g-dev build-essential libssl-dev libreadline-d
ev libyaml-dev libsqlite3-dev sqlite3 libxml2-dev libxslt1-dev libcurl4-openssl-dev py
thon-software-properties libffi-dev
```

Next we're going to be installing Ruby using one of three methods. Each have their own benefits, most people prefer using rbenv these days, but if you're familiar with rvm you can follow those steps as well. I've included instructions for installing from source as well, but in general, you'll want to choose either rbenv or rvm.

Choose one method. Some of these conflict with each other, so choose the one that sounds the most interesting to you, or go with my suggestion, rbenv.

Now I will install Ruby 2.3.1

Method 1: Install with rbenv

Installing with rbenv is a simple two step process. First you install rbenv, and then ruby-build:

```
cd
git clone https://github.com/rbenv/rbenv.git ~/.rbenv
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(rbenv init -)"' >> ~/.bashrc
exec $SHELL

git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
echo 'export PATH="$HOME/.rbenv/plugins/ruby-build/bin:$PATH"' >> ~/.bashrc
exec $SHELL

rbenv install 2.3.1
rbenv global 2.3.1
ruby -v
```

Method 2: Install with rvm

The installation for rvm is pretty simple:

```
sudo apt-get install libgdbm-dev libncurses5-dev automake libtool bison libffi-dev
gpg --keyserver hkp://keys.gnupg.net --recv-keys 409B6B1796C275462A1703113804BB82D39DC
0E3
curl -sSL https://get.rvm.io | bash -s stable
source ~/.rvm/scripts/rvm
rvm install 2.3.1
rvm use 2.3.1 --default
ruby -v
```

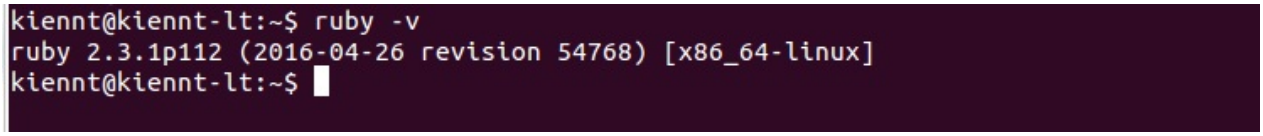
Method 3: Install from source

Arguably the least useful Ruby setup for development is installing from source, but I thought I'd give you the steps anyways:

```
cd
wget http://ftp.ruby-lang.org/pub/ruby/2.3/ruby-2.3.1.tar.gz
tar -xzvf ruby-2.3.1.tar.gz
cd ruby-2.3.1/
./configure
make
sudo make install
ruby -v
```

Last step

Check ruby version



The last step is to install Bundler

```
gem install bundler
```

Setting Up MySQL

You can install MySQL server and client from the packages in the Ubuntu repository. As part of the installation process, you'll set the password for the root user. This information will go into your Rails app's database.yml file in the future.

```
sudo apt-get install mysql-server mysql-client libmysqlclient-dev
```

Installing the libmysqlclient-dev gives you the necessary files to compile the mysql2 gem which is what Rails will use to connect to MySQL when you setup your Rails app.

Setting Up PostgreSQL

For PostgreSQL, we're going to add a new repository to easily install a recent version of Postgres.

```
sudo sh -c "echo 'deb http://apt.postgresql.org/pub/repos/apt/ xenial-pgdg main' > /et
c/apt/sources.list.d/pgdg.list"
wget --quiet -O - http://apt.postgresql.org/pub/repos/apt/ACCC4CF8.asc | sudo apt-key
add -
sudo apt-get update
sudo apt-get install postgresql-common
sudo apt-get install postgresql-9.5 libpq-dev
```

The postgres installation doesn't setup a user for you, so you'll need to follow these steps to create a user with permission to create databases. Feel free to replace chris with your username.

```
sudo -u postgres createuser chris -s

# If you would like to set a password for the user, you can do the following
sudo -u postgres psql
postgres=# \password chris
```

View more

OS command line

Ubuntu

SN	Command lines	Description
1	`sudo apt-get install APPLICATION`	where APPLICATION is the name of an app; it installs APPLICATION (an application)
2	`sudo apt-get remove APPLICATION`	it removes an application
3	`sudo apt-get update`	it updates the repositories
4	`sudo apt-get upgrade`	upgrades your installed application with their latest versions from Ubuntu repositories
5	`killall APPLICATION_NAME`	kills (terminates an application)
6	`kill APPLICATION_PID`	kills an application; where APPLICATION_PID can be found by typing the next command below
7	`ps -e`	displays currently running processes
8	`wget http://path_to_file.com`	downloads a file from the web to current directory
9	`cd /PATH/TO/DIR`	changes current directory to DIR. Use cd to change the current directory into any dir
10	`cd ..`	Like ms-dos, goes up one directory
11	`dir`	or ls: lists directory content
12	`man COMMAND`	displays the manual for a command
13	`cp ORIGINALFILE NEWFILE`	Copy a file
14	`mv SOURCE DESTINATION`	Move a file
15	`rm – FILENAME`	Remove a file
16	`rm -r DIRNAME`	Removes a directory and all it's contents
17	`mkdir FOLDERNAME`	Make directory
18	`free`	Brag about how much free RAM you have on Linux
19	`whereis [app]`	Where is that application I just installed (all directories)
20	`df -h`	Disk space usage
21	`ls -R > playlist.m3u`	To make a playlist of the audio files in a folder
22	`ps aux | sort -nrk 4 | head`	Display the top ten running processes - sorted by memory usage

Editor for Developer

Sublime 3

Common shortcut for Sublime 3

Key to the Keys

```
⌘ : Command key
^ : Control key
⌫ : Delete key
↓ : Down arrow key
⌥ : Option or Alt key
↩ : Return or Enter key
⇧ : Shift key
↑ : Up arrow key
```

SN	Command	MacOSX	Window	Ubuntu
1	Open Cmd Prompt	⌘ + ⇧ + P	Ctrl + ⇧ + P	Ctrl + ⇧ + P
2	Toggle Side Bar	⌘ + K, ⌘ + B	Ctrl + KB	Ctrl + KB
3	Show Scope (Status Bar)	^ + ⇧ + P	Ctrl + ⇧ + Alt + P	Ctrl + ⇧ + Alt + P
4	Select File Language	⌘ + ⇧ + P [language]	Ctrl + ⇧ + P [language]	Ctrl + ⇧ + P [language]
5	Delete Line	⌘ + X	Ctrl + X	Ctrl + X
6	Insert Line After	⌘ + ↩	Ctrl + ↩	Ctrl + ↩

View more

Essential Basic

Table of Content

Basic Syntax

Declaring Variable Names

In ruby, when declaring variable names always use snake*case, not camelCase. Each time you need to name something that requires multiple words, separate those words using underscores* , and make the whole title lowercase. Names should never start or contain uppercase letters.

Comments

In ruby a single line comment looks like this:

```
# this is a single line comment
```

A multiple line comment looks like this:

```
=begin
This is a multiple line comment
This is line number two
This is line number three
=end
```

Symbols

Symbols are primarily used as hash keys, or to reference method names. Once a symbol is created, it can not be changed, and only one copy of a symbol can exist at any given time. There can not be multiple symbols with the same value. Symbols look like this: :symbol. They have a colon, and then are followed by a name.

Prints, Puts, Return, & Yield

Prints:

The print command takes whatever instructions you give it, and puts it on the screen, on the same line.

```
print "Hello world!"
print "Hello world!"
```

Output:

```
"Hello world!Hello world!"
```

Puts

The puts command creates a new line for each thing that you have it print.

```
puts "Hello world!"
puts "Hello world!"
```

Output:

```
"Hello world!"
"Hello world!"
```

Return

The return command just returns the value of something. If you don't tell ruby what to return, it will always return the last expression in the method code block.

```
def double(n)
  return n * 2
end

output = double(6)
output += 2
puts output
# puts ==> 14
```

In the example above, we defined a method called double. Inside the method, we return n * 2. We then set a variable output which is equal to double(6). with the argument of 6. We then add two to output, and we puts the variable output. If you don't tell ruby what to return, it will always return the result of the last expression in the method code block.

Yield

The yield command allows for methods (that don't have the capability already) to accept a block of code. Methods that accept blocks have a way of transferring control from the method to the block and back to the method again. You can build this into methods by using the yield command.

```
def block_test
    puts "We're in the method!"
    puts "Yielding to the block…"
    yield
    puts "We're back in the method!"
end

block_test { puts "--- We're in the block!" }
```

Output

```
"We're in the method!"
"Yielding to the block…"
"--- We're in the block!"
"We're back in the method!"
```

You can also pass parameters to the yield command.

```
def yield_name(name)
    puts "In the method! Let's yield."
    yield("Kim")
    puts "In between the yields!"
    yield(name)
    puts "Block complete! Back in the method."
end

yield_name("Eric") { |n| puts "My name is #{n}." }
```

Output

```
"In the method! Let's yield!"
"My name is Kim."
"In between the yields!"
"My name is Eric."
"Block complete! Back in the method!"
```

The yield_name method is defined with one parameter, name. On line 8, we call the yield_name method and supply the argument "Eric" for the name parameter. Since yield_name has a yield statement, we will also need to supply a block. Inside the method, on line 2, we first puts an introductory statement. Then we yield to the block and pass in "Kim".

In the block, n is now equal to "Kim" and we puts out "My name is Kim." Back in the method, we puts out that we are in between the yields. Then we yield to the block again. This time, we pass in "Eric" which we stored in the name parameter. In the block, n is now equal to "Eric" and we puts out "My name is Eric." Finally, we `puts out a closing statement.

Ruby Loops

Table of Contents

A 'for' loops

```
for i in 1..50
    puts "#{i}"
end
```

1. For every number between 1 - 50
2. Using the inclusive .. operator tells the computer to include the last number in the loop
3. Will puts numbers 1 - 50
4. Using the exclusive ... operator tells the computer to exclude the last number from the loop
5. Will puts the numbers 1-49
6. Display the number on the console
7. Ends instructions for the loop

While loops

```
i = 1
while i <= 50
    puts "#{i}"
    i += 1
end
```

1. Sets variable i equal to one
2. While i is less than or equal to 50, do the block of code (will print numbers 1 -50)
3. Display the number on the console`
4. Increment i by one every time, that way the loop will end at some point

5. Ends instructions for the loop

Until loops

```
i = 1
until i > 50
    puts "#{i}"
    i += 1
end
```

1. Sets variable i equal to one
2. Until i is greater than 50, run the block of code (will print numbers 1-50)
3. Display i on the console
4. Increment i by one every time, that way the loop will end at some point
5. End the instructions for the loop

Loop loops

```
i = 1
loop do
    puts "#{i}"
    i += 1
    break if i > 50
end
```

1. Sets variable i equal to one
2. Create a loop, and run the block of code
3. Display i on the console
4. Increment i by one every time, that way the loop will end at some point
5. Break the loop if i becomes greater than 50
6. End the instructions for the loop

.times iterator loop

```
5.times do
    puts "Hello world!"
end
```

1. Run this loop five times, and do the block of code
2. Display the string on the console
3. End the instructions for this loop

Data Structures

This chapter will cover the different ways to store data in Ruby. These different ways include Arrays, Hashes, and Blocks.

Arrays

General Syntax:

If you want to save a range of numbers, or pack multiple values into one variable, then you use the data structure called an array. An array is a list of items between square brackets. Data inside an array does not have to be in any order, and arrays can hold any type of data, whether it be a string, an integer, booleans, objects, even additional arrays.

Array syntax looks like this:

```
example_array = ["String", 5, "Birds", 231, "baseball"]
```

Accessing Items in an Array:

You can access items inside an array like so: example_array[0]. Arrays start counting at number 0. So the element in the first position is at index 0, then the seconds is at index 1, and the third is at index 2, etc.

Adding to An Array:

Adding to an array can be done in two ways: The first looks like this: example_array.push("string") orexample_array << "string".

Iterating over Arrays:

Iterating over arrays uses syntax that looks like this: example_array { |i| puts i }

Iterating over multi-dimensional arrays is a little bit different:

```ruby
# s stands for sandwiches
s = [["ham", "swiss"], ["turkey", "cheddar"], ["roast beef", "gruyere"]]
# if we just wanted to access "swiss", we would do this: s[0][1]
# We access "s", then we access the array in the 0th position, then the element in the
 1nth position
# if we wanted to print out every element, we would do this:
s.each do |sub_array|
    sub_array.each do |x|
        puts x
    end
end
# first we iterate over each object inside 's'
# then we iterate over each object inside 'sub_array'
```

Hashes

General Syntax:

Hashes are similar to JavaScript objects or Python dictionaries. Hashes are a collection of key-value pairs. Keys are the names of the values. Values are assigned to keys using the => operator, called a hash-rocket.

Creating a Hash:

You can create an empty hash using constructor notation like so: my_hash = Hash.new. Then you would add to the hash using bracket notation, like so: my_hash["key1"] = value1. If you are adding a string as a value, then you would use quotations, if not, you can leave the value as-is.

Or you can use hash literal notation which looks like this:

```
hash_one = {
    "key1" => value1,
    "key2" => value2,
    "key3" => value3
}
```

Accessing a Hash:

You can access objects inside of a hash just like an array: hash_one["key1"]. All keys need to strings inside brackets, and inside the hash. When using quotes around a key, you are looking for that specific key, inside a hash. When not using quotes, you are looking for a variable, inside that hash.

Iterating over a Hash:

Iterating over hashes needs two placeholder variables to represent each key-value pair. You would use syntax like so:

```
hash_one.each do | key, value |
    puts "#{key} has the value of #{value}"
end
```

Using Symbols as Keys:

Since Ruby 1.9.0 was released, common consensus in the Ruby community is to use symbols as keys, instead of strings. Ruby symbols are sort of like a name. Symbols are not strings. There can be multiple strings that all have the same value, but there can only be one copy of any particular symbol at any given time. Symbols are good for hash keys for a few reasons. Symbols can not be changed once they've been created. Only one copy of any symbol exists at any given time, so they save a lot of memory. Symbols as keys are faster than strings as keys, because of the past two reasons.

For example:

```
movies = {
    top_gun: "Good",
    finding_nemo: "Awesome",
    lion_king: "Great"
}
```

Even though the colon : is at the end of the key, it is still accessed the same. Meaning, if you were to call the key, you would do it like this: movies[:top_gun] or movies[:lion_king]. So you won't be needing the hash-rocket anymore :(. If you are familiar with JavaScript objects or Python dictionaries, then this should look familiar.

Blocks

General Syntax:

Blocks are a set of code that explains what to do. It is nothing more than a set of instructions. Blocks are on of the only things in Ruby that is not an object. Since blocks are not objects, they can not be stored to variables and don't have all the powers that other objects have. Blocks are similar to anonymous functions in JavaScript and lambdas in Python. Blocks can be defined with either the key words: do and end, or with curly braces { }.

For example:

```
1.times do
    puts "I'm a code block!"
end
```

or

```
1.times { puts "I'm a code block too!" }
```

Passing Blocks to Methods:

Passing a block to a method is a great way of abstracting certain tasks from the method and defining those tasks whenwe call a method. Abstraction is an important idea in computer science, and you can think of it as meaning "making something simpler."

For example:

```
# The block, {|i| puts i}, is passed the current array item
# each time it is evaluated. This block prints the item
[1, 2, 3, 4, 5].each { |i| puts i }

# This block prints the number 5 for each item.
# It chooses to ignore the passed item, which is allowed
[1, 2, 3, 4, 5].each { |i| puts 5 }
Whenever the each method is called, it is passed a block as a parameter, with a set of
 instructions as to what needs to be done.
```

Blocks and Methods

You have seen how a block and a method can be associated with each other. You normally invoke a block by using the yield statement from a method that has the same name as that of the block. Therefore, you write:

```
#!/usr/bin/ruby

def test
  yield
end
test{ puts "Hello world"}
```

This example is the simplest way to implement a block. You call the test block by using the yield statement.

But if the last argument of a method is preceded by &, then you can pass a block to this method and this block will be assigned to the last parameter. In case both * and & are present in the argument list, & should come later.

```
def test(&block)
   block.call
end
test { puts "Hello World!"}
```

This will produce the following result:

```
Hello World!
```

Control Flow in Ruby

if Statements:

if statements takes an expression, and runs it to see if it is true or false. If the expression is true, then it runs the block of code. If the expression is false, then it moves onto the next piece of instructions.

An example:

```
if 4 < 5
    puts "Four is less than five."
end
```

If an if statement is short, and only requires one line, then you can use a simpler syntax. The statement has a syntax like this: expression if boolean. It would be said like so: Do this expression if the boolean is true or false (which ever one you put). The order they go in, is extremely important! If they're not in that order, it won't work.

Using the example above, it would look like this:

```
puts "Four is less than five" if 4 < 5
```

if/else Statements:

The partner to the if statement, is the else statement. An if/else statement says to ruby: "if this expression is true, run this code block; otherwise, run the code block after the else.

For example:

```
if 4 > 5
    puts "This should not run because four is not greater than five."

# Realize there is no end statement after this if statement

else
    puts "The if statement is false, so the else code is being ran."
end
```

The more concise version of the if/else statement is called the ternary conditional expression. It is called ternary because it takes three arguments: a boolean, an expression to evaluate if the statement is true, and an expression for if the statement is false. It has syntax which looks like this: boolean ? Do this if true: Do this if false. It would be said like so: Evaluate this boolean, if it's true do the first expression, if it's false do the second expression. The order they go in is extremely important! If they're not in that order, it won't work.

For example:

```
puts 4 < 5 ? "Four is less than five" : "Four is not less than five"
```

elsif Statements

If you need more than just two options, then you can implement an elsif statement.

For example:

```
if x < y  # Assumes x and y are defined
    puts "x is less than y!"
elsif x > y
    puts "x is greater than y!"
else
    puts "x equals y!"
end
```

unless Statements

Sometimes you want to use control flow to see if a statement is false, rather than if it's true. This is where you use the unless statement. Lets say that you don't want to eat unless you're hungry, and when you're not hungry then you want to draw pictures.

You would write something like this:

```
hungry = false

unless hungry
    puts "Let's go draw a picture!"
else
    puts "Let's go get some food."
end
```

If an unless statement is short, and only requires one line, then you can use a simpler syntax. The statement has a syntax like this: expression unless boolean. It would be said like so: Do this expression unless the boolean is true or false (which ever one you put). The order they go in is extremely important! If they're not in that order, it won't work.

Using the example above, it would look like this:

```
hungry = false
puts "lets go draw a picture" unless hungry
```

switch Statements

When you have multiple options, you can use the switch statement. It allows you to put multiple when statements, or conditions, and then which ever one is true, it will run that condition. It is exactly like an elsif statement, but better for situations where there are multiple choices, because we don't want to use too many elsif statements.

The syntax looks like this:

```
puts "What is your favorite color?"
color = gets.chomp

case color
    when 'blue'
        puts "Blue"
    when 'red'
        puts "Red"
    when 'green'
        puts "Green"
    when 'purple'
        puts "Purple"
    when 'yellow'
        puts "Yellow"
    else
        puts "You did not put a color"
end
```

However, there is a simpler way to do it, if the conditions don't have a big block of code. Meaning, if they only have one line of instructions, then you can fold it up into a simpler syntax like so:

```
puts "What is your favorite color?"
color = gets.chomp

case color
    when ‘blue’ then puts “Blue”
    when ‘red’ then puts “Red”
    when ‘green’ then puts “Green”
    when ‘purple’ then puts “Purple”
    when ‘yellow’ then puts “Yellow”
    else puts “You did not put a color”
end
```

Variables

Variables are the memory locations which hold any data to be used by any program.

There are five types of variables supported by Ruby. You already have gone through a small description of these variables in previous chapter as well. These five types of variables are explained in this chapter.

Ruby Global Variables:

Global variables begin with $. Uninitialized global variables have the value nil and produce warnings with the -w option.

Assignment to global variables alters global status. It is not recommended to use global variables. They make programs cryptic.

Here is an example showing usage of global variable.

```
#!/usr/bin/ruby

$global_variable = 10

class Class1

  def print_global

     puts "Global variable in Class1 is #$global_variable"

  end

end

class Class2

  def print_global

     puts "Global variable in Class2 is #$global_variable"

  end

end

class1obj = Class1.new

class1obj.print_global

class2obj = Class2.new

class2obj.print_global
```

Here $global_variable is a global variable. This will produce the following result:

NOTE: In Ruby you CAN access value of any variable or constant by putting a hash (#) character just before that variable or constant.

```
Global variable in Class1 is 10

Global variable in Class2 is 10
```

Ruby Instance Variables:

Instance variables begin with @. Uninitialized instance variables have the value nil and produce warnings with the -w option.

Here is an example showing usage of Instance Variables.

```
#!/usr/bin/ruby

class Customer

   def initialize(id, name, addr)

      @cust_id=id

      @cust_name=name

      @cust_addr=addr

   end

   def display_details()

      puts "Customer id #@cust_id"

      puts "Customer name #@cust_name"

      puts "Customer address #@cust_addr"

    end

end

# Create Objects

cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")

cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# Call Methods

cust1.display_details()

cust2.display_details()
```

Here, @cust_id, @cust_name and @cust_addr are instance variables. This will produce the following result:

```
Customer id 1

Customer name John

Customer address Wisdom Apartments, Ludhiya

Customer id 2

Customer name Poul

Customer address New Empire road, Khandala
```

Ruby Class Variables:

Class variables begin with @@ and must be initialized before they can be used in method definitions.

Referencing an uninitialized class variable produces an error. Class variables are shared among descendants of the class or module in which the class variables are defined.

Overriding class variables produce warnings with the -w option.

Here is an example showing usage of class variable:

```
#!/usr/bin/ruby

class Customer

   @@no_of_customers=0

   def initialize(id, name, addr)

      @cust_id=id

      @cust_name=name

      @cust_addr=addr

      @@no_of_customers += 1

   end

   def display_details()

      puts "Customer id #@cust_id"

      puts "Customer name #@cust_name"

      puts "Customer address #@cust_addr"

    end

    def total_no_of_customers()

       puts "Total number of customers: #@@no_of_customers"

    end

end

# Create Objects

cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")

cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# Call Methods

cust1.total_no_of_customers()

cust2.total_no_of_customers()
```

Here @@no_of_customers is a class variable. This will produce the following result:

```
Total number of customers: 1

Total number of customers: 2
```

Ruby Local Variables:

Local variables begin with a lowercase letter or _. The scope of a local variable ranges from class, module, def, or do to the corresponding end or from a block's opening brace to its close brace {}.

When an uninitialized local variable is referenced, it is interpreted as a call to a method that has no arguments.

Assignment to uninitialized local variables also serves as variable declaration. The variables start to exist until the end of the current scope is reached. The lifetime of local variables is determined when Ruby parses the program.

In the above example local variables are id, name and addr.

Ruby Constants:

Constants begin with an uppercase letter. Constants defined within a class or module can be accessed from within that class or module, and those defined outside a class or module can be accessed globally.

Constants may not be defined within methods. Referencing an uninitialized constant produces an error. Making an assignment to a constant that is already initialized produces a warning.

```
#!/usr/bin/ruby

class Example

   VAR1 = 100

   VAR2 = 200

   def show

       puts "Value of first Constant is #{VAR1}"

       puts "Value of second Constant is #{VAR2}"

   end

end

# Create Objects

object=Example.new()

object.show
```

Here VAR1 and VAR2 are constant. This will produce the following result:

```
Value of first Constant is 100

Value of second Constant is 200
```

Ruby Pseudo-Variables:

They are special variables that have the appearance of local variables but behave like constants. You can not assign any value to these variables.

- **self**: The receiver object of the current method.
- **true**: Value representing true.
- **false**: Value representing false.
- **nil**: Value representing undefined.
- **FILE**: The name of the current source file.
- **LINE**: The current line number in the source file.

Ruby Methods

Since everything in Ruby is an object, everything in Ruby has built in abilities called methods. You can think of methods as "skills" that these objects can perform.

Calling methods on an object:

Calling a method uses syntax like this: object.method where you first type the object you want to use, then use the . operator, and lastly the method you want to use

Defining methods:

It starts with the keyword def, which is short for define. Methods have three parts:

1. The Header - Includes the def keyword, the name of the method, and any arguments the method takes
2. The Body - This is the code block that will be run. It describes the procedures the method carries out
3. End - The method is then closed with the keyword end

An example would look like this:

```
def method_name
    puts “This is a new method!”
end
```

Calling a method:

Calling a method all you have to do is type in the name of the method. Nothing else. So for the example above, if we wanted to call that method, we would do this: method_name on a new line inside the file. That’s all that is needed to call the method, is to just type in the name of it on a new line, and then it will call the method and proceed with the instructions for that method.

Parameters & arguments:

If a method takes arguments, we say that it accepts or expects those arguments. The argument is the actual piece of code you put between the methods parenthesis when you call it. The parameter is the name that you put between the parenthesis when you define it.

An example would look like this:

```
def square(n)
    puts n ** 2
end
square(10)
# ==> prints “100”
```

In the example above, we gave the method square a parameter of n, and when we called the method, we gave it an argument of 10.

Splat Arguments:

On regular methods, you never really know exactly what argument you’re going to get, so you use the parameter as a temporary placeholder. However, what if you don’t know how many arguments you’re going to get? Then you use the splat argument: (*). You place your splat argument then follow it with the parameter you give it. It signals to ruby: “Hey Ruby, I don’t know how many arguments there are about to be, but it could be more than one."

For example:

```
def what_up(greeting, *bros)
    bros.each { |bro| puts "#{greeting}, #{bro}!" }
end
what_up("What up", "Justin", "Ben", "Kevin Sorbo")
```

Output:

```
"What up, Justin!"
"What up, Ben!"
"What up, Kevin Sorbo!"
```

Classes

General Syntax:

Ruby is an object-oriented-programming language (OOP), which means it manipulates programming constructs called objects. All objects have methods and attributes. For example: Matz.length #==> 4. In this case, the “Matz” object is a string, with a method of .length, and an attribute of 4. What makes Matz a string though? The fact that it is an instance of the class String. A class is just a way of organizing and producing objects with similar methods and attributes.

Class syntax looks like this:

```
class ClassName
 # Do something
end
```

Class Names:

Class names start with an uppercase letter and use camelCase instead of snake_case. In this example, we tell ruby that we can’t to create a new class, then we put the ClassName to give the class a name, and then we insert whatever code inside that class. ClassName is now a class, just like “Hello!”, is a String (which is a class) and 4 is a Fixnum (which is also a class).

Initialize Method & Creating a Class:

When creating a class, you want to create a method called initialize. When you create a new instance, it will run the initialize method, and execute any code in that method. initialize. takes however many arguments you tell it to take. In ruby, we use the @ operator before a variable to signify that it is an instance variable. This means that the variable is attached to the instance of the class. They are unique to whatever instance is

For example:

```
class Car
 def initialize(make, model)
  @make = make
  @model = model
 end
 end

time_machine = Car.new(DMC-12, DeLorean)
# ==> This is the DeLorean from Back to The Future
```

In this example, we created an instance time_machine of the class Car. time_machine has its own make of DeLorean and its own model of DMC-12. So these variables only belong to this instance, thus, this makes these variables instance variables.

Creating a New Instance of a Class:

Creating a new instance of a class looks like this: me = Person.new(“Eric”). Where me is a new instance of the class Person, and it takes an argument of “Eric”.

Different Kinds of Variables:

Another important aspect of Ruby classes is scope. The scope of a variable is the context in which it’s visible to the program. Not all variables are accessible to all parts of the program. When dealing with classes, you can have variables that accessible to all parts of the program (global variables), variables that are only available to certain methods (local variables), and variables that are only available to a particular instance of a class (instance variables). The same goes for methods, some are available everywhere, some can only be accessible to members of that class, and some are only accessible to a particular instance.

Naming Your Variables:

Instance variables start with the @operator. Class variables are like instance variables, but instead of belonging to an instance of a class, they belong to the class itself. Class variables always start with the @@ operator. Global variables can be declared in two ways: The first is just to define the variable outside of any classes or methods, then it is global. If you want to make a variable global from inside a method or class, you start it with the $operator.

An example of a class with different kinds of variables:

```
class Person
  @@people_count = 0
  def initialize(name, age, profession)
    @name = name
    @age = age
    @profession = profession
    @@people_count += 1
  end

  def number_of_instances
    return @@people_count
  end
end


batman = Person.new(“Bruce Wayne”, 35, “Superhero”)
superman = Person.new(“Clark Kent”, 35, “Superhero”)
bill_gates = Person.new(“Bill Gates”, 45, “Software Engineer”)

puts Person.number_of_instances
# ==> It would print 3, because we have batman, superman, and bill_gates
```

In this example, we create a class variable called @@people_count and set it to 0. Then we add it to the initialize method, and every time an instance is initialized, it adds one to people_count. We define a new method called number_of_instances and have it return the current @@people_count. We then print the number_of_instances method to the console to see how many instances have been created of the People class.

Class Methods:

Just like how there are class variables, and instance variables, the same is true for methods. A class method belongs to the class as a whole, while an instance method belongs to certain instances.

Here is the comparison between the two:

```
class Person
  @@people = []
  # This is a class variable that is equal to an empty array

  def initialize(name, age, profession)
    @name = name
    @age = age
    @profession = profession
    @@people << name
    # We are pushing the names of the people into the array
  end

  def Person.people_names
    # This is a class method. Syntax: ClassName.method_name
    puts @@people
  end

  # This is an instance method, that only applies to whatever instance we call it on
  def greeting
    puts “Hello, my name is #{name}, I’m #{age} years old and I work as a #{profession
}!”
  end
end

eric = Person.new(“Eric”, 25, “Doctor”)

# Calling them would look like this:

Person.people_names
# We’re printing all the names of every instance of the class People

Person.greeting
# This would not work because Person doesn’t exist, it’s just a class

eric.greeting
# This would work, because Eric is an actual instance of the class Person
```

Real-World Classes:

However, in the real world, most classes will not look like the examples above. They will more closely resemble real-world objects.

Here is an example of a class that would most likely be seen in commercial software:

```
def create_record(attributes, raise_error = false)
  record = build_record(attributes)
  yield(record) if block_given?
  saved = record.save
  set_new_record(record)
  raise RecordInvalid.new(record) if !saved && raise_error
  record
end
```

Notice how it creates an instance of the RecordInvalid class. The syntax should all be familiar, except the raise bit. All it does is generate a new RecordInvalid error if the user tries to create or save an invalid record.

Inheritance:

Inheritance is the process by which one class takes on the methods and attributes of another, and it's used to express an is-a relationship. For example, a human is-a mammal, so it could inherit it's methods and attributes from the mammal class, but a human is not a fox, so it could not inherit its methods and attributes from the fox, despite them both being mammals. The class inheriting from another class is called the sub-class or the derived-class. The class it is inheriting from is called the super-class, the parent, or the base class.

Inheritance syntax works like so:

```
class SubClass < BaseClass
  # Do something
end
```

Where InheritingClass is the new class that is being made, and the BaseClass is the class from which it is inheriting its methods and attributes from. You can read < as "inheriting from".

Overriding Inheritance:

Sometimes you will want one class that inherits from another to not only take on the parent's methods and attributes, but to also override one or more of them. For example: You might have an Email class that inherits from the Message class. Both classes will have a send method, but sending an email might require different protocols that Message doesn't have. Instead of just creating a new method in your Email class specifically for that, you can just create a method named send and it will override the send method that it inherited from Message. Also, sometimes you will be working inside a sub-class and realize you have overwritten a method or attribute from its super-class that you actually need. No problem!

You can directly access a super-class's methods or attributes by using Ruby's built in super keyword. When you call the super keyword form inside a sub-class method or attribute, it will tell Ruby to look in the super-class and find a method or attribute with the same name as the one from which super is called. Ruby will then use the super-class method/attribute instead.

Multiple Inheritances:

Any given Ruby class can only have one parent class. Some languages allow a class to have more than one parent, which is called a multiple inheritance. This can get really ugly really fast, which is why Ruby doesn't allow it. However, there are circumstances where you want to incorporate data or behavior from several classes into one sub-class. Ruby allows this through the use of mix-ins.

For example:

```
class Creature
  def initialize(powers)
    @powers = powers
  end
end

class Human
  def initialize(name)
    @name = name
  end
end

class Dragon < Creature; end
# This is shorthand for creating a class
class Dragon < Human; end
# This is shorthand for creating a class

# ==> Outputs error: "superclass mismatch for class Dragon”
```

We tried to have Dragon inherit from both Human and Creature, but Ruby gave us an error that basically says "Dragon seems to be inheriting from more than one class, please fix this so that Dragon only has one super-class."

Public and Private Methods:

Ruby allows you to explicitly make some methods public, and some methods private. public methods allow for an interface with the rest of the program. They say: "Hey! Ask me if you need to know something about my class or its instances!". private methods on the other hand are for your classes to do their own work undisturbed. They don't want anyone asking

them anything, so they make themselves unreachable. Methods in Ruby are public by default. If you want to make it clear whether your method is public or private, then you have to put either public or private before your method definition.

Like so:

```
class ClassName
  # Class stuff

  public
  def public_method
    # Code here
  end

  private
  def private_method
    # Code here
  end
end
```

Note that everything after the public keyword until the end of the class definition will then be public, unless we state otherwise. Private methods are private to the object that they are defined in. This means you can only call the method from inside other methods within that object. Another way to say it is that the method cannot be called using an explicit receiver. Receivers are the objects that methods are called on. For example: object.method, where object is the receiver of the method. In order to access private information, we have to create public methods that know how to get it. This separates the private implementation from the public interface.

Example of Putting it All Together:

```
class Account
  attr_reader :balance
  attr_reader :name
  def initialize(name, balance=100)
    @name = name
    @balance = balance
  end

  private

  def pin
    @pin = 1234
  end

  def pin_error
    return "Access denied: incorrect PIN."
  end

  public

  def display_balance(pin_number)
    if pin_number == pin
      puts "Balance: #{@balance}"
    else
      puts pin_error
    end
  end

  def withdraw(pin_number, amount)
    if pin_number == @pin
      @blanace -= amount
      puts "Withdrew #{amount}. New balance: #{balance}"
    else
      puts pin_error
    end
  end
end


checking_account = Account.new(“John Doe", 100_000_000)
checking_account.display_balance(1234)
```

Getter Setter in Ruby

Let's say you have a class Person.

```
class Person
end

person = Person.new
person.name # => no method error
```

Obviously we never defined method name. Let's do that.

```
class Person
  def name
    @name # simply returning an instance variable @name
  end
end

person = Person.new
person.name # => nil
person.name = "Dennis" # => no method error
```

We can read the name, but that doesn't mean we can assign the name. Those are two different methods. Former called reader and latter called writer. We didn't create the writer yet so let's do that.

```
class Person
  def name
    @name
  end

  def name=(str)
    @name = str
  end
end

person = Person.new
person.name = 'Dennis'
person.name # => "Dennis"
```

Awesome. Now we can write and read instance variable @name using reader and writer methods. Except, this is done so frequently, why waste time writing these methods every time? We can do it easier.

```
class Person
  attr_reader :name
  attr_writer :name
end
Even this can get repetitive. When you want both reader and writer just use accessor!

class Person
  attr_accessor :name
end

person = Person.new
person.name = "Dennis"
person.name # => "Dennis"
```

Works the same way! And guess what: the instance variable @name in our person object will be set just like when we did it manually, so you can use it in other methods.

```
class Person
  attr_accessor :name

  def greeting
    "Hello #{@name}"
  end
end

person = Person.new
person.name = "Dennis"
person.greeting # => "Hello Dennis"
```

That's it. In order to understand how attr_reader, attr_writer, and attr_accessor methods actually generate methods for you, read other answers, books, ruby docs.

Procs:

Summary:

A proc is a saved block we can use over and over again.

General Syntax:

Since blocks are not objects, and cannot be stored in variables along with all the other powers that objects have, we have procs to fill in this role. Procs are sort of like a “saved block”. Just like you can give a bit of code a name and turn it into a method, you can give blocks a name and turn it into a proc. With a block, you have to write your code out every time you use it. However, with procs, you can store this block code, write it only once, and then use it multiple times!

Creating a Proc:

To create a proc, all you have to do is call Proc.new, and then pass in the block of code that you want to store. Any time you pass a proc to a method that usually expects a block, you have to use the ampersand (&) operator to then call that proc’s instructions (or block of code). Whenever calling a proc in a method that usually takes a block, make sure to put the proc inside parenthesis like so: .some_method(&some_proc). Or a simpler version is to use the .call method. It would look like this: some_proc.call.

For example:

```
multiples_of_3 = Proc.new do |n|
    n % 3 == 0
end

(1..100).to_a.select(&multiples_of_3)

# ==> 3, 6, 9, 12, 15, 18, 21, 24, 27 ...
```

Another example:

```
numbers = [5, 10, 15, 20, 25, 30]

cubed = Proc.new { |n| puts n ** 3 }
numbers.map( &cubed )

# ==> Cubes each element in the number array, and then puts them on the console
```

Blocks vs. Procs:

The benefits of using procs is that since procs are full-fledged objects, they have all the powers and abilities that objects do. Also, unlike blocks, procs can be called over and over again without rewriting them. This saves time and prevents you from having to write the block multiple times.

You can also convert symbols into procs using the ampersand (&) operator.

For example:

```
strings = ["1", "2", "3"]
nums = strings.map(&:to_i)

# ==> [1, 2 ,3]
Another example:

numbers_array = [1, 2, 3, 4, 5]
    strings_array = numbers_array.map(&:to_s)

# ==> ["1", "2", "3", "4", "5"]
```

Lambdas

Summary:

A lambda is just like a proc, only it cares about the number of arguments it gets and it returns to its calling method rather than returning immediately.

General Syntax:

Like procs, lambdas are objects too. With the exception of a little syntax and a few behavioral quirks, lambdas are identical to procs. Lambdas are defined like this: lambda { |param| block }. Lambdas are useful in the same situation that you would use procs.

For example:

```
strings = ["leonardo", "donatello", "raphel", "michaelangelo"]

symbolize = lambda { |name| name.to_sym } nil
strings.collect(&symbolize)

# ==> [:leonardo, :donatello, :raphel, :michaelangelo]
```

Lambdas vs. Procs:

There are not many differences between lambdas and procs. However, there are a few: First, lambdas checks the number of arguments passed to it, while procs do not. This means that lambdas will throw in an error if you pass in the wrong number of arguments, and a proc will just ignore any unexpected arguments and assign nil to any that are missing. Second, when a lambda returns a value, it sends the control back to the calling method. When procs return a value, they do it immediately without going back to the calling method.

An example between procs & lambdas:

```
def batman_ironman_proc
    victor = Proc.new { return "Batman will win!" }
    victor.call
    "Iron Man will win!"
end

puts batman_ironman_proc

def batman_ironman_lambda
    victor = lambda { return "Batman will win!" }
    victor.call
    "Iron Man will win!"
end

puts batman_ironman_lambda

# ==> Batman will win!
# ==> Ironman will win!
```

See how the proc prints out "Batman will win!"? That shows that the proc instantly returns the value and doesn't shift the control back to the method. The lambda however returns the value then gives the control back to the method. Since methods return the last expression if no return is defined, "Ironman will win!" gets printed to the console.

Passing Lambdas to Methods:

Just like procs also, we need to use the ampersand (&) operator at the beginning of the lambda name when we pass it to a method. This will convert the lambda into a block for the method to run.

Another example:

```
my_array = ["raindrops", :kettles, "whiskers", :mittens, :packages]

symbol_filter = lambda { |x| x.is_a? Symbol }
symbols = my_array.select(&symbol_filter)

# ==> [:kettles, :mittens, :packages]
```

Modules

General Syntax:

A module is like a toolbox that contains a set of methods and constants. Modules are very much like classes, except they can't create instances and can't have subclasses. They're just a way to store things that you might use later. It doesn't make much sense to store variables in a module, for by definition, variables change or vary. Constants however, should always stay the same, so including helpful constants inside a module is a good idea. Ruby constants are written in ALL_CAPS. So whenever you are creating a constant, it should be formatted accordingly.

Creating a Module:

Creating a module is almost identical to creating a class. Like class names, module names are written in CapitilizedCamelCase as well.

It looks like this:

```
module ModuleName
    # Insert methods and constants to use later
end
```

Name-spacing & Scope Operators:

One of the main purposes of a module is to separate methods and constants into named spaces. This is called (conveniently), namespacing, and it's how Ruby doesn't confuse Math::PI and Circle::PI. The double colon in the previous example is called a scope resolution operator. It is just a way to tell ruby where you are looking for a specific piece of code. If you say Math::PI, then it tells Ruby to look inside the Math module and search for PI, and not any other PI such as in the module Circle.

Importing External Modules:

Some modules, like MATH, are already present in the interpreter. However, sometimes you will need to be explicitly imported. We can do this using the require keyword. Like so: require 'module'. We can do more than just require a module, we can also include it. Any class that include's a certain module can use that module's methods and constants. You use include like so: include Math. Notice how it doesn't use the quotes to call on the module. Also,

whenever you include a module inside a class, that class no longer has to use the scope resolution operator (::) to define the module to look inside. Instead of writing Math::PI, you can just write PI.

Modules & Classes:

Whenever a module is used to mix in additional information and behavior into a class, it is called a mixin. Mixins allow us to customize a class without having to write additional code.

Example:

```
module Action
    def jump
        @distance = rand(4) + 2
        puts "I jumped forward #{@distance} feet!"
    end
end

class Rabbit
    include Action
    attr_reader :name

    def initialize(name)
        @name = name
    end
end

class Cricket
    include Action
    attr_reader :name

    def initialize(name)
        @name = name
    end
end

peter = Rabbit.new("Peter")
jiminy = Cricket.new("Jiminy")

peter.jump
jiminy.jump
```

Notice how we define the `jump` method inside the Action module, the we include it into the `Rabbit` and `Cricket` class.

Extending Include:

The `include` keyword only mixes that modules methods and information in at the instance level, to where only instances of that particular class can use the information. However, the extend keyword mixes the modules methods into the class level. This means the class itself can use that module's information, as opposed to just instances of that class.

For example:

```
module ThePresent
    def now
        puts "It's #{Time.new.hour > 12 ? Time.new.hour - 12 : Time.new.hour}:#{Time.n
ew.min} #{Time.new.hour > 12 ? 'PM' : 'AM'} (GMT)."
    end
end

class TheHereAnd
    extend ThePresent
end

TheHereAnd.now
```

Notice how we `extended` the `now` method from `ThePresent` module into `TheHereAnd` class. This now allows the class to use the `ThePresent` module. Unlike before, where only instances of `TheHereAnd` class would have been able to use it.

Include vs Extend in Ruby

Let's cover the difference between include and extend in regards to modules. Include is for adding methods to an instance of a class and extend is for adding class methods. Let's take a look at a small example.

```
module Foo
  def foo
    puts 'heyyyyoooo!'
  end
end

class Bar
  include Foo
end

Bar.new.foo # heyyyyoooo!
Bar.foo # NoMethodError: undefined method ‘foo’ for Bar:Class

class Baz
  extend Foo
end

Baz.foo # heyyyyoooo!
Baz.new.foo # NoMethodError: undefined method ‘foo’ for #<Baz:0x1e708>
```

As you can see, include makes the foo method available to an instance of a class and extend makes the foo method available to the class itself.

Include Example

If you want to see more examples of using include to share methods among models, you can read my article on how I added simple permissions to an app. The permissions module in that article is then included in a few models thus sharing the methods in it. That is all I'll say here, so if you want to see more check out that article.

Extend Example

I've also got a simple example of using extend that I've plucked from the Twitter gem. Basically, Twitter supports two methods for authentication—httpauth and oauth. In order to share the maximum amount of code when using these two different authentication methods,

I use a lot of delegation. Basically, the Twitter::Base class takes an instance of a “client”. A client is an instance of either Twitter::HTTPAuth or Twitter::OAuth.

Anytime a request is made from the Twitter::Base object, either get or post is called on the client. The Twitter::HTTPAuth client defines the get and post methods, but the Twitter::OAuth client does not. Twitter::OAuth is just a thin wrapper around the OAuth gem and the OAuth gem actually provides get and post methods on the access token, which automatically handles passing the OAuth information around with each request.

Important Feature and Topics

Table of Contents

Closure

In this chapter, you will learn about the basics of closures and how you can use them to execute code in different execution contexts.

What is closure?

A closure is an anonymous function that carries its creation context where ever it goes.

Block Objects are Closures

Changing the Value Outside a Block

Let's write a simple program to illustrate what happens to the block object when we change the values of a local variable.

```
x = 0
seconds = -&gt; { x }
p seconds.call
```

This prints:

```
0
```

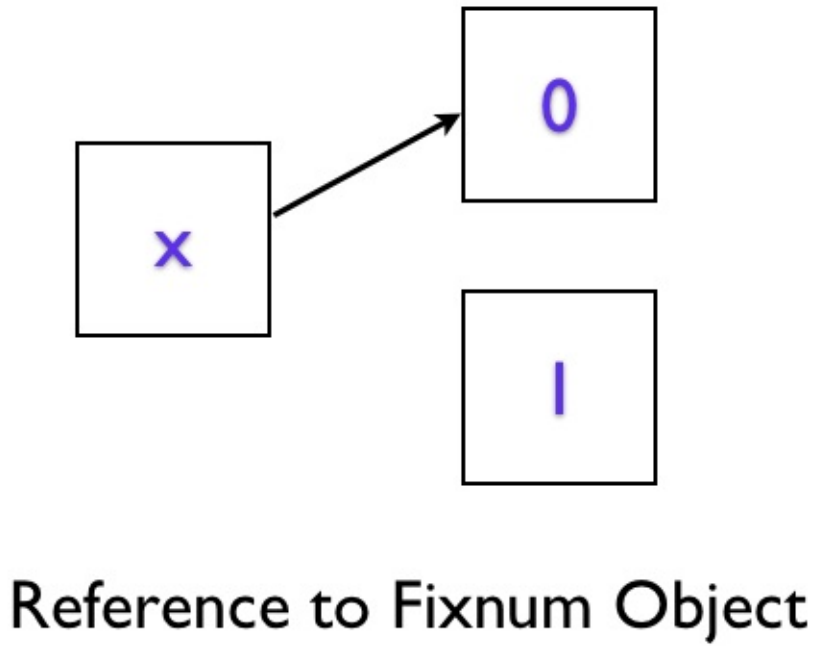
Let's change the value of x and print it.

```
x = 0
seconds = -&gt; { x }
p seconds.call

x = 1
p seconds.call
```

This prints:

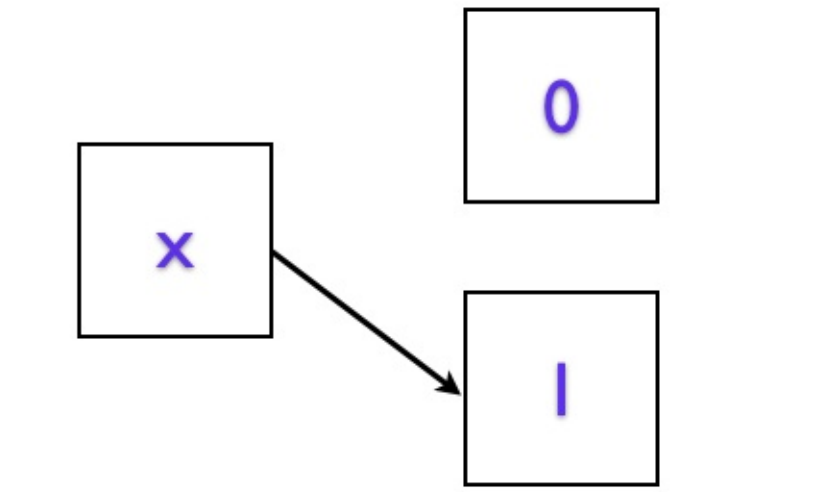
```
0
1
```

The value of x is changed to 1 after the block object is created. But, the change is reflected when we execute the code in the block object. This illustrates that the identifier x is actually a reference to Fixnum object.



Reference to Fixnum Object

If you change that reference to point to a different Fixnum object, it will point to it.



Reference to Fixnum Object

Changing the Value Inside a Block

Let's write a simple program to illustrate what happens when the value changes inside the block.

```
x = 0
seconds = -&gt; { x += 1 }
p seconds.call
p seconds.call
p seconds.call
p seconds.call
```

This prints:

```
1
2
3
4
```

The counter increases by one on each call.

Carrying the State Around

The block encapsulates the state. Earlier, we saw that we can pass this Proc object as an argument to a method. What happens when we have another variable with the same name in that method?

```
x = 0
seconds = -&gt; { x += 1 }

def tester\(s\)
  x = 100
  p s.call
  p s.call
  p s.call
  p s.call
end

tester\(seconds\)
```

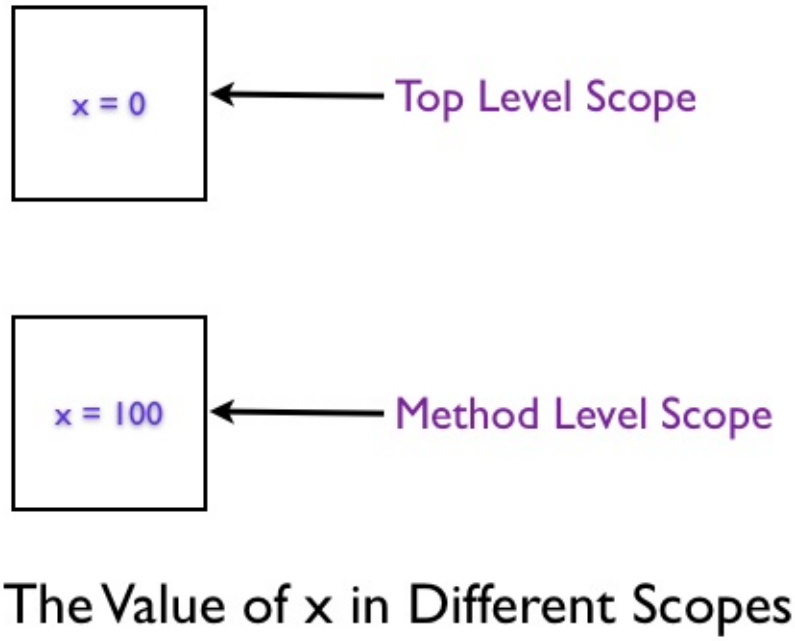
This prints:

```
1
2
3
4
```

The variable x with the value 100 inside the tester method did not have any effect on the Proc object. This illustrates an important concept. The block object carries the state found at the point of its creation. In our example, it is this line:

```
seconds = -&gt; { x += 1 }
```

At this line, the value of x was 0. It carries this value into the new scope of the tester method. We already know that methods create a new scope and x = 100 is in a new scope. The identifier names are the same but they belong to different execution contexts. One at the top level scope and the other at the method definition scope.

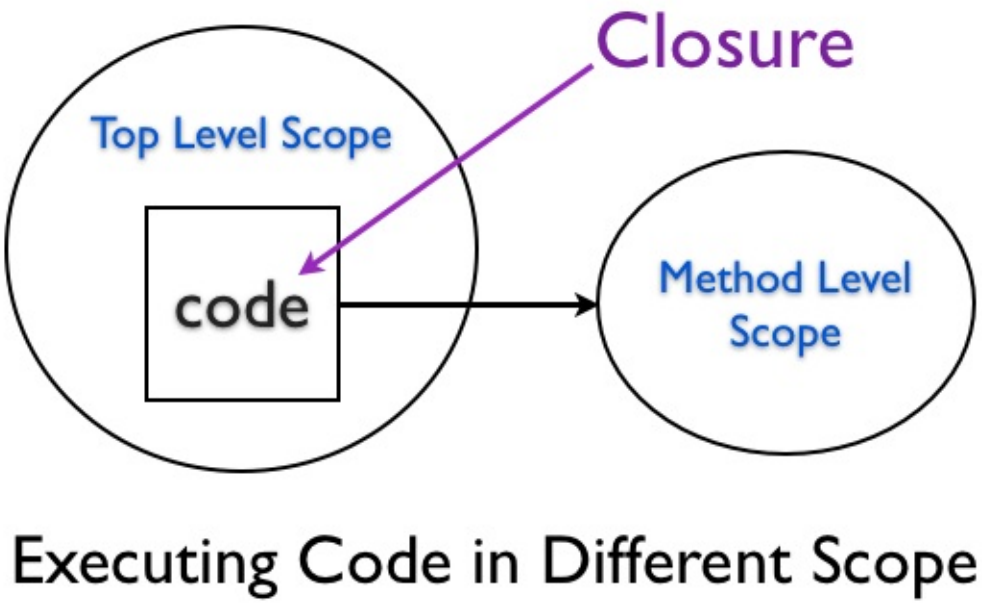


The Value of x in Different Scopes

The block object encapsulates the state. In this case, the value of x. The x gets incremented every time we call the block object by sending the call message. When a piece of code carries its creation context around with it like this, we call it closure.

Insight

You can execute code in a different execution context without using eval by using closures. Executing Code using Closure



Executing Code in Different Scope

We captured the binding at the top level scope in a block object and executed the code in the block object in the method level scope.

Twin Analogy

Haylee and Kaylee are twins who live together in San Francisco. Twins

Haylee is going on a business trip to New York.

She packs a tooth paste that is 100% full in her suitcase. The packing of the suitcase is the creation of the proc object using the literal constructor, ->. The top level context is San Francisco. The InNewYork class represents New York location.

```
toothpaste_level = 100
p "In SF : #{toothpaste_level}"

brush = -&gt; { toothpaste_level -= 5 }
brush.call

p "After brushing in SF : #{toothpaste_level}"

class InNewYork
  def get_ready(block)
    p "Brushing in NY"
    current_level = block.call
    p "In NY : #{current_level}"
  end
end

InNewYork.new.get_ready(brush)
p "In SF : #{toothpaste_level}"
```

This prints:

```
In SF : 100
After brushing in SF : 95
Brushing in NY
In NY : 90
In SF : 90
Toothpaste
```

When Haylee brushes her teeth in New York, the mirror image of that toothpaste in San Francisco is affected. Kaylee in San Francisco observes the toothpaste usage of her twin in New York. Of course, in reality physical objects cannot be in two locations at the same time. But, that's how closures behave in a programming environment.

Evaluating Code using Binding Object

Execute Code in Top Level Context

The block objects provide a binding method that we can use to execute code. Here is an example to illustrate that concept.

```
x = 1
o = -&gt; { x }

def tester(b)
  x = 10
  eval('x', b)
end

p tester(o.binding)
```

This prints 1. Inside the tester method the value of x is 10. But, we execute code in the top level execution context by passing in the binding of the block object. Thus, the value of x is 1 inside the tester method. Execute Code in Method Level Context

How can we switch the execution context to the method level scope? Here is an example. x = 1 o = -> { x }

def tester x = 10 eval('x', binding) end

p tester This prints 10. The binding call inside the tester method provides the execution context within that method. Thus, the x = 10 initialized value is available in the method level scope.

Execute Code in an Object Context

In the previous chapter, we could not call the private binding method of an object. Here is that example.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end
end

car = Car.new('red')
p eval("@color", car.binding)
```

This gave us the error:

```
NoMethodError: private method ‘binding’ called for #<Car:0x0570 @color="red">
```

We can make this example work by accessing the binding within the car object.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end

  def null_proc
    -> { }
  end

end

car = Car.new('red')
p eval("@color", car.null_proc.binding)
```

This prints red. We can also use the binding of a block object to access the instance variable.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end

  def null_proc
    -> { }
  end

end

car = Car.new('red')
p eval("@color", car.null_proc.binding)
```

This also prints red. This illustrates the concept that the null_proc is a closure. Let's check the class of the null_proc.

```
p car.null_proc.class
```

This prints Proc. The null_proc is bound to the execution context that can access the instance variable of the car object. The binding instance method on the Proc class exposes the execution context. You can also replace the existing null_proc implementation with:

```
Proc.new{ }
```

This creates a proc object from an empty block. This will still work.

Summary

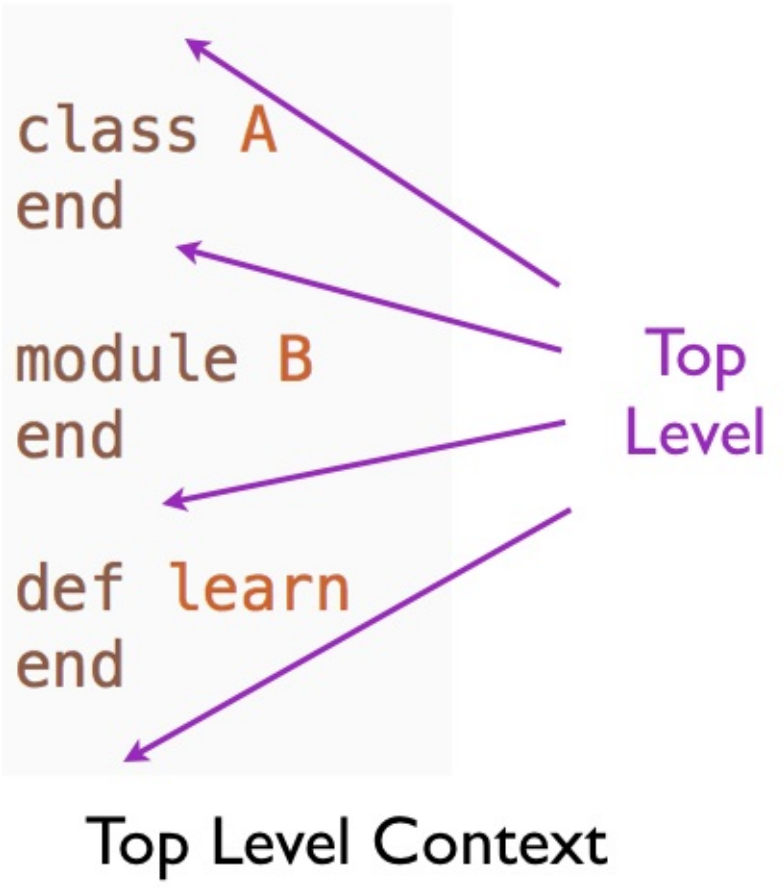
In this chapter, you learned the basics of closures. You also learned how to execute code in different contexts using closures.

The Top Level Context

In this chapter we will discuss about the top level context in Ruby program. We will experiment sending messages with and without explicit receiver.

Top Level

What is top level? You are in top level when you have not entered into a class, module or method definition. Or exited from all class, module or method definitions.



Top Level Context

Hello without Receiver

Let's write a simple program that prints hello at the top level.+

```
puts 'hello'
```

As you would expect, this prints:+

```
hello
```

The IO Class

Can we use an explicit receiver? Let's ask Ruby for the public instance methods of IO class.

```
puts IO.public_instance_methods(false).grep(/put/)
```

The grep searches for methods that has put in its name. The result shows that the puts is a public instance method of IO class.+

```
putc
puts
```

The false argument to the method filters out the methods from the super-class of IO class.+

Hello with Receiver

The puts is an instance method in IO class. Let's call the public instance method puts in the IO class.

```
io = IO.new(1)
io.puts 'hello'
```

The argument to IO constructor, 1, tells Ruby to print to the standard output. This prints hello to the terminal.+

Standard Output Global Variable

It's the same as doing:+

```
$stdout.puts 'hello'
```

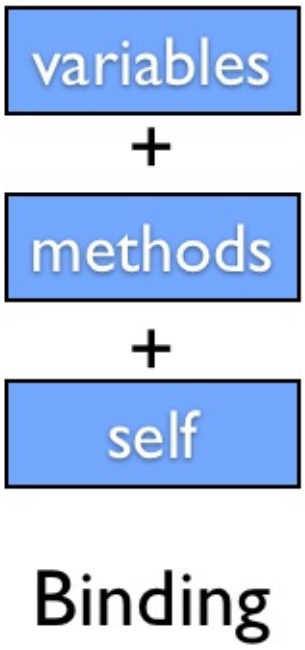
Here the $stdout is the Ruby built-in global variable for standard output. Global variables can be accessed from anywhere.

Binding

In this chapter you will learn the basics of binding and how we can execute code in different execution contexts.

Background

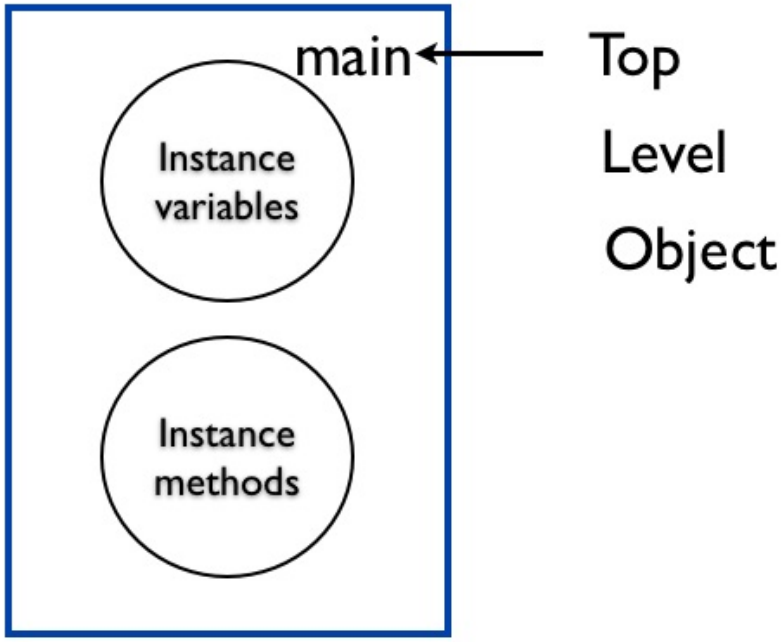
We have already covered the basic concepts of variables, methods and self. This chapter will combine all those concepts into one.



What is Binding?

A Binding object encapsulates the execution context at a particular place in the program. The execution context consists of the variables, methods and value of self. This context can be later accessed via the built-in function binding. We can create the binding object by using Kernel#binding method. The Kernel#eval method takes binding object as the second argument. Thus, the binding object can establish an environment for code evaluation.

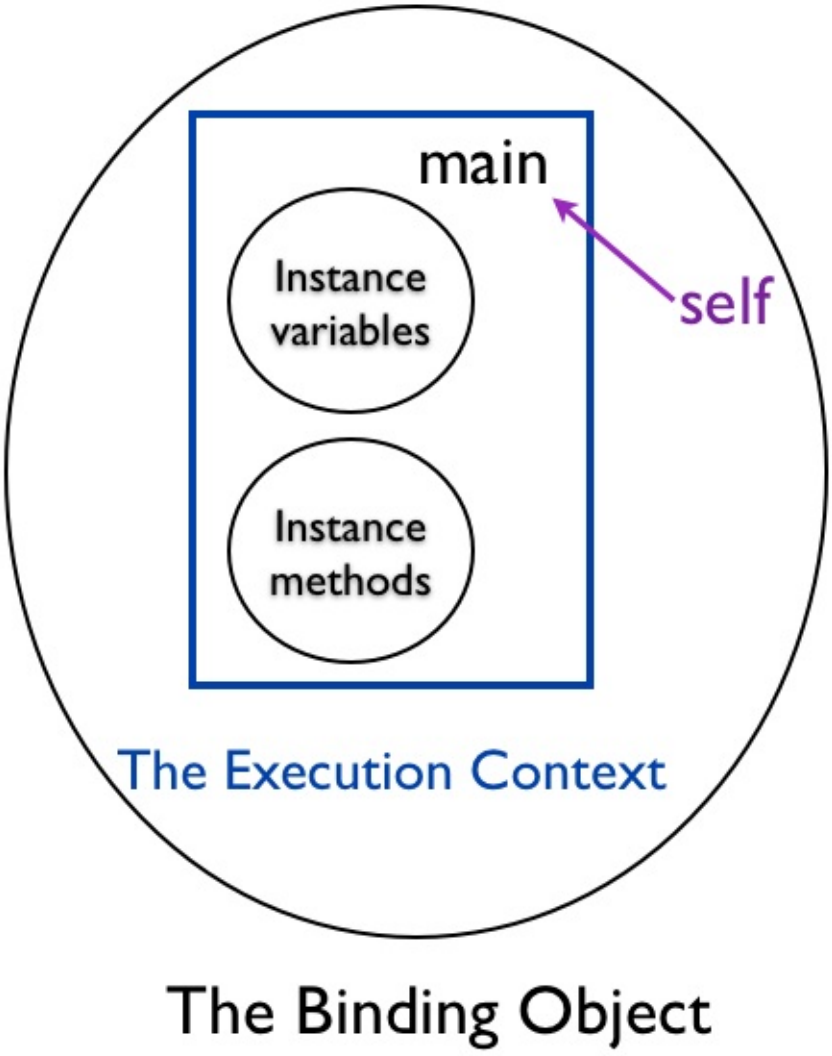
The Execution Context



The main bound instance methods and instance variables

In The main Object chapter, you saw this diagram:

The particular place in the program in this example is the top level. The value of self is main. We also saw how the instance variables and the instance methods are bound to the main object. The above diagram is the execution context for the top level. The diagram can be redrawn to make the binding concept clear.



Why do we need Binding? Code does not run in a vacuum. Code combined with an execution context becomes a running program.

Code, Execution Context and Running Program

Self at the Top Level

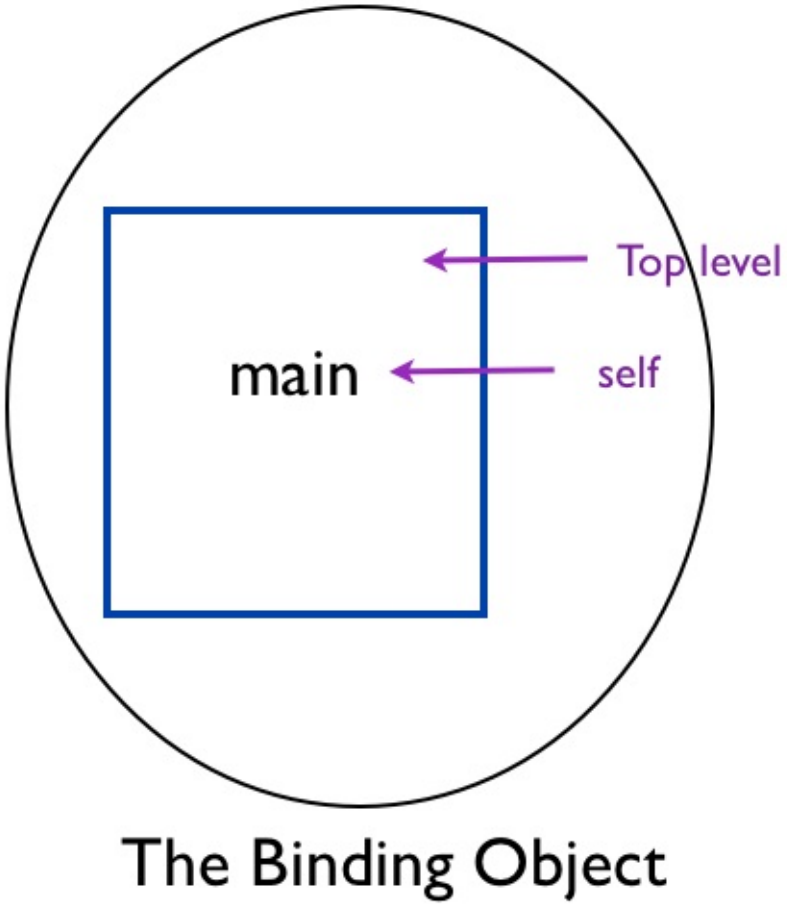
We can verify that the binding object at the top level context has main as the value of self.+

```
p TOPLEVEL_BINDING.receiver
```

This prints:

```
main
```

The TOPLEVEL_BINDING is a Ruby built-in constant that captures the binding at the top level.+



Local Variables at the Top Level

Let's check for local variables defined at the top level.+

```
p TOPLEVEL_BINDING.local_variables
```

This prints an empty array.

```
[]
```

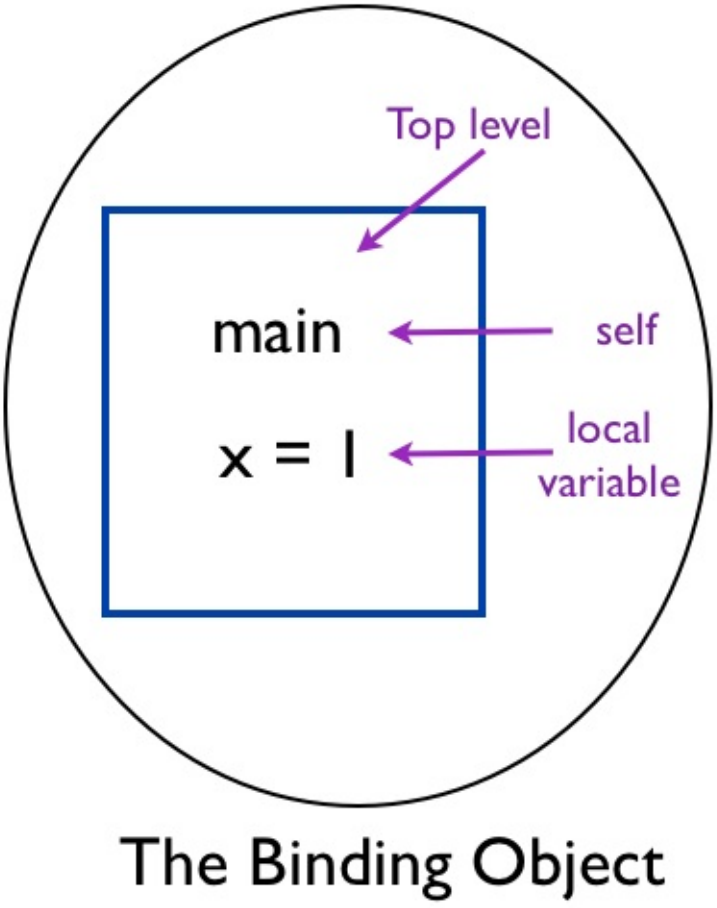
Let's define a local variable at the top level.

```
p TOPLEVEL_BINDING.local_variables
x = 1
```

This prints :

[:x]

Ruby read all the statements at the top level, so it was able to print the value of x that comes even after the print statement.



Instance Variable at the Top Level

How can we inspect the instance variable at the top level in the binding object?

```
@y = 0
p TOPLEVEL_BINDING.instance_variables
```

This prints an empty array.

```
[]
```

However, if we set the instance variable dynamically, we can print it.

```
TOPLEVEL_BINDING.instance_variable_set('@y', 0)
p TOPLEVEL_BINDING.instance_variables
```

This prints:

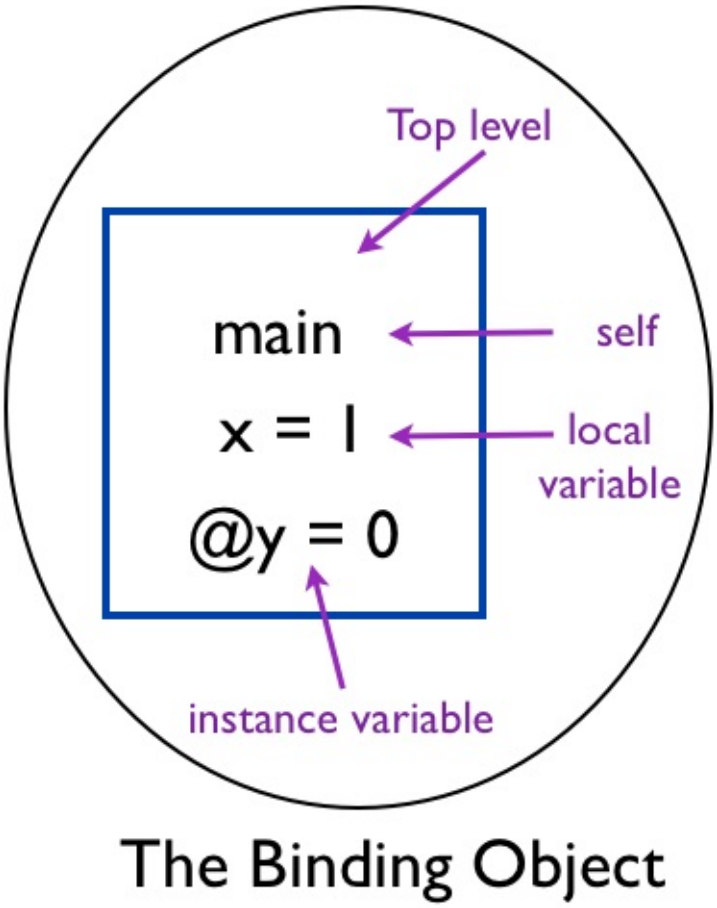
```
[:@y]
```

We can also read the value of the instance variable at the top level.

```
TOPLEVEL_BINDING.instance_variable_set('@y', 0)
p TOPLEVEL_BINDING.instance_variable_get('@y')
```

This prints:

```
0
```



The Binding Object

Binding Object Instance Variable

Accessing the Local Variable at the Top Level

Let's use eval to access the local variable at the top level.

```
binding_before_x = binding
p "Before defining x : #{eval("x", binding_before_x)}"

x = 1

binding_after_x = binding
p "After defining x : #{eval("x", binding_after_x)}"
```

This prints:

```
Before defining x :
After defining x : 1
```

The local variable x did not have any value before the assignment statement. Local Variable Execution Context

You can see the difference in this program from the previous section Instance Variable at the Top Level. The eval evaluates the value of x at the top level at the point at which it encounters. We then print the value before and after the local variable is initialized.

Object Context

Finding the self using Binding Object

Here is the example we saw in Same Sender and Receiver chapter.

```
class Car
  def drive
    p "self is : #{self}"
    start
  end

  private
  def start
    p 'starting...'
  end
end

c = Car.new
p "receiver is : #{c}"
c.drive
```

Let's rewrite the above example to use the binding object to find the receiver.2

```
class Car
  def drive
    p "self is : #{self}"
    p "receiver is : #{binding.receiver}"
    start
  end

  private
  def start
    p 'starting...'
  end
end

c = Car.new
c.drive
```

This example uses:

```
binding.receiver
```

to print the receiver object. The output is:

```
self is : #<Car:0x007fe373864dc8>
receiver is : #<Car:0x007fe373864dc8>
starting...
```

The self and the receiver is the same car object.

Fabio Asks

Can I find the sender using the binding object? No, binding object does not have a sender method that can give us the sender object.

Accessing the Instance Variable

In What is an Object chapter we could not access the color instance variable of the car object.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end
end

car = Car.new('red')
p car.color
```

This example gave the error:

```
NoMethodError: undefined method ‘color’ for #<Car:0x007f9be4025bc0 @color="red">
```

How can we access the color instance variable in the car object using binding? We know eval method takes the code as the first argument and binding as the second argument. Let's print the result of eval that takes the color instance variable and binding of the car object.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end
end

car = Car.new('red')
p eval("@color", car.binding)
```

This results in the error:

```
NoMethodError: private method ‘binding’ called for #<Car:0x007ffb639adbc8 @color="red"
>
```

Kernel module defines the binding method. Thus, it is available as a private method in the Object. Let's define a my_binding method that will provide us access to the execution context.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end

  def my_binding
    binding
  end
end

car = Car.new('red')
p eval("@color", car.my_binding)
```

This prints:

```
red
```

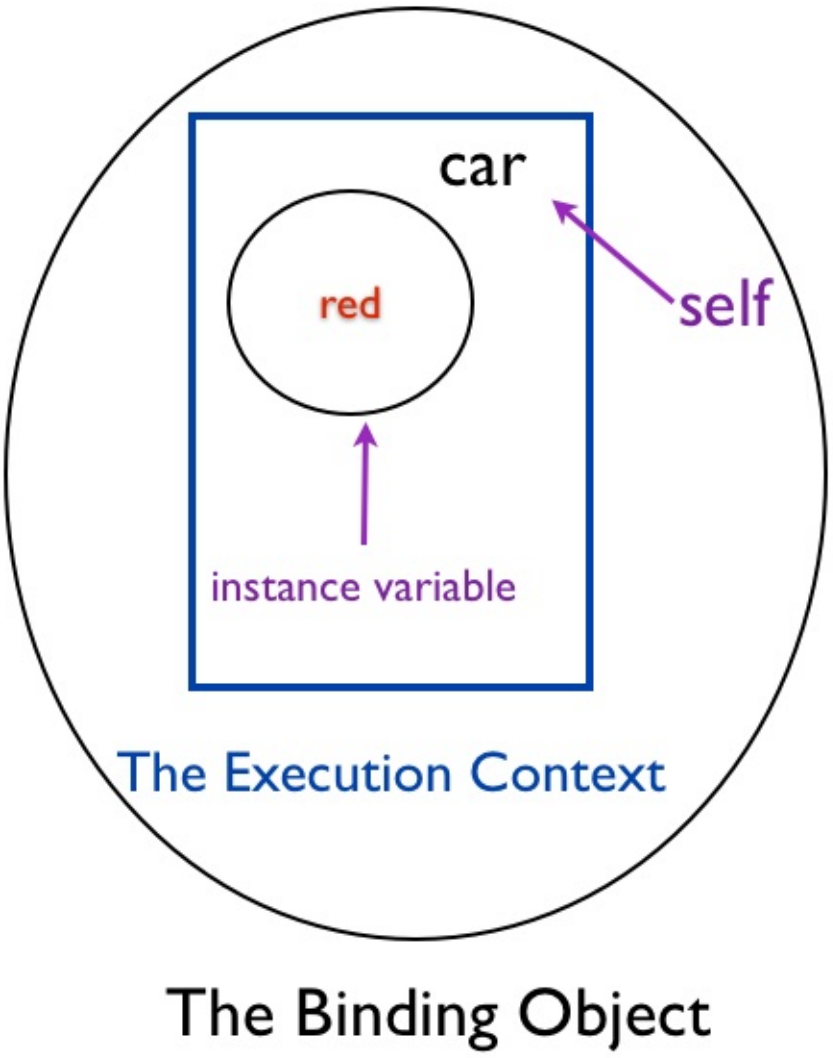
We are able to take a peek at the instance variable in the binding object. The binding object captures the value of self inside the my_binding method. We know that the value of self inside the my_binding method is a car object. The above example is the same as the

following example:

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end

  def my_binding
    puts @color
  end
end

car = Car.new('red')
car.my_binding
```

This also prints:

```
red
```



We can verify the value of self by running the following example.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    return 'driving'
  end

  def my_binding
    puts self
    puts @color
  end
end

car = Car.new('red')
car.my_binding
```

This prints the car object memory location and the color.

```
#<Car:0x007fa132018e90>
red
```

Execution Context Analogy

It's like using a probe to send some piece of code to execute in a different context. Probe

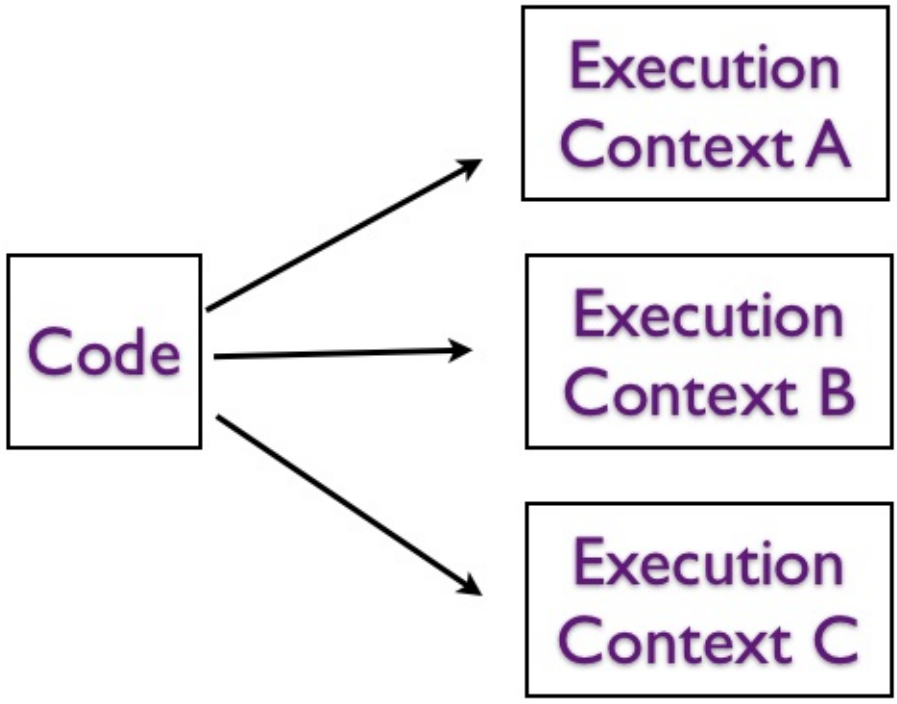
Executing Code in Different Execution Contexts

The power of binding is in the ability to run the same code in different contexts. Let's take a look at an example.

```
class Car
  def initialize(color)
    @color = color
  end

  def my_binding
    binding
  end
end

red_car = Car.new('red')
black_car = Car.new('black')

code = "@color"

p eval(code, red_car.my_binding)
p eval(code, black_car.my_binding)
```

This prints:

```
red
black
```



Executing Code in Different Execution Contexts

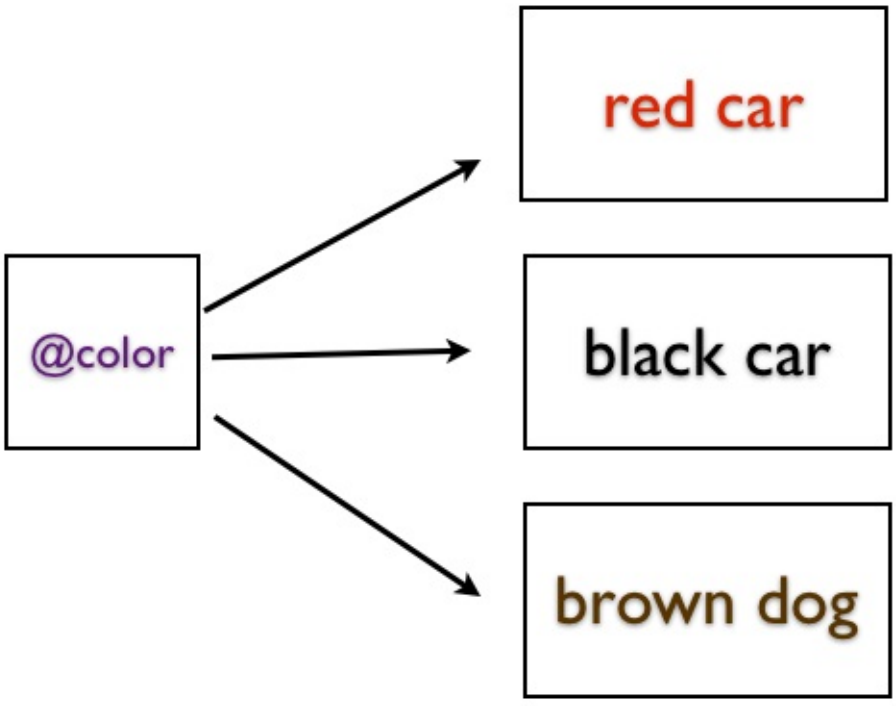
There is no restriction on which object should provide the binding. We can have another class, let's say dog, that provides an execution context for the same code.

```
class Dog
  def initialize(color)
    @color = color
  end

  def my_binding
    binding
  end
end

dog = Dog.new('Brown')
code = "@color"
p eval(code, dog.my_binding)
```

This prints:

```
Brown
```

The my_binding method is like a probe. Code is put inside of this method.



Executing Code in Different Execution Contexts

Rhonda Asks

Why do we need the ability to run the code in different execution contexts?

Ruby is dynamic in nature. This allows us to reuse code in scenarios that we might not have imagined when we wrote the code.

Executing Code in Different Scope

Execute Code in an Object Scope from Top Level Scope

Let's look at an example that executes code in the rabbit object scope from the top level scope.

```
class Rabbit
  def context
    @first = 'Bugs'
    last = 'Bunny'

    binding
  end
end

binding = Rabbit.new.context
# Scope here has changed because this is top level scope.
# But we are executing the following code in the rabbit object scope by using eval.
p eval("self", binding)
p eval("last.size", binding)
p eval('@first', binding)

# This uses binding only, no eval is used to get the local variable
p binding.local_variable_get('last')
```

This prints:

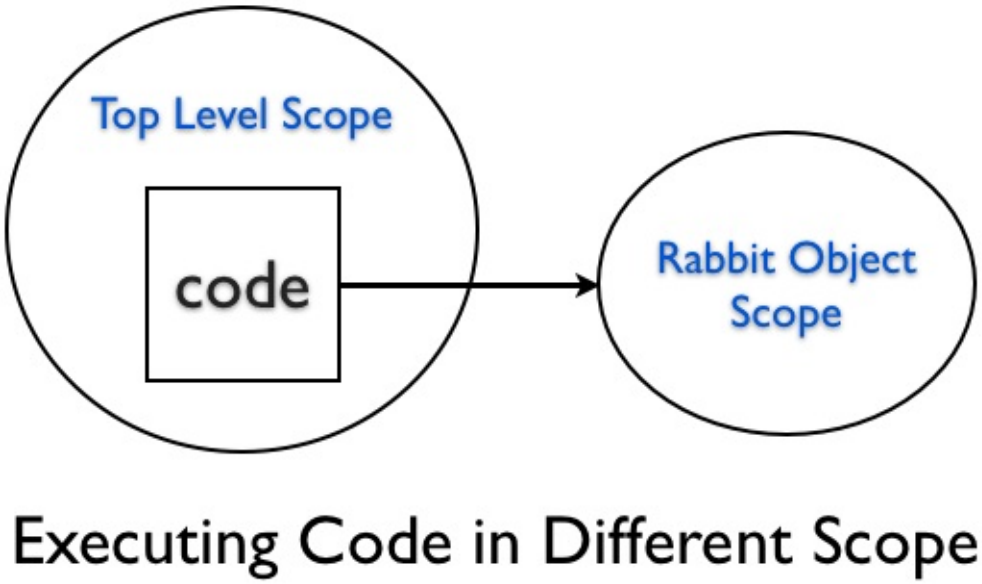
```
#<Rabbit:0x007fc2a406c318 @first="Bugs">
5
Bugs
Bunny
```

We were able to take a look at the local variable, instance variable, value of self and execute methods on the local variable. It is as if we were inside the context method like this:

```
class Rabbit
  def context
    @first = 'Bugs'
    last = 'Bunny'

    puts self
    puts @first
    puts last
    puts last.size
  end
end

Rabbit.new.context
```

This prints:

```
#<Rabbit:0x007fe3f3881038>
Bugs
Bunny
5
```



Executing Code in Different Scope

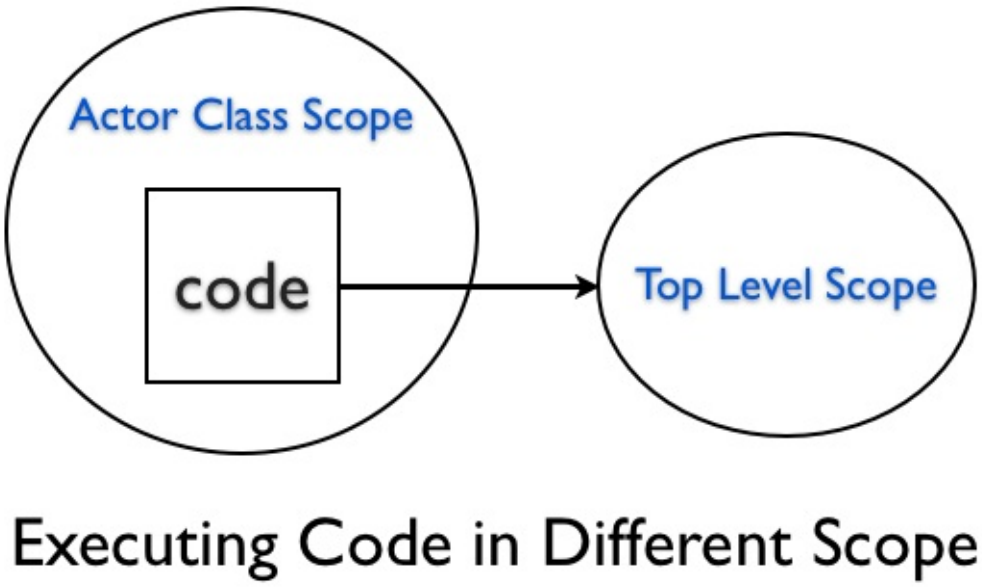
Execute Code in Top Level Scope from an Object Scope

Let's now see an example where we execute code in the top level scope when we are in an object scope.

```
@actor = 'Daffy'

class Actor
  def self.act
    eval("@actor", TOPLEVEL_BINDING)
  end
end

p Actor.act
```

This prints:

```
Daffy
```



Executing Code in Different Scope

If you access the top level instance variable inside the method from the Actor class scope, you will not be able to access the value.

```
@actor = 'Daffy'
class Actor
  def self.act
    @actor
  end
end

p Actor.act
```

This will print nil, because, the @actor inside the act method is in a different scope and it is not initialized. They are two different variables that happens to have the same name.

The Default Receiver

In this chapter, you will learn that when you don't provide an explicit receiver in the code, Ruby uses self as the default receiver object.

The self Within Car

Let's define a Car class and print the value of self inside the Car class.

```
class Car
  puts self
end
```

This prints:

```
Car
```

The drive Class Method

Let's define a drive() class method in the Car class.

```
class Car
  def self.drive
    p 'driving'
  end
end
```

We know that the value of self is Car. This is the same as:

```
class Car
  def Car.drive
    p 'driving'
  end
end
```

Call the Class Method

Explicit Receiver

We can call the drive() class method inside the Car class.

```
class Car
  def self.drive
    p 'driving'
  end

  Car.drive
end
```

This prints:

```
driving
```

We can also use self instead of Car to call the drive() class method.

```
class Car
  def self.drive
    p 'driving'
  end

  self.drive
end
```

This also prints:

```
driving
```

Call the Class Method

No Receiver

We know the value of self inside the Car class is Car. We can omit the self from the above example.

```
class Car
  def self.drive
    p 'driving'
  end

  drive
end
```

This prints:

```
driving
```

We don't have the dot notation that sends a message or the receiver object. The message is sent to the current value of self whenever you call a method with no receiver. Since the current value of self is Car, we are able to call the drive class method. In this example, Car is the default receiver object.

Fabio Asks

Why is it called default receiver?

Because, when you don't provide an explicit receiver, the current value of self defaults as the receiver.

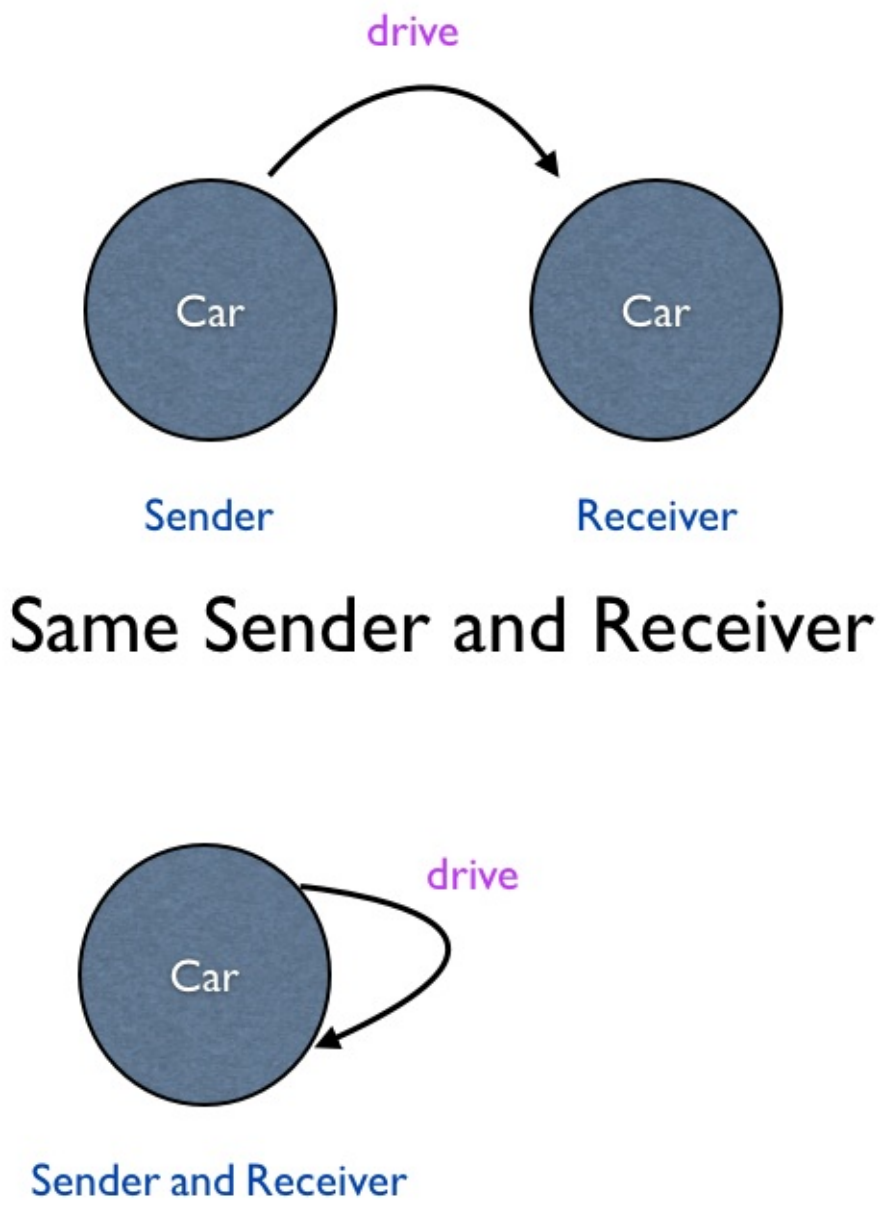
Rhonda Asks

Why do we need a receiver to send a message?

In a OO language like Ruby, all messages are sent to some object.

Insight

There is another way to think about this example. You can say that whenever the sender object and the receiver object is the same, you can omit the receiver.



We will revisit this idea in an upcoming chapter.

Subclass Calling Class Method

The example may seem trivial, but this allows us to write code like this:

```
class Car
  def self.drive
    p 'driving'
  end
end

class Beetle < Car
  drive
end
```

This example prints:

```
driving
```

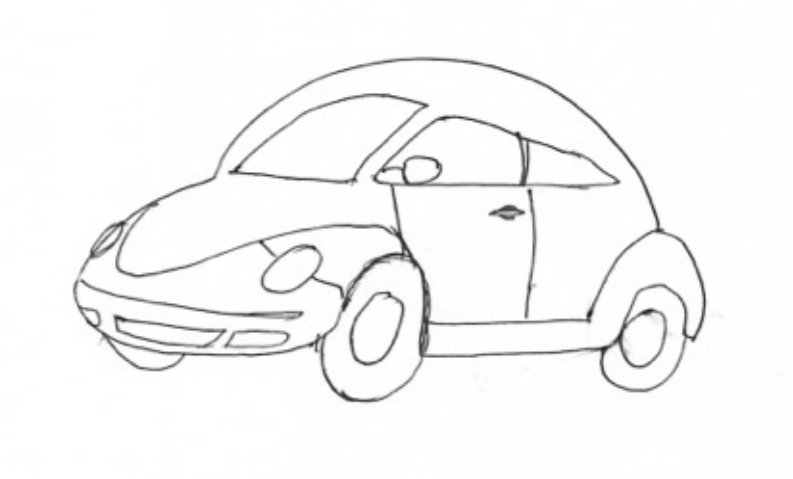
Rails Example

This is like ActiveRecord library in Rails. Imagine that ActiveRecord find() implementation is like this:

```
class ActiveRecord
  def self.find(id)
    p "Find record with id : #{id}"
  end
end
```

In our web application we have a Beetle subclass of ActiveRecord as the model.

```
class Beetle < ActiveRecord
end
```



In the controller, we use the find method:

```
Beetle.find(1)
```

This prints:

```
"Find record with id : 1"
```

The dot notation makes sending messages explicit. If the receiver and the sender is the same, you can omit the receiver and the dot. In this case, Ruby will use value of self as the receiver. Thus, self is the default receiver.

Message Sending Expression

In this chapter, we will identify the sender, receiver, message and arguments in a message sending expression.

Background

By now, you already know:

- There is always a receiver.
- There is always a sender.
- There is always a message that passes between the sender and the receiver.

Let's see an example that ties all these concepts together.

Messaging

Let's write the simplest program to illustrate the three key takeaways of this book so far.

```
x = 1 + 2

p x
```

This prints 3.

Explicit Message

The above program can be rewritten in an equal way.

```
x = 1.+ 2

p x
```

This also prints 3. This example makes sending a message explicit by using the dot notation.

Explicit Argument

The above program can be rewritten in an another way.

```
x = 1.+(2)

p x
```

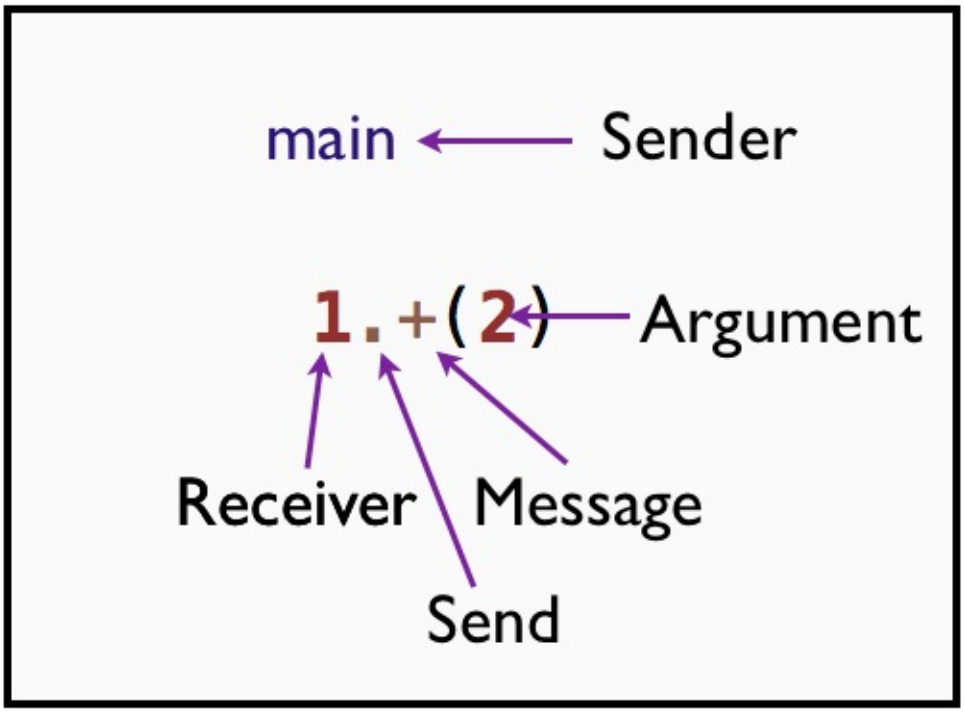
This also prints 3. This example makes the argument, 2 to the message + explicit.

Identify Receiver

It's now clear that 1 is the receiver. We already know that this object is Fixnum.

Identify Sender

The sender is implicit because we have written this code at the top level context. We know that main object is the sender.



Sending a Message

The + is not an operator in Ruby. The + is a message that passes between the sender and the receiver. Let's verify it.

```
p 1.respond_to?(:+)
```

The respond_to? method takes a symbol of the method name as the argument and returns either true or false depending on whether it can respond to it or not. This prints:

```
true
```

This proves that Fixnum object responds to the + message. We can explicitly send the message to Fixnum object.

```
p 1.send(:+, 2)
```

The send method is defined in Kernel module and is mixed into the Object. So, it's available everywhere. The first argument to the send method is the symbol of the method name and the second argument is the argument for that method. This prints 3.

Convert Message to an Object

We can convert the + message sent to the Fixnum object into an object and call it.

```
 > addition = 1.method(:+)
 => #<Method: Fixnum#+>
 > addition.call(2)
```

The method called method is defined in Kernel module. Since the Kernel module is mixed into the Object, it's available everywhere. This returns a Method object. We send the call message with 2 as the argument to add 1 and 2. This prints 3.

Fabio Asks

Are there other messages in Ruby that looks like an operator? Yes, you can experiment in the IRB and get a list of those messages:

```
> 1.public_methods(false)
 => [:%, :&, :*, :+, :-, :/, :<, :>, :^, :|, :~, :-@, :**, :<=>, :<<, :>>, :<=, :>=, :
==, :===, :[],...]
```

The false argument is used to list only the methods found in Fixnum class. We can also send the instance_methods message to the Fixnum class.

```
> Fixnum.instance_methods(false)
```

This prints:

```
[:-@, :+, :-, :*, :/, :%, :**,  :==, :===, :<=>, :>, :>=, :<, :<=, :~, :&, :|, :^, :[]
, :<<, :>>, ...]
```

Key Takeaway

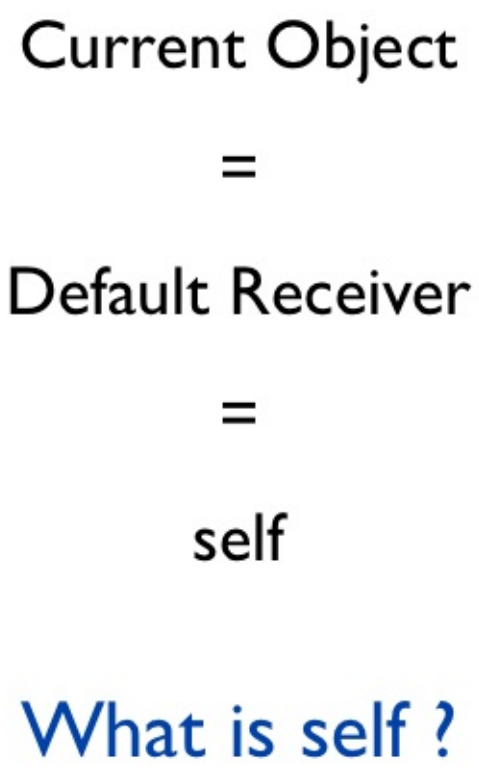
You must read 1 + 2 as, the object 1 is sent the message +, with the object 2 as the argument.

The self at the Top Level

In this chapter, you will learn about the current object, self at the top level.

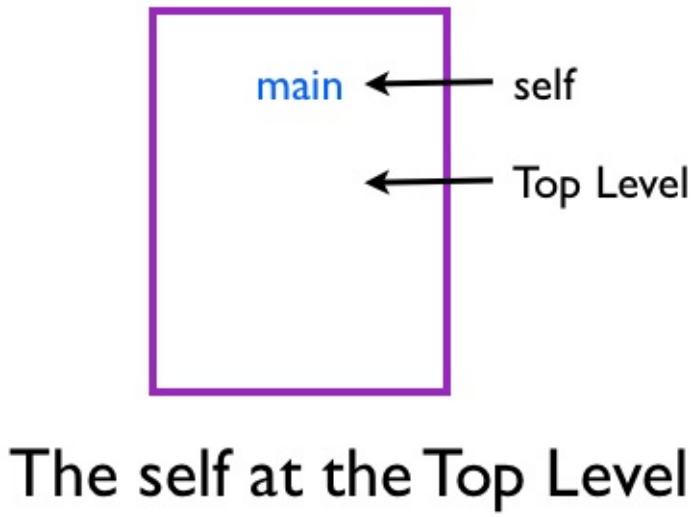
What is self?

In Ruby, there is always one object that plays the role of current object at any given instant. The current object provides an execution context for the code. This current object is the default receiver. When the receiver is not provided, the message is sent to the default receiver. This is the self.



Self at the Top Level

Let's see the value of self at the top level. puts self This prints: main This tells us that Ruby has created an object called main for us at the top level. And all the code we write at the top level will use main as the receiver in method calls.



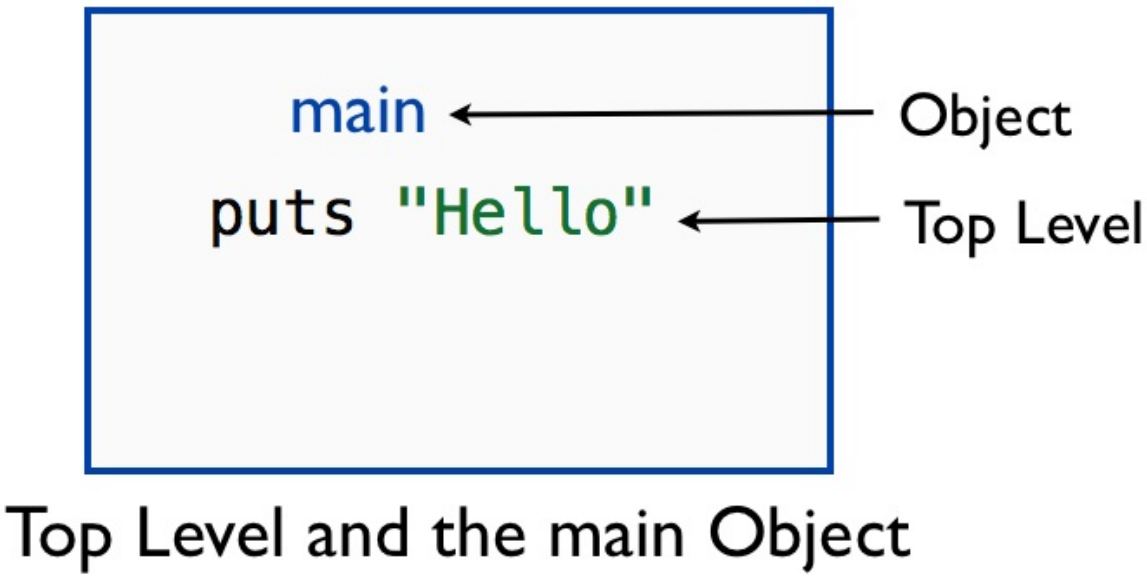
The self at the Top Level

The main Object

If main is an object, it must be instance of some class. We can ask Ruby to tell us which class main is an instance of: puts self.class This prints: Object We now know, Ruby did something like this: main = Object.new This provides us an object context to execute our code at the top level. Thus, providing us the default receiver main at the top level. The main is an instance of the Object class.

Hello at the Top Level

Let's print hello to the standard output. puts 'hello' As expected, this prints hello.



Top Level and the main Object

Is main a Receiver Object?

Can we use main, the instance of Object, to call puts?

```
main.puts 'hi'
```

We get:

```
NameError: undefined local variable or method ‘main’ for main:Object
```

Ruby prints main as the current object at the top level. But, there is no such variable called main.

Human Visible main Object

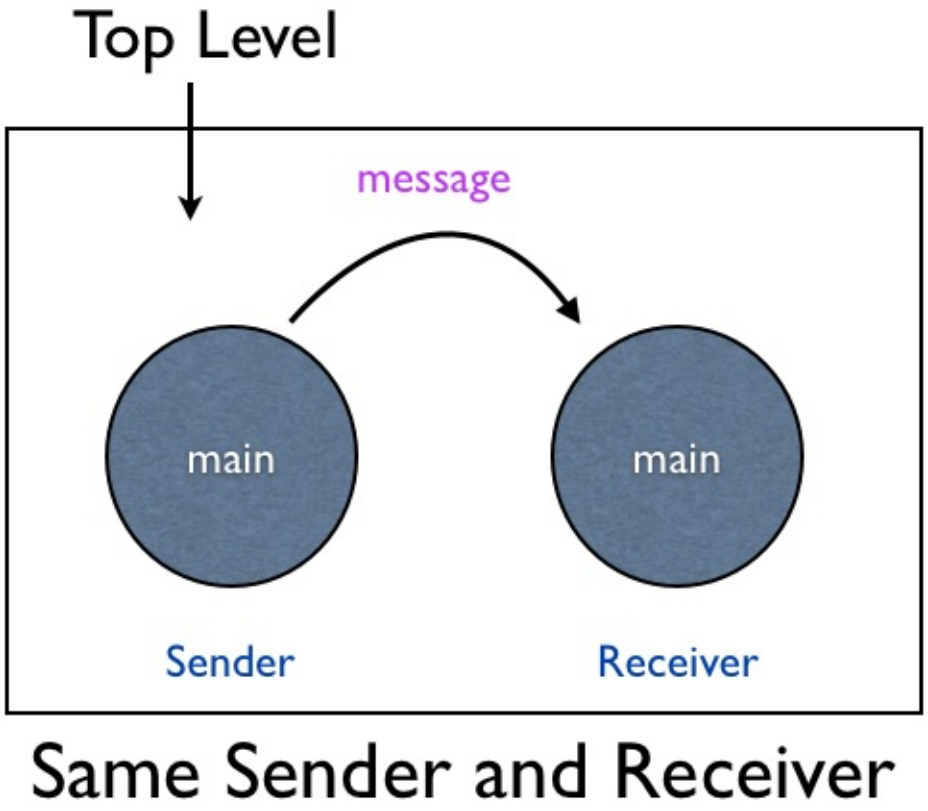
The main is the human visible representation of the current object. Let's see this in action in the IRB console.

```
> self
=> main
> self.inspect
=> "main"
> self.to_s
=> "main"
```

Fabio Asks

Why does Ruby create main object at the top level?

In a OO language like Ruby, there is always a sender and receiver involved in a message send. We did not explicitly create a sender object. Thus, Ruby created a sender object for us. It is implicit, because it is not visible in the code.

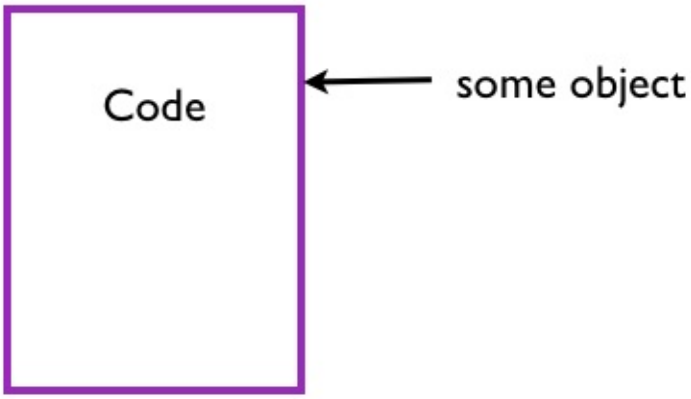


Same Sender and Receiver

In an upcoming chapter, we will see that both sender and receiver are main at the top level.

Rhonda Asks

If I don't create a receiver object, does Ruby create a receiver object? No. Ruby does not create a receiver object; it uses the existing value of self as the default receiver.



Code Executes in Current Object

Ruby Pattern

Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

```
class Quest
  attr_accessor :difficulty, :hero

  def initialize(difficulty)
    @difficulty = difficulty
    @hero = nil
  end

  def finish
    @hero.exp += calculate_experience
  end

  def calculate_experience
    @difficulty * 50 / @hero.level
  end
end

class Hero
  attr_accessor :level, :exp, :quests

  def initialize
    @level = 1
    @exp = 0
    @quests = []
  end

  def take_quest(quest)
    @quests << (quest.hero = self)
  end

  def finish_quest(quest)
    quest.finish
    @quests.delete quest
```

```
    end
  end

  class OldQuest
    attr_accessor :hero, :difficulty, :experience

    def initialize
      @difficulty = 3
      @experience = 10
    end

    def done
      difficulty * experience
    end
  end

  class QuestAdapter
    attr_accessor :hero

    def initialize(old_quest, difficulty)
      @old_quest = old_quest
      @old_quest.difficulty = difficulty
      @hero = nil
    end

    def finish
      @hero.exp += @old_quest.done
    end
  end

  # Usage
  hero = Hero.new
  quest = Quest.new 5
  hero.take_quest quest
  hero.finish_quest quest
  puts hero.exp
  # => 250
  some_old_quest = OldQuest.new
  old_quest_adapted = QuestAdapter.new(some_old_quest, 5)
  hero.take_quest old_quest_adapted
  hero.finish_quest old_quest_adapted
  puts hero.exp
  # => 300
```

Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

```ruby
class BoardBuilder
  def initialize(width, height)
    @board = Board.new
    @board.width = width
    @board.height = height
    @board.tiles = []
    @board.monsters = []
  end

  def add_tiles(n)
    n.times{ @board.tiles << Tile.new }
  end

  def add_monsters(n)
    n.times{ @board.monsters << Monster.new }
  end

  def board
    @board
  end
end

class Board
  attr_accessor :width, :height, :tiles, :monsters
  def initialize
  end
end

class Tile; end
class Monster; end

# Usage
builder = BoardBuilder.new 2, 3
puts builder.board
# => Board Object
board = builder.board
puts board.width
# => 2
builder.add_tiles(3)
builder.add_monsters(2)
puts board.tiles.size
# => 3
puts board.monsters.size
# => 2
```

Observer

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

```
module Observable
  attr_reader :observers

  def initialize(attrs = {})
    @observers = []
  end

  def add_observer(observer)
    @observers << observer
  end

  def notify_observers
    @observers.each{ |observer| observer.update }
  end
end

class Tile
  include Observable

  def initialize(attrs = {})
    super
    @cursed = attrs.fetch(:cursed, :false)
  end

  def cursed?
    @cursed
  end

  def activate_curse
    notify_observers
  end
end

class Hero
  attr_reader :health

  def initialize
    @cursed = false
    @health = 10
  end

  def damage(hit)
    @health -= hit
  end

  def cursed?
    @cursed
  end

  def discover(tile)
    if tile.cursed?
      @cursed = true
```

```
        tile.add_observer(self)
      end
    end

    def update
      damage(4)
    end
  end

  # Usage
  hero = Hero.new
  tile = Tile.new cursed: true
  hero.discover(tile)
  tile.activate_curse
  puts hero.health
  # => 6
  puts hero.cursed?
  # => true
```

Singleton

Define a unique instance of an object.

```
  class HeroFactory
    @@instance = nil

    def self.instance
      @@instance ||= HeroFactory.send(:new)
    end

    def create_warrior
      Warrior.new
    end

    def create_mage
      Mage.new
    end

    private_class_method :new
  end

  # Usage
  factory = HeroFactory.instance
  another_factory = HeroFactory.instance
  puts factory == another_factory
  # => true
  HeroFactory.new
  # => Raise Exception
```

Chapter II: Web Frameworks

What is Nginx

Nginx is a lightweight, high performance web server designed to deliver large amounts of static content quickly with efficient use of system resources. Nginx's strong point is its ability to efficiently serve static content, like plain HTML and media files. Some consider it a less than ideal server for dynamic content.

Unlike Apache, which uses a threaded or process-oriented approach to handle requests, Nginx uses an asynchronous event-driven model which provides more predictable performance under load. Rather than using the embedded interpreter approach, Nginx hands off dynamic content to CGI, FastCGI, or even other web servers like Apache, which is then passed back to Nginx for delivery to the client.

This leads to a more complex setup for certain deployments. For these and other reasons, the configuration of Nginx can feel complicated and unintuitive at first. This document should familiarize you with basic Nginx parameters and conventions. We'll be going through Nginx's primary configuration file.

All Nginx configuration files are located in the `/etc/nginx/` directory. The primary configuration file is `/etc/nginx/nginx.conf`.

> Note
>
> This is where the files will be located if you install Nginx from the package manager. Other possible locations include /opt/nginx/conf/.

Web server with nginx

The following sections cover a few concepts and safeguards that should be reviewed before making any changes to your Nginx configuration.

Preserve a Working Configuration

Sometimes, server configuration files can get so corrupted or convoluted that they become unusable, so it's always a good idea to have a working copy of the essential files on hand. A good first step is to copy any configuration file before you begin making changes to it, like so:

```
cp /etc/nginx/nginx.conf /etc/nginx/nginx.conf.backup
```

For even better restoration options, we recommend making regular backups of your Nginx configuration. You might want to store your entire /etc/nginx/ directory in a Git repository so you can save the original settings and all the versions from all your different changes. Another option is to periodically create dated copies of your files. You can accomplish this by issuing the following command:

```
cp /etc/nginx/nginx.conf /etc/nginx/nginx.conf.$(date "+%b_%d_%Y_%H.%M.%S")
```

Start, Stop, Reload

Now you're ready to make changes to your Nginx configuration. Whenever you make a change to the `nginx.conf` file, you need to reload your configuration before the change will take effect. You can do this by issuing the following command:

```
service nginx reload
```

To completely stop or start the service, replace `reload` with `start` or `stop`.

Introduction to Syntax

This section showcases an excerpt from the beginning of the default configuration file, `/etc/nginx/nginx.conf`:

```
user www-data;
worker_processes 4;
pid /run/nginx.pid;

events {
        worker_connections 768;
        # multi_accept on;
}
```

Normally, you will not need to change anything in this first section. Instead, we'll use this as an opportunity to explain the syntax of the configuration file.

- All lines preceded by a **pound sign** or **hash (#)** are comments. This means that they are not interpreted by Nginx. Programmers use comments to explain what certain blocks of code do, or leave suggestions on how to edit values. The default file has several commented sections for explanatory purposes, and you can add your own if desired. Programmers will also leave optional statements as comments. They can be "turned on" if needed by removing the pound sign.

  An example of this is shown above. The directive `multi_accept on;` has a `#` in front of it. This isn't a comment for the user, but rather a directive that can be "activated" by removing the `#`.

- Settings begin with the variable name and then state an argument or series of arguments separated by spaces. Examples include `worker_processes 1;` and `error_log logs/error.log notice;`. All statements end with a semi-colon (;).
- Some settings, like the `events` variable above, have arguments that are themselves settings with arguments. All sub-directives are enclosed in one set of curly brackets (`{ }`).
- Brackets are sometimes nested inside each other for multiple sets of sub-directives. If you add or edit a section with nested brackets, make sure they all come in opening and closing pairs.
- Tabs or multiple spaces are interpreted by nginx as a single space. Using tabs or spaces for indentation in a standardized way will greatly improve the readability of your file and make it easier to maintain.

Understanding How Nginx Works

Now that we understand the syntax, let's get into the nuts and bolts of Nginx. First we'll go over the core directives in the `nginx.conf` file, which define the basic behavior of the web server. Then we'll explain the `HTTP` block in greater detail and some of the more commonly

adjusted variables. From there we'll move to the `server` block, and virtual host configuration files. This is the section you will edit the most when defining the websites you'll host with Nginx.

Defining the Directives

We'll begin by explaining the core directives in `/etc/nginx/nginx.conf` Let's go back to that first section:

```
user www-data;
worker_processes 4;
pid /run/nginx.pid;

events {
        worker_connections 768;
        # multi_accept on;
}
```

user	Defines which Linux system user will own and run the Nginx server. Most Debian-based distributions use www-data but this may be different in other distros. There are certain use cases that benefit from changing the user; for instance if you run two simultaneous web servers, or need another program's user to have control over Nginx.
worker_process	Defines how many threads, or simultaneous instances, of Nginx to run. You can learn more about this directive and the values of adjusting
pid	Defines where Nginx will write its master process ID, or PID. The PID is used by the operating system to keep track of and send signals to the Nginx process.

HTTP (Universal Configuration)

The next section of the `nginx.conf` file covers the universal directives for Nginx as it handles HTTP web traffic. The first part of the HTTP block is shown below:

File: `/etc/nginx/nginx.conf`

```
http {

    ##
    # Basic Settings
    ##

    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 65;
    types_hash_max_size 2048;
    # server_tokens off;

    # server_names_hash_bucket_size 64;
    # server_name_in_redirect off;

    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    ##
    # Logging Settings
    ##

    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;

    ##
    # Gzip Settings
    ##

    gzip on;
    gzip_disable "msie6";
```

Most of the http { } block should work as-is for most Nginx configurations. We do, however, want to draw your attention to the following configuration options:

include

The include statement at the beginning of this section includes the file mime.types located at /opt/nginx/conf/mime.types. What this means is that anything written in the file mime.types is interpreted as if it was written inside the http { } block. This lets you include a lengthy amount of information in the http { } block without having it clutter up the main configuration file. Try to avoid too many chained inclusions (i.e., including a file that itself includes a file, etc.) Keep it to one or two levels of inclusion if possible, for readability purposes. You can always include all files in a certain directory with the directive:

```
include /etc/nginx/sites-enabled/*;
```

Or to be more specific, you can include all `.conf` files in a directory:

```
include /etc/nginx/conf.d/*.conf;
```

gzip The `gzip` directive tells the server to use on-the-fly gzip compression to limit the amount of bandwidth used and speed up some transfers. This is equivalent to Apache's `mod_deflate`. Additional settings can be uncommented in this section of the http block to modify the gzip behavior: /etc/nginx/nginx.conf

```
# gzip_vary on;
# gzip_proxied any;
# gzip_comp_level 6;
# gzip_buffers 16 8k;
# gzip_http_version 1.1;
# gzip_types text/plain text/css application/json application/x-javascript text/xml ap
plication/xml application/xml+rss text/javascript;
```

For full definitions of the gzip options, check out this page from the Nginx docs.

> If you keep gzip compression enabled here, note that you are trading increased CPU costs in exchange for your lower bandwidth use. Set the gzip_comp_level to a value between 1 and 9, where 9 requires the greatest amount of CPU resources and 1 requires the least. The default value is 1.

Note that the code snippet shown above does not include the closing bracket (}), because the HTTP section isn't finished. For detailed explanations of every directive in the HTTP block, check out this page from the Nginx documentation.

Server (Virtual Domains Configuration)

The HTTP block of the `nginx.conf` file contains the statement `include /etc/nginx/sites-enabled/*;`. This allows for server block configurations to be loaded in from separate files found in the `sites-enabled` sub-directory. Usually these are symlinks to files stored in `/etc/nginx/sites-available/`. By using symlinks you can quickly enable or disable a virtual server while preserving its configuration file. Nginx provides a single default virtual host file, which can be used as a template to create virtual host files for other domains:

```
cp /etc/nginx/sites-available/default /etc/nginx/sites-available/example.com
```

Now let's go over the directives and settings that make up the `server` block:

File excerpt: /etc/nginx/sites-available/default

```
server {
        listen 80 default_server;
        listen [::]:80 default_server ipv6only=on;

        root /usr/share/nginx/html;
        index index.html index.htm;

        # Make site accessible from http://localhost/
        server_name localhost;

        location / {
                # First attempt to serve request as file, then
                # as directory, then fall back to displaying a 404.
                try_files $uri $uri/ /index.html;
                # Uncomment to enable naxsi on this location
                # include /etc/nginx/naxsi.rules
        }
}
```

The `server` block is where the typical Nginx user will make most of his or her changes to the default configuration. Generally, you'll want to make a separate file with its own `server` block for each virtual domain on your server. More configuration options for the server block are shown in the following sections.

Listening Ports

The `listen` directive, which is located in the `server` block, tells Nginx the hostname/IP and the TCP port where it should listen for HTTP connections. By default, Nginx will listen for HTTP connections on port `80`.

Next we'll present a few common examples for the `listen` directive.

> You can use more than one listen directive, if needed.

File excerpt: /etc/nginx/sites-available/default

listen 80 default_server; listen [::]:80 default_server ipv6only=on; These are the default listen statements in the default virtual host file. The argument default_server means this virtual host will answer requests on port 80 that don't specifically match another virtual host's listen statement. The second statement listens over IPv6 and behaves in the same way.

File excerpt: **/etc/nginx/sites-available/example.com**

```
listen     127.0.0.1:80;
listen     localhost:80;
```

These two examples direct Nginx to listen on `127.0.0.1`; that is, the local loopback interface. localhost is conventionally set as the hostname for `127.0.0.1` in `/etc/hosts`.

File excerpt: **/etc/nginx/sites-available/example.com**

```
listen     127.0.0.1:8080;
listen     localhost:8080;
```

The third pair of examples also listen on localhost, but they listen for responses on port 8080 instead of port 80.

File excerpt: **/etc/nginx/sites-available/example.com**

```
listen     192.168.3.105:80;
listen     192.168.3.105:8080;
```

The fourth pair of examples specify a server listening for requests on the IP address `192.168.3.105`. The first listens on port `80` and the second on port `8080`.

File excerpt: **/etc/nginx/sites-available/example.com**

```
listen     80;
listen     *:80;
listen     8080;
listen     *:8080;
```

The fifth set of examples tell Nginx to listen on all domains and IP addresses on a specific port. listen 80; is equivalent to `listen *:80;`, and `listen 8080;` is equivalent to `listen *:8080;`.

File excerpt: **/etc/nginx/sites-available/example.com**

```
listen     12.34.56.77:80;
listen     12.34.56.78:80;
listen     12.34.56.79:80;
```

Finally, the last set of examples instruct the server to listen for requests on port `80` for the IP addresses `12.34.56.77`, `12.34.56.78`, and `12.34.56.79`.

Name-Based Virtual Hosting

The `server_name` directive, which is located in the `server` block, lets the administrator provide name-based virtual hosting. This allows multiple domains to be served from a single IP address. The server decides which domain to serve based on the request header it receives (for example, when someone requests a particular URL).

Typically, you will want to create one file per domain you want to host on your server. Each file should have its own `server` block, and the `server_name` directive is where you specify which domain this file affects.

Next we'll present a few common examples for the `server_name` directive:

File excerpt: **/etc/nginx/sites-available/example.com**

```
server_name   example.com;
```

The example above directs Nginx to process requests for example.com. This is the most basic configuration.

File excerpt: **/etc/nginx/sites-available/example.com**

```
server_name   example.com www.example.com;
```

The second example instructs the server to process requests for both example.com and www.example.com.

File excerpt: **/etc/nginx/sites-available/example.com**

```
server_name   *.example.com;
server_name   .example.com;
```

These two examples are equivalent. *.example.com and .example.com both instruct the server to process requests for all subdomains of example.com, including www.example.com, foo.example.com, etc.

File excerpt: **/etc/nginx/sites-available/example**

```
server_name   example.*;
```

The fourth example instructs the server to process requests for all domain names beginning with example., including example.com, example.org, example.net, example.foo.com, etc.

File excerpt: **/etc/nginx/sites-available/multi-list**

```
server_name   example.com linode.com icann.org;
```

The fifth example instructs the server to process requests for three different domain names. Note that any combination of domain names can be listed in a single server_name directive.

File excerpt: **/etc/nginx/sites-available/local**

```
server_name   localhost linode galloway;
```

Nginx allows you to specify names for virtual hosts that are not valid domain names. Nginx uses the name from the HTTP header to answer requests; it doesn't matter to Nginx whether the domain name is valid or not. In this case, the hostames can be specified in the /etc/hosts file.

Using non-domain hostnames may be useful if your Nginx server is deployed on a LAN, or if you already know all of the clients that will be making requests of the server. This includes front-end proxy servers that preconfigured /etc/hosts entries for the IP address on which Nginx is listening.

File excerpt: **/etc/nginx/sites-available/catchall**

```
server_name   "";
```

Finally, if you set server_name to the empty quote set (""), Nginx will process all requests that either do not have a hostname, or that have an unspecified hostname, such as requests for the IP address itself.

> Individual names are separated with a space. You can use regular expressions if desired.

Access Logs

The `access_log` directive can be set in the `http` block in `nginx.conf` or in the `server` block for a specific virtual domain. It sets the location of the Nginx access log. By defining the access log to a different path in each `server` block, you can sort the output specific to each virtual domain into its own file. An `access_log` directive defined in the `http` block can be used to log all access to a single file, or as a catch-all for access to virtual hosts that don't define their own log files.

You can use a path relative to the current directory:

File excerpt: **/etc/nginx/nginx.conf**

```
access_log logs/example.access.log;
```

Or, you can use a full path:

File excerpt: **/etc/nginx/sites-available/example.com**

```
access_log /srv/www/example.com/logs/access.log;
```

You can also disable the access log, although this is not recommended:

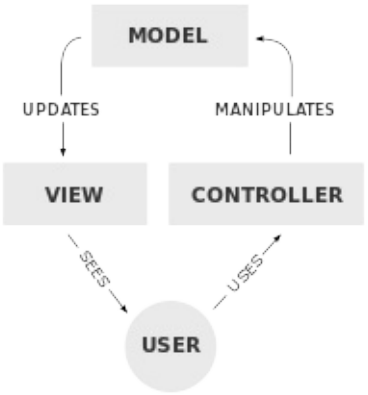
File excerpt: **/etc/nginx/nginx.conf**

```
access_log off;
```

MVC Architecture

Table of Contents

Description



Model–view–controller (MVC) is a software design pattern for implementing user interfaces on computers. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.[1][2]

Components

The central component of MVC, the model, captures the behavior of the application in terms of its problem domain, independent of the user interface.

- The model directly manages the data, logic, and rules of the application.
- A view can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
- The third part, the controller, accepts input and converts it to commands for the model or view.

Interactions

In addition to dividing the application into three kinds of components, the model–view–controller design defines the interactions between them.

- A model stores data that is retrieved according to commands from the controller and displayed in the view.
- A view generates new output to the user based on changes in the model.
- A controller can send commands to the model to update the model's state (e.g., editing a document). It can also send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document).

History

One of the seminal insights in the early development of graphical user interfaces, MVC became one of the first approaches to describe and implement software constructs in terms of their responsibilities.

Trygve Reenskaug introduced MVC into Smalltalk-76 while visiting the Xerox Palo Alto Research Center (PARC)[9][10] in the 1970s. In the 1980s, Jim Althoff and others implemented a version of MVC for the Smalltalk-80 class library. Only later did a 1988 article in The Journal of Object Technology (JOT) express MVC as a general concept.

The MVC pattern has subsequently evolved, giving rise to variants such as hierarchical model–view–controller (HMVC), model–view–adapter (MVA), model–view–presenter (MVP), model–view–viewmodel (MVVM), and others that adapted MVC to different contexts.[citation needed]

The use of the MVC pattern in web applications exploded in popularity after the introduction of Apple's WebObjects in 1996, which was originally written in Objective-C (that borrowed heavily from Smalltalk) and helped enforce MVC principles. Later, the MVC pattern became popular with Java developers when WebObjects was ported to Java. Later frameworks for Java, such as Spring (released in 2002), continued the strong bond between Java and MVC. The introduction of the frameworks Rails (December 2005, for Ruby) and Django (July 2005, for Python), both of which had a strong emphasis on rapid deployment, increased MVC's popularity outside the traditional enterprise environment in which it has long been popular. MVC web frameworks now hold large market-shares relative to non-MVC web toolkits.

Use in Web Application

Although originally developed for desktop computing, model–view–controller has been widely adopted as an architecture for World Wide Web applications in major programming languages. Several commercial and noncommercial web frameworks have been created

that enforce the pattern. These software frameworks vary in their interpretations, mainly in the way that the MVC responsibilities are divided between the client and server.

Early web MVC frameworks took a thin client approach that placed almost the entire model, view and controller logic on the server. This is still reflected in popular frameworks such as Ruby on Rails, Django, ASP.NET MVC. In this approach, the client sends either hyperlink requests or form input to the controller and then receives a complete and updated web page (or other document) from the view; the model exists entirely on the server. As client technologies have matured, frameworks such as AngularJS, EmberJS, JavaScriptMVC and Backbone have been created that allow the MVC components to execute partly on the client (also see Ajax)

What is Sinatra

Sinatra is a DSL for quickly creating web applications in Ruby with minimal effort:

```
# myapp.rb
require 'sinatra'

get '/' do
  'Hello world!'
end
```

Install the gem:

```
gem install sinatra
```

And run with:

```
ruby myapp.rb
```

View at: `http://localhost:4567`

It is recommended to also run gem install thin, which Sinatra will pick up if available.

Routes

In Sinatra, a route is an HTTP method paired with a URL-matching pattern. Each route is associated with a block:

```
get '/' do
  .. show something ..
end

post '/' do
  .. create something ..
end

put '/' do
  .. replace something ..
end

patch '/' do
  .. modify something ..
end

delete '/' do
  .. annihilate something ..
end

options '/' do
  .. appease something ..
end

link '/' do
  .. affiliate something ..
end

unlink '/' do
  .. separate something ..
end
```

Routes are matched in the order they are defined. The first route that matches the request is invoked.

Route patterns may include named parameters, accessible via the params hash:

```
get '/hello/:name' do
  # matches "GET /hello/foo" and "GET /hello/bar"
  # params['name'] is 'foo' or 'bar'
  "Hello #{params['name']}!"
end
```

You can also access named parameters via block parameters:

```
get '/hello/:name' do |n|
  # matches "GET /hello/foo" and "GET /hello/bar"
  # params['name'] is 'foo' or 'bar'
  # n stores params['name']
  "Hello #{n}!"
end
```

Route patterns may also include splat (or wildcard) parameters, accessible via the params['splat'] array:

```
get '/say/*/to/*' do
  # matches /say/hello/to/world
  params['splat'] # => ["hello", "world"]
end

get '/download/*.*' do
  # matches /download/path/to/file.xml
  params['splat'] # => ["path/to/file", "xml"]
end
```

Or with block parameters:

```
get '/download/*.*' do |path, ext|
  [path, ext] # => ["path/to/file", "xml"]
end
```

Route matching with Regular Expressions:

```
get /\A\/hello\/([\w]+)\z/ do
  "Hello, #{params['captures'].first}!"
end
```

Or with a block parameter:

```
get %r{/hello/([\w]+)} do |c|
  # Matches "GET /meta/hello/world", "GET /hello/world/1234" etc.
  "Hello, #{c}!"
end
```

Route patterns may have optional parameters:

```
get '/posts.?:format?' do
  # matches "GET /posts" and any extension "GET /posts.json", "GET /posts.xml" etc.
end
```

Routes may also utilize query parameters:

```
get '/posts' do
  # matches "GET /posts?title=foo&author=bar"
  title = params['title']
  author = params['author']
  # uses title and author variables; query is optional to the /posts route
end
```

By the way, unless you disable the path traversal attack protection (see below), the request path might be modified before matching against your routes.

Conditions

Routes may include a variety of matching conditions, such as the user agent:

```
get '/foo', :agent => /Songbird (\d\.\d)[\d\/]*?/ do
  "You're using Songbird version #{params['agent'][0]}"
end

get '/foo' do
  # Matches non-songbird browsers
end
```

Other available conditions are host_name and provides:

```
get '/', :host_name => /^admin\./ do
  "Admin Area, Access denied!"
end

get '/', :provides => 'html' do
  haml :index
end

get '/', :provides => ['rss', 'atom', 'xml'] do
  builder :feed
end
```

provides searches the request's Accept header.

You can easily define your own conditions:

```
set(:probability) { |value| condition { rand <= value } }

get '/win_a_car', :probability => 0.1 do
  "You won!"
end

get '/win_a_car' do
  "Sorry, you lost."
end
```

For a condition that takes multiple values use a splat:

```
set(:auth) do |*roles|   # <- notice the splat here
  condition do
    unless logged_in? && roles.any? {|role| current_user.in_role? role }
      redirect "/login/", 303
    end
  end
end

get "/my/account/", :auth => [:user, :admin] do
  "Your Account Details"
end

get "/only/admin/", :auth => :admin do
  "Only admins are allowed here!"
end
```

Return Values

The return value of a route block determines at least the response body passed on to the HTTP client, or at least the next middleware in the Rack stack. Most commonly, this is a string, as in the above examples. But other values are also accepted.

You can return any object that would either be a valid Rack response, Rack body object or HTTP status code:

- An Array with three elements: [status (Fixnum), headers (Hash), response body (responds to #each)]
- An Array with two elements: [status (Fixnum), response body (responds to #each)]
- An object that responds to #each and passes nothing but strings to the given block
- A Fixnum representing the status code

That way we can, for instance, easily implement a streaming example:

```
class Stream
  def each
    100.times { \|i\| yield "\#{i}\n" }
  end
end

get\('\/'\) { Stream.new }
```

You can also use the stream helper method (described below) to reduce boiler plate and embed the streaming logic in the route.

Custom Route Matchers

As shown above, Sinatra ships with built-in support for using String patterns and regular expressions as route matches. However, it does not stop there. You can easily define your own matchers:

```
class AllButPattern
  Match = Struct.new\(:captures\)

def initialize\(except\)
    @except   = except
    @captures = Match.new\(\[\]\)
  end

def match\(str\)
    @captures unless @except === str
  end
end

def all\_but\(pattern\)
  AllButPattern.new\(pattern\)
end

get all\_but\("\/index"\) do

# ...

end
```

Note that the above example might be over-engineered, as it can also be expressed as:

```
get \/\/ do
  pass if request.path\_info == "\/index"

# ...

end
```

Or, using negative look ahead:

```
get %r{^\(?!\/index$\)} do

# ...

end
```

Custom Route Matchers

As shown above, Sinatra ships with built-in support for using String patterns and regular expressions as route matches. However, it does not stop there. You can easily define your own matchers:

```
class AllButPattern
  Match = Struct.new(:captures)

  def initialize(except)
    @except   = except
    @captures = Match.new([])
  end

  def match(str)
    @captures unless @except === str
  end
end

def all_but(pattern)
  AllButPattern.new(pattern)
end

get all_but("/index") do
  # ...
end
```

Note that the above example might be over-engineered, as it can also be expressed as:

```
get // do
  pass if request.path_info == "/index"
  # ...
end
Or, using negative look ahead:

get %r{^(?!/index$)} do
  # ...
end
```

Views / Templates

Each template language is exposed via its own rendering method. These methods simply return a string:

```
get '/' do
  erb :index
end
```

This renders views/index.erb.

Instead of a template name, you can also just pass in the template content directly:

```
get '/' do
  code = "<%= Time.now %>"
  erb code
end
```

Templates take a second argument, the options hash:

```
get '/' do
  erb :index, :layout => :post
end
```

This will render views/index.erb embedded in the views/post.erb (default is views/layout.erb, if it exists).

Hello World in Rails

What is Rails

Rails is a web application development framework written in the Ruby language. It is designed to make programming web applications easier by making assumptions about what every developer needs to get started. It allows you to write less code while accomplishing more than many other languages and frameworks. Experienced Rails developers also report that it makes web application development more fun.

Rails is opinionated software. It makes the assumption that there is a "best" way to do things, and it's designed to encourage that way - and in some cases to discourage alternatives. If you learn "The Rails Way" you'll probably discover a tremendous increase in productivity. If you persist in bringing old habits from other languages to your Rails development, and trying to use patterns you learned elsewhere, you may have a less happy experience.

The Rails philosophy includes two major guiding principles:

- **Don't Repeat Yourself:** DRY is a principle of software development which states that "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." By not writing the same information over and over again, our code is more maintainable, more extensible, and less buggy.
- **Convention Over Configuration:** Rails has opinions about the best way to do many things in a web application, and defaults to this set of conventions, rather than require that you specify every minutiae through endless configuration files.

Creating a New Rails Project

The best way to read this guide is to follow it step by step. All steps are essential to run this example application and no additional code or steps are needed.

By following along with this guide, you'll create a Rails project called blog, a (very) simple weblog. Before you can start building the application, you need to make sure that you have Rails itself installed.

Installing Rails

Open up a command line prompt. On Mac OS X open Terminal.app, on Windows choose "Run" from your Start menu and type 'cmd.exe'. Any commands prefaced with a dollar sign $ should be run in the command line. Verify that you have a current version of Ruby installed:

```
$ ruby -v
ruby 2.3.0p0
```

Many popular UNIX-like OSes ship with an acceptable version of SQLite3. On Windows, if you installed Rails through Rails Installer, you already have SQLite installed. Others can find installation instructions at the SQLite3 website. Verify that it is correctly installed and in your PATH:

```
$ sqlite3 --version
```

The program should report its version.

To install Rails, use the gem install command provided by RubyGems:

```
$ gem install rails
```

To verify that you have everything installed correctly, you should be able to run the following:

```
$ rails --version
```

If it says something like "Rails 5.0.0", you are ready to continue.

Creating the Blog Application

Rails comes with a number of scripts called generators that are designed to make your development life easier by creating everything that's necessary to start working on a particular task. One of these is the new application generator, which will provide you with the foundation of a fresh Rails application so that you don't have to write it yourself.

To use this generator, open a terminal, navigate to a directory where you have rights to create files, and type:

```
$ rails new blog
```

This will create a Rails application called Blog in a blog directory and install the gem dependencies that are already mentioned in Gemfile using bundle install.

After you create the blog application, switch to its folder:

```
$ cd blog
```

The blog directory has a number of auto-generated files and folders that make up the structure of a Rails application. Most of the work in this tutorial will happen in the app folder, but here's a basic rundown on the function of each of the files and folders that Rails created by default:

File/Folder	Purpose
app/	Contains the controllers, models, views, helpers, mailers and assets for your application. You'll focus on this folder for the remainder of this guide.
bin/	Contains the rails script that starts your app and can contain other scripts you use to setup, update, deploy or run your application.
config/	Configure your application's routes, database, and more. This is covered in more detail in Configuring Rails Applications.
config.ru	Rack configuration for Rack based servers used to start the application.
db/	Contains your current database schema, as well as the database migrations.
Gemfile Gemfile.lock	These files allow you to specify what gem dependencies are needed for your Rails application. These files are used by the Bundler gem. For more information about Bundler, see the Bundler website.
lib/	Extended modules for your application.
log/	Application log files.
public/	The only folder seen by the world as-is. Contains static files and compiled assets.
Rakefile	This file locates and loads tasks that can be run from the command line. The task definitions are defined throughout the components of Rails. Rather than changing Rakefile, you should add your own tasks by adding files to the lib/tasks directory of your application.
README.md	This is a brief instruction manual for your application. You should edit this file to tell others what your application does, how to set it up, and so on.
test/	Unit tests, fixtures, and other test apparatus. These are covered in Testing Rails Applications.
tmp/	Temporary files (like cache and pid files).
vendor/	A place for all third-party code. In a typical Rails application this includes vendored gems.

Hello, Rails!

To begin with, let's get some text up on screen quickly. To do this, you need to get your Rails application server running.

Starting up the Web Server

You actually have a functional Rails application already. To see it, you need to start a web server on your development machine. You can do this by running the following in the blog directory:

```
$ bin/rails server
```

This will fire up Puma, a web server distributed with Rails by default. To see your application in action, open a browser window and navigate to http://localhost:3000. You should see the Rails default information page:

To stop the web server, hit Ctrl+C in the terminal window where it's running. To verify the server has stopped you should see your command prompt cursor again. For most UNIX-like systems including Mac OS X this will be a dollar sign $. In development mode, Rails does not generally require you to restart the server; changes you make in files will be automatically picked up by the server.

The "Welcome aboard" page is the smoke test for a new Rails application: it makes sure that you have your software configured correctly enough to serve a page.

Say "Hello", Rails

To get Rails saying "Hello", you need to create at minimum a controller and a view.

A controller's purpose is to receive specific requests for the application. Routing decides which controller receives which requests. Often, there is more than one route to each controller, and different routes can be served by different actions. Each action's purpose is to collect information to provide it to a view.

A view's purpose is to display this information in a human readable format. An important distinction to make is that it is the controller, not the view, where information is collected. The view should just display that information. By default, view templates are written in a language called eRuby (Embedded Ruby) which is processed by the request cycle in Rails before being sent to the user.

To create a new controller, you will need to run the "controller" generator and tell it you want a controller called "Welcome" with an action called "index", just like this:

```
$ bin/rails generate controller Welcome index
```

Rails will create several files and a route for you.

```
create  app/controllers/welcome_controller.rb
 route  get 'welcome/index'
invoke  erb
create    app/views/welcome
create    app/views/welcome/index.html.erb
invoke  test_unit
create    test/controllers/welcome_controller_test.rb
invoke  helper
create    app/helpers/welcome_helper.rb
invoke  assets
invoke    coffee
create      app/assets/javascripts/welcome.coffee
invoke    scss
create      app/assets/stylesheets/welcome.scss
```

Most important of these are of course the controller, `located at app/controllers/welcome_controller.rb` and the view, located at `app/views/welcome/index.html.erb`

Open the app/views/welcome/index.html.erb file in your text editor. Delete all of the existing code in the file, and replace it with the following single line of code:

```
<h1>Hello, Rails!</h1>
```

Setting the Application Home Page

Now that we have made the controller and view, we need to tell Rails when we want "Hello, Rails!" to show up. In our case, we want it to show up when we navigate to the root URL of our site, http://localhost:3000. At the moment, "Welcome aboard" is occupying that spot.

Next, you have to tell Rails where your actual home page is located.

Open the file config/routes.rb in your editor.

```
Rails.application.routes.draw do
  get 'welcome/index'

  # For details on the DSL available within this file, see http://guides.rubyonrails.o
rg/routing.html
end
```

This is your application's routing file which holds entries in a special DSL (domain-specific language) that tells Rails how to connect incoming requests to controllers and actions. Edit this file by adding the line of code root 'welcome#index'. It should look something like the following:

```
Rails.application.routes.draw do
  get 'welcome/index'

  root 'welcome#index'
end
```

`root 'welcome#index'` tells Rails to map requests to the root of the application to the welcome controller's index action and `get 'welcome/index'` tells Rails to map requests to http://localhost:3000/welcome/index to the welcome controller's index action. This was created earlier when you ran the controller generator (`bin/rails generate controller Welcome index`).

Launch the web server again if you stopped it to generate the controller (`bin/rails server`) and navigate to http://localhost:3000 in your browser. You'll see the "Hello, Rails!" message you put into `app/views/welcome/index.html.erb` , indicating that this new route is indeed going to `WelcomeController's` index action and is rendering the view correctly.

CRUD with Rails

Getting Up and Running

Now that you've seen how to create a controller, an action and a view, let's create something with a bit more substance.

In the Blog application, you will now create a new resource. A resource is the term used for a collection of similar objects, such as articles, people or animals. You can create, read, update and destroy items for a resource and these operations are referred to as CRUD operations.

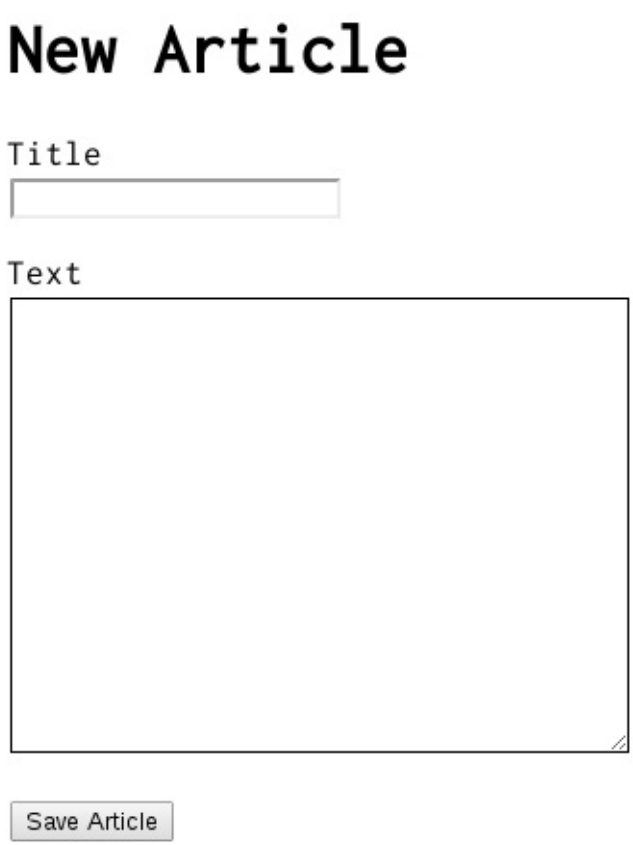
Rails provides a resources method which can be used to declare a standard REST resource. You need to add the article resource to the config/routes.rb so the file will look as follows:

```
Rails.application.routes.draw do

  resources :articles

  root 'welcome#index'
end
```

If you run bin/rails routes, you'll see that it has defined routes for all the standard RESTful actions. The meaning of the prefix column (and other columns) will be seen later, but for now notice that Rails has inferred the singular form article and makes meaningful use of the distinction.

```
$ bin/rails routes
      Prefix Verb   URI Pattern                  Controller#Action
    articles GET    /articles(.:format)          articles#index
             POST   /articles(.:format)          articles#create
 new_article GET    /articles/new(.:format)      articles#new
edit_article GET    /articles/:id/edit(.:format) articles#edit
     article GET    /articles/:id(.:format)      articles#show
             PATCH  /articles/:id(.:format)      articles#update
             PUT    /articles/:id(.:format)      articles#update
             DELETE /articles/:id(.:format)      articles#destroy
        root GET    /                            welcome#index
```

In the next section, you will add the ability to create new articles in your application and be able to view them. This is the "C" and the "R" from CRUD: create and read. The form for doing this will look like this:



It will look a little basic for now, but that's ok. We'll look at improving the styling for it afterwards.

1. Laying down the ground work

Firstly, you need a place within the application to create a new article. A great place for that would be at /articles/new. With the route already defined, requests can now be made to /articles/new in the application. Navigate to http://localhost:3000/articles/new and you'll see a routing error:

This error occurs because the route needs to have a controller defined in order to serve the request. The solution to this particular problem is simple: create a controller called ArticlesController. You can do this by running this command:

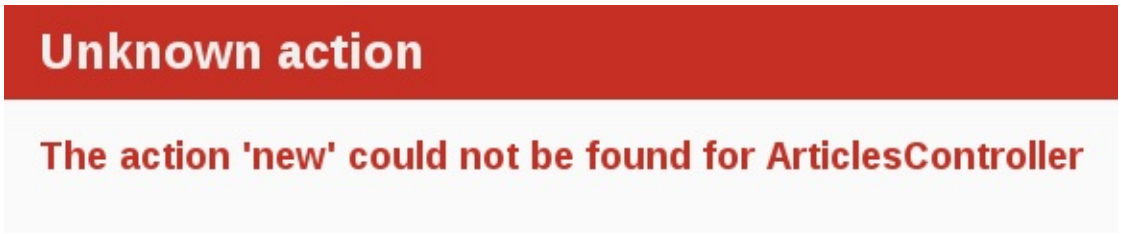
```
$ bin/rails generate controller Articles
```

If you open up the newly generated app/controllers/articles_controller.rb you'll see a fairly empty controller:

```
class ArticlesController < ApplicationController
end
```

A controller is simply a class that is defined to inherit from ApplicationController. It's inside this class that you'll define methods that will become the actions for this controller. These actions will perform CRUD operations on the articles within our system.

If you refresh http://localhost:3000/articles/new now, you'll get a new error:

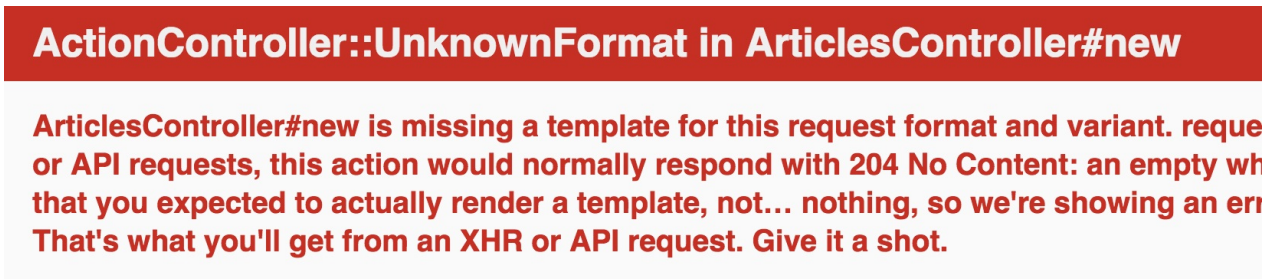


This error indicates that Rails cannot find the new action inside the ArticlesController that you just generated. This is because when controllers are generated in Rails they are empty by default, unless you tell it your desired actions during the generation process.

To manually define an action inside a controller, all you need to do is to define a new method inside the controller. Open app/controllers/articles_controller.rb and inside the ArticlesController class, define the new method so that your controller now looks like this:

```
class ArticlesController < ApplicationController
  def new
  end
end
```

With the new method defined in ArticlesController, if you refresh http://localhost:3000/articles/new you'll see another error:



You're getting this error now because Rails expects plain actions like this one to have views associated with them to display their information. With no view available, Rails will raise an exception.

In the above image, the bottom line has been truncated. Let's see what the full error message looks like:

> ArticlesController#new is missing a template for this request format and variant. request.formats: ["text/html"] request.variant: [] NOTE! For XHR/Ajax or API requests, this action would normally respond with 204 No Content: an empty white screen. Since you're loading it in a web browser, we assume that you expected to actually render a template, not… nothing, so we're showing an error to be extra-clear. If you expect 204 No Content, carry on. That's what you'll get from an XHR or API request. Give it a shot.

That's quite a lot of text! Let's quickly go through and understand what each part of it means.

The first part identifies which template is missing. In this case, it's the articles/new template. Rails will first look for this template. If not found, then it will attempt to load a template called application/new. It looks for one here because the ArticlesController inherits from ApplicationController.

The next part of the message contains a hash. The :locale key in this hash simply indicates which spoken language template should be retrieved. By default, this is the English - or "en" - template. The next key, :formats specifies the format of template to be served in response. The default format is :html, and so Rails is looking for an HTML template. The final key, :handlers, is telling us what template handlers could be used to render our template. :erb is most commonly used for HTML templates, :builder is used for XML templates, and :coffee uses CoffeeScript to build JavaScript templates.

The message also contains request.formats which specifies the format of template to be served in response. It is set to text/html as we requested this page via browser, so Rails is looking for an HTML template.

The simplest template that would work in this case would be one located at app/views/articles/new.html.erb. The extension of this file name is important: the first extension is the format of the template, and the second extension is the handler that will be used. Rails is attempting to find a template called articles/new within app/views for the application. The format for this template can only be html and the handler must be one of erb, builder or coffee. :erb is most commonly used for HTML templates, :builder is used for XML templates, and :coffee uses CoffeeScript to build JavaScript templates. Because you want to create a new HTML form, you will be using the ERB language which is designed to embed Ruby in HTML.

Therefore the file should be called articles/new.html.erb and needs to be located inside the app/views directory of the application.

Go ahead now and create a new file at app/views/articles/new.html.erb and write this content in it:

```
<h1>New Article</h1>
```

When you refresh http://localhost:3000/articles/new you'll now see that the page has a title. The route, controller, action and view are now working harmoniously! It's time to create the form for a new article.

2. The first form

To create a form within this template, you will use a form builder. The primary form builder for Rails is provided by a helper method called form_for. To use this method, add this code into app/views/articles/new.html.erb:

```
<%= form_for :article do |f| %>
  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>
<% end %>
```

If you refresh the page now, you'll see the exact same form as in the example. Building forms in Rails is really just that easy!

When you call form_for, you pass it an identifying object for this form. In this case, it's the symbol :article. This tells the form_for helper what this form is for. Inside the block for this method, the FormBuilder object - represented by f - is used to build two labels and two text fields, one each for the title and text of an article. Finally, a call to submit on the f object will create a submit button for the form.

There's one problem with this form though. If you inspect the HTML that is generated, by viewing the source of the page, you will see that the action attribute for the form is pointing at /articles/new. This is a problem because this route goes to the very page that you're on right at the moment, and that route should only be used to display the form for a new article.

The form needs to use a different URL in order to go somewhere else. This can be done quite simply with the :url option of form_for. Typically in Rails, the action that is used for new form submissions like this is called "create", and so the form should be pointed to that action.

Edit the form_for line inside app/views/articles/new.html.erb to look like this:

```
<%= form_for :article, url: articles_path do |f| %>
```

In this example, the articles_path helper is passed to the :url option. To see what Rails will do with this, we look back at the output of bin/rails routes:

```
$ bin/rails routes
      Prefix Verb   URI Pattern                  Controller#Action
    articles GET    /articles(.:format)          articles#index
             POST   /articles(.:format)          articles#create
 new_article GET    /articles/new(.:format)      articles#new
edit_article GET    /articles/:id/edit(.:format) articles#edit
     article GET    /articles/:id(.:format)      articles#show
             PATCH  /articles/:id(.:format)      articles#update
             PUT    /articles/:id(.:format)      articles#update
             DELETE /articles/:id(.:format)      articles#destroy
        root GET    /                            welcome#index
```

The articles_path helper tells Rails to point the form to the URI Pattern associated with the articles prefix; and the form will (by default) send a POST request to that route. This is associated with the create action of the current controller, the ArticlesController.

With the form and its associated route defined, you will be able to fill in the form and then click the submit button to begin the process of creating a new article, so go ahead and do that. When you submit the form, you should see a familiar error:

You now need to create the create action within the ArticlesController for this to work.

3. Creating articles

To make the "Unknown action" go away, you can define a create action within the ArticlesController class in app/controllers/articles_controller.rb, underneath the new action, as shown:

```
class ArticlesController < ApplicationController
  def new
  end

  def create
  end
end
```

If you re-submit the form now, you may not see any change on the page. Don't worry! This is because Rails by default returns 204 No Content response for an action if we don't specify what the response should be. We just added the create action but didn't specify anything about how the response should be. In this case, the create action should save our new article to the database.

When a form is submitted, the fields of the form are sent to Rails as parameters. These parameters can then be referenced inside the controller actions, typically to perform a particular task. To see what these parameters look like, change the create action to this:

```
def create
  render plain: params[:article].inspect
end
```

The render method here is taking a very simple hash with a key of :plain and value of params[:article].inspect. The params method is the object which represents the parameters (or fields) coming in from the form. The params method returns an ActionController::Parameters object, which allows you to access the keys of the hash using either strings or symbols. In this situation, the only parameters that matter are the ones from the form.

> Ensure you have a firm grasp of the params method, as you'll use it fairly regularly. Let's consider an example URL: http://www.example.com/?username=dhh&email=dhh@email.com. In this URL, params[:username] would equal "dhh" and params[:email] would equal "dhh@email.com".

If you re-submit the form one more time you'll now no longer get the missing template error. Instead, you'll see something that looks like the following:

```
<ActionController::Parameters {"title"=>"First Article!", "text"=>"This is my first ar
ticle."} permitted: false>
```

This action is now displaying the parameters for the article that are coming in from the form. However, this isn't really all that helpful. Yes, you can see the parameters but nothing in particular is being done with them.

4. Creating the Article model

Models in Rails use a singular name, and their corresponding database tables use a plural name. Rails provides a generator for creating models, which most Rails developers tend to use when creating new models. To create the new model, run this command in your terminal:

```
$ bin/rails generate model Article title:string text:text
```

With that command we told Rails that we want an Article model, together with a title attribute of type string, and a text attribute of type text. Those attributes are automatically added to the articles table in the database and mapped to the Article model.

Rails responded by creating a bunch of files. For now, we're only interested in app/models/article.rb and db/migrate/20140120191729_create_articles.rb (your name could be a bit different). The latter is responsible for creating the database structure, which is what we'll look at next.

> Active Record is smart enough to automatically map column names to model attributes, which means you don't have to declare attributes inside Rails models, as that will be done automatically by Active Record.

5. Running a Migration

As we've just seen, bin/rails generate model created a database migration file inside the db/migrate directory. Migrations are Ruby classes that are designed to make it simple to create and modify database tables. Rails uses rake commands to run migrations, and it's possible to undo a migration after it's been applied to your database. Migration filenames include a timestamp to ensure that they're processed in the order that they were created.

If you look in the db/migrate/YYYYMMDDHHMMSS_create_articles.rb file (remember, yours will have a slightly different name), here's what you'll find:

```
class CreateArticles < ActiveRecord::Migration[5.0]
  def change
    create_table :articles do |t|
      t.string :title
      t.text :text

      t.timestamps
    end
  end
end
```

The above migration creates a method named change which will be called when you run this migration. The action defined in this method is also reversible, which means Rails knows how to reverse the change made by this migration, in case you want to reverse it later. When you run this migration it will create an articles table with one string column and a text column. It also creates two timestamp fields to allow Rails to track article creation and update times.

> For more information about migrations, refer to Rails Database Migrations.

At this point, you can use a bin/rails command to run the migration:

```
$ bin/rails db:migrate
```

Rails will execute this migration command and tell you it created the Articles table.

```
==  CreateArticles: migrating ================================================
-- create_table(:articles)
   -> 0.0019s
==  CreateArticles: migrated (0.0020s) =======================================
```

> Because you're working in the development environment by default, this command will apply to the database defined in the development section of your config/database.yml file. If you would like to execute migrations in another environment, for instance in production, you must explicitly pass it when invoking the command: bin/rails db:migrate RAILS_ENV=production.

6.Saving data in the controller

Back in ArticlesController, we need to change the create action to use the new Article model to save the data in the database. Open app/controllers/articles_controller.rb and change the create action to look like this:

```
def create
  @article = Article.new(params[:article])

  @article.save
  redirect_to @article
end
```

Here's what's going on: every Rails model can be initialized with its respective attributes, which are automatically mapped to the respective database columns. In the first line we do just that (remember that `params[:article]` contains the attributes we're interested in). Then, `@article.save` is responsible for saving the model in the database. Finally, we redirect the user to the show action, which we'll define later.

> You might be wondering why the A in `Article.new` is capitalized above, whereas most other references to articles in this guide have used lowercase. In this context, we are referring to the class named `Article` that is defined in `app/models/article.rb`. Class names in Ruby must begin with a capital letter.
>
> As we'll see later, `@article.save` returns a boolean indicating whether the article was saved or not.

If you now go to http://localhost:3000/articles/new you'll almost be able to create an article. Try it! You should get an error that looks like this:

Rails has several security features that help you write secure applications, and you're running into one of them now. This one is called strong parameters, which requires us to tell Rails exactly which parameters are allowed into our controller actions.

Why do you have to bother? The ability to grab and automatically assign all controller parameters to your model in one shot makes the programmer's job easier, but this convenience also allows malicious use. What if a request to the server was crafted to look like a new article form submit but also included extra fields with values that violated your application's integrity? They would be 'mass assigned' into your model and then into the database along with the good stuff - potentially breaking your application or worse.

We have to whitelist our controller parameters to prevent wrongful mass assignment. In this case, we want to both allow and require the title and text parameters for valid use of create. The syntax for this introduces require and permit. The change will involve one line in the create action:

```
@article = Article.new(params.require(:article).permit(:title, :text))
```

This is often factored out into its own method so it can be reused by multiple actions in the same controller, for example create and update. Above and beyond mass assignment issues, the method is often made private to make sure it can't be called outside its intended context. Here is the result:

```
def create
  @article = Article.new(article_params)

  @article.save
  redirect_to @article
end

private
  def article_params
    params.require(:article).permit(:title, :text)
  end
```

For more information, refer to the reference above and this blog article about Strong Parameters.

7 Showing Articles

If you submit the form again now, Rails will complain about not finding the show action. That's not very useful though, so let's add the show action before proceeding.

As we have seen in the output of `bin/rails` routes, the route for show action is as follows:

```
article GET    /articles/:id(.:format)      articles#show
```

The special syntax `:id` tells rails that this route expects an `:id` parameter, which in our case will be the id of the article.

As we did before, we need to add the `show` action in `app/controllers/articles_controller.rb` and its respective view.

> A frequent practice is to place the standard CRUD actions in each controller in the following order: `index, show, new, edit, create, update` and `destroy`. You may use any order you choose, but keep in mind that these are public methods; as mentioned earlier in this guide, they must be placed before any private or protected method in the controller in order to work.

Given that, let's add the `show` action, as follows:

```
class ArticlesController < ApplicationController
  def show
    @article = Article.find(params[:id])
  end

  def new
  end

  # snippet for brevity
```

A couple of things to note. We use `Article.find` to find the article we're interested in, passing in `params[:id]` to get the `:id` parameter from the request. We also use an instance variable (prefixed with @) to hold a reference to the article object. We do this because Rails will pass all instance variables to the view.

Now, create a new file `app/views/articles/show.html.erb` with the following content:

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>
```

With this change, you should finally be able to create new articles. Visit http://localhost:3000/articles/new and give it a try!

Show action for articles

8 Listing all articles

We still need a way to list all our articles, so let's do that. The route for this as per output of `bin/rails routes` is:

```
articles GET    /articles(.:format)          articles#index
```

Add the corresponding index action for that route inside the ArticlesController in the `app/controllers/articles_controller.rb` file. When we write an index action, the usual practice is to place it as the first method in the controller. Let's do it:

```
class ArticlesController < ApplicationController
  def index
    @articles = Article.all
  end

  def show
    @article = Article.find(params[:id])
  end

  def new
  end

  # snippet for brevity
```

And then finally, add the view for this action, located at `app/views/articles/index.html.erb`:

```
<h1>Listing articles</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
  </tr>

  <% @articles.each do |article| %>
    <tr>
      <td><%= article.title %></td>
      <td><%= article.text %></td>
      <td><%= link_to 'Show', article_path(article) %></td>
    </tr>
  <% end %>
</table>
```

Now if you go to `http://localhost:3000/articles` you will see a list of all the articles that you have created.

9 Adding links

You can now create, show, and list articles. Now let's add some links to navigate through pages.

Open `app/views/welcome/index.html.erb` and modify it as follows:

```
<h1>Hello, Rails!</h1>
<%= link_to 'My Blog', controller: 'articles' %>
```

The `link_to` method is one of Rails' built-in view helpers. It creates a hyperlink based on text to display and where to go - in this case, to the path for articles.

Let's add links to the other views as well, starting with adding this "New Article" link to `app/views/articles/index.html.erb`, placing it above the `<table>` tag:

```
<%= link_to 'New article', new_article_path %>
```

This link will allow you to bring up the form that lets you create a new article.

Now, add another link in app/views/articles/new.html.erb, underneath the form, to go back to the index action:

```
<%= form_for :article, url: articles_path do |f| %>
  ...
<% end %>

<%= link_to 'Back', articles_path %>
```

Finally, add a link to the app/views/articles/show.html.erb template to go back to the index action as well, so that people who are viewing a single article can go back and view the whole list again:

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

<%= link_to 'Back', articles_path %>
```

> If you want to link to an action in the same controller, you don't need to specify the :controller option, as Rails will use the current controller by default.
>
> In development mode (which is what you're working in by default), Rails reloads your application with every browser request, so there's no need to stop and restart the web server when a change is made.

10 Adding Some Validation

The model file, `app/models/article.rb` is about as simple as it can get:

```
class Article < ApplicationRecord
end
```

There isn't much to this file - but note that the `Article` class inherits from `ApplicationRecord`. `ApplicationRecord` inherits from `ActiveRecord::Base` which supplies a great deal of functionality to your Rails models for free, including basic database CRUD (Create, Read, Update, Destroy) operations, data validation, as well as sophisticated search support and the ability to relate multiple models to one another.

Rails includes methods to help you validate the data that you send to models. Open the `app/models/article.rb` file and edit it:

```
class Article < ApplicationRecord
  validates :title, presence: true,
                    length: { minimum: 5 }
end
```

These changes will ensure that all articles have a title that is at least five characters long. Rails can validate a variety of conditions in a model, including the presence or uniqueness of columns, their format, and the existence of associated objects. Validations are covered in detail in Active Record Validations.

With the validation now in place, when you call `@article.save` on an invalid article, it will return false. If you open `app/controllers/articles_controller.rb` again, you'll notice that we don't check the result of calling `@article.save` inside the create action. If `@article.save` fails in this situation, we need to show the form back to the user. To do this, change the new and create actions inside `app/controllers/articles_controller.rb` to these:

```
def new
  @article = Article.new
end

def create
  @article = Article.new(article_params)

  if @article.save
    redirect_to @article
  else
    render 'new'
  end
end

private
  def article_params
    params.require(:article).permit(:title, :text)
  end
```

The `new` action is now creating a new instance variable called `@article`, and you'll see why that is in just a few moments.

Notice that inside the `create` action we use `render` instead of `redirect_to` when `save` returns `false`. The render method is used so that the @article object is passed back to the new template when it is rendered. This rendering is done within the same request as the form submission, whereas the `redirect_to` will tell the browser to issue another request.

If you reload `http://localhost:3000/articles/new` and try to save an article without a title, Rails will send you back to the form, but that's not very useful. You need to tell the user that something went wrong. To do that, you'll modify `app/views/articles/new.html.erb` to check for error messages:

```
<%= form_for :article, url: articles_path do |f| %>

  <% if @article.errors.any? %>
    <div id="error_explanation">
      <h2>
        <%= pluralize(@article.errors.count, "error") %> prohibited
        this article from being saved:
      </h2>
      <ul>
        <% @article.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>

<% end %>

<%= link_to 'Back', articles_path %>
```

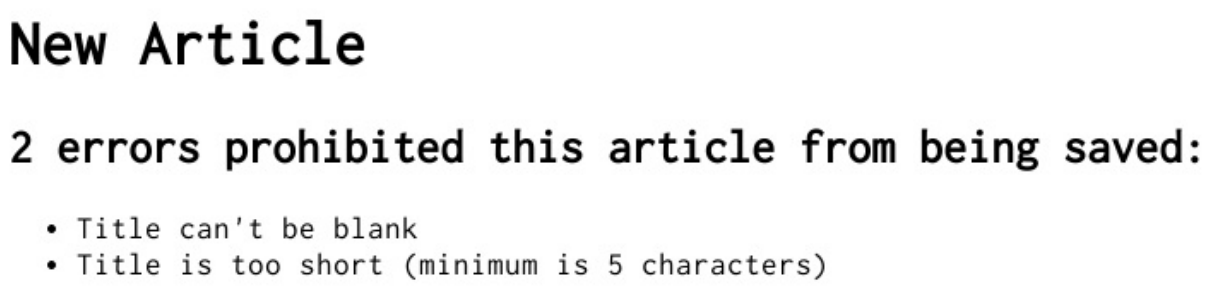
A few things are going on. We check if there are any errors with `@article.errors.any?`, and in that case we show a list of all errors with `@article.errors.full_messages`.

`pluralize` is a rails helper that takes a number and a string as its arguments. If the number is greater than one, the string will be automatically pluralized.

The reason why we added `@article = Article.new` in the `ArticlesController` is that otherwise `@article` would be `nil` in our view, and calling `@article.errors.any?` would throw an error.

> Rails automatically wraps fields that contain an error with a div with class `field_with_errors`. You can define a css rule to make them standout.

Now you'll get a nice error message when saving an article without title when you attempt to do just that on the new article form `http://localhost:3000/articles/new`:



11 Updating Articles

We've covered the "CR" part of CRUD. Now let's focus on the "U" part, updating articles.

The first step we'll take is adding an edit action to the ArticlesController, generally between the new and create actions, as shown:

```
def new
  @article = Article.new
end

def edit
  @article = Article.find(params[:id])
end

def create
  @article = Article.new(article_params)

  if @article.save
    redirect_to @article
  else
    render 'new'
  end
end
```

The view will contain a form similar to the one we used when creating new articles. Create a file called `app/views/articles/edit.html.erb` and make it look as follows:

```
<h1>Editing article</h1>

<%= form_for :article, url: article_path(@article), method: :patch do |f| %>

  <% if @article.errors.any? %>
    <div id="error_explanation">
      <h2>
        <%= pluralize(@article.errors.count, "error") %> prohibited
        this article from being saved:
      </h2>
      <ul>
        <% @article.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>

<% end %>

<%= link_to 'Back', articles_path %>
```

This time we point the form to the update action, which is not defined yet but will be very soon.

The `method: :patch` option tells Rails that we want this form to be submitted via the `PATCH` HTTP method which is the HTTP method you're expected to use to update resources according to the REST protocol.

The first parameter of `form_for` can be an object, say, `@article` which would cause the helper to fill in the form with the fields of the object. Passing in a symbol (`:article`) with the same name as the instance variable (`@article`) also automagically leads to the same behavior. This is what is happening here. More details can be found in `form_for` documentation.

Next, we need to create the update action in `app/controllers/articles_controller.rb`. Add it between the create action and the private method:

```
def create
  @article = Article.new(article_params)

  if @article.save
    redirect_to @article
  else
    render 'new'
  end
end

def update
  @article = Article.find(params[:id])

  if @article.update(article_params)
    redirect_to @article
  else
    render 'edit'
  end
end

private
  def article_params
    params.require(:article).permit(:title, :text)
  end
```

The `new` method, `update`, is used when you want to update a record that already exists, and it accepts a hash containing the attributes that you want to update. As before, if there was an error updating the article we want to show the form back to the user.

We reuse the article_params method that we defined earlier for the create action.

> It is not necessary to pass all the attributes to update. For example, if @article.update(title: 'A new title') was called, Rails would only update the title attribute, leaving all other attributes untouched.

Finally, we want to show a link to the edit action in the list of all the articles, so let's add that now to `app/views/articles/index.html.erb` to make it appear next to the "Show" link:

```
<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
    <th colspan="2"></th>
  </tr>

  <% @articles.each do |article| %>
    <tr>
      <td><%= article.title %></td>
      <td><%= article.text %></td>
      <td><%= link_to 'Show', article_path(article) %></td>
      <td><%= link_to 'Edit', edit_article_path(article) %></td>
    </tr>
  <% end %>
</table>
```

And we'll also add one to the `app/views/articles/show.html.erb` template as well, so that there's also an "Edit" link on an article's page. Add this at the bottom of the template:

```
...

<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>
```

And here's how our app looks so far:

12 Using partials to clean up duplication in views

Our edit page looks very similar to the new page; in fact, they both share the same code for displaying the form. Let's remove this duplication by using a view partial. By convention, partial files are prefixed with an underscore.

You can read more about partials in the Layouts and Rendering in Rails guide.

Create a new file `app/views/articles/_form.html.erb` with the following content:

```
<%= form_for @article do |f| %>

  <% if @article.errors.any? %>
    <div id="error_explanation">
      <h2>
        <%= pluralize(@article.errors.count, "error") %> prohibited
        this article from being saved:
      </h2>
      <ul>
        <% @article.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>

<% end %>
```

Everything except for the form_for declaration remained the same. The reason we can use this shorter, simpler `form_for` declaration to stand in for either of the other forms is that `@article` is a resource corresponding to a full set of RESTful routes, and Rails is able to infer which URI and method to use. For more information about this use of form_for, see Resource-oriented style.

Now, let's update the `app/views/articles/new.html.erb` view to use this new partial, rewriting it completely:

```
<h1>New article</h1>

<%= render 'form' %>

<%= link_to 'Back', articles_path %>
Then do the same for the app/views/articles/edit.html.erb view:

<h1>Edit article</h1>

<%= render 'form' %>

<%= link_to 'Back', articles_path %>
```

13 Deleting Articles

We're now ready to cover the "D" part of CRUD, deleting articles from the database. Following the REST convention, the route for deleting articles as per output of bin/rails routes is:

```
DELETE /articles/:id(.:format)      articles#destroy
```

The delete routing method should be used for routes that destroy resources. If this was left as a typical get route, it could be possible for people to craft malicious URLs like this:

```
<a href='http://example.com/articles/1/destroy'>look at this cat!</a>
```

We use the delete method for destroying resources, and this route is mapped to the destroy action inside `app/controllers/articles_controller.rb`, which doesn't exist yet. The destroy method is generally the last CRUD action in the controller, and like the other public CRUD actions, it must be placed before any private or protected methods. Let's add it:

```
def destroy
  @article = Article.find(params[:id])
  @article.destroy

  redirect_to articles_path
end
```

The complete ArticlesController in the `app/controllers/articles_controller.rb` file should now look like this:

```ruby
class ArticlesController < ApplicationController
  def index
    @articles = Article.all
  end

  def show
    @article = Article.find(params[:id])
  end

  def new
    @article = Article.new
  end

  def edit
    @article = Article.find(params[:id])
  end

  def create
    @article = Article.new(article_params)

    if @article.save
      redirect_to @article
    else
      render 'new'
    end
  end

  def update
    @article = Article.find(params[:id])

    if @article.update(article_params)
      redirect_to @article
    else
      render 'edit'
    end
  end

  def destroy
    @article = Article.find(params[:id])
    @article.destroy

    redirect_to articles_path
  end

  private
    def article_params
      params.require(:article).permit(:title, :text)
    end
end
```

You can call destroy on `Active Record` objects when you want to delete them from the database. Note that we don't need to add a view for this action since we're redirecting to the index action.

Finally, add a 'Destroy' link to your index action template (`app/views/articles/index.html.erb`) to wrap everything together.

```
<h1>Listing Articles</h1>
<%= link_to 'New article', new_article_path %>
<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
    <th colspan="3"></th>
  </tr>

  <% @articles.each do |article| %>
    <tr>
      <td><%= article.title %></td>
      <td><%= article.text %></td>
      <td><%= link_to 'Show', article_path(article) %></td>
      <td><%= link_to 'Edit', edit_article_path(article) %></td>
      <td><%= link_to 'Destroy', article_path(article),
              method: :delete,
              data: { confirm: 'Are you sure?' } %></td>
    </tr>
  <% end %>
</table>
```

Here we're using `link_to` in a different way. We pass the named route as the second argument, and then the options as another argument. The `method: :delete` and `data: { confirm: 'Are you sure?' }` options are used as HTML5 attributes so that when the link is clicked, Rails will first show a confirm dialog to the user, and then submit the link with method delete. This is done via the JavaScript file jquery_ujs which is automatically included in your application's layout (`app/views/layouts/application.html.erb`) when you generated the application. Without this file, the confirmation dialog box won't appear.

> Learn more about jQuery Unobtrusive Adapter (jQuery UJS) on Working With JavaScript in Rails guide.

Congratulations, you can now create, show, list, update and destroy articles.

> In general, Rails encourages using resources objects instead of declaring routes manually. For more information about routing, see Rails Routing from the Outside In.

Model Advanced

Table of Contents

1. Migrations

1.1 Creating a Migration

1.1.1. Creating a Standalone Migration

Migrations are stored as files in the `db/migrate` directory, one for each migration class. The name of the file is of the form `YYYYMMDDHHMMSS_create_products.rb`, that is to say a UTC timestamp identifying the migration followed by an underscore followed by the name of the migration. The name of the migration class (CamelCased version) should match the latter part of the file name. For example `20080906120000_create_products.rb` should define class `CreateProducts` and `20080906120001_add_details_to_products.rb` should define `AddDetailsToProducts`. Rails uses this timestamp to determine which migration should be run and in what order, so if you're copying a migration from another application or generate a file yourself, be aware of its position in the order.

Of course, calculating timestamps is no fun, so Active Record provides a generator to handle making it for you:

```
$ bin/rails generate migration AddPartNumberToProducts
```

This will create an empty but appropriately named migration:

```
class AddPartNumberToProducts < ActiveRecord::Migration[5.0]
  def change
  end
end
```

If the migration name is of the form "AddXXXToYYY" or "RemoveXXXFromYYY" and is followed by a list of column names and types then a migration containing the appropriate `add_column` and `remove_column` statements will be created.

```
$ bin/rails generate migration AddPartNumberToProducts part_number:string
```

will generate

```
class AddPartNumberToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
  end
end
```

If you'd like to add an index on the new column, you can do that as well:

$ bin/rails generate migration AddPartNumberToProducts part_number:string:index will generate

```
class AddPartNumberToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
    add_index :products, :part_number
  end
end
```

Similarly, you can generate a migration to remove a column from the command line:

```
$ bin/rails generate migration RemovePartNumberFromProducts part_number:string
```

generates

```
class RemovePartNumberFromProducts < ActiveRecord::Migration[5.0]
  def change
    remove_column :products, :part_number, :string
  end
end
```

You are not limited to one magically generated column. For example:

```
$ bin/rails generate migration AddDetailsToProducts part_number:string price:decimal
```

generates

```
class AddDetailsToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
    add_column :products, :price, :decimal
  end
end
```

If the migration name is of the form "CreateXXX" and is followed by a list of column names and types then a migration creating the table XXX with the columns listed will be generated. For example:

```
$ bin/rails generate migration CreateProducts name:string part_number:string
```

generates

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :products do |t|
      t.string :name
      t.string :part_number
    end
  end
end
```

As always, what has been generated for you is just a starting point. You can add or remove from it as you see fit by editing the `db/migrate/YYYYMMDDHHMMSS_add_details_to_products.rb` file.

Also, the generator accepts column type as references(also available as belongs_to). For instance:

```
$ bin/rails generate migration AddUserRefToProducts user:references
```

generates

```
class AddUserRefToProducts < ActiveRecord::Migration[5.0]
  def change
    add_reference :products, :user, index: true, foreign_key: true
  end
end
```

This migration will create a user_id column and appropriate index. For more add_reference options, visit the API documentation.

There is also a generator which will produce join tables if JoinTable is part of the name:

```
$ bin/rails g migration CreateJoinTableCustomerProduct customer product
```

will produce the following migration:

```
class CreateJoinTableCustomerProduct < ActiveRecord::Migration[5.0]
  def change
    create_join_table :customers, :products do |t|
      # t.index [:customer_id, :product_id]
      # t.index [:product_id, :customer_id]
    end
  end
end
```

1.1.2. Model Generators

The model and scaffold generators will create migrations appropriate for adding a new model. This migration will already contain instructions for creating the relevant table. If you tell Rails what columns you want, then statements for adding these columns will also be created. For example, running:

```
$ bin/rails generate model Product name:string description:text
```

will create a migration that looks like this

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

You can append as many column name/type pairs as you want.

1.1.3. Passing Modifiers

Some commonly used type modifiers can be passed directly on the command line. They are enclosed by curly braces and follow the field type:

For instance, running:

```
$ bin/rails generate migration AddDetailsToProducts 'price:decimal{5,2}' supplier:refe
rences{polymorphic}
```

will produce a migration that looks like this

```
class AddDetailsToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :price, :decimal, precision: 5, scale: 2
    add_reference :products, :supplier, polymorphic: true, index: true
  end
end
```

Have a look at the generators help output for further details.

1.2. Writing a Migration

Once you have created your migration using one of the generators it's time to get to work!

1.2.1 Creating a Table

The create_table method is one of the most fundamental, but most of the time, will be generated for you from using a model or scaffold generator. A typical use would be

```
create_table :products do |t|
  t.string :name
end
```

which creates a products table with a column called name (and as discussed below, an implicit id column).

By default, create_table will create a primary key called id. You can change the name of the primary key with the `:primary_key` option (don't forget to update the corresponding model) or, if you don't want a primary key at all, you can pass the option id: false. If you need to pass database specific options you can place an SQL fragment in the :options option. For example:

```
create_table :products, options: "ENGINE=BLACKHOLE" do |t|
  t.string :name, null: false
end
```

will append ENGINE=BLACKHOLE to the SQL statement used to create the table (when using MySQL or MariaDB, the default is ENGINE=InnoDB).

Also you can pass the `:comment` option with any description for the table that will be stored in database itself and can be viewed with database administration tools, such as MySQL Workbench or PgAdmin III. It's highly recommended to specify comments in migrations for applications with large databases as it helps people to understand data model and generate documentation. Currently only the MySQL and PostgreSQL adapters support comments.

1.2.2 Creating a Join Table

The migration method create_join_table creates an HABTM (has and belongs to many) join table. A typical use would be:

```
create_join_table :products, :categories
```

which creates a categories_products table with two columns called category_id and product_id. These columns have the option :null set to false by default. This can be overridden by specifying the `:column_options` option:

```
create_join_table :products, :categories, column_options: { null: true }
```

By default, the name of the join table comes from the union of the first two arguments provided to create_join_table, in alphabetical order. To customize the name of the table, provide a `:table_name` option:

```
create_join_table :products, :categories, table_name: :categorization
```

creates a categorization table.

create_join_table also accepts a block, which you can use to add indices (which are not created by default) or additional columns:

```
create_join_table :products, :categories do |t|
  t.index :product_id
  t.index :category_id
end
```

1.2.3. Changing Tables

A close cousin of create_table is change_table, used for changing existing tables. It is used in a similar fashion to create_table but the object yielded to the block knows more tricks. For example:

```
change_table :products do |t|
  t.remove :description, :name
  t.string :part_number
  t.index :part_number
  t.rename :upccode, :upc_code
end
```

removes the description and name columns, creates a part_number string column and adds an index on it. Finally it renames the upccode column.

1.2.4. Changing Columns

Like the remove_column and add_column Rails provides the change_column migration method.

```
change_column :products, :part_number, :text
```

This changes the column part_number on products table to be a `:text field`. Note that change_column command is irreversible.

Besides change_column, the change_column_null and change_column_default methods are used specifically to change a not null constraint and default values of a column.

```
change_column_null :products, :name, false
change_column_default :products, :approved, from: true, to: false
```

This sets `:name field` on products to a NOT NULL column and the default value of the :approved field from true to false.

Note: You could also write the above change_column_default migration as change_column_default :products, :approved, false, but unlike the previous example, this would make your migration irreversible.

1.2.5. Column Modifiers

Column modifiers can be applied when creating or changing a column:

- limit Sets the maximum size of the string/text/binary/integer fields.
- precision Defines the precision for the decimal fields, representing the total number of digits in the number.
- scale Defines the scale for the decimal fields, representing the number of digits after the decimal point.
- polymorphic Adds a type column for belongs_to associations.
- null Allows or disallows NULL values in the column.

- default Allows to set a default value on the column. Note that if you are using a dynamic value (such as a date), the default will only be calculated the first time (i.e. on the date the migration is applied).
- index Adds an index for the column.
- comment Adds a comment for the column. Some adapters may support additional options; see the adapter specific API docs for further information.

1.2.6. Foreign Keys

While it's not required you might want to add foreign key constraints to guarantee referential integrity.

```
add_foreign_key :articles, :authors
```

This adds a new foreign key to the `author_id` column of the `articles` table. The key references the id column of the authors table. If the column names can not be derived from the table names, you can use the `:column` and `:primary_key` options.

Rails will generate a name for every foreign key starting with fk*rails* followed by 10 characters which are deterministically generated from the from_table and column. There is a :name option to specify a different name if needed.

> Active Record only supports single column foreign keys. execute and structure.sql are required to use composite foreign keys. See Schema Dumping and You.

Removing a foreign key is easy as well:

```
# let Active Record figure out the column name
remove_foreign_key :accounts, :branches

# remove foreign key for a specific column
remove_foreign_key :accounts, column: :owner_id

# remove foreign key by name
remove_foreign_key :accounts, name: :special_fk_name
```

2. Validations

2.1 Validations Overview

Here's an example of a very simple validation:

```
class Person < ApplicationRecord
  validates :name, presence: true
end

Person.create(name: "John Doe").valid? # => true
Person.create(name: nil).valid? # => false
```

As you can see, our validation lets us know that our Person is not valid without a name attribute. The second Person will not be persisted to the database.

Before we dig into more details, let's talk about how validations fit into the big picture of your application.

2.1.1 Why Use Validations?

Validations are used to ensure that only valid data is saved into your database. For example, it may be important to your application to ensure that every user provides a valid email address and mailing address. Model-level validations are the best way to ensure that only valid data is saved into your database. They are database agnostic, cannot be bypassed by end users, and are convenient to test and maintain. Rails makes them easy to use, provides built-in helpers for common needs, and allows you to create your own validation methods as well.

There are several other ways to validate data before it is saved into your database, including native database constraints, client-side validations and controller-level validations. Here's a summary of the pros and cons:

- Database constraints and/or stored procedures make the validation mechanisms database-dependent and can make testing and maintenance more difficult. However, if your database is used by other applications, it may be a good idea to use some constraints at the database level. Additionally, database-level validations can safely handle some things (such as uniqueness in heavily-used tables) that can be difficult to implement otherwise.
- Client-side validations can be useful, but are generally unreliable if used alone. If they are implemented using JavaScript, they may be bypassed if JavaScript is turned off in the user's browser. However, if combined with other techniques, client-side validation can be a convenient way to provide users with immediate feedback as they use your site.
- Controller-level validations can be tempting to use, but often become unwieldy and difficult to test and maintain. Whenever possible, it's a good idea to keep your controllers skinny, as it will make your application a pleasure to work with in the long run.

Choose these in certain, specific cases. It's the opinion of the Rails team that model-level validations are the most appropriate in most circumstances.

2.1.2 When Does Validation Happen?

There are two kinds of Active Record objects: those that correspond to a row inside your database and those that do not. When you create a fresh object, for example using the new method, that object does not belong to the database yet. Once you call save upon that object it will be saved into the appropriate database table. Active Record uses the new_record? instance method to determine whether an object is already in the database or not. Consider the following simple `Active Record` class:

```
class Person < ApplicationRecord
end
```

We can see how it works by looking at some rails console output:

```
$ bin/rails console
>> p = Person.new(name: "John Doe")
=> #<Person id: nil, name: "John Doe", created_at: nil, updated_at: nil>
>> p.new_record?
=> true
>> p.save
=> true
>> p.new_record?
=> false
```

Creating and saving a new record will send an SQL INSERT operation to the database. Updating an existing record will send an SQL UPDATE operation instead. Validations are typically run before these commands are sent to the database. If any validations fail, the object will be marked as invalid and Active Record will not perform the INSERT or UPDATE operation. This avoids storing an invalid object in the database. You can choose to have specific validations run when an object is created, saved, or updated.

> There are many ways to change the state of an object in the database. Some methods will trigger validations, but some will not. This means that it's possible to save an object in the database in an invalid state if you aren't careful.

The following methods trigger validations, and will save the object to the database only if the object is valid:

```
- create
- create!
- save
- save!
- update
- update!
```

The bang versions (e.g. save!) raise an exception if the record is invalid. The non-bang versions don't: save and update return false, and create just returns the object.

2.1.3 Skipping Validations

The following methods skip validations, and will save the object to the database regardless of its validity. They should be used with caution.

```
- decrement!
- decrement_counter
- increment!
- increment_counter
- toggle!
- touch
- update_all
- update_attribute
- update_column
- update_columns
- update_counters
```

Note that save also has the ability to skip validations if passed validate: false as an argument. This technique should be used with caution.

- save(validate: false)

2.1.4 valid? and invalid?

Before saving an Active Record object, Rails runs your validations. If these validations produce any errors, Rails does not save the object.

You can also run these validations on your own. valid? triggers your validations and returns true if no errors were found in the object, and false otherwise. As you saw above:

```ruby
class Person < ApplicationRecord
  validates :name, presence: true
end

Person.create(name: "John Doe").valid? # => true
Person.create(name: nil).valid? # => false
```

After Active Record has performed validations, any errors found can be accessed through the errors.messages instance method, which returns a collection of errors. By definition, an object is valid if this collection is empty after running validations.

Note that an object instantiated with new will not report errors even if it's technically invalid, because validations are automatically run only when the object is saved, such as with the create or save methods.

```
class Person < ApplicationRecord
  validates :name, presence: true
end

>> p = Person.new
# => #<Person id: nil, name: nil>
>> p.errors.messages
# => {}

>> p.valid?
# => false
>> p.errors.messages
# => {name:["can't be blank"]}

>> p = Person.create
# => #<Person id: nil, name: nil>
>> p.errors.messages
# => {name:["can't be blank"]}

>> p.save
# => false

>> p.save!
# => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank

>> Person.create!
# => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank
```

`invalid?` is simply the inverse of valid?. It triggers your validations, returning true if any errors were found in the object, and false otherwise.

2.1.5 errors[]

To verify whether or not a particular attribute of an object is valid, you can use errors[:attribute]. It returns an array of all the errors for :attribute. If there are no errors on the specified attribute, an empty array is returned.

This method is only useful after validations have been run, because it only inspects the errors collection and does not trigger validations itself. It's different from the ActiveRecord::Base#invalid? method explained above because it doesn't verify the validity

of the object as a whole. It only checks to see whether there are errors found on an individual attribute of the object.

```
class Person < ApplicationRecord
  validates :name, presence: true
end

>> Person.new.errors[:name].any? # => false
>> Person.create.errors[:name].any? # => true
```

We'll cover validation errors in greater depth in the Working with Validation Errors section.

2.1.6 errors.details

To check which validations failed on an invalid attribute, you can use errors.details[:attribute]. It returns an array of hashes with an :error key to get the symbol of the validator:

```
class Person < ApplicationRecord
  validates :name, presence: true
end

>> person = Person.new
>> person.valid?
>> person.errors.details[:name] # => [{error: :blank}]
```

Using details with custom validators is covered in the Working with Validation Errors section.

2.2. Validation Helpers

Active Record offers many pre-defined validation helpers that you can use directly inside your class definitions. These helpers provide common validation rules. Every time a validation fails, an error message is added to the object's errors collection, and this message is associated with the attribute being validated.

Each helper accepts an arbitrary number of attribute names, so with a single line of code you can add the same kind of validation to several attributes.

All of them accept the :on and :message options, which define when the validation should be run and what message should be added to the errors collection if it fails, respectively. The :on option takes one of the values :create or :update. There is a default error message for each one of the validation helpers. These messages are used when the :message option isn't specified. Let's take a look at each one of the available helpers.

2.2.1 acceptance

This method validates that a checkbox on the user interface was checked when a form was submitted. This is typically used when the user needs to agree to your application's terms of service, confirm that some text is read, or any similar concept.

```
class Person < ApplicationRecord
  validates :terms_of_service, acceptance: true
end
```

This check is performed only if terms_of_service is not nil. The default error message for this helper is "must be accepted". You can also pass custom message via the message option.

```
class Person < ApplicationRecord
  validates :terms_of_service, acceptance: true, message: 'must be abided'
end
```

It can also receive an :accept option, which determines the allowed values that will be considered as accepted. It defaults to ['1', true] and can be easily changed.

```
class Person < ApplicationRecord
  validates :terms_of_service, acceptance: { accept: 'yes' }
  validates :eula, acceptance: { accept: ['TRUE', 'accepted'] }
end
```

This validation is very specific to web applications and this 'acceptance' does not need to be recorded anywhere in your database. If you don't have a field for it, the helper will just create a virtual attribute. If the field does exist in your database, the accept option must be set to or include true or else the validation will not run.

2.2.2. validates_associated

You should use this helper when your model has associations with other models and they also need to be validated. When you try to save your object, valid? will be called upon each one of the associated objects.

```
class Library < ApplicationRecord
  has_many :books
  validates_associated :books
end
```

This validation will work with all of the association types.

Don't use validates_associated on both ends of your associations. They would call each other in an infinite loop.

The default error message for validates_associated is "is invalid". Note that each associated object will contain its own errors collection; errors do not bubble up to the calling model.

2.2.3. confirmation

You should use this helper when you have two text fields that should receive exactly the same content. For example, you may want to confirm an email address or a password. This validation creates a virtual attribute whose name is the name of the field that has to be confirmed with "_confirmation" appended.

```
class Person < ApplicationRecord
  validates :email, confirmation: true
end
```

In your view template you could use something like

```
<%= text_field :person, :email %>
<%= text_field :person, :email_confirmation %>
```

This check is performed only if email_confirmation is not nil. To require confirmation, make sure to add a presence check for the confirmation attribute (we'll take a look at presence later on in this guide):

```
class Person < ApplicationRecord
  validates :email, confirmation: true
  validates :email_confirmation, presence: true
end
```

There is also a `:case_sensitive` option that you can use to define whether the confirmation constraint will be case sensitive or not. This option defaults to true.

```
class Person < ApplicationRecord
  validates :email, confirmation: { case_sensitive: false }
end
```

The default error message for this helper is "doesn't match confirmation".

2.2.4 exclusion

This helper validates that the attributes' values are not included in a given set. In fact, this set can be any enumerable object.

```
class Account < ApplicationRecord
  validates :subdomain, exclusion: { in: %w(www us ca jp),
    message: "%{value} is reserved." }
end
```

The exclusion helper has an option :in that receives the set of values that will not be accepted for the validated attributes. The :in option has an alias called :within that you can use for the same purpose, if you'd like to. This example uses the :message option to show how you can include the attribute's value.

The default error message is "is reserved".

2.2.5 format

This helper validates the attributes' values by testing whether they match a given regular expression, which is specified using the :with option.

```
class Product < ApplicationRecord
  validates :legacy_code, format: { with: /\A[a-zA-Z]+\z/,
    message: "only allows letters" }
end
```

Alternatively, you can require that the specified attribute does not match the regular expression by using the :without option.

The default error message is "is invalid".

2.2.6 inclusion

This helper validates that the attributes' values are included in a given set. In fact, this set can be any enumerable object.

```
class Coffee < ApplicationRecord
  validates :size, inclusion: { in: %w(small medium large),
    message: "%{value} is not a valid size" }
end
```

The inclusion helper has an option :in that receives the set of values that will be accepted. The :in option has an alias called :within that you can use for the same purpose, if you'd like to. The previous example uses the :message option to show how you can include the attribute's value.

The default error message for this helper is "is not included in the list".

2.2.7 length

This helper validates the length of the attributes' values. It provides a variety of options, so you can specify length constraints in different ways:

```
class Person < ApplicationRecord
  validates :name, length: { minimum: 2 }
  validates :bio, length: { maximum: 500 }
  validates :password, length: { in: 6..20 }
  validates :registration_number, length: { is: 6 }
end
```

The possible length constraint options are:

- :minimum - The attribute cannot have less than the specified length.
- :maximum - The attribute cannot have more than the specified length.
- :in (or :within) - The attribute length must be included in a given interval. The value for this option must be a range.
- :is - The attribute length must be equal to the given value.

The default error messages depend on the type of length validation being performed. You can personalize these messages using the :wrong_length, `:too_long`, and `:too_short` options and %{count} as a placeholder for the number corresponding to the length constraint being used. You can still use the :message option to specify an error message.

```
class Person < ApplicationRecord
  validates :bio, length: { maximum: 1000,
    too_long: "%{count} characters is the maximum allowed" }
end
```

Note that the default error messages are plural (e.g., "is too short (minimum is %{count} characters)"). For this reason, when :minimum is 1 you should provide a personalized message or use presence: true instead. When :in or :within have a lower limit of 1, you should either provide a personalized message or call presence prior to length.

2.2.8 numericality

This helper validates that your attributes have only numeric values. By default, it will match an optional sign followed by an integral or floating point number. To specify that only integral numbers are allowed set :only_integer to true.

If you set :only_integer to true, then it will use the

```
/\A[+-]?\d+\z/
```

regular expression to validate the attribute's value. Otherwise, it will try to convert the value to a number using Float.

Note that the regular expression above allows a trailing newline character.

```
class Player < ApplicationRecord
  validates :points, numericality: true
  validates :games_played, numericality: { only_integer: true }
end
```

Besides `:only_integer`, this helper also accepts the following options to add constraints to acceptable values:

- :greater_than - Specifies the value must be greater than the supplied value. The default error message for this option is "must be greater than %{count}".
- :greater_than_or_equal_to - Specifies the value must be greater than or equal to the supplied value. The default error message for this option is "must be greater than or equal to %{count}".
- :equal_to - Specifies the value must be equal to the supplied value. The default error message for this option is "must be equal to %{count}".
- :less_than - Specifies the value must be less than the supplied value. The default error message for this option is "must be less than %{count}".
- :less_than_or_equal_to - Specifies the value must be less than or equal to the supplied value. The default error message for this option is "must be less than or equal to %{count}".
- :other_than - Specifies the value must be other than the supplied value. The default error message for this option is "must be other than %{count}".
- :odd - Specifies the value must be an odd number if set to true. The default error message for this option is "must be odd".
- :even - Specifies the value must be an even number if set to true. The default error message for this option is "must be even".

By default, numericality doesn't allow nil values. You can use allow_nil: true option to permit it.

The default error message is "is not a number".

2.2.9 presence

This helper validates that the specified attributes are not empty. It uses the blank? method to check if the value is either nil or a blank string, that is, a string that is either empty or consists of whitespace.

```
class Person < ApplicationRecord
  validates :name, :login, :email, presence: true
end
```

If you want to be sure that an association is present, you'll need to test whether the associated object itself is present, and not the foreign key used to map the association.

```
class LineItem < ApplicationRecord
  belongs_to :order
  validates :order, presence: true
end
```

In order to validate associated records whose presence is required, you must specify the :inverse_of option for the association:

```
class Order < ApplicationRecord
  has_many :line_items, inverse_of: :order
end
```

If you validate the presence of an object associated via a has_one or has_many relationship, it will check that the object is neither blank? nor marked_for_destruction?.

Since false.blank? is true, if you want to validate the presence of a boolean field you should use one of the following validations:

```
validates :boolean_field_name, inclusion: { in: [true, false] }
validates :boolean_field_name, exclusion: { in: [nil] }
```

By using one of these validations, you will ensure the value will NOT be nil which would result in a NULL value in most cases.

2.2.10 absence

This helper validates that the specified attributes are absent. It uses the present? method to check if the value is not either nil or a blank string, that is, a string that is either empty or consists of whitespace.

```
class Person < ApplicationRecord
  validates :name, :login, :email, absence: true
end
```

If you want to be sure that an association is absent, you'll need to test whether the associated object itself is absent, and not the foreign key used to map the association.

```
class LineItem < ApplicationRecord
  belongs_to :order
  validates :order, absence: true
end
```

In order to validate associated records whose absence is required, you must specify the :inverse_of option for the association:

```
class Order < ApplicationRecord
  has_many :line_items, inverse_of: :order
end
```

If you validate the absence of an object associated via a has_one or has_many relationship, it will check that the object is neither present? nor marked_for_destruction?.

Since false.present? is false, if you want to validate the absence of a boolean field you should use validates `:field_name`, exclusion: { in: [true, false] }.

The default error message is "must be blank".

2.2.11 uniqueness

This helper validates that the attribute's value is unique right before the object gets saved. It does not create a uniqueness constraint in the database, so it may happen that two different database connections create two records with the same value for a column that you intend to be unique. To avoid that, you must create a unique index on that column in your database.

```
class Account < ApplicationRecord
  validates :email, uniqueness: true
end
```

The validation happens by performing an SQL query into the model's table, searching for an existing record with the same value in that attribute.

There is a :scope option that you can use to specify one or more attributes that are used to limit the uniqueness check:

```
class Holiday < ApplicationRecord
  validates :name, uniqueness: { scope: :year,
    message: "should happen once per year" }
end
```

Should you wish to create a database constraint to prevent possible violations of a uniqueness validation using the `:scope` option, you must create a unique index on both columns in your database. See the MySQL manual for more details about multiple column indexes or the PostgreSQL manual for examples of unique constraints that refer to a group of columns.

There is also a :case_sensitive option that you can use to define whether the uniqueness constraint will be case sensitive or not. This option defaults to true.

```
class Person < ApplicationRecord
  validates :name, uniqueness: { case_sensitive: false }
end
```

Note that some databases are configured to perform case-insensitive searches anyway.

The default error message is "has already been taken".

2.2.12 validates_with

This helper passes the record to a separate class for validation.

```
class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if record.first_name == "Evil"
      record.errors[:base] << "This person is evil"
    end
  end
end

class Person < ApplicationRecord
  validates_with GoodnessValidator
end
```

Errors added to record.errors[:base] relate to the state of the record as a whole, and not to a specific attribute.

The validates_with helper takes a class, or a list of classes to use for validation. There is no default error message for validates_with. You must manually add errors to the record's errors collection in the validator class.

To implement the validate method, you must have a record parameter defined, which is the record to be validated.

Like all other validations, validates_with takes the :if, `:unless` and `:on` options. If you pass any other options, it will send those options to the validator class as options:

```
class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if options[:fields].any?{|field| record.send(field) == "Evil" }
      record.errors[:base] << "This person is evil"
    end
  end
end

class Person < ApplicationRecord
  validates_with GoodnessValidator, fields: [:first_name, :last_name]
end
```

Note that the validator will be initialized only once for the whole application life cycle, and not on each validation run, so be careful about using instance variables inside it.

If your validator is complex enough that you want instance variables, you can easily use a plain old Ruby object instead:

```
class Person < ApplicationRecord
  validate do |person|
    GoodnessValidator.new(person).validate
  end
end

class GoodnessValidator
  def initialize(person)
    @person = person
  end

  def validate
    if some_complex_condition_involving_ivars_and_private_methods?
      @person.errors[:base] << "This person is evil"
    end
  end

  # ...
end
```

2.2.13 validates_each

This helper validates attributes against a block. It doesn't have a predefined validation function. You should create one using a block, and every attribute passed to validates_each will be tested against it. In the following example, we don't want names and surnames to begin with lower case.

```
class Person < ApplicationRecord
  validates_each :name, :surname do |record, attr, value|
    record.errors.add(attr, 'must start with upper case') if value =~ /\A[[:lower:]]/
  end
end
```

The block receives the record, the attribute's name and the attribute's value. You can do anything you like to check for valid data within the block. If your validation fails, you should add an error message to the model, therefore making it invalid.

3. Associations

3.1 Why Associations?

In Rails, an association is a connection between two Active Record models. Why do we need associations between models? Because they make common operations simpler and easier in your code. For example, consider a simple Rails application that includes a model for authors and a model for books. Each author can have many books. Without associations, the model declarations would look like this:

```
class Author < ApplicationRecord
end

class Book < ApplicationRecord
end
```

Now, suppose we wanted to add a new book for an existing author. We'd need to do something like this:

```
@book = Book.create(published_at: Time.now, author_id: @author.id)
```

Or consider deleting an author, and ensuring that all of its books get deleted as well:

```
@books = Book.where(author_id: @author.id)
@books.each do |book|
  book.destroy
end
@author.destroy
```

With Active Record associations, we can streamline these - and other - operations by declaratively telling Rails that there is a connection between the two models. Here's the revised code for setting up authors and books:

```
class Author < ApplicationRecord
  has_many :books, dependent: :destroy
end

class Book < ApplicationRecord
  belongs_to :author
end
```

With this change, creating a new book for a particular author is easier:

```
@book = @author.books.create(published_at: Time.now)
```

Deleting an author and all of its books is much easier:

```
@author.destroy
```

To learn more about the different types of associations, read the next section of this guide. That's followed by some tips and tricks for working with associations, and then by a complete reference to the methods and options for associations in Rails.

3.2 The Types of Associations

Rails supports six types of associations:

```
belongs_to
has_one
has_many
has_many :through
has_one :through
has_and_belongs_to_many
```

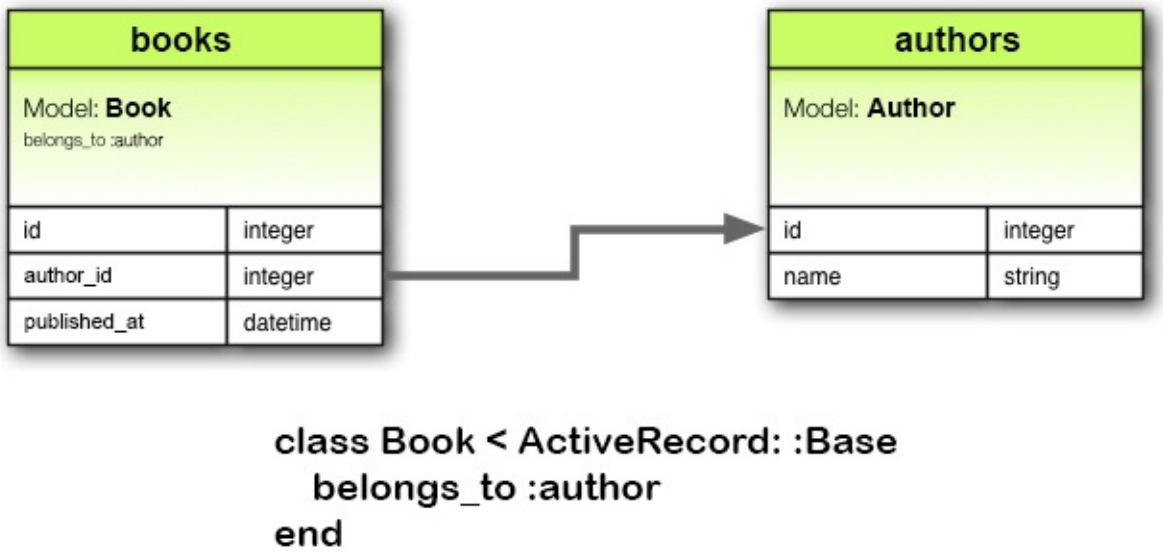
Associations are implemented using macro-style calls, so that you can declaratively add features to your models. For example, by declaring that one model belongs_to another, you instruct Rails to maintain Primary Key-Foreign Key information between instances of the two models, and you also get a number of utility methods added to your model.

In the remainder of this guide, you'll learn how to declare and use the various forms of associations. But first, a quick introduction to the situations where each association type is appropriate.

3.2.1 The belongs_to Association

A belongs_to association sets up a one-to-one connection with another model, such that each instance of the declaring model "belongs to" one instance of the other model. For example, if your application includes authors and books, and each book can be assigned to exactly one author, you'd declare the book model this way:

```
class Book < ApplicationRecord
  belongs_to :author
end
```



> belongs_to associations must use the singular term. If you used the pluralized form in the above example for the author association in the Book model, you would be told that there was an "uninitialized constant Book::Authors". This is because Rails automatically infers the class name from the association name. If the association name is wrongly pluralized, then the inferred class will be wrongly pluralized too.

The corresponding migration might look like this:

```
class CreateBooks < ActiveRecord::Migration[5.0]
  def change
    create_table :authors do |t|
      t.string :name
      t.timestamps
    end

    create_table :books do |t|
      t.belongs_to :author, index: true
      t.datetime :published_at
      t.timestamps
    end
  end
end
```

2.2 The has_one Association A has_one association also sets up a one-to-one connection with another model, but with somewhat different semantics (and consequences). This association indicates that each instance of a model contains or possesses one instance of another model. For example, if each supplier in your application has only one account, you'd declare the supplier model like this:

class Supplier < ApplicationRecord has_one :account end has_one Association Diagram

The corresponding migration might look like this:

class CreateSuppliers < ActiveRecord::Migration[5.0] def change create_table :suppliers do |t| t.string :name t.timestamps end

```
create_table :accounts do |t|
  t.belongs_to :supplier, index: true
  t.string :account_number
  t.timestamps
end
```

end end Depending on the use case, you might also need to create a unique index and/or a foreign key constraint on the supplier column for the accounts table. In this case, the column definition might look like this:

create_table :accounts do |t| t.belongs_to :supplier, index: true, unique: true, foreign_key: true

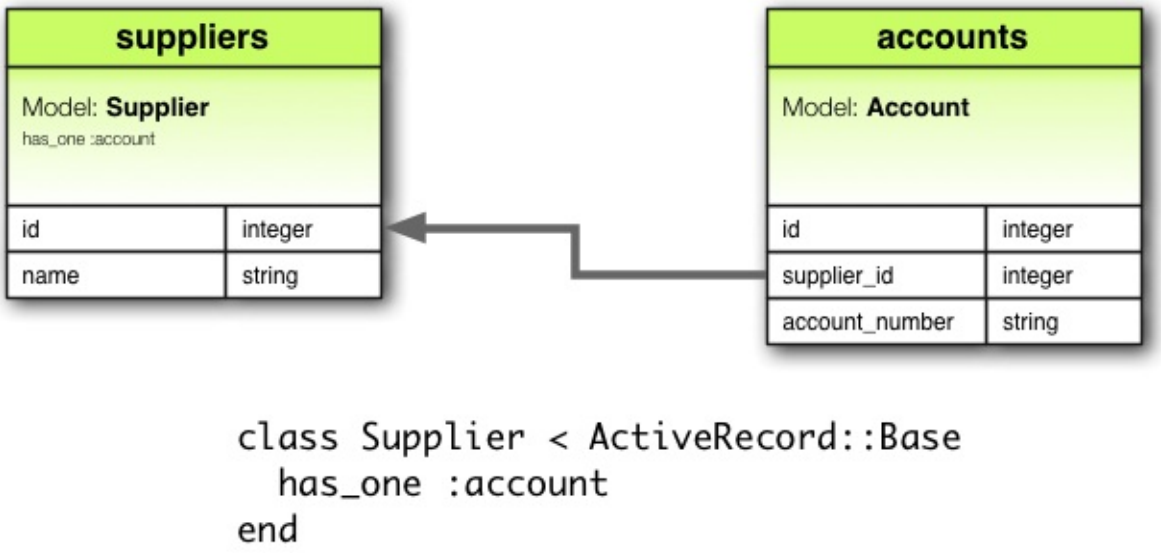
▪▪▪

end 2.3 The has_many Association

A has_many association indicates a one-to-many connection with another model. You'll often find this association on the "other side" of a belongs_to association. This association indicates that each instance of the model has zero or more instances of another model. For example, in an application containing authors and books, the author model could be declared like this:

```
class Author < ApplicationRecord
  has_many :books
end
```

The name of the other model is pluralized when declaring a has_many association.



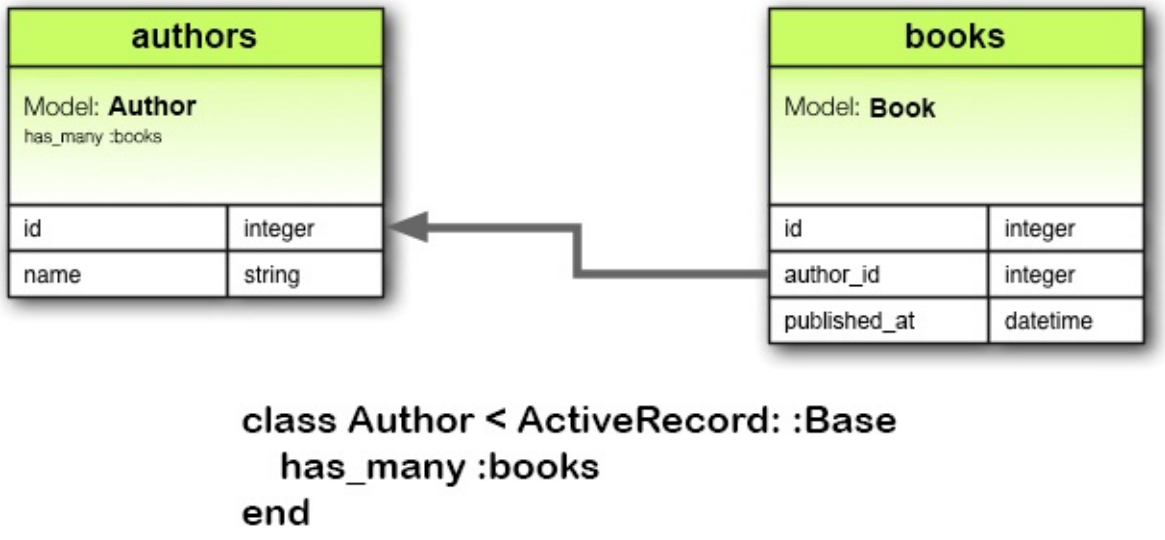
The corresponding migration might look like this:

```
class CreateAuthors < ActiveRecord::Migration[5.0]
  def change
    create_table :authors do |t|
      t.string :name
      t.timestamps
    end

    create_table :books do |t|
      t.belongs_to :author, index: true
      t.datetime :published_at
      t.timestamps
    end
  end
end
```

3.2.4 The has_many :through Association

A has_many :through association is often used to set up a many-to-many connection with another model. This association indicates that the declaring model can be matched with zero or more instances of another model by proceeding through a third model. For example, consider a medical practice where patients make appointments to see physicians. The relevant association declarations could look like this:

```
class Physician < ApplicationRecord
  has_many :appointments
  has_many :patients, through: :appointments
end

class Appointment < ApplicationRecord
  belongs_to :physician
  belongs_to :patient
end

class Patient < ApplicationRecord
  has_many :appointments
  has_many :physicians, through: :appointments
end
```



The corresponding migration might look like this:

```
class CreateAppointments < ActiveRecord::Migration[5.0]
  def change
    create_table :physicians do |t|
      t.string :name
      t.timestamps
    end

    create_table :patients do |t|
      t.string :name
      t.timestamps
    end

    create_table :appointments do |t|
      t.belongs_to :physician, index: true
      t.belongs_to :patient, index: true
      t.datetime :appointment_date
      t.timestamps
    end
  end
end
```

The collection of join models can be managed via the has_many association methods. For example, if you assign:

```
physician.patients = patients
```

Then new join models are automatically created for the newly associated objects. If some that existed previously are now missing, then their join rows are automatically deleted.

Automatic deletion of join models is direct, no destroy callbacks are triggered.

The has_many :through association is also useful for setting up "shortcuts" through nested has_many associations. For example, if a document has many sections, and a section has many paragraphs, you may sometimes want to get a simple collection of all paragraphs in the document. You could set that up this way:

```
class Document < ApplicationRecord
  has_many :sections
  has_many :paragraphs, through: :sections
end

class Section < ApplicationRecord
  belongs_to :document
  has_many :paragraphs
end

class Paragraph < ApplicationRecord
  belongs_to :section
end
```

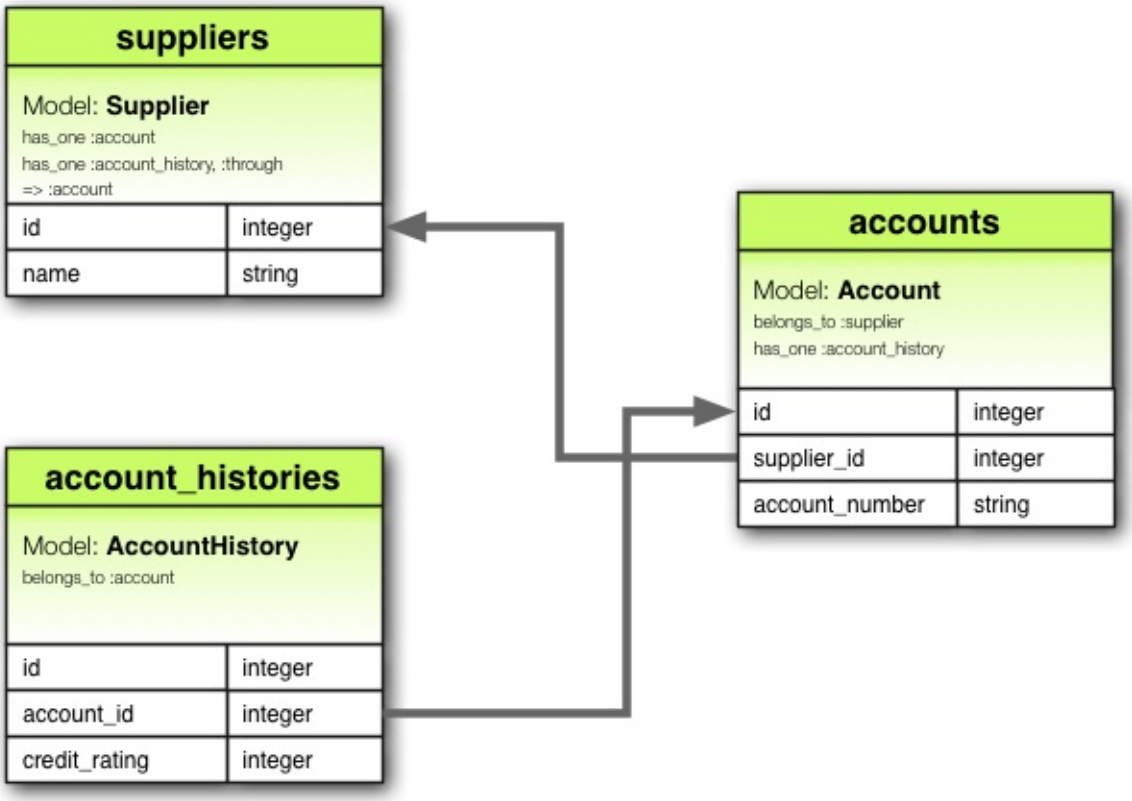
With through: :sections specified, Rails will now understand:

```
@document.paragraphs
```

3.2.5 The has_one :through Association

A has_one :through association sets up a one-to-one connection with another model. This association indicates that the declaring model can be matched with one instance of another model by proceeding through a third model. For example, if each supplier has one account, and each account is associated with one account history, then the supplier model could look like this:

```
class Supplier < ApplicationRecord
  has_one :account
  has_one :account_history, through: :account
end

class Account < ApplicationRecord
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ApplicationRecord
  belongs_to :account
end
```




```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

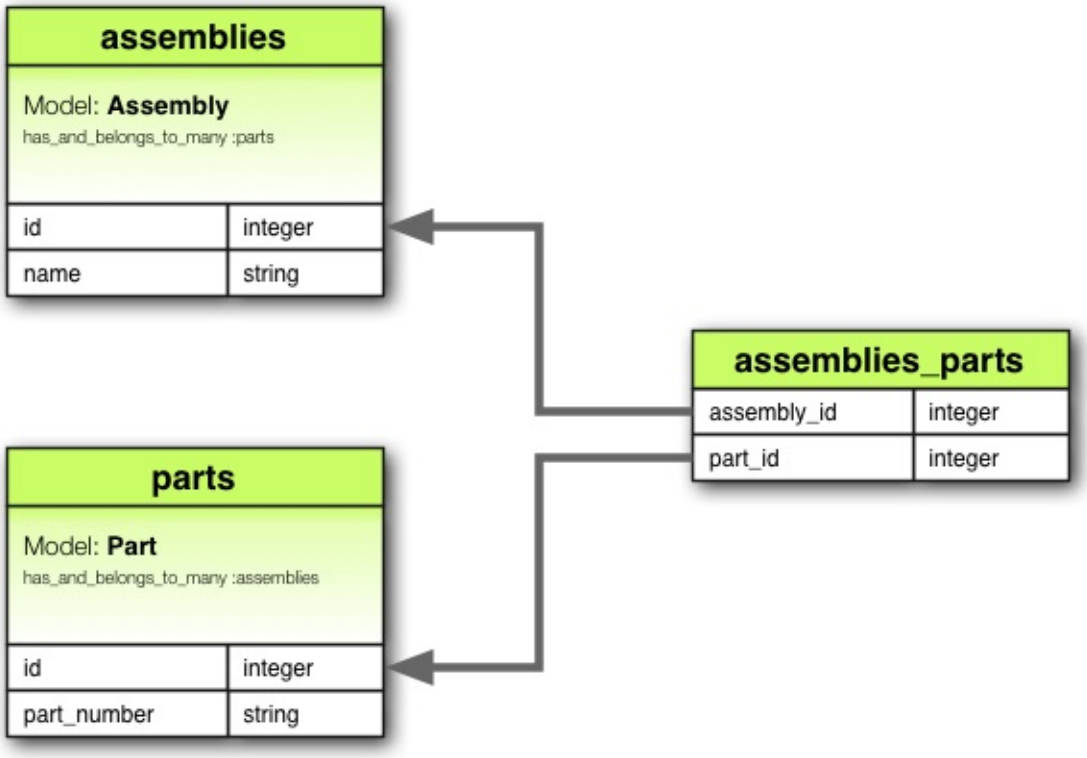
The corresponding migration might look like this:

```
class CreateAccountHistories < ActiveRecord::Migration[5.0]
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.belongs_to :supplier, index: true
      t.string :account_number
      t.timestamps
    end

    create_table :account_histories do |t|
      t.belongs_to :account, index: true
      t.integer :credit_rating
      t.timestamps
    end
  end
end
```

3.2.6 The has_and_belongs_to_many Association

A has_and_belongs_to_many association creates a direct many-to-many connection with another model, with no intervening model. For example, if your application includes assemblies and parts, with each assembly having many parts and each part appearing in many assemblies, you could declare the models this way:

```
class Assembly < ApplicationRecord
  has_and_belongs_to_many :parts
end

class Part < ApplicationRecord
  has_and_belongs_to_many :assemblies
end
```




```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

The corresponding migration might look like this:

```
class CreateAssembliesAndParts < ActiveRecord::Migration[5.0]
  def change
    create_table :assemblies do |t|
      t.string :name
      t.timestamps
    end

    create_table :parts do |t|
      t.string :part_number
      t.timestamps
    end

    create_table :assemblies_parts, id: false do |t|
      t.belongs_to :assembly, index: true
      t.belongs_to :part, index: true
    end
  end
end
```

2.7 Choosing Between belongs_to and has_one

If you want to set up a one-to-one relationship between two models, you'll need to add belongs_to to one, and has_one to the other. How do you know which is which?

The distinction is in where you place the foreign key (it goes on the table for the class declaring the belongs_to association), but you should give some thought to the actual meaning of the data as well. The has_one relationship says that one of something is yours - that is, that something points back to you. For example, it makes more sense to say that a supplier owns an account than that an account owns a supplier. This suggests that the correct relationships are like this:

```
class Supplier < ApplicationRecord
  has_one :account
end

class Account < ApplicationRecord
  belongs_to :supplier
end
```

The corresponding migration might look like this:

```
class CreateSuppliers < ActiveRecord::Migration[5.0]
  def change
    create_table :suppliers do |t|
      t.string  :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.integer :supplier_id
      t.string  :account_number
      t.timestamps
    end

    add_index :accounts, :supplier_id
  end
end
```

Using t.integer :supplier_id makes the foreign key naming obvious and explicit. In current versions of Rails, you can abstract away this implementation detail by using t.references :supplier instead.

2.8 Choosing Between has_many :through and has_and_belongs_to_many

Rails offers two different ways to declare a many-to-many relationship between models. The simpler way is to use has_and_belongs_to_many, which allows you to make the association directly:

```
class Assembly < ApplicationRecord
  has_and_belongs_to_many :parts
end

class Part < ApplicationRecord
  has_and_belongs_to_many :assemblies
end
```

The second way to declare a many-to-many relationship is to use has_many :through. This makes the association indirectly, through a join model:

```
class Assembly < ApplicationRecord
  has_many :manifests
  has_many :parts, through: :manifests
end

class Manifest < ApplicationRecord
  belongs_to :assembly
  belongs_to :part
end

class Part < ApplicationRecord
  has_many :manifests
  has_many :assemblies, through: :manifests
end
```

The simplest rule of thumb is that you should set up a has_many :through relationship if you need to work with the relationship model as an independent entity. If you don't need to do anything with the relationship model, it may be simpler to set up a has_and_belongs_to_many relationship (though you'll need to remember to create the joining table in the database).

You should use has_many :through if you need validations, callbacks or extra attributes on the join model.

3.2.9 Polymorphic Associations

A slightly more advanced twist on associations is the polymorphic association. With polymorphic associations, a model can belong to more than one other model, on a single association. For example, you might have a picture model that belongs to either an employee model or a product model. Here's how this could be declared:

```
class Picture < ApplicationRecord
  belongs_to :imageable, polymorphic: true
end

class Employee < ApplicationRecord
  has_many :pictures, as: :imageable
end

class Product < ApplicationRecord
  has_many :pictures, as: :imageable
end
```

You can think of a polymorphic belongs_to declaration as setting up an interface that any other model can use. From an instance of the Employee model, you can retrieve a collection of pictures: @employee.pictures.

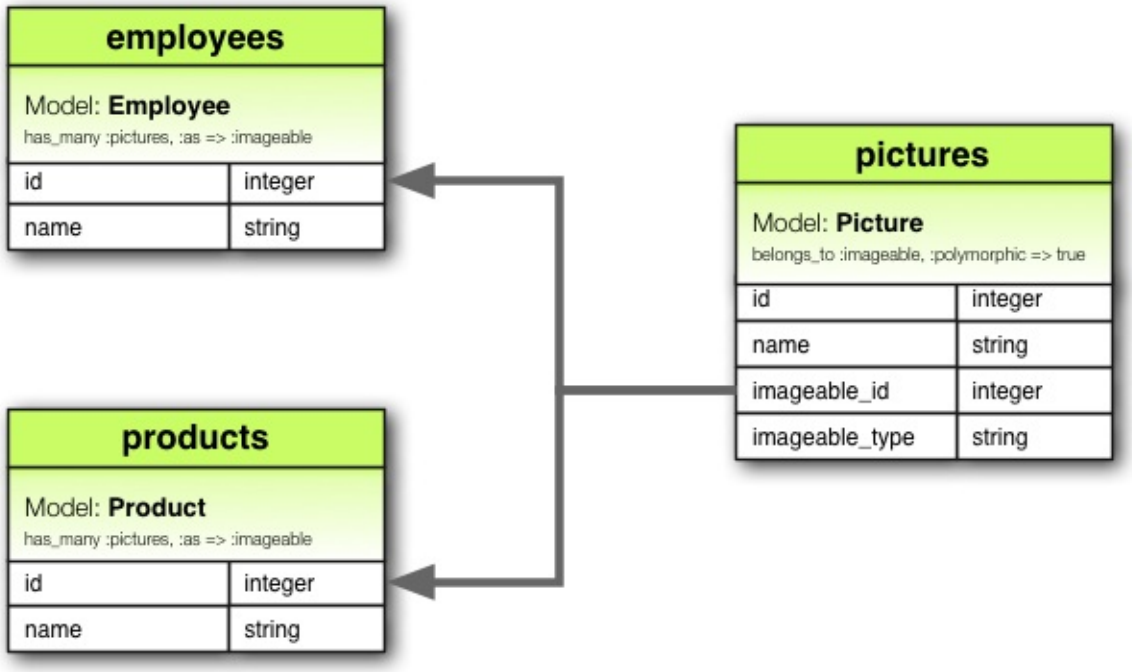
Similarly, you can retrieve @product.pictures.

If you have an instance of the Picture model, you can get to its parent via @picture.imageable. To make this work, you need to declare both a foreign key column and a type column in the model that declares the polymorphic interface:

```
class CreatePictures < ActiveRecord::Migration[5.0]
  def change
    create_table :pictures do |t|
      t.string  :name
      t.integer :imageable_id
      t.string  :imageable_type
      t.timestamps
    end

    add_index :pictures, [:imageable_type, :imageable_id]
  end
end
```

This migration can be simplified by using the t.references form:

```
class CreatePictures < ActiveRecord::Migration[5.0]
  def change
    create_table :pictures do |t|
      t.string :name
      t.references :imageable, polymorphic: true, index: true
      t.timestamps
    end
  end
end
```




```
class Picture < ActiveRecord::Base
  belongs_to :imageable, :polymorphic => true
end

class Employee < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end
```

3.2.10 Self Joins

In designing a data model, you will sometimes find a model that should have a relation to itself. For example, you may want to store all employees in a single database model, but be able to trace relationships such as between manager and subordinates. This situation can be modeled with self-joining associations:

```
class Employee < ApplicationRecord
  has_many :subordinates, class_name: "Employee",
                          foreign_key: "manager_id"

  belongs_to :manager, class_name: "Employee"
end
```

With this setup, you can retrieve @employee.subordinates and @employee.manager.

In your migrations/schema, you will add a references column to the model itself.

```
class CreateEmployees < ActiveRecord::Migration[5.0]
  def change
    create_table :employees do |t|
      t.references :manager, index: true
      t.timestamps
    end
  end
end
```

Controller Advance

1 Methods and Actions

A controller is a Ruby class which inherits from ApplicationController and has methods just like any other class. When your application receives a request, the routing will determine which controller and action to run, then Rails creates an instance of that controller and runs the method with the same name as the action.

```
class ClientsController < ApplicationController
  def new
  end
end
```

As an example, if a user goes to `/clients/new` in your application to add a new client, Rails will create an instance of ClientsController and call its new method. Note that the empty method from the example above would work just fine because Rails will by default render the new.html.erb view unless the action says otherwise. The new method could make available to the view a @client instance variable by creating a new Client:

```
def new
  @client = Client.new
end
```

The Layouts & Rendering Guide explains this in more detail.

ApplicationController inherits from ActionController::Base, which defines a number of helpful methods. This guide will cover some of these, but if you're curious to see what's in there, you can see all of them in the API documentation or in the source itself.

Only public methods are callable as actions. It is a best practice to lower the visibility of methods (with private or protected) which are not intended to be actions, like auxiliary methods or filters.

2 Parameters

You will probably want to access data sent in by the user or other parameters in your controller actions. There are two kinds of parameters possible in a web application. The first are parameters that are sent as part of the URL, called query string parameters. The query string is everything after "?" in the URL. The second type of parameter is usually referred to

as POST data. This information usually comes from an HTML form which has been filled in by the user. It's called POST data because it can only be sent as part of an HTTP POST request. Rails does not make any distinction between query string parameters and POST parameters, and both are available in the params hash in your controller:

```
class ClientsController < ApplicationController
  # This action uses query string parameters because it gets run
  # by an HTTP GET request, but this does not make any difference
  # to the way in which the parameters are accessed. The URL for
  # this action would look like this in order to list activated
  # clients: /clients?status=activated
  def index
    if params[:status] == "activated"
      @clients = Client.activated
    else
      @clients = Client.inactivated
    end
  end

  # This action uses POST parameters. They are most likely coming
  # from an HTML form which the user has submitted. The URL for
  # this RESTful request will be "/clients", and the data will be
  # sent as part of the request body.
  def create
    @client = Client.new(params[:client])
    if @client.save
      redirect_to @client
    else
      # This line overrides the default rendering behavior, which
      # would have been to render the "create" view.
      render "new"
    end
  end
end
```

2.1 Hash and Array Parameters

The params hash is not limited to one-dimensional keys and values. It can contain nested arrays and hashes. To send an array of values, append an empty pair of square brackets "[]" to the key name:

```
GET /clients?ids[]=1&ids[]=2&ids[]=3
```

> The actual URL in this example will be encoded as "/clients?ids%5b%5d=1&ids%5b%5d=2&ids%5b%5d=3" as the "[" and "]" characters are not allowed in URLs. Most of the time you don't have to worry about this because the browser will encode it for you, and Rails will decode it automatically, but if you ever find yourself having to send those requests to the server manually you should keep this in mind.

The value of params[:ids] will now be ["1", "2", "3"]. Note that parameter values are always strings; Rails makes no attempt to guess or cast the type.

Values such as [nil] or [nil, nil, ...] in params are replaced with [] for security reasons by default. See Security Guide for more information.

To send a hash, you include the key name inside the brackets:

```
<form accept-charset="UTF-8" action="/clients" method="post">
  <input type="text" name="client[name]" value="Acme" />
  <input type="text" name="client[phone]" value="12345" />
  <input type="text" name="client[address][postcode]" value="12345" />
  <input type="text" name="client[address][city]" value="Carrot City" />
</form>
```

When this form is submitted, the value of params[:client] will be { "name" => "Acme", "phone" => "12345", "address" => { "postcode" => "12345", "city" => "Carrot City" } }. Note the nested hash in params[:client][:address].

The params object acts like a Hash, but lets you use symbols and strings interchangeably as keys.

2.2 JSON parameters

If you're writing a web service application, you might find yourself more comfortable accepting parameters in JSON format. If the "Content-Type" header of your request is set to "application/json", Rails will automatically load your parameters into the params hash, which you can access as you would normally.

So for example, if you are sending this JSON content:

```
{ "company": { "name": "acme", "address": "123 Carrot Street" } }
```

Your controller will receive params[:company] as { "name" => "acme", "address" => "123 Carrot Street" }.

Also, if you've turned on config.wrap_parameters in your initializer or called wrap_parameters in your controller, you can safely omit the root element in the JSON parameter. In this case, the parameters will be cloned and wrapped with a key chosen based on your controller's name. So the above JSON POST can be written as:

```
{ "name": "acme", "address": "123 Carrot Street" }
```

And, assuming that you're sending the data to CompaniesController, it would then be wrapped within the :company key like this:

```
{ name: "acme", address: "123 Carrot Street", company: { name: "acme", address: "123 C
arrot Street" } }
```

You can customize the name of the key or specific parameters you want to wrap by consulting the API documentation

Support for parsing XML parameters has been extracted into a gem named actionpack-xml_parser.

2.3 Routing Parameters

The params hash will always contain the :controller and :action keys, but you should use the methods controller_name and action_name instead to access these values. Any other parameters defined by the routing, such as :id, will also be available. As an example, consider a listing of clients where the list can show either active or inactive clients. We can add a route which captures the :status parameter in a "pretty" URL:

```
get '/clients/:status' => 'clients#index', foo: 'bar'
```

In this case, when a user opens the URL /clients/active, params[:status] will be set to "active". When this route is used, params[:foo] will also be set to "bar", as if it were passed in the query string. Your controller will also receive params[:action] as "index" and params[:controller] as "clients".

2.4 default_url_options

You can set global default parameters for URL generation by defining a method called default_url_options in your controller. Such a method must return a hash with the desired defaults, whose keys must be symbols:

```
class ApplicationController < ActionController::Base
  def default_url_options
    { locale: I18n.locale }
  end
end
```

These options will be used as a starting point when generating URLs, so it's possible they'll be overridden by the options passed to url_for calls.

If you define default_url_options in ApplicationController, as in the example above, these defaults will be used for all URL generation. The method can also be defined in a specific controller, in which case it only affects URLs generated there.

In a given request, the method is not actually called for every single generated URL; for performance reasons, the returned hash is cached, there is at most one invocation per request.

2.5 Strong Parameters

With strong parameters, Action Controller parameters are forbidden to be used in Active Model mass assignments until they have been whitelisted. This means that you'll have to make a conscious decision about which attributes to allow for mass update. This is a better security practice to help prevent accidentally allowing users to update sensitive model attributes.

In addition, parameters can be marked as required and will flow through a predefined raise/rescue flow to end up as a 400 Bad Request.

```
class PeopleController < ActionController::Base
  # This will raise an ActiveModel::ForbiddenAttributes exception
  # because it's using mass assignment without an explicit permit
  # step.
  def create
    Person.create(params[:person])
  end

  # This will pass with flying colors as long as there's a person key
  # in the parameters, otherwise it'll raise a
  # ActionController::ParameterMissing exception, which will get
  # caught by ActionController::Base and turned into that 400 Bad
  # Request reply.
  def update
    person = current_account.people.find(params[:id])
    person.update!(person_params)
    redirect_to person
  end

  private
    # Using a private method to encapsulate the permissible parameters
    # is just a good pattern since you'll be able to reuse the same
    # permit list between create and update. Also, you can specialize
    # this method with per-user checking of permissible attributes.
    def person_params
      params.require(:person).permit(:name, :age)
    end
end
```

2.5.1 Permitted Scalar Values

Given

```
params.permit(:id)
```

the key :id will pass the whitelisting if it appears in params and it has a permitted scalar value associated. Otherwise, the key is going to be filtered out, so arrays, hashes, or any other objects cannot be injected.

The permitted scalar types are String, Symbol, NilClass, Numeric, TrueClass, FalseClass, Date, Time, DateTime, StringIO, IO, ActionDispatch::Http::UploadedFile, and Rack::Test::UploadedFile.

To declare that the value in params must be an array of permitted scalar values, map the key to an empty array:

```
params.permit(id: [])
```

To whitelist an entire hash of parameters, the permit! method can be used:

```
params.require(:log_entry).permit!
```

This will mark the :log_entry parameters hash and any sub-hash of it as permitted. Extreme care should be taken when using permit!, as it will allow all current and future model attributes to be mass-assigned.

2.5.2 Nested Parameters

You can also use permit on nested parameters, like:

```
params.permit(:name, { emails: [] },
              friends: [ :name,
                         { family: [ :name ], hobbies: [] }])
```

This declaration whitelists the name, emails, and friends attributes. It is expected that emails will be an array of permitted scalar values, and that friends will be an array of resources with specific attributes: they should have a name attribute (any permitted scalar values allowed), a hobbies attribute as an array of permitted scalar values, and a family attribute which is restricted to having a name (any permitted scalar values allowed here, too).

2.5.3 More Examples

You may want to also use the permitted attributes in your new action. This raises the problem that you can't use require on the root key because, normally, it does not exist when calling new:

```
# using `fetch` you can supply a default and use
# the Strong Parameters API from there.
params.fetch(:blog, {}).permit(:title, :author)
```

The model class method accepts_nested_attributes_for allows you to update and destroy associated records. This is based on the id and _destroy parameters:

```
# permit :id and :_destroy
params.require(:author).permit(:name, books_attributes: [:title, :id, :_destroy])
```

Hashes with integer keys are treated differently, and you can declare the attributes as if they were direct children. You get these kinds of parameters when you use accepts_nested_attributes_for in combination with a has_many association:

```
# To whitelist the following data:
# {"book" => {"title" => "Some Book",
#             "chapters_attributes" => { "1" => {"title" => "First Chapter"},
#                                        "2" => {"title" => "Second Chapter"}}}}

params.require(:book).permit(:title, chapters_attributes: [:title])
```

2.5.4 Outside the Scope of Strong Parameters

The strong parameter API was designed with the most common use cases in mind. It is not meant as a silver bullet to handle all of your whitelisting problems. However, you can easily mix the API with your own code to adapt to your situation.

Imagine a scenario where you have parameters representing a product name and a hash of arbitrary data associated with that product, and you want to whitelist the product name attribute and also the whole data hash. The strong parameters API doesn't let you directly whitelist the whole of a nested hash with any keys, but you can use the keys of your nested hash to declare what to whitelist:

```
def product_params
  params.require(:product).permit(:name, data: params[:product][:data].try(:keys))
end
```

Chapter III: Questions and Quiz

Table of Content