# Build It

# ©2012-2013 Jeff Cohen and Raghu Betina

# On Your Marks...

## What is computer programming?

Wikipedia describes it like this:

> *A computer program (also software, or just a program) is a sequence of instructions written to perform a specified task with a computer.*

This is actually a very good, concise description. This sentence encapsulates the task in front of us. If we want to develop a web application, we're going to have to learn how give the computer a sequence of instructions that it can perform. In other words, we have to teach the computer how we want our application to work.

We can distill our job into an even shorter description:

> *software is a sequence of instructions*

To build any piece of software, like our web application, we apparently need to figure out how to formulate it as a sequence of instructions in order for the computer to do what we want.

What are these instructions? And how do we determine the proper sequence?

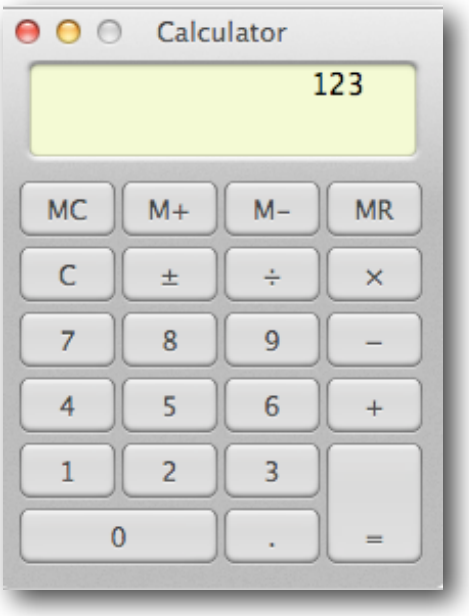
## IPO

Every computer program ever built has operated under a single, fundamental, and recurring pattern of operation.

- **Input.** Some sort of input is fed into the computer.
- **Processing**. Some kind of calculation or manipulation is performed.

• **Output**. The computer outputs the result of the processing in some fashion.

Let's take a look at a computer that you're likely already very familiar with.




A calculator is a computer that, like all computers, follows the IPO pattern.

**Input**: Can you identify where the input will take places? Right: the push buttons provide a human interface so that the user can specify the desired input.

**Processing**: If this were a real handheld calculator, where would the processing take palce? Inside the calculator, if we opened it up with a screwdriver, we would find a computer chip responsible for performing all of the processing.

**Output**: What output facilities does the calculator have? Just one: the LCD display screen at the top. In the old days, calculators didn't have screens. They used rolls of paper, and the calculator would physically print the output onto the paper. All software must have some kind of output. Other examples of output mechanisms include a visual display screen, printed paper, a file saved to disk, and electronic data transmitted as an email, web response, or text message.

Here's another example of the IPO pattern in action:


**Input:** Can you find at least two sources of input on this digital camera? How about the shutter button on top, and the lens which receives light into the camera chamber?

**Processing**: When the shutter button is pressed, the camera processes the light by using light-sensitive sensors, transforming

color patterns into digital information.

**Output**: A modern digital camera has two output paths. One is a digital SD card, where the image data is saved for later processing. The other is on a visual display on the back of the camera:


While we're here, notice all of the additional input buttons that are available on the back of the camera.

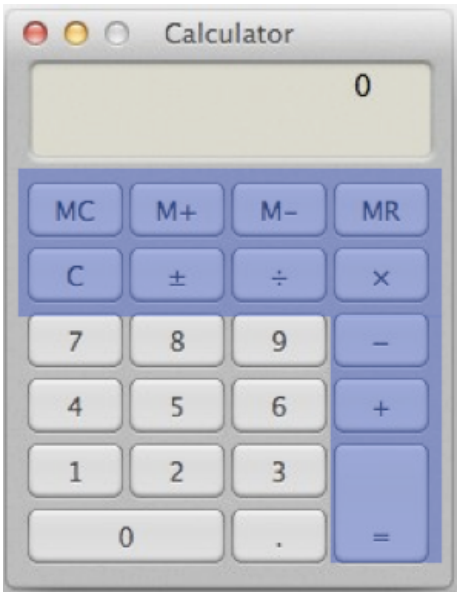
## Input: Instructions vs. Data

Let's be more specific about the term *input*. There are actually two types of input that we talk about when we're writing a computer program.

- **Instructions**. A computer program consists primarily of a sequence of instructions to the computer. Each instruction is assigned a unique number. When a particular instruction number is given to the computer, it will perform the action associated with that number. Programming *languages*, like Ruby, allow us to use English words instead of numbers, making our job a whole lot easer.

- **Data**. The other kind of input that we'll often use provides *data* to go along with the instructions. Data can be numbers, words, or even entire data *structures* that can contain longs lists of numbers or words. Data elements can't make the computer do anything on their own. But many instructions expect some data to come along for the ride, so that the instruction has something to do.

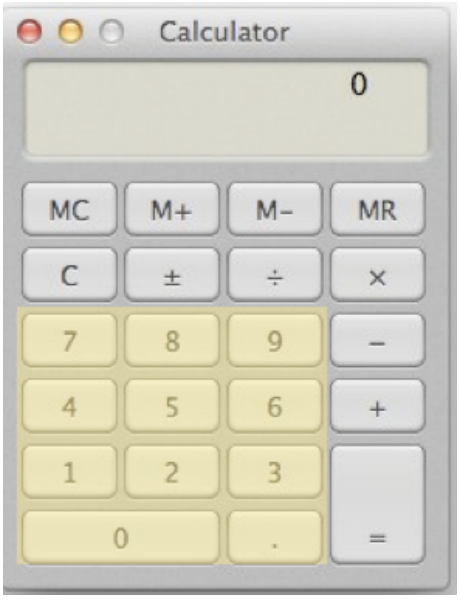
Let's take a look at our calculator again. Among the input buttons, which ones are responsible for sending *instructions*, and which provide *data* for the instructions to operate on?




The buttons in purple are used to provide instructions. *Add*. *Subtract. Multiply. Divide. Evaluate. Clear. Store in Memory. Clear Memory. Add to Memory. Subtract from Memory.*

When you tap one of the instruction buttons, what do you expect the calculator to do?

Imagine that you've entered *5 + 5,* and then you tap the = key. You would expect the calculator to add up the numbers (*process the data*) and display *10* in the LCD display (*output*).




Now: which buttons are used to input data instead of instructions?

You guessed it: the ones shown here in yellow. We use these buttons to provide the calculator with the numbers we want to operate upon.

## The Calculator Language

Using this kind of calculator now seems intuitive, but I bet there was a time, if you can recall, when you learned how to use a calculator for the very first time, ever. Maybe it was a parent or teacher that demonstrated how to use it.

The first demonstration you got was very important. It showed you the *sequence* that was necessary to have the calculator solve your math problem. Calculators expect to receive both instructions and data, and in a particular sequence. This is the *programming language* that the calculator can understand.

Like spoken languages, programming languages have nouns, verbs, and grammar rules. We tend to call the grammar rules "language syntax" but it's the same idea.

If we took all of the buttons on our calculator and grouped them into nouns and verbs, what it would look like? Something like this:

| **Nouns** | **Verbs** |
|---|---|
| 1 2 3 4 5 6 7 8 9 0 . | + – × ÷ ± = C MC M+ M- MR |

The nouns are the data we want to act upon.
The verbs are actions. They represent our choices for how to manipulate the data we have.

Here is a sentence in Calculator Language:

# 5 + 4 - 3 =

Can you locate the nouns and verbs in this sentence?

Ok, now here comes the important part. It's a crucial component of every language and will be very important to keep in view as we begin to learn other programming languages.

Ths crucial component is the *syntax*. The syntax is the rules by which we are allowed to put the nouns and verbs together. It's the order in which these symbols can appear together.

The syntax of a language is paramount. Suppose we decided to change the sequence of our nouns and verbs:

# 5 - 4 + 3 =

Would we get the same result? No, we wouldn't. In Calculator Language, expressions are evaluated left to right. Therefore, the answer would be very different from what we obtained before. Seems obvious, right?

Now, try this one:

# 5 4 + 3 -

Whoa, what's this? Looks like... maybe 54 (but if that's the case, why isn't the 4 right up next to the 5?), plus 3... and then we've begun a subtraction operation but didn't finish it. This is obviously an incomplete expression and violates the syntax rules unless we complete it.

However, there's another Calculator Language where the above syntax is legal! It's called RPN: *reverse Polish notation*. In RPN, the above syntax is required to provide the original, expected answer of 6.

So it is with programming languages. All three components - nouns, verbs, and syntax - are necessary to properly define how a language works. Change any one of them, and you have an entirely different language.

Each language has a way of letting you indicate what's a noun - that is, some data - from what's a verb - that is, an instruction. Let's take a look at the most important language on the web, *hypertext markup language*, which we call HTML for short.

## HTML

Instructions in HTML are surrounded with angle brackets. Here are some examples:

```
<h1>
<p>
<ul>
<li>
```

Meanwhile, data elements in HTML are simple text values:

```
I am an HTML element.
```

Most instructions will collaborate with some data. The data is said to be the "content" of the HTML element. For most HTML elements that contain data content, a closing tag is used to indicate the end of the partnership:

```
<h1>This is a title!</h1>
<p>This is a simple paragraph.</p>
```

Sometimes HTML tags are also adorned with attributes:

```
<h1 class="page-title">This is a heading!</h1>
<a href="http://www.google.com">Google</a>
```

And a few elements do not require closing tags at all:

```
<img src="/images/logo.png">
<link rel="stylesheet" href="/stylesheets/my_styles.css">
```

Elements can contain other elements in order to convey a semantic relationship:

```
<h1>Meeting Agenda</h1>
<ul>
  <li>Introduction</li>
  <li>Budget Summary</li>
  <li>Future Outlook</li>
  <li>Conclusion</li>
</ul>
```

Notice how the <li> elements are contained inside the <ul> element by ensuring that the closing </ul> tag is placed below the <li> elements.

Because the angle brackets have significant meaning to an HTML processor - they are not displayed, but instead denote instruction tags - it can be tricky when you want to display one on your HTML page.  In such cases, an HTML entity must be used:

```
&lt;
```

That's the "entity representation" for a literal less-than sign.  Try entering it into your HTML page and viewing it in your browser.

## Ruby

Open up your text editor, type in this Ruby program, and save it under the name `hello.rb`:

```
puts “Hello!”
```

This Ruby program contains one instruction, **`puts`**, and one piece of data, **`“Hello!”`**

Let’s run this program by using the Ruby processor:

```
ruby hello.rb
```

If all goes well, you should see “Hello!” displayed in your Terminal window.

Congratulations! You’ve just written your first Ruby program.

- Ruby has a number of built-in data types or *classes*.
- The classes you want to learn at the start are: String, Array, Fixnum, Hash, and Symbol.
- A *Symbol* is like a lightweight-string constant. It is a human-readable identifier and becomes globally defined upon first use. It starts with a colon and does not require quotations marks. Examples:

```
:my_symbol
:purple
:hockey
```

- Symbols are similar to *enums* in other languages: they let you define a human-readable word to a constant. In Ruby, you don’t care about the actual constant value, and you never define it yourself.
- A *Fixnum* is the baseclass for fixed-point numerical types. Think “integer” when you see a *Fixnum* object.
- You can ask any variable for its class by calling the **`.class`** method on it:

```
“hello”.class                    # => String
:hello.class                     # => Symbol
[19, :apple, “Tea”].class        # => Array
```

## Learning Everything with Interactive Ruby

Every Ruby installation comes with a power utility that every Ruby developer uses: *interactive ruby,* known as the *irb* command.

The best way to learn about Ruby is to start *irb* and begin playing around. The more curious you are, the faster you will learn everything you want to know.

```
$ irb
irb(main):001:0> "hello".length
=> 5
irb(main):002:0> exit
```

Every time you enter a line of Ruby, irb will do two things:

1. Interpret and execute you line of code
2. Emit the final return value of your code

To end your irb session, type *exit* or press *Ctrl-D*.

## Expressions

- Every Ruby method returns a value.
- Ruby has a built-in object **`nil`** that can be used to represent “nothing.”
- Every Ruby method must return a value, even if that value is the **`nil`** object.
- Every Ruby statement, therefore, is a Ruby expression that will always boil down to some value.
- Read the above again. :-)
- More proof that everything is an expression: the **`puts`** method returns a **`nil`** value:

```
irb(main):001:0> puts “Wazzzzup!”
```

```
Wazzzzup!
=> nil
```

- You can see that after the **puts** method does its work, it returns a **nil** value as shown by the irb intepreter.
- Variables hold the result of an expression. Their class is determined dynamically based on actual value that's being assigned:

```
sum = 2 + 2
puts sum.class       # => Fixnum

sum = "Hello " + "there!"
puts sum.class       # => String
```

- We never indicate the class or datatype of variables. Ruby determines the type automatically at runtime.
- There is no compilation step in Ruby. Everything is interpreted on the fly.

## Objects and Methods

Now that we know that everything in Ruby is built as a series of expressions, and every expression boils down to some value (or, more technically, one object), we are ready to see how objects work in Ruby.

Objects store data, but they also are a container of methods that can help you act on that data. Objects hold data and provide services that can perform activities based on their data.

Here is a String object. Notice how we use *dot notation* to invoke the services (methods) that are available on that object.

```
name = "Charlie Brown"
```

```
puts name.length              # => 13
puts name.reverse             # => “nworB eilrahC”
puts name.upcase              # => “CHARLIE BROWN”
```

Another string object with different data provides the same services but will yield different results, because the data is different:

```
name = “Linus Van Pelt”


puts name.length              # => 14
puts name.reverse             # => “tleP naV siniL”
puts name.upcase              # => “LINUS VAN PELT”
```

## Lists of Things

In which we extol the virtues of arrays and hashes.

**Key Ideas**

1. Arrays
2. Hashes
3. Special Hash Syntax

Ruby provides us with two built-in classes that will help us maintain a list of objects: **`Array`** and **`Hash`**.

Let’s start with arrays. There are two ways to construct an array.  First, let’s look at the easy way, which uses “array literal” syntax for constructing a new, empty array:

```
my_favorites = [“purple”, “cookies”]
```

This line of code creates a new variable, my_favorites, which is an Array instance holding two objects representing my favorite color and favorite snack.

You don't have to provide any initial elements when you create the Array:

```
my_favorites = []
```

Another alternative is to use the new method on the class itself:

```
my_favorites = Array.new
```

There is no way to provide any initial elements when you use the **new** method.

We can add more objects at anytime by calling the **push** method. Here I am adding my favorite kind of music to the list:

```
my_favorites.push "jazz"

# my_favorites is now ["purple", "cookies", "jazz"]
```

There's a popular synonym for push called the "shovel" operator:

```
my_favorites << "jazz"
```

It's identical to calling **push**. This will happen sometimes with Ruby classes: there will be more than one method available to perform the same job. The choice of which one to use is left entirely up to the developer's taste.

While **push** always adds data to the end of the array, **insert** will let you add it at a specific position. Check the Ruby docs to learn how to use the **insert** method..

Now that we've got a list of things, how do we retrieve specific elements out?

With Ruby arrays, there are many methods available to us. The choice of which one to use depends upon what you are trying to do. Do you need the first item? Or maybe the last?

```
my_favorites.first  # => “purple”
my_favorites.last   # => “jazz”
```

Sometimes you will need an element somewhere in between the first and last. Use the **`[]`** method to access any element you want. You must provide the *zero-based index* of the element you want to retrieve:

```
my_favorites[1]     # => “cookies”
```

Here, passing a 1 means “give me the second item”. Here’s how we would retrieve the very first item (if we didn’t like using the **`first`** method shown above):

```
my_favorites[0]     # => “purple”
```

It turns out that the **`[]`** method is just a synonym for **`at`**. Here are all the ways we can retrieve the first element from an array named **`my_favorites`**:

```
my_favorites[0]      # => “purple”
my_favorites.first   # => “purple”
my_favorites.at(0)   # => “purple”
```

There are several other methods you’ll want to become familiar with when working with arrays. Be sure to read up on **`length`**, and its synonyms **`size`** and **`count`**; **`sort`**; **`compact`**; **`empty?`**; and **`join`**.

Now let’s look at the **`Hash`**. Like an array, it also stores a list of objects. But it’s often a better choice, because you don’t access elements with a confusing, zero-based index. Instead, you provide a *key* that is to be associated with the object. (Some other languages call them *associative arrays* for this reason).

In a hash, we have a list of *key-value pairs*. Here’s how we can use hash literal syntax to create a hash and initailize it with some data:

```
my_favorites = { “color” => “purple”, :food => “cookies” }
```

Notice how we have to provide a key for each value, and we use the “rocket” or “hash rocket” operator to indicate which pairs go together. Pay close attention to where the commas go, and where the rockets go.

Keys must be unique, because that’s how we’re going to try to retrieve values from the hash.

Notice in our example that we used a string for one of the keys, and a symbol for the other. Indeed, the keys can be any type of object you want. Symbols are a common choice, since they give us string-like readability of a name for something, without the overhead of a string object.

To show that keys and values can be of any type, here’s another example:

```
person = { :name => “Jeff”, :faves => [“purple”, “cookies”] }
```

Here we chose to use symbols for both keys, but one value is a string while the other value is an array. We could even use another hash as a value:

```
person = { :name => “Jeff”,
           :faves => { “color” => “purple”,
                       “food” => “cookies }
         }
```

We can add more key-value pairs after initial construction with the **`merge`** method:

```
my_favorites.merge :music => “jazz”
```

More common is to use the synonymous **`[]`** method combined with an assignment:

```
my_favorites[:music] = “jazz”
```

How do we retrieve data from a hash? Easiest way is to use the `[]` method:

```
my_favorites[:color]     # => “purple”
person[:faves][“color”]  # => “purple”
```

Notice how we must use successive `[]` method invocations to “dive in” to the nested hash.

Finally, we must mention an alternative hash syntax for providing key-value pairs. It only applies when the key is a symbol.

Instead of:

```
favorites = { :color => “purple”, :food => “cookies }
```

we can notice that the keys are symbols and opt for an alternative syntax:

```
favorites = { color: “purple”, food: “cookies }
```

**Remember**, this only works when the key is a symbol. If the key is any other type of object, the classic `=>` syntax must be used instead.

## Loops and Iterators

**Key Ideas**

1. Array#each
2. code blocks
3. block variables

When working with lists, we often want to perform an operation on each and every item in the list. For example, suppose we had this Ruby array:

```
favorites = [“purple”, “cookies”, “jazz”]
```

and we wanted to construct an HTML list from this data:

```
<ol>
  <li>Purple</li>
  <li>Cookies</li>
  <li>Jazz</li>
</ol>
```

How would we go about doing that? Forget about computer programming for a minute. If you had to write this HTML with a pen on a paper napkin, and all I told you was the contents of my array, what you do?

Think about it for a few minutes. Try to describe out loud how you would "know" what to write on the napkin, the exact steps you would take if you had to explain it to someone else.

You're back so soon? Ok, let's walk through it. We know we're going to have to do something with each element in the array - basically, put **<li>** tags around them - but before we even get that far, we have to start with the **<ol>** tag:

```
<ol>
</ol>
```

Next up: write down each list element. To do that, we start with an **<li>** tag:

```
<ol>
  <li>
</ol>
```

Next, we grab the first element from our array:

```
<ol>
  <li>purple
</ol>
```

but we capitalize it:

```
<ol>
  <li>Purple
</ol>
```

and finish with the closing tag:

```
<ol>
  <li>Purple</li>
</ol>
```

Are we done? No. We know that there are more items in our array we haven't gotten to yet. So we repeat the same exact procedure with the second item:

```
<ol>
  <li>Purple</li>
  <li>Cookies</li>
</ol>
```

Are we done? No. There's another item in our list:

```
<ol>
  <li>Purple</li>
  <li>Cookies</li>
  <li>Jazz</li>
</ol>
```

Are we done? Yes. How do we know? Because I've gone through each item in the list, and I've reached the end.

Does this all sound silly? That we're describing something seemingly obvious in such detail? It probably does. But that's exactly how we need to think - in microsteps - in order to write code.

Now that we have a manual procedure for how to transform our list of favorites into a HTML ordered list, we can write those instructions down. If we write the instructions in Ruby, then the computer can do the work for us! (That's why we call it computer programming.)

We now reach one of the most interesting, distinctive, and vital features of the Ruby programming language: *enumerators*. Using an enumerator is a powerful way to step through each element of an array, one at a time, and perform some work with each one. Let's look at the code and then describe how it works.

```
1   favorites = [“purple”, “cookies”, “jazz”]
2
3   puts “<ol>”
4   favorites.each do |favorite_thing|
5     puts “<li>” + favorite_thing.capitalize + “</li>”
6   end
7   puts “</ol>
```

The new stuff is on lines 4-7. (On line 1, we create an array, and on line 3 we emit the **<ol>** tag.)

On line 4, we call a method on our array named **each**:

```
4   favorites.each do |favorite_thing|
```

This method that will look at the **favorites** array, and begin to hand us each element in turn so that we can perform some work with it. We provide a *block* of code between the **do...end** pairing. The block starts with the keyword **do** on line 4, and continues through the keyword **end** on line 6.

A block is a section of code that describes a particular unit of work. When we provide our block to the **each** method, our block will be called automatically, once for each and every element contained inside the array.

The `each` method allows us to receive the current element as it works its way through the array. We do that by providing a *block variable.* Here we've named `favorite_thing`. A block variable is like a local variable that only has meaning within the scope of the block. It's very similar to a method argument, except we surround the argument vertical pipes `||` instead of parentheses.

The actual value held in the `favorite_thing` variable will change as the `each` method works its way through the array. We have three elements, so our block will execute three times, and the `favorite_thing` variable will change three times. The first time the block is executed, it will have the value "purple". The next time, "cookies". And finally, "jazz".

When the `each` method finally runs out of elements, it considers its work to be done. Our program then continues on line 7, which emits the closing tag for the list.

Here's what you need to know about arrays and hashes:

- They are used to contain lists of other objects.
- Arrays connect a positional index with particular value
- Hashes connect a given key object with a value
- You can retrieve a specific item if you know how (a zero-based position index for an array, or the key object for a hash)
- You can iterate through the elements of an array or hash with the `each` method
- Other iterator methods are `select`, `collect`, `map`, and `detect`
- *Blocks* are sections of code enclosed in a `do`...`end` pair

## Mash It Up

JSON

API via open-uri

Generate a web page with Ruby

## Ride The Rails

Put into a Rails app controller

# All Aboard

When you want to see Google's home page, you know to type "www.google.com" into the address bar in your browser, and then you press the Enter key on your keyboard. About 50 millesconds later, you're seeing Google's logo and a search box appear on your screen.

How did that happen?

No really.. how, *exactly,* did that work?

You might say, "I'm just using the web to search for something."

Well, sure. But when we say that we are using "the web" or "the internet," what is our computer *exactly* doing? Your computer somehow knows how to display Google's home page and how to communicate with Google's search engine.

It's actually simpler than you might think. The "internet" is a group of computers that are able to send text, photos, and videos, to each other. When you view a web page on your computer, it's only because *your computer* is one of those computers "on the internet," and it is able to send and receive data with any computer in the group.

Oh and when we say "group," we actually mean a "very large gigantic humongous group of interconnected networks." Or for short: the *internet*.

Very few people on the planet understand the actual workings of this group communication. Those few are called "web developers," and we're about to join the club. So let's learn about how computers "on the internet" exchange information. In

other words, how do your computer “know” how to go get Google’s home page and display it on your screen?

It turns out that in order for all of these computers to be able to talk to each other, they first had to agree upon a common language to use so that they could understand each other. That language - or more technically speaking, that *protocol* - was called the HyperText Transfer Protocol.

Nowadays we just call it HTTP for short. HTTP is a standard protocol for exchanging information between computers.

## HTTP Basics

**Key Ideas**

1. HTTP Request
2. HTTP Response
3. Uniform Resource Locator

- One computer, the *client*, sends an *HTTP request* to another computer, the *server*.
- I lied a bit just there. That last statement was an oversimplification. It’s really just one piece of software on the client that sends an *HTTP request* to another piece of software on the server.
- Ok so I lied again. Two computers aren’t actually necessary at all. The client and server software can actually exist on the same computer, and everything would still work the same. This is unusual in the real world, but extremely common when you’re developing your own web application. For most of this course you’ll be simulating a web experience by running both the client and server software on your computer.
- Anyway... once the HTTP request is received, the server *processes* the request and replies with a *response*.
- HTTP IPO diagram goes here
- (Sidebar) Something important must be mentioned at this point. HTTP is *stateless*. No state is maintained by the server. Each HTTP request must be standalone and

independent, each carrying sufficient information in each request for the server to perform its work. The server is not required to "remember" anything about the previous request.

- A web application receives the HTTP request as the *input* and is responsible for generating HTTP responses as the *output*.
- Therefore, understanding the structure of an HTTP request is paramount. Let's get into some of the details.
- An HTTP request packet contains a list of *headers* and a *body.*
- A *header* is a simple name-value pair of strings separated by a colon. Examples: "Accept: text/html", "Server: Apache", "Pragma: no-cache"
- A *URL* is a text string that uniquely identifies a specific *resource* on the internet. We typically think of URLs as representing web pages, but URLs can identify resources of all kinds: sogns, individual photos, search results, flight information, etc.
- The three most important headers in an HTTP request are the *HTTP method,* the *URL*, and the *acceptable format*.
- The HTTP method can be one of: GET, POST, PUT, or DELETE. (The HTTP specification also defines the methods OPTIONS, TRACE and CONNECT, but they are generally not used in most apps and are not a concern for us in this course.)
- An HTTP response packet contains a list of *headers* and a response *body*.
- All HTTP packets consist of plain text. Binary files must be encoded in some kind of textual representation in order to be transmitted.
- The body of an HTTP response generally contains HTML, Javascript, or CSS. But HTTP responses can also contain PDF data, JSON structured strings, CSV data, images, audio files, and more.

# HTTP on Rails

So much for theory. Let's build a real web app. We will write some Ruby code to send HTTP requests and responses back and forth between a client (our browser) and a server (our app).

**Key Ideas**

1. **Routes**
2. **Actions**
3. **View Templates**

Remember, the job of every web application is pretty simple: receive a stream of HTTP requests, one after the other, and generate an HTTP response for each one. That's it.

- A web application does just one thing, and it does it over, and over, and over again:
  1. The app waits for an incoming HTTP request.
  2. The app inspects the data inside the request.
  3. The app performs some work associated with the request.
  4. The app returns an HTTP response back to the client
- Nice graphic of http cycle goes here
- We are going to write our application in Ruby. So now the question becomes, how do we write some Ruby to receive HTTP requests and generate HTTP responses? That's where Rails comes in.
- The reason developers like to use a pre-written *web framework* like Rails is to enable the developer to write code that can respond to incoming HTTP requests by easily as possible. The framework will abstract away low-level network protocol details, giving the developer a jumpstart on working on the more interesting parts of the application.
- Our app will receive requests for different *resources*.
- The first thing the Rails framework will require from our app is to specify the rules by which incoming HTTP requests can be handed off to specific Ruby methods.
- The rule syntax that is used to map an HTTP request to a Ruby method consists of:

  - a URL pattern string
  - an *HTTP method* (GET, POST, PUT, or DELETE).

- Each rule that maps a specific URL pattern-with-HTTP-method to a Ruby method is called a *route*.
- Each route must be connected to a Ruby method that can return an appropriate HTTP response.
- The Ruby method chosen to provide the response must be an instance method of a Ruby class.
- The Ruby class can contain many such methods that each provide responses for distinct, routed requests.
- Such Ruby classes are termed *controllers.* Controllers are generally considered to be the "brains" of the application, and perform the crucial work of generating HTTP responses.
- A Ruby method inside of a controller class that is generates an HTTP response is called an *action* or *action method*.
- Controller classes live in the app/controllers folder and derive from the *ApplicationController* base class.
- We never create instances of controller classes ourselves.
- Rails creates a fresh instance of the appropriate controller class for every HTTP request that is received.
- Any instance variables that are assigned during the running of an action method will be thrown away as soon as the HTTP response is generated. Therefore, other methods will never share those instance variables. This is by design.
- Instance variables are therefore only useful to *views*.
- A view can provide an HTML response template, as an alternative to embedding the entire HTML string inside of the action method.
-

# Routes

- All routes for the app are specified in a single file: config/routes.rb.
- When an HTTP request is received by the Rails dispatch mechanism, the applications's routes are consulted to see if the application can respond to the incoming request. The routes file is consulted by processing the routes in order, starting at the top and proceeding downward. The first routing rule that matches is used to determine the Ruby method that should be invoked.
- The Ruby methods that are specified in the routes file are called *actions* or *action methods*.

## Controllers and Actions

- Action methods are instance methods of *controller classes*.
- Routes must therefore map an incoming URL pattern/HTTP method pair to a Ruby method by specifying the controller and method names. The controller name is not the full controller class name, but rather a string providing the first part of the class name without the "Controller" suffix. The method name must be exact as it appears in the method definition.
- There are two ways to define routes: by providing a URL string and a hash of options, or a hash which uses a url string as one of its keys.
- Sample routes go here.
- Controller classes derive from certain base classes which provide built-in support for HTTP request inspection and HTTP response generation.
- Actions are simple Ruby methods that accept zero method arguments.
- Actions can use the inherited *params* method to inspect a hash of HTTP request data.
- Actions must generate an HTTP response. This is an action's sole responsibility.
- The *render* method is used to generate an HTTP response packet.
- There are four basic choices when rendering a response:

1. Call the *render* method, passing a Ruby hash that provides complete data for the response headers and body.
2. Call the *redirect_to* method, passing a URL string. This is equivalent to calling *render(status: 302, location: "url string", body: nil)*.
3. Call the render method, passing the filename of an ERb template. If a bare filename without a path is provided (i.e the filename does not contain a slash), the file is assumed to reside in a subfolder of the *views* folder, in which the subfolder is named after the name of the controller. If a path is provided, it is assumed to be a path relative to the *views* folder.
4. Do not call render nor redirect. Rely on implicit template rendering instead.

- Example RCAV walkthrough goes here: render text: "hello world", status: 200

## View Templates, Partials, and Layouts

- Example of explicit render of a view template goes here
- Instead of explicitly rendering a template or redirect code, actions can choose to omit any explict invocation whatsoever, and instead rely on Rails' convention-based, *implicit template response* functionality.

- An *implicit tempate response* allows the Rails framework to automatically render the appropriate ERb file template. This will only succeed if the filename of the ERb template to be rendered can be inferred by the framework, by following very specific path filenaming conventions which are based on the combination of both the action method name and the name of the containing controller class.
- Diagram of implicit render conventions here
- Inside an action method, the hash provided by the *params* method is often very helpful. The hash enables you to inspect the incoming HTTP request data. Specifically, the params hash can provide:
  1. URL path components
  2. URL query string parameters
  3. data submitted from an HTML form

**Key Point**

Every action must generate an HTTP response. You only have three choices:

1. Call **render**
2. Call **redirect_to**
3. Rely on an implicit template-based response.

- ERb Templates are primarily HTML files that can use *embedded Ruby*. Special markup tags <% ... %> are used to surround Ruby logic and expressions. Embedding Ruby into an HTML template file allows looping structures, method calls, and Ruby expression evaluation.
- ERb templates can choose to inject the result of a Ruby expression into the resulting HTML stream by using the equal-sign version of the embedded Ruby tags: <%= ... %>.
- ERb templates cannot normally call methods defined in the controller. Data prepared in the controller for the view to use are held in Ruby instance variables instead. Instance variables survive the scope of the action method, and are made available to the corresponding view template.
- Use implicit render with @var for hello world example.
- Show HTML <a> links and <img> tags on the page.

- Example of link_to and image_tag helpers.
- Example of header and footer in application layout.
- Example of partials here.
- Methods can be defined inside of views, but Rails has a better place for Ruby methods that are to be used inside of views: app/helpers.

## Capturing User Input

- The primary distinction between a “web application” and a simple “website” is that applications can interact with the user.
- Many websites are *static:* they look the same every second of every day, and the user can read the content on the site, and that’s about it.
- Web *applications*, on the other hand, interact with the user, display content that can change every minute, might allow users to create personal accounts, and in a myriad of other ways, are engaging for the user.
- Despite the wide variety of interactive applications on the internet today, all of them end up receivng input from the user in a limited number of ways:

**Key Point**

There are 3 ways for users to send input to your app:

1. URL path
2. URL querystring
3. Form submission

Rails always puts all incoming user input into the ***params*** hash.

1. By specifying a URL. The URL is the primary way for the user to choose what page they want to see next.

   **Example:** http://www.mysite.com/teams/chicago

   The *path components* of a URL are those parts that appear after the domain name and are separated with slashes. In this example, the two path components are the strings *teams* and *chicago*.

2. By providing extra parameters in the *querystring*. This is done by appending a a question mark, followed by a list of key-value pairs.

**Example:** http://www.mysite.com/teams?sort=name&limit=25

Here, the query string provides two key-value pairs: the key *sort* has a value of *name*, and the key *limit* has a value of *25*. Since a URL is a string, the key-value pairs are always strings as well, even if they look like integers to the human eye.

3. By filling in a form and clickin a "submit" button (or equivalent). Forms with only one text field, like a search form, can often omit a physical button; the browser will interpret a keypress of the Enter key the signal to submit the form.

**Example**: screenshot of a form goes here

- Rails makes it easy for the developer to inspect the input coming from the user, regardless of whether the input is received by URL, querystring parameter, or form data.
- Rails makes available to all controllers and views a predefined Ruby hash that always contains any user input data.
- The hash is accessible by calling an inherited method named *params* (it's short for *parameters)*. This method returns a hash of key-value pairs, making it easy to use any user input into any controller action method.
- The *params hash*, as it is called, is used in controller code to enable the action to respond *dynamically*, customizing the output depending on the incoming data.
- The params hash is technically an object of class *HashWithIndifferentAccess*, which derives from the built-in Hash class. This subclass extends the Hash class in one, very important way: although keys and values are always strings in the params hash, values can be retreived by providing a string key ***or*** a symbol key. This means that the following two lines of code are equivalent:

  **params["limit"] # => "25"**
  **params[:limit] # => "25"**

  Most Rails developers opt to use a symbol key, because it's less typing and more performant. (Footnote needed: memory vs. speed in symbols vs. strings).

  **Remember:** Normal Ruby hash lookup is normally based on exact object identity, but the params hash in Rails is an exception, accepting either a symbol or string key to perform the lookup.

- The values in the parms hash aren't restricted to strings; often the value associated to key is another hash! This is common when using HTML forms: the key identifies to form, and the value is another hash, in which the keys correspond to field labels and the values

correspond to user-provided input:

**Example params hash with nested form data:**
{ “user” => { “name” => “Cookie Monster”, “fur_color” => “blue” } }

- The params hash can be easily seen by viewing the **Rails server log** during the incoming request.
- Browsers send form data as a single list of key-value pairs, so the only way to achieve a nested hash like the above example is to use a specific convention when composing key names. If square brackets are used in HTML input element names, Rails will use that as a signal that a nested hash should be generated for similar elements:

**Example of form element names and resulting server log entry goes here**

# Models

- A model is something that represents a real-world thing.
- It’s not the actual real-world thing; it just models it.
- Models let us use the same vocabulary in our software as we do in the real world.
- Users, Movies, Flights, Friends - these are all examples of things that are models.
- In Rails, a model is a Ruby class, that happens to live in the app/models folder.
- Again: A *model* is just fancy name for a normal Ruby class that has the particular responsiblity of representing a real-world thing in the vocabulary of our application.
- In Rails, models are database-backed.
- In Rails, we don’t access the database directly. Instead we use manipulate Ruby objects - instances of our models - and the database will be kept in sync automatically.
- Models provide the data our application will need to operate.
- For most Rails apps, most of the views are just templates for displaying model data.

- In Rails, models are database-backed: they can save the contents of themselves to the database, and they can be rehydrated again from the database when needed.
- A database is where we can store a lot of related data together in one place.
- A relational database is a lot like an Excel notebook.
- A relational database is comprised of a series of tables.
- Each table consists of a set of columns and rows.
- A table corresponds to a single spreadsheet.
- **Diagram of three related tables just to give the overall idea**
- A table's columns correspond to a model's attribute definitions
- A table's rows correspond to specific model instances.
- Each row can be hydrated into a specific model instance.
- Models can be related, or associated, to other models. A particular Flight will be associated to a particular set of Passengers. A Movie will be associated to a particular Director.
- Every model has an associated database table.
- Models automatically sync to their underlying data table based on an established naming convention that ties a Ruby model class name to a database table name.
- Model names in Ruby as *singular*, and the underlying database table name is *plural*.
- A model with the class name Team will expect a database table named teams.
- Columns of the underlying table are accessed by using same-named Ruby attribute accessors on model instances.
- Implementing a model in Rails requires two steps: a Ruby class definition, and a database table definition.
- Database table definitions can be provided in Ruby. We use *migration* classes to describe the definition of a single table.
- Migrations allow us to write Ruby code to describe a table, and Rails will translate our Ruby into SQL DDL automatically.
- Migrations can define tables, columns, indexes, and database-level constraints.

- Writing a migration class from hand can be tricky at first.
- Rails provides a *model generator* to help us get a headstart at writing botht the migration class and the model class.
- The generator writes code that you should inspect before trusting it.
- The generator does not execute the migration file
- In order to run the migration file and actually create the table, we use the rake command.
- rake db:migrate is the *rake task* that will run any migrations that have not yet been executed.
- **Sidebar on generators and getting help on syntax**
- **Example model generation here**
- You can find out if your migration and model are working properly by using the *Rails console* utility
- **Example of rails console here**
- You can rollback the most recent migration with **rake db:rollback**
- You can roll forward again with **rake db:migrate**

---

# CRUD: The Four Pillars of Every App

*Legend: Model, instance, Relation*

- There are four fundamental actions we can do that affect the data in our app.
- Let's use our model class as an example.
- In Rails, models represent the data our app can work with.
- Each model class is an object representation of a database table.
- Each model *instance* is an object representation of a single database row.

- CREATE: We can create a new row in the table by creating a new in-memory instance of our model, setting its attributes, and then calling the *instance.save* method to create a new row in the table.
- CREATE: We can also set the attribute values at the same time we create a new instance by passing a hash to *Model.new*, and then calling *instance.*save as usual.
- CREATE: We can set the attribute values and save the instance all at the same time with the *Model.create* method.
- READ: We can find out how many rows exist in the table with *Model.count*, get the first and last rows with *Model.first* and *Model.last*, get all of the rows back with *Model.all*, or just the first *n* rows with *Model.limit*.
- READ: We can also retrieve an array of rows that match a given set of criteria with the *Model.where* or *Relation.where* method.
- READ: We can ask the database to sort the data for us by using the *Model.order* or *Relation.order* method.
- READ: We can chain these methods together: *Model.order("name asc").limit(10)*.
- UPDATE: Once we have retrieved a database row into a hydrated model instance, we can modify its attributes, and then call *instance.save* to update the the row in the database.
- UPDATE: We can also update a set of rows en-masse by calling *Model.update_all* or *Relation.update_all*.
- DELETE: Given a model instance, we can delete the underlying table data by calling *instance.destroy*.
- DELETE: We can also update a set of rows with *Model.destroy_all* or *Relation.destroy_all*.
- SQL fragments are sometimes required as arguments to the *where, update_all,* and *destroy_all* methods.
- You now know how to do 90% of all of the data management you'll ever need to do in a Rails app.

# RESTful Conventions

- Now that we understand RCAV, we can actually build any website we can think of.
- Most database-backed web apps follow certain recurring patterns
- The primary pattern: users want to manage a finite collection of *resources*
- Resources represent the value proposition of the application
- Think of your app in terms of resources, not web pages
- Rails expects you to embrace the resource concept
- Users managing a resource is equivalent to web-based CRUD.
- Web-based CRUD maps nicely to a controller with seven specific actions.
- Of the 7 actions: four render views (index, show, new, edit), and three redirect.
- Why 7 instead of 4? There are two “READ” actions: index and show; plus two that display forms: new and edit.
- Rails embraces a particular web philosophy known as “REST” in which all HTTP methods are available for use, not just HTML-supported GET and POST.
- REST pairs up a URL with an HTTP method in order to convey CRUD semantics.
- There are two dominant URL patterns for each resource: /products for the list, and /products/{id} for a specific product.
- Two extra URLs are used to show the forms: products/new and products/{id}/edit
- Rails enables the use of PUT and DELETE even though HTML does not natively support them.
- Most controllers which manage a user-contributed resource will end up with these 7 actions and four views.
- Rails provides generators to give you a jumpstart on writing these actions and views.

- When a form is intended to represent the data for a new or existing model instance, form_for is easier to use than form_tag and generates special hidden fields that will automatically simulate the PUT method when needed.
- form_for provides a block variable that makes it much easier to draw individual form elements when those elements are tied to a model attribute or column.
- Routes can be given names.  Named routes generated view helpers that simplify code that displays forms or links.
- Named routes are the preferred alternative to hardcoding URL paths into views.