On whiteboard before I begin:

Setup:

Please clone down the repo for this mini-lesson:
`$ git clone git@github.com:trivett/refactoring_intro.git`
Make sure that you have `pry` installed globally by running `$ gem list pry`. If you don't have it just run `$ gem install pry`.

Terms:

`Refactoring`

`Extract class`

`Extract method`

`Composition`

9:30

Hello hello.

Today I want to plant some seeds in your head regarding clean, readable, updatable object oriented code and the practice of making it.

Who has heard the term "refactoring"?
What does it mean?

Not less code. Usually you end up with more, but its easier to understand and update.

Before we begin I want to stress that though this little lesson will be delivered in Ruby, refactoring and recognizing code smells is a skill that will serve you in JavaScript, Java, Python or whatever language you happen to be working in. It's a programming skill, not a Ruby skill.

9:35

> Do I need to refactor?

Short answer, yes. And the answer is yes forever. My biggest challenge in preparing this lessone was knowing when to quit!

Longer answer is to look for smells.

- Squint test
- Excessive comments
- Long parameter lists
- You start reading a method or a class and by the time you are at the end, you forget the beginning
- Long method
- A class that knows way too much
- Methods that are not reusable
- Local variables inside methods
- Inflexible -- expensive to update or expand upon

As you progress in your career, these 'smells' will stink to high heaven and you won't be able to bear it until you fix it up. As you introduce stinky code, you will just start cleaning it up without even thinking about it.

I should note that that is not a comprehensive list, and no, I don't have all of the canonical named code smells/refactoring moves off the top of my head. In the Readme for this repo, I linked to some resources for further study.

Let's look at some code that ticks those boxes.

9:40

Open up person.rb and look for smells by solliciting feedback from the students

> Squint test

> You start reading a method or a class and by the time you are at the end, you forget the beginning

Yeah this is clearly too much going on in one method. You can tell in just seconds

> Excessive comments

> Long parameter lists

> Long method

> A class that knows way too much

Are people born with exactly one bank account? Let's say we wanted to include pin number, balance, interest rate, etc. That would currently have to go on Person. You can imagine that getting completely out of hand.

> Methods that are not reusable

> Inflexible

We have three attrs related to bank account. But check it out. What if someone has a savings account too? This works now, but not it isn't prepared for the future as is.

> Local variables inside methods

This is one of the most overt signs that a method can be *extracted*.

Check for understanding:

Can anyone give me one reason why this code is bad?

9:45

Extract Method:

> I look at a method that is too long or look at code that needs a comment to understand its purpose. I then turn that fragment of code into its own method. I prefer short, well-named methods for several reasons. First, it increases the chances that other methods can use a method when the method is finely grained. Second, it allows the higher-level methods to read more like a series of comments.

Let's identify some stuff inside `introduce` that can be extracted.

With student input, identify at least two methods that can be extracted.

Extract `full_name`

Let students extract `age`

Does it make sense for Person to respond to `age` and `full_name`? Yes it does.

9:50

We pared down the verbs into their own things, but now let's look at the 900 pound gorilla here that I want to introduce: dealing with the nouns in your project. Can you find the obvious noun here hiding in plain sight on Person?

A Person isn't born knowing what their bank account is. Some have none. Some have more than one. Bank account should be separate from Person. We should be able to initialize a Person without a bank account.

We are going to do this through *composing* our code into classes and dividing responsibility amongst them. Unlike inheritance, which is best pictured in terms of animal cladistics, (i.e. a cat *is a* mammal etc), composition involves relating things together, like cells that make tissues that make organs that make animals.

Code along and create a new Bank Account class.

lib/bank_account.rb:

```
1  class BankAccount
2    attr_accessor :name, :routing_number,
   :account_number, :account_balance
3
4    def initialize(name, routing_number, account_number,
   account_balance)
5      @name = name
6      @routing_number = routing_number
7      @account_number = account_balance
8      @account_balance = account_balance
9    end
10 end
```

10:00

Remove the attrs and initializer for bank account on Person, comment out the bank account portion of

app.rb:

```ruby
alice = Person.new("Alice", "Abigail", "Anderson", "1984-07-13",  "118 Eagle St", "", 11222)

alice.bank_account = BankAccount.new("TD", 12345, 253367435, 500)
```

Confirm that it works.

Then re-add the bank account parts of introduce.

```ruby

    puts @bank_account.name
    puts "Routing # #{@bank_account.routing_number}"
    puts "Acct # #{@bank_account.account_number}"
    puts "Balance #{@bank_account.account_balance}"
```

10:10

So nothing really changed. What if we want to add a new bank account? Easy!

Change bank_account to checking_account

Add savings_account attr_accessor

move the part of introduce about bank account to BankAccount as to_s

```ruby
  def to_s
    puts "Account Info"
    puts @name
    puts "Routing # #{@routing_number}"
    puts "Acct # #{@account_number}"
    puts "Balance #{@account_balance}"
    puts "Balance #{@account_balance}"
  end
```

10:15

In the highly unlikely event of having extra time, let the students pair on extracting the address class.

Further practice to get those reps in:

Introduce cash register app.

Wrap up, check for understanding as to how recomposing the Person class made it better and easier to expand on.