Programming Fundamentals (Part 6) - Classes

In our last several lessons, we learned some of the programming fundamentals that we'll need when writing applications. The next step is to combine these new concepts with real data from a SQL database - that's where the real fun begins!

But before we can get there, we need to briefly cover one more programming concept: *Classes*.

What is a Class?

A *class* is actually short for classification, which better describes its role in programming - the classification of a piece of data. Let's look at some of the data types we started with last week.

```
"Hello, world!"

"Tacos are delish."

"This is the way."
```

What type of things are these? Or, put another way, how would we classify each of these pieces of data? These are all *strings*. They come from the *String* class. Although they are all different, they are all from the *String* class - we say they are *instances* or *objects* of the String class.

Practically speaking, what that means is that they have some shared behavior. For example, although the content is different, each has a method `.to_upcase` and knows what to do when that method is called:

```
"Hello, world!".upcase # "HELLO, WORLD!"

"Tacos are delish.".upcase # "TACOS ARE DELISH."

"This is the way.".upcase # "THIS IS THE WAY."
```

The same is true for the other data we learned about:

```
5
```

```
2
```

Both `5` and `2` are integers - instances of the *Integer* class. And the *Integer* class gives them both a shared method `.to_f` which converts an integer into a different class: *Float*.

```
5.to_f # 5.0

2.to_f # 2.0
```

We can ask any object what kind of thing it is (i.e. what's its class):

```
"Hello, world!".class # String



5.class # Integer

5.to_f.class # Float

5.to_s.class # String



2.class # Integer

2.to_f.class # Float

2.to_s.class # String
```

More examples:

Here we have arrays, which, you can guess are instances of the *Array* class. And some of the shared methods that we've already explored include `.sort` and `.uniq`.

```
[4, 8, 15, 16, 23, 42]

["Rachel", "Monica", "Phoebe", "Ross", "Chandler", "Joey"]
```

And lastly, our hash data, which are instances of the *Hash* class.

```
{ "color" => "purple", "number" => 17, "computer" => "Apple" }
```

```
{ "name" => "Ben", "location" => "Chicago, IL", "status" => "Staying
warm!" }
```

Code-along

Let's get some practice.

Go to this lesson's repository and open it in Ona. You'll notice a couple things are different than the previous *Programming Fundamentals* repository. First, there's some load time when it first opens in Ona - that's because we're installing several libraries and running a few files as setup for the following lessons. Second, the file structure in the left panel is much bigger than before. We're now in a complete Rails application repository which has A LOT of extra directories and files. We won't get into what that all means right now - we'll learn more about it as we go through the course. Don't worry - all in due time.

Let's look inside the `code-along` directory and open the file `0-classes.rb`. Here's another difference - to run this file, we'll stay at our main directory in terminal which should read `/workspace/ruby-and-sql`. From there, we'll use a command `rails runner code-along/0-classes.rb`. This is similar to when we used `ruby` and the file name, but now we're also loading in other libraries that we'll need. Again, don't worry too much about it now, just see if you can run the file without error (there shouldn't be any output).

Ok, in the file, let's create a few strings:

```
my_favorite_food = "ice cream"

puts my_favorite_food



your_favorite_food = "tacos"

puts your_favorite_food
```

When we run the file with `rails runner code-along/0-classes.rb` we should see the output

```
ice cream

tacos
```

Let's check what their classes are by adding this code:

```
puts my_favorite_food.class

puts your_favorite_food.class
```

At this point, hopefully no surprise, the output is

```
String

String
```

These are instances of the *String* class. And because they're from the *String* class, they have all of the shared String methods like `capitalize` and `upcase`. How about we capitalize them:

```
puts my_favorite_food.capitalize

puts your_favorite_food.capitalize
```

And when you run the file, you'll see both strings are capitalized. Our complete output from running this file is now:

```
ice cream

tacos

String

String

Ice cream

Tacos
```

This might be interesting and clarify some of our earlier lessons, but how do we put this knowledge to use?

Custom Classes

Classes are kind of like factories or templates. Think of a car factory - it can build cars that are all similar but slightly different - each car has its own VIN (an identification number) and perhaps variations in color, but the cars all have shared behavior like being able to turn on and drive.

The built-in ruby classes are useful, but often we also need our own custom factories to build things that our application needs. For example, let's say we want to build virtual dogs and give them all some shared functionality. Go ahead and add the following code to our `0-classes.rb` file.

```
class Dog



  def speak

    puts "woof!"

  end




end
```

This is a `Dog` class with a method `speak`. The code above is like building a factory - we're now able to create dogs, but we haven't yet. If you run the file (`rails runner code-along/0-classes.rb`), nothing has changed. Now let's use our class to create a new dog instance:

```
lassie = Dog.new

lassie.speak
```

```
rover = Dog.new
```

```
rover.speak
```

We've built 2 different dogs that both can "speak" (i.e. output "woof!"). And the output you now see should be:

```
woof!
```

```
woof!
```

Interestingly, you didn't need to type `puts` in our file. That's because the method itself already does that.

Defining a Class

When we define a class, we capitalize the name and use the singular - this is an important pattern. If you think again about a Car factory - a Car (singular) factory produces cars (plural). In the same way, our Dog (singular) class produces instances of dogs (plural). And it's capitalized to differentiate from all the variables that will inevitably be in our code (like `lassie` and `rover`). Don't forget about this pattern - if you do, expect to see some errors later on 😉.

That's about as far as we're going to take this idea for now. We could spend weeks on this topic, but the only reason it has any meaning at the moment is in light of our next topic - models.

Migrations

The CRM domain model will be helpful in this unit.

Back in the SQL unit, we learned how to write SQL `CREATE TABLE` statements to modify the database structure. Here in our ruby application, we don't just need a table, we also need the corresponding model to interact with the table data. For example, we want a table for the company data in our domain model and a `Company` model to interact with that data.

Generating Models and Migrations

If you haven't already, go to this lesson's repository and open it in Ona. The file for this lesson is `1-migrations.rb` in the `code-along` directory. Note - we won't end up running this file; the code from this unit will be commands in terminal and code in other files. The code-along file is only for informational purposes.

We need to do two things:

- We want to create a table that can hold data about companies
- We want to create a `Company` model and have it connect to this table

To accomplish the first step, we'll need a *migration* file. A migration is just Ruby code that, when executed, will talk to the database and add a new table into it. And for the second step, we just need to create a file for the `Company` class.

We could create these files manually, but Rails (the ruby application we've been working with) comes with several commands to help generate files that we might need. From terminal, type the following command and hit enter:

```
rails generate model Company
```

The first 3 words in this command will always be the same. The last word will be the name of the model (singular and capitalized) from the domain model that you want to add to the application.

The output you'll see indicates that 2 files were created by this command.

```
      create    db/migrate/*_create_companies.rb

      create    app/models/company.rb
```

The first is the migration file that will modify the database structure - there will be a long set of numbers where the asterisk (*) is in the file name above. The second is the model class file. Let's quickly look at the model since that's easier and we won't need to do anything to that file right now. If you open `company.rb` in the `app/models` directory, you'll see this code:

```
class Company < ApplicationRecord

end
```

This is the model class and is responsible for connecting `Company` ruby code with the `companies` table. At some point, we may choose to add custom code here, but that's later. You can close this file.

Now open the migration file which has a bunch of numbers and then `_create_companies.rb` as its name. It's in the `db/migrate` directory.

Creating Database Tables

Migration files modify the database structure, so you can imagine that we don't want to run them more than once - it wouldn't make sense to create a table and then try to create it again. The order of the files also matters since it wouldn't make sense to add a column to a table unless the migration that created that table was already executed. That's where the numbers in the file name are useful - they're the timestamp of when this file was generated. The timestamp keeps the order of the files and also ensures file names are unique.

Check out the code.

```
class CreateCompanies < ActiveRecord::Migration[7.0]

  def change

    create_table :companies do |t|



      t.timestamps

    end

  end

end
```

We haven't seen this code before, but it's part of the `ActiveRecord` library and you can probably guess what it does - it writes and executes a SQL `CREATE TABLE` statement that will add a table named `companies`. When we ran the generator, we used the singular model name, but this library is smart enough to know that the table name should be pluralized - cool, right?!

Inside this file, we need to add the columns that we want this table to have. The format for adding a column is `column_type :column_name`. Looking back at the domain model, the `companies` table has a `name` column, a `city` column, a `state` columns, and a `url` column. It makes sense that those would all be strings, so let's add them to the code.

```
class CreateCompanies < ActiveRecord::Migration[7.0]

  def change

    create_table :companies do |t|

      t.string "name"

      t.string "city"

      t.string "state"

      t.string "url"



      t.timestamps

    end

  end

end
```

The `t.` is just part of the ruby code that tells this table object to add a column - another deep dive we won't get into here.

We don't have any numeric columns, but if we did, we could use the `integer` column type, for example `t.integer "age"`. There are several column types, but these are the 2 most common.

Running Migrations

Now that our migration file has the table columns, we're almost done, but nothing's actually happened yet! Simply defining the migration file won't physically alter the database. If you check the `db/schema.rb` file, you won't see the `companies` table there. Same if you hop into `sqlite` and check `.schema` there. We need to run the migration file in order to execute the code, make a live connection to the database, and insert the new table. At the terminal prompt, run the following command:

```
rails db:migrate
```

You should see some output confirming that it was successful. You can also check the schema again to verify the table now exists.

As explained above, this migration file only runs once - it wouldn't make sense (and in fact would raise an error) if we tried to create another `companies` table. So, if you try rerunning the migrate command from above, nothing will happen because Rails is tracking which files have run and knows there aren't any new files.

That said, if you need to make a change, you can't just edit the migration file since you can't rerun it. We'll learn some advance ways to modify the tables later, but for now, you'd need to actually delete the entire database and start again. Fortunately (or unfortunately), doing so is very easy, but be aware, this will delete ALL of your data from all of your tables. Since this is all development data so far, it might be ok, but it's still a process to recreate data. If you want to proceed, find the sqlite file `db/development.sqlite3` and delete it. You can then modify the code in the migration file and migrate again with the command:

```
rails db:migrate
```

Contacts

With the `companies` table in place, we'll be able to add rows to the table - we'll do that in the next unit. For now, let's repeat the steps to create a model and table for `Contact`.

Step 1: generate the model and table.

```
rails generate model Contact
```

Step 2: modify the migration file `db/migrate/*_create_contacts.rb` with columns matching the domain model.

```
class CreateContacts < ActiveRecord::Migration[7.0]
```

```
  def change

    create_table :contacts do |t|

      t.string "first_name"

      t.string "last_name"

      t.string "email"

      t.integer "company_id"



      t.timestamps

    end

  end

end
```

Note that this table has a foreign key column because each contact is associated to a company. We designed that relationship back in our domain model unit. The foreign key is an integer and will reference the `id` column in a row in the `companies` table that we just created.

Step 3: execute the file to actually create the table in our database.

```
rails db:migrate
```

That's it, we now have models and tables for our `Company` and `Contact` entities in our domain model.

Lab

Time for a lab to practice - in this lab we'll add the other tables in our core domain model. Instructions are in the file `1-migrations.rb` in the `labs` directory.

Models

Now that some programming fundamentals are in our tool belt and we know a bit about classes, we're ready to connect our ruby code to our SQL database.

Why?

It's a good question - why do we care?

1. It's easier.
   SQL isn't really a programming language - at least not one that you can build an application with. It's a critical part of the technical stack (the technologies used in an application), but there's no easy way to expose SQL data to a user or create business logic around how data gets created. For that, we need a programming language like ruby. We could figure out how to connect the 2 technologies and write raw SQL statements (and sometimes we do), but then we would need to do a lot of context switching. It's nicer to just focus on 1 language (e.g. ruby) and worry less about raw SQL code.
2. It's more consistent.
   Along the same lines, if we can just work in our main application code, we can build a team and focus our resources around that expertise. There's an organizational benefit to doing so. Perhaps we need 1 or 2 SQL experts, but the majority of developers can just focus on the application layer.
3. It's more flexible.
   Lastly, sometimes we want to switch the underlying database technology. In fact, in this course we're using sqlite in our development, but when we eventually want to ship our code to production (i.e. make it live for users), we'll want to upgrade to an enterprise version of SQL (e.g. postgresql). If we were writing raw SQL, our code might need to change when the database software changes. But if we're using ruby to talk to our database, very little code will need to change to accommodate the new software.

ActiveRecord (ORM)

So how do we connect ruby code with SQL? We use an ORM, Object-relational mapper. Every programming language that needs to talk to a database has an ORM - in ruby, we'll use a library called `ActiveRecord`. It's a bridge that gives us a robust but easy-to-write ruby code that will translate to SQL without us even thinking about it.

Using ActiveRecord, we can create custom classes called *Models* that are dedicated to representing a single table in the database. For example, an `events` table in our database would have a matching `Event` model in our code. And instances of the `Event` model represent rows in the `events` table. Let's see it in action.

Go to this lesson's repository and open it in Ona. Let's open the file `2-models.rb` from the `code-along` directory. There's a bit of code already in the file, we'll come back to that later. There are some code comments here and we'll follow them as we learn about models.

Domain Model

Since models represent tables in the database, we need a domain model to work with. We'll use the CRM domain model from our domain modeling lesson:

- Wireframes & User Stories
- Domain Model (scroll to the bottom for the final structure)

From the previous lesson on migrations, this application now has several tables from the domain model: `companies`, `contacts`, `salespeople`, and `activities`. There are a couple ways to check them out.

First, we can open our sqlite database just like we used to. The sqlite database file is in the `db` directory. To open it from terminal, run:

```
sqlite3 db/development.sqlite3
```

You'll be at the sqlite prompt `sqlite>`. Use the `.schema` command to see what tables are in the database - you should see our 2 tables and their respective columns. Don't forget to `.exit` afterwards - we'll need to be back at the usual terminal prompt to run our ruby file.

Next, we can also check a ruby version of the schema. In the `db` directory, open the `schema.rb` file. This is ruby code (part of the ORM library) that mirrors the SQL schema. Again, you'll see the 2 tables that are already part of the database and their columns.

To interact with these tables, each has a corresponding model. In the `app/models` directory, open up the files `company.rb` and `contact.rb`.

```
class Company < ApplicationRecord

end

class Contact < ApplicationRecord

end
```

They're empty, but these model classes are the building blocks of our application.

Creating Data

So the tables exist, but what data is in them? We could go back into sqlite and use a `SELECT` statement to see the data. But let's start using ruby.

In the `code-along/2-models.rb` file, add the following line of code:

```
puts "There are #{Company.all.count} companies."
```

Then run the file with `rails runner code-along/2-models.rb`. The output should be

```
There are 0 companies.
```

`Company.all.count` is calling a method on the `Company` class which is talking to the database and asking for a count of the rows in the `companies` table. In fact, under the hood that `.all` method writes raw SQL that looks like

```
SELECT * FROM companies
```

And `.all.count` similarly writes SQL like

```
SELECT COUNT(*) FROM companies
```

To prove it, open up the `log/development.log` file and scroll to the bottom. There's some noise there, but if you look closely you'll see that a `SELECT COUNT(*)` query was performed. (Side note: this log file tracks all of the application activity and is often used by development teams when trying to debug a tricky issue - it's very useful history.) If you want, you can hop back into the sqlite prompt and run the same SQL there too.

Ok, now we know the `companies` table is empty. Let's insert a new row. First, let's build a new instance of the Company class:

```
apple = Company.new
```

This is similar to how we built a new dog from the `Dog` class.

Next, let's assign the company's attributes one at a time:

```
apple["name"] = "Apple Inc."
```

```
apple["url"] = "https://apple.com"

apple["city"] = "Cupertino"

apple["state"] = "CA"
```

These attributes match the columns in the table `name`, `url`, `city`, and `state`.

Last step, we want to actually save (i.e. `INSERT`) it into the database:

```
apple.save
```

All together, the code looks like this:

```
apple = Company.new

apple["name"] = "Apple Inc."

apple["url"] = "https://apple.com"

apple["city"] = "Cupertino"

apple["state"] = "CA"

apple.save
```

How do we know if it worked? The same ways we checked if the table was empty before. First, let's add another line of code checking the row count:

```
puts "There are #{Company.all.count} companies."
```

When you run the file now, you should see

```
There are 0 companies.

There are 1 companies.
```

Not grammatically correct, but at least it seems like our insert worked. You can also check the log file to see if the SQL insert statement is there. And last, you can hop into the sqlite prompt once again and check the results of a SQL select statement. We're going to stop referencing these checks and trust the ruby code, but feel free to use them whenever you feel they're useful.

Let's insert another row and another count. We'll leave the `url` column blank this time (we'll update it later).

```
amazon = Company.new

amazon["name"] = "Amazon.com, Inc."

amazon["city"] = "Seattle"

amazon["state"] = "WA"

amazon.save



puts "There are #{Company.all.count} companies."
```

The output is now

```
There are 0 companies.

There are 1 companies.

There are 2 companies.
```

There's another way to assign attributes that you may see if you're doing any googling. We'll start with a new company instance:

```
tesla = Company.new
```

And we'll assign the attributes:

```
tesla["name"] = "Tesla, Inc."

tesla["url"] = "https://tesla.com"
```

```
tesla["city"] = "Palo Alto"

tesla["state"] = "CA"
```

We do this so often that the `ActiveRecord` library gives us a shortcut:

```
tesla.name = "Tesla, Inc."

tesla.url = "https://tesla.com"

tesla.city = "Palo Alto"

tesla.state = "CA"
```

Instead of using the hash syntax where we use square brackets and the hash key, we can use this dot-notation (`.` + the name of the attribute). These are called "setter" methods (because they set, or write, attribute data). They're no different than what we've been doing, but some developers prefer this dot-notation syntax. For now, we'll keep using the hash syntax, but you can use whichever you prefer. We'll also see some other ways to assign attributes later on.

And now just save the record:

```
tesla.save
```

All together, it looks like this (along with the count):

```
tesla = Company.new

tesla["name"] = "Tesla, Inc."

tesla["url"] = "https://tesla.com"

tesla["city"] = "Palo Alto"

tesla["state"] = "CA"

tesla.save
```

```
puts "There are #{Company.all.count} companies."
```

Reading Data

Now that there is data in the table, let's look at how to query data. We've actually already used a query: `Company.all`. The `.all` method unsurprisingly will return all records from the table. It writes a `SELECT *` statement from the corresponding table.

To filter down from all records, we can use the `.where` method with a hash of conditions. For example, if we want all companies in the state of California, we can use

```
Company.where({"state" => "CA"})
```

This returns an array of records. At a glance, the returned object is called an `ActiveRecord::Relation`, but if you look closely you'll see the square brackets `[]` which indicates this is an array - a special kind of array, but still an array. We can be more specific by using multiple conditions

```
Company.where({"state" => "CA", "name" => "Apple Inc."})
```

At this point, there should only be 1 "Apple Inc." company in our database (certainly only 1 that is in California). If we want to pull out that object from the array, we use its index `[0]`:

```
company = Company.where({"state" => "CA", "name" => "Apple Inc."})[0]
puts company
```

When we *know* there's only one row in our table matching the criteria, the use of `[0]` to find the first element in the array gets annoying. So, as with many things in ruby and rails code, there's a more succinct way:

```
company = Company.find_by({"state" => "CA", "name" => "Apple Inc."})

puts company
```

The `find_by` method is the same as the above, but it reads a little nicer - it's obvious in the code that we're expecting only 1 result from our query.

If you add this code to the code-along file and run it, you'll see something a little odd:

```
#<Company:0x00007f785cd9cc00>
```

The characters you see will be different, but it will look similar. This is the identifier of where in the computer's memory this object is being held. It's not very useful to us - it would be great to see the object with the underlying data from that row in the table. To do so, we'll use `puts company.inspect` which will "inspect" or open up the object. Once we make the change, the output is more useful.

```
#<Company id: 1, name: "Apple Inc.", url: "https://apple.com", city:
"Cupertino", state: "CA", slogan: nil, created_at: "2022-02-01
09:53:44.087974000 +0000", updated_at: "2022-02-01 09:53:44.102738000
+0000">
```

Note, your output will be slightly different - the `created_at` and `updated_at` timestamps will obviously reflect when you ran your code. And the `id` may be different depending on how many times you've run the file.

So what is this thing? It's the row from our companies table that we queried for - the one with the name "Apple Inc." and the state "CA". Look a little closer at it and you may notice that it sort of looks like a hash with key-value pairs! If we check its class (`puts company.class`), we'll find out that it's not a *Hash* - it's a *Company*, meaning it comes from our custom Company class. That might not be surprising considering when we inserted this row, we used `Company.new`. But it does behave like a hash in many ways.

Now that we have an individual row from the database, let's read its individual values. Imagine we want to create a sentence like "Visit the Apple Inc. website at https://apple.com". To read specific attributes (i.e. column values), we'll use hash syntax:

```
puts company["name"]

puts company["url"]
```

This outputs

```
Apple Inc.

https://apple.com
```

Similar to when we were assigning attributes, if you do some googling, you may see it written slightly differently using dot-notation:

```
puts company.name
puts company.url
```

These are called "getter" methods (because they get, or read, attribute data) and, again, some developer prefer them. But we'll keep using the hash syntax:

```
puts "Visit the #{company["name"]} website at #{company["url"]}"
```

Updating Data

We can now create data and read data. Another common behavior is modifying an existing row in the database. We actually already know the code for updating - individual assignment and `.save`. Let's update the `url` column for Amazon.

In the code-along file, we need to first query to find the Amazon row.

```
amazon = Company.find_by({"name" => "Amazon.com, Inc."})
```

And now we can assign a value to the `url` column and save the row.

```
amazon["url"] = "https://www.amazon.com"

amazon.save
```

The `.save` method is interesting here. Previously, we used it to insert a new row. But here, we're using it to update an existing row. If you open up the `log/development.log` file, you'll see the corresponding SQL is an `UPDATE companies` statement. If we were writing our own SQL, we would need a lot of logic to determine if we should use an `INSERT` or an `UPDATE`, but the ORM takes care of it for us and we can write some simple ruby code. Lucky us!

And that's it, we've updated the record. We can verify by pulling the row back out of the database just like we did before:

```
company = Company.find_by({"name" => "Amazon.com, Inc."})

puts company.inspect
```

And now you'll see that `url` is updated.

```
#<Company id: 1, name: "Amazon.com, Inc.", url: "https://www.amazon.com",
city: "Seattle", state: "WA", created_at: "2022-02-01 09:53:44.087974000
+0000", updated_at: "2022-02-01 10:03:44.102738000 +0000">
```

Note that the `updated_at` timestamp has changed also since it reflects the last time this row was edited.

Deleting Data

Occasionally, we need to remove a row from the database - maybe due to an error when inserting or maybe because a record is no longer relevant to our application. Whatever the reason may be, there is a method `.destroy` that will permanently delete a row.

Let's first insert a new row that has a mistyped name:

```
company = Compan.new

company["name"] = "Tesler"

company.save
```

We have it already stored in this variable, but let's pretend that we need to query it from the database since that's a more common use case.

```
tesler = Company.find_by({"name" => "Tesler"})
```

Now we can delete it:

```
tesler.destroy
```

Adding a few `puts Company.all.count` lines of code before and after the `.destroy` will clarify that there were `4` companies and now there are `3`. We can also check out the log file to see the corresponding SQL `DELETE` statement. You'll notice that just as we instructed in the SQL unit, there's a `WHERE id = ...` clause to ensure only the intended row is deleted.

Resetting the database

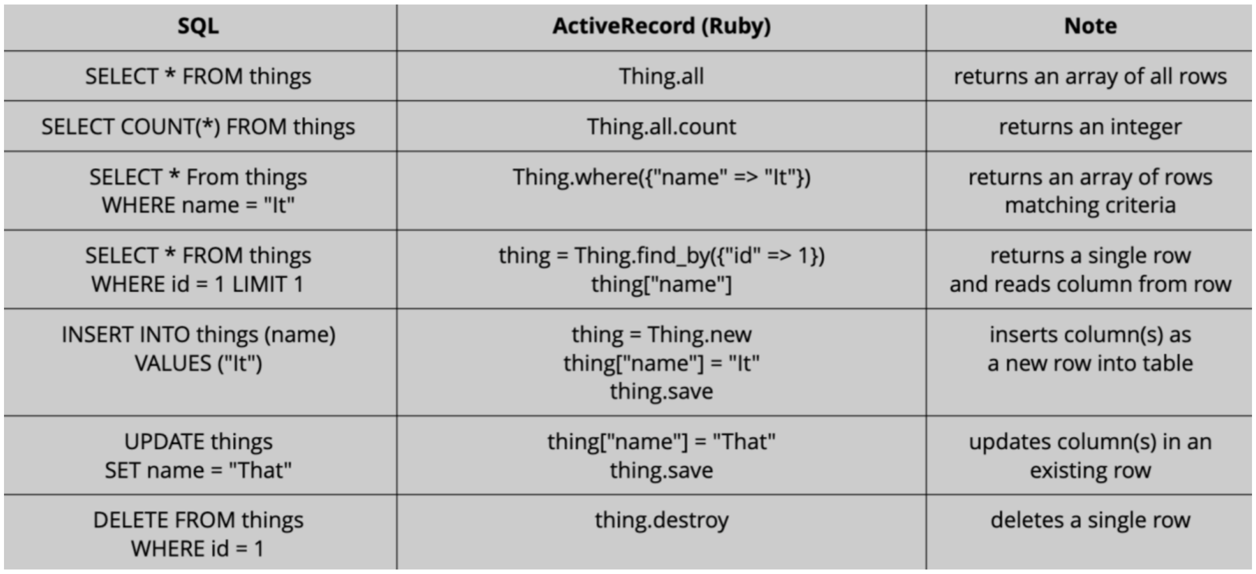
At some point, you may have realized that we've executed this `2-models.rb` file several times and yet the table hasn't grown - it still only has 3 rows. How? Shouldn't the table grow by 3 new rows each time the file runs? Where are all the previously inserted rows?

Look back up at the top of the file at the code `Company.destroy_all`. That line of code might make more sense now. Each time this file runs, the first thing that happens is this line of code deletes all the rows in the `companies` table. Doing so ensures you can rerun this file as many times as you want without creating duplicate rows. This also explains why the `id` values for your rows may differ - they will auto-increment and never be reused, so if you have 3 rows and run the file 3 times, the `id` values will be `7`, `8`, and `9`.

Just be careful!!! Running this file deletes **all** of the rows in this table - not just the rows created in this file. Eventually the table will have more data created outside of this file and then we probably should avoid running this file. It's only safe for now because we're starting with an empty table.

TL;DR - The `.destroy_all` method is dangerous, so use it with extreme caution.

Cheatsheet



SQL	ActiveRecord (Ruby)	Note
SELECT * FROM things	Thing.all	returns an array of all rows
SELECT COUNT(*) FROM things	Thing.all.count	returns an integer

SELECT * From things
WHERE name = "It"
	Thing.where({"name" => "It"})	returns an array of rows
		matching criteria

SELECT * FROM things
WHERE id = 1 LIMIT 1
	thing = Thing.find_by({"id" => 1})
	thing["name"]
		returns a single row
		and reads column from row

INSERT INTO things (name)
VALUES ("It")
	thing = Thing.new
	thing["name"] = "It"
	thing.save
		inserts column(s) as
		a new row into table

UPDATE things
SET name = "That"
	thing["name"] = "That"
	thing.save
		updates column(s) in an
		existing row

DELETE FROM things
WHERE id = 1
	thing.destroy	deletes a single row

Inheritance

One final note - how do our models know about all these methods? Remember back in the classes unit, we had to tell our `Dog` class how to speak with a method. Where are the `.all` and the `.where` and the `.find_by` and the `.save` methods?

If we look again at the code in the model, there's a tiny bit of code `< ApplicationRecord` - tiny but powerful.

```
class Company < ApplicationRecord
```

```
end
```

As a programming concept, we say that our `Company` class *inherits* from the `ApplicationRecord` class (which is part of the `ActiveRecord` ORM library we're using). Inheritance means that our model, which appears to have no custom code in it, gets all the methods and functionality defined in `ApplicationRecord`. If we were to go look at the source code for that class, we would find the methods we've been using.

For models that correspond with a database table, that tiny bit of inheritance code takes care of connecting the model with the database table and providing all of the ORM methods that help us interact with the table.

Inheritance is another topic that we could spend weeks on, but isn't that important to our current goals. We're just pointing it out here so that you don't think it's all magic and you can explore it more on your own if you so choose.

Lab

Time for a lab to practice. Instructions are in the file `2-models.rb` in the `labs` directory.

Associations

The next step in our ruby + SQL unit is understanding how to read data from multiple tables. Data from a single table is fine, but frequently we need the context from multiple tables to properly use the data in an application. For example, in our domain model we have a `contacts` table, but it's important to know the company that a contact is associated to - context matters.

In the domain modeling unit, we spent a good deal of time thinking about the relationships between tables. Now we want to introduce those relationships in our ruby code. As a prerequisite, be sure you've created a few rows in the `contacts` table (if you completed the previous lab, this is already done).

If you haven't already, go to this lesson's repository and open it in Ona. The file for this unit is `3-associations.rb` in the `code-along` directory.

To confirm there are at least a couple rows in both the `contacts` and `companies` tables, add the following code to our ruby file and then run it (`rails runner code-along/2-associations.rb`).

```
puts Company.all.count


puts Contact.all.count
```

Finding Records by Foreign Key

If we look back to the CRM domain model (scroll to the bottom for the final structure), the relationship that we're modeling is a *1-to-many* where a company can have many contacts and a contact belongs to a single company. To establish this relationship in the data, the `contacts` table has a `company_id` foreign key column which correlates with the `companies` table `id` column. To find all contacts for a given company, we write a query using the company's id.

For example, if we want all of Apple's contacts, we can first find Apple's `id` in the database:

```
apple = Company.find_by({"name" => "Apple Inc."})

puts apple["id"]
```

That value might be different in each database, but let's just imagine that the output above is `18` - meaning Apple's id in the `companies` table of this database is `18`. We could then write a query like this:

```
contacts = Contact.where({"company_id" => 18})
```

This is *ok*, but not great - a hardcoded value like an id is fragile. Again, `18` might not be the id for Apple - multiple developers might have different data in their databases so each would have to change this value to match their data. It would be better if this code was more dynamic and could adapt to the underlying data.

Actually, it turns out that's pretty easy. Instead of displaying the id and then hardcoding whatever you see in the output, we can just use `apple["id"]` in our query which will always work (assuming there's a row somewhere in the table for Apple Inc.).

```
contacts = Contact.where({"company_id" => apple["id"]})
```

Now it won't matter if the id is `18` or `1` or `99` - it will find the associated contacts regardless.

Next let's practice loops and display all of these contact records (you can reuse your code from the previous lab):

```
puts "Contacts at Apple: #{contacts.count}"

for contact in contacts
```

```
  puts "#{contact["first_name"]} #{contact["last_name"]}"

end
```

Lab

Time for a lab to practice. Instructions are in the file `3-associations.rb` in the `labs` directory.