Ruby Classes 2

Prerequisites

Ruby Classes & OOP

What you'll learn

- class variables
- private
- protected
- overriding
- super

OOP and classes are a big topic. This is an assorted list of things that were missing from part 1.

Class Variables

```
class Table
  @@next_table_id = 1
end
```

A variable prefixed with two ampersands “@@” is a class-scoped variable.

If you imagine the class as a factory, then a class-scoped variable is an attribute of the factory as a whole, rather than an attribute of a particular product.

In this case, it’s an attribute of the Table factory, not an attribute of a single table. I’d like the Table factory to keep track of the serial number for the next Table that rolls out the door.

Class Variables

```
Table.@@next_table_id

syntax error, unexpected
tCVAR, expecting '('
```

Class variables and instance variables have similar scopes. They're both private by default. That means you can't access them directly.

Ruby will always give you an syntax error if you try to directly access a variable that begins with an ampersand (@).

Class Variables

```
class Table
  @@next_table_id = 1

  def self.next_table_id
    @@id
  end
end

p Table.next_table_id
```

Just like with instance variables, to access a class variable you need to define a method. The method exposes the secret to the outside world (the world outside the `Table` class).

Instance variables require an instance method to be exposed.
Class variables require a class method. A class method is just like an instance method, but it's name starts with the world `self`.

Class Variables

```
p Table.next_table_id

t = Table.new
p t.next_table_id
```

undefined method ‘next_table_id’

To review method scope:

A class-scoped method is accessed via the class (`Table`). A class-scoped method can expose class-scoped variables (whose names start with `@@`).

An instance-scoped method is accessed via an instance of the class (`Table.new`). An instance-scoped method can expose instance-scoped variables (whose names start with `@`).

You can’t call a class-scoped method from an instance. That generates an “undefined method” error.

Similarly, you can’t call an instance-scoped method from a class.

Class Variables

```
class Table
  @@next_table_id = 1
  attr_reader :id

  def initialize
    @id = @@next_table_id
    @@next_table_id += 1
  end
end
```

Why would you want to use a class variable?

Say your `Table` factory stamps a serial number id on every table it creates. The `Table` factory doesn't want to give any two tables the same id. The numbers must be unique.

To make sure they ids are unique, once the factory stamps an id number, it increments it by one. So the first table with get id=0. The next table will get id=1. And so on.

Class Variables

```
t = Table.new
p t.id
```
=> 1

```
t2 = Table.new
p t2.id
```
=> 2

We can check if this works by creating 2 Table instances.

The first has id = 0. The second has id = 1.

So the `Table` *factory* (rather than any individual `Table` instance) is keeping track of some data. Imagine that there's a big sign inside the factory that says "The next table we create is going to have id=3". When the 3rd table is created, someone updates the sign to say the next id will be 4.

Class Variables

```
class Table
  @@next_table_id = 1
  attr_reader :id

  def initialize
    @id = @@next_table_id
    @@next_table_id += 1
  end
end
```

Another way to think about it:

Each table gets it's own `@id` instance variable. Each table has a different id. No two values are shared.

The table factory holds the `@@next_table_id` class variable. There's only one "next table id" in my program. And all the tables share it.

What do I mean when I say the tables all "share" that value?

Wi-Fi Enabled Tables

```
class Table
  @@next_table_id = 1
  attr_reader :id

  def initialize
    @id = @@next_table_id
    @@next_table_id += 1
  end
end
```

When I say all the table instances share the class variable, I mean that every table produced by this factory has built-in Wi-Fi.

Wat?

Check out the `initialize` method in our Table class. `initialize` is an instance-scoped method. It doesn't start with the word `self`.

Even though I'm inside an instance method (`initialize`), I'm using a class variable (`@@next_table_id`). Now only am I reading the class variable, but I'm also changing it's value.

Wi-Fi Enabled Tables

```
p Table.next_table_id

t = Table.new
p t.next_table_id
```

undefined method ‘next_table_id’

Didn’t I just say instances can’t access class methods a few slides ago?

In this code, I create an instance of a table, but the instance can’t access the class method `next_table_id`.

So which is it? Can a particular table affect the table factory: yes or no?

The answer is both. Instances don’t have access to class *methods*. A table instance can’t access a table factory method.

...

Wi-Fi Enabled Tables

```
class Table
  @@next_table_id = 1
  attr_reader :id

  def initialize
    @id = @@next_table_id
    @@next_table_id += 1
  end
end
```

But instance methods *do* have access to class *variables*. A table instance *can* access a table factory variable.

It's as if every table comes with built-in Wi-Fi. Why?

Because every table, even after it leaves the factory, can call back to the factory to preview what the next table is going to look like. Every table can expose factory secrets.

Let's see what that would look like.

Wi-Fi Enabled Tables

```
class Table
  @@next_table_id = 1
  attr_reader :id

  def initialize
    @id = @@next_table_id
    @@next_table_id += 1
  end

  def preview_next_id
    @@next_table_id
  end
end
```

Let's explicitly enable table wi-fi.

I've created a new instance method named `preview_next_id`.

The `preview_next_id` method returns the value of `@@next_table_id`. What does that mean? That means an instance of a table can preview what the id will be for a table that hasn't been created yet. Every table can now reveal a factory secret, internal knowledge that only factory workers would know.

Wi-Fi Enabled Tables

```
table1 = Table.new
p table1.id
p table1.preview_next_id

table2 = Table.new
p table2.id
```

Let's use this new feature.

First, we'll create a table named `table1`. `table1` has an id assigned to it. That id should be 1 because it's the first table we've created.

Using `table1`, I can ask to preview the next table id. It's as if this `table1`, a plain table now sitting in my house somewhere, can somehow send a message back to the table factory and get a response containing a preview of the next table that's about to be rolled out the door.

You can call this Wi-Fi. Or maybe it's a phone call. Or a fax. Or some sort of magical psychic connection. Either way, each table can now expose factory secrets.

To prove that the secret is true, we create another table: `table2`. Just as we expect, `table2` contains the id that `table1` predicted it would have. In this case, it should be 2.

Wi-Fi Enabled Tables

```
class Table
  @@next_table_id = 1

  def change_next_id(next_id)
    @@next_table_id = next_id
  end

  def self.preview_next_id
    @@next_table_id
  end
end
```

So each of my tables can pull back data from the table factory. But can they change it as well?

Yes, they can.

I've created an instance method named `change_next_id`. When I call `change_next_id` and pass it an argument, it changes the value of `@@next_table_id`.

It's as if I could press a hidden button in my table and magically affect the next table that comes out of the factory.

I've also changed `preview_next_id` from an instance method to a class method (`self.preview_next_id`). To check that this code really works, I'm going to call the Table factory directly to see if the data really changed.

Wi-Fi Enabled Tables

```
t = Table.new
p Table.preview_next_id

t.change_next_id 100
p Table.preview_next_id
```

Let's try this new feature out.

I'm going to create a new `Table` (named `t`). I'm going check in with the Table factory to preview the next table id. Like before, I should get a 2. `t` is table 1. The next table should be table 2.

Unlike last time, I'm going to tell the table factory that the next table id should be 100. But rather than call the factory, I'm going to do it with my magic wi-fi table, `t`.

After I'm done, I call the factory to confirm that they got my message. And they did. When I ask the factory to tell me what the next table id will be, they'll say 100.

What this "wi-fi feature" demonstrates is that class variables are *shared*. There's only one (in the "table factory"), but every table can access it *and* change it. It's still technically a "secret", but it's a poorly held one.

If we were imagining deities instead of factories, you could say each table has a mainline back to Tablos, the god of tables, where they send prayers and requests for favors. Or something.

This is certainly *not* being used as a secret social network where tables can

coordinate their conspiracy to take over the world. Not at all.

Why Not Class Variables?

```
class Table
  @@next_table_id = 1
  attr_reader :id

  def initialize
    @id = @@next_table_id
    @@next_table_id += 1
  end
end
```

There are many valid uses for class variables, like this table id generator we just created.

Or, instead of an id, you could keep a count of how many tables you've created in your program. Or keep a list of all the tables ever created.

In general, there are valid reasons for wanting to know what's going on back at the "table factory".

But class variables should be treated as exceptions rather than rules. Here are a few reasons:

First since class variables are "poorly held secrets", they suffer from some of the same evil that plagues global variables.

If you create 100 tables, it's probably not a good idea that one of those tables can make a change that affects the other 99.

Second, sometimes Ruby class variables can be very confusing to use. For example, they may not work the way you expect if you use inheritance. We won't get into those details.

Third, if you have a choice of using instance variables or class variables in a project, you're probably better off using instance variables.

For example, a common mistake among Wyncode students is to assume - since I'm never going to create more than one of a thing in my program (a tip calculator, for example) - then I should use class variables as a shortcut.

TODO: tip calculator example code

Don't do this. Don't turn your "table factory" into a "table". Even if there will only ever be one tip calculator in your program, always keep separate the idea of the "tip calculator factory" from the "tip calculator" product.

Private

```
class Table
   def a_public_method
      a_private_method
   end

   private
   def a_private_method
   end
end
```

New topic:

Within a class, the "`private`" keyword allows me to define a group of methods that are private. All the methods between `private` and the end of the class are private.

What does that mean? A private method is only accessible from inside the class. A private method has a limited scope. It can only be used between `class` at the top and `end` at the bottom.

Private

```
t = Table.new
t.a_public_method

t.a_private_method
```

private method `a_private_method' called

For example, when I create a new table, I can call `a_public_method` - no problem. Methods are public by default. Public methods are accessible from any table.

In the last slide, `a_public_method` calls `a_private_method`. That's still ok. The call to `a_private_method` is within the boundaries of the class.

But I can't call `a_private_method` directly. Ruby will let me know that I attempted to call a private method and will block me by generating an error.

Private By Default By

```
t = Table.new

t.initialize
private method `initialize' called
```

Almost all methods are public by default. Except one interesting case.

`initialize` is *always* a private method. Why?

Think about what `initialize` promises to do. It should only run once, when the object is created. If you expect the initialize method to only ever run once, allowing someone to re-initialize could cause trouble. They should just create a new object instead.

Every other method is public by default (unless I'm forgetting something).

Private

```
t = Table.new
t.a_public_method
#t.a_private_method
t.send :a_private_method
```

Privacy is good design. The more secrets you can keep, the less likely someone dumb or evil will come along and try to access or change your secret data.

However, on principle, Ruby as a language doesn't believe in limiting your freedom to do whatever you want with your code. So privacy isn't enforced very strictly by Ruby.

For example, I can get around privacy protection by using `send` and passing it the name of a method I'd like to call. Ruby will allow me to call any method that way, even private ones.

So in Ruby, unlike some other languages, privacy doesn't prevent other developers from doing bad things. It only makes it harder for them to do bad things. If someone uses the `send` method, that person must take responsibility for the results.

Besides, they can't hide. Searching your code for places where this method is being used will tip you off to potentially dumb or evil members of your team.

Private

```
class Bank
    def transfer
        withdraw
        deposit
    end

    private
    def withdraw; end
    def deposit; end
end
```

Private methods are useful when you'd like to refactor your class to add additional methods without making those methods accessible.

For example, say you create a method that represents a transaction. A transaction is a set of steps that need to occur together. For example, a bank transfer is a deduction from one account and a deposit into another. You don't want someone to be able to do one without the other.

Using private, you can prevent a developer from *accidentally* calling one without the other. It won't stop a malicious developer from using `send`, but it'll stop someone from making a mistake.

Protected

```
class Parent
    protected
    def a_protected_method; end
end

class Child < Parent
    def a_public_method
        a_protected_method
    end
end
```

Between `public` (the default) and `private` is the `protected` keyword. A protected method is accessible from within a class and within it's subclasses, but not from the outside.

In this example, Child inherits from Parent. So Child has access to `a_protected_method`.

Protected

```
p = Parent.new
p.a_protected_method
protected method `a_protected_method' called

c = Child.new
c.a_public_method
c.a_protected_method
protected method `a_protected_method' called
```

From an instance of `Parent`, I can't access protected methods from the "outside".

From an instance of `Child`, I can't access `a_protected_method` either.

I can access `a_protected_method` via the Child's `a_public_method`.

You'd use a protected group of methods in the same places you'd use a private group of methods. The only difference is that the method's scope extends to include all subclasses, not just methods in the current class.

Overriding

```
class Parent
   def whoami; puts "I'm a parent"; end
end

class Child < Parent
   def whoami; puts "I'm a child"; end
end
```

Subclasses inherit all the methods in the superclass. But subclasses are also allowed to define their own methods. And sometimes those methods can even override superclass methods.

So for example, say you have a Parent class that defines a `whoami` method. Then say you have a Child class that inherits from the Parent class, but defines it's own `whoami` method.

Overriding

```
p = Parent.new
p.whoami
```
=> “I’m a parent”

```
c = Child.new
c.whoami
```
=> “I’m a child”

When a `Parent` uses `whoami`, the Parent’s own `whoami` method is executed. But when a `Child` uses the `whoami` method, the child’s own version of the method is used instead. The child’s version of the method overrides (replaces) the parent’s version.

Overriding means a subclass has some discretion over which methods it would like to inherit. A subclass can change any methods it doesn’t like.

Super

```
class Person
    def speak; "I'm a person"; end
end

class Parent < Person
    def speak
        super + " who is a parent"
    end
end
```

Sometimes a subclass doesn't want to override a superclass entirely. Maybe the superclass had some good ideas, but the subclass wants to do a little bit more.

If you'd like to call the overridden method in your subclass, use the word `super`. When you see `super`, imagine that the keyword is being replaced by a call to a method of the same name in the superclass.

In this example, `Person` is a superclass. `Parent` inherits the `speak` method from person. The `speak` is nice, but `Parent` would like to do more. So `Parent`'s version of the speak method calls `super`, then adds more.

Super

```
p = Person.new
p p.speak
```
=> "I'm a person"

```
c = Parent.new
p c.speak
```
=> "I'm a person who is a parent"

This is how that looks. When the `Parent` subclass speaks, it calls the `Person` superclass to generate part of the String, then adds more.

In this case, the `Parent` class is said to "*embrace-and-extend*" the `speak` method in the superclass, rather than overriding the `speak` method in the superclass.

In programming speak, if you see something you don't like, you *override* it. That means it's bad.

If you see something you do like, you *embrace-and-extend* it. You embrace the good ideas, but add your own. That means it's mostly good.

Embrace-and-extend can be a euphemism for *override*. If someone is trying to be ironic (which, online, they usually are), then *embrace-and-extend* is a euphemism for an *override* that is pretending to be something it's not.

Super

```
class Parent
   def speak(arg); puts arg; end
end

class Child < Parent
   def speak; super "I'm a child"; end
end
```

If a superclass defines a method that accepts an argument, and a subclass defines a method with the exact same name that doesn't take an argument, the subclass method *still* overrides the superclass method. The only thing that matters is that the names match. The argument list doesn't matter.

If the subclass would like to call up to a version of the same method in the superclass that uses arguments, it can pass arguments to super. As before, just imagine the word `super` is replaced with a method by the same name in the parent.

In this example. the `Child` calls "`super`" with an argument, which calls `Parent`'s `speak` method with the argument.

Super

```
p = Parent.new
p.speak "I'm a parent"
```
=> I'm a parent

```
c = Child.new
c.speak "I'm a child"
c.speak
```
=> I'm a child

To see this in action, I can use `Parent`'s `speak` method with an argument. But I can't use the `Child`'s `speak` method with an argument. That method was replaced. Instead, I have to call the new version, without an argument.

Super

```
class Parent
    def speak1(arg); puts arg; end
end

class Child < Parent
    def speak2
        super.speak1 "I'm a child"
    end
end
```

Let's say, instead of overriding, you want to create a new method that simply uses a method from the superclass. How do you call any arbitrary superclass method from a subclass?

In this example, how would the `Child`'s `speak2` method call up to the `Parent`'s `speak1` method? Who thinks this will work?

Super

```
class Parent
    def speak1(arg); puts arg; end
end

class Child < Parent
    def speak2
        super.speak1 "I'm a child"
    end
end
```

This doesn't work. `super` calls the same *method* in the superclass. It doesn't give you access to any other method.

`super` is not an object. It's a method call.

In other languages that user `super`, `super` is an object, so this can be confusing.

Super

```
class Parent
    def speak1(arg); puts arg; end
end

class Child < Parent
    def speak2
        speak1 "I'm a child"
    end
end
```

Instead, you can skip `super`. Just directly call the `speak1` method. `Child` inherited `speak1`.

Super

```
class Parent
   def speak(arg); puts arg; end
end

class Child < Parent
   def speak(arg); super; end
end
```

Say you'd like to override a method with another method that matches both the name *and* the arguments?

That pattern is so common that there's a shortcut. If the child method and the parent method are exactly the same (same name *and* same arguments), then just calling super will forward your arguments to the parent for you, automatically.

In this case, `speak`, doesn't have to call "`super arg`". It just needs to call `super`. Ruby passes the `args` along for you, automatically.

Super

```
class Parent
   def speak; end
end

class Child < Parent
   def speak(arg); super(); end
end
```

If that's not what you want, if you don't want Ruby to forward your arguments for you, then call `super` with an explicitly empty list of arguments.

In this case, the `Child speak` method wants to call the `Parent speak` method, but doesn't want `arg` passed up to the `Parent`. So the child calls `super()` with an explicitly empty list of arguments, which blocks Ruby from automatically forwarding `arg` along.