Refactoring the Code

Now we're going to have a go at refactoring the code for the Play Your Cards Right Sinatra app that we created in the last lesson.

Refactoring code is the process of improving its structure and maintainability without actually changing its behavior. What we're going to do is replace some of the chunks of code with methods. This will make the code easier to follow and easier to maintain, because if we want to make a change to some of the functionality, all we'll need to do is change the method in just one place.

Create a file called *play_your_cards_right_v2.rb*:

```
require 'sinatra'
enable :sessions

helpers do

  def set_up_game
    session[:deck] = []
    suits = %w[ Hearts Diamonds Clubs Spades ]
    values = %w[ Ace 2 3 4 5 6 7 8 9 10 Jack Queen King ]
    suits.each do |suit|
      values.each do |value|
        session[:deck] << "#{value} of #{suit}"
      end
    end
    session[:deck].shuffle!
    session[:guesses] = -1
  end

  def value_of card
    case card[0]
      when "J" then 11
      when "Q" then 12
      when "K" then 13
      else card.to_i
    end
```

```
  end

  def player_has_a_losing value
    (params[:guess] == 'higher' and value < session[:value]) or (params[:guess] == 'lower' and val
  end

  def draw_card
    session[:deck].pop
  end

  def game_over card
    "Game Over! The card was the #{ card }. You managed to make #{session[:guesses]} correct guess
  end

  def update_session_with value
    session[:value] = value
    session[:guesses]  += 1
  end

  def ask_about card
    "The card is the #{ card }. Do you think the next card will be <a href='/play/higher'>Higher</
  end
end

get '/' do
  set_up_game
  redirect to('/play/cards')
end

get '/play/:guess' do
  card = draw_card
  value = value_of card

  if player_has_a_losing value
    game_over card
  else
    update_session_with value
    ask_about card
  end
end
```

I've chosen some descriptive method names making the code more readable, almost like English in some places.

Let's take a look at these route handlers. We'll start with the first one. It sets up the game and then redirects the player to the play cards route.

We have a method called `set_up_game` that has all the code to set up the game. It tidies up the route handler, and makes it clear what's happening.

draw_card Method

The other route handler deals with the game play, and this is also being rewritten to make it easier to follow what's happening. It's also much shorter than it was before.

We start with a method called `draw_card` and it assigns a card to the variable `card`. We also use a method to find the value of the card (`value_of`).

Next we have a big `if` statement to check if the player has guessed correctly on it. Instead of using all those long conditions that we had before, I've extracted all of that logic into the method called `player_has_a_losing`, that takes an argument called `value`. After this, we use some more methods to deal with what happens if the player has a losing value or not.

Method game_over

`game_over` takes an argument of `card`, which will be the current card.

If the player doesn't lose, we go into the `else` section, and do two things. First of all, we update the session with the value of the card.

Method update_session_with

The method is called `update_session_with`; the value is an argument representing the value of the card.

The next method is called `ask_about` and it also takes the current card as an argument.

Both of these methods are very descriptive because they describe exactly what we want to do. We want to update the session with the value of the card, and then we want to ask the player about the card.

Helper Methods

Sinatra uses the concepts of **helper methods** to describe methods that are used in route handlers and views. These are placed in a `helpers` block, and it can actually go anywhere in the code, but it's usually placed near the beginning.

Helper methods can be used inside the route handlers as well as inside the views. Let's take a look at some of these helper methods. Most of the code is very similar to the code that we used in the last lesson, so it should be very familiar to you.

The first helper method is the `set_up_game` method that sets a `session` variable as an array and then creates a deck of cards by iterating through two arrays like we've done in the lessons earlier. It then shuffles that deck of cards that's stored in the session and also sets up the number of guesses starting at negative one.

Method value_of

The next method is called `value_of` and this has to be given an argument of card and this will be provided as string and it will look at the string to find out what the value of the card is. We're going to look at the first letter and check

what that is, if it's a letter we'll give it a value depending on which card it is.

to_i Method

Otherwise, we'll use the `to_i` method to just return the integer that the string starts with.

The next helper method is used to check if the player has guessed correctly or not. It's actually called `player_has_a_losing`, and value is entered from the value of the card, as an argument. This method returns `true` or `false`. It compares card's value to the player's guess, which is stored in the `params` hash with a key of `guess`.

The next helper method is the `draw_card`. It simply takes the session, the deck of cards held in the session, and uses the `pop` method to take a card from the end of the array. It doesn't really do much except it just makes things more descriptive because we want to actually write. We're going to draw the card because in our game that's what we're doing and that's actually more descriptive than using the word `session` and the method `pop`.

Then we have the `game_over` function and this simply returns a string telling the player that the game is over. Again this is a long string that clogged up the route handler in the earlier lesson making it harder to follow. So it's much better to be extracted out as a helper method. This needs to take the current card as an argument, so that we can tell the player what the card was, using string interpolation.

Using String Interpolation.

And you can see that here, so the current card is entered as an argument, and it's used inside the string there.The `update_session_with` method takes an argument of `value`. This will be the value of the current card. This does exactly what it says: it updates the value stored in the session with the value of the current card so it's stored for the next round and it also increases the number of guesses that's stored in the session by one.

Helper Method called ask_about

The last helper method is called `ask_about`, and it takes the current card as an argument. This simply returns a string and asks the player whether the card is higher or lower. Now again, although it's only a basic method that doesn't really do much except return a string, it helps to make the route handlers much tidier and easier to read, as we saw when we were going through the route handlers.

Now one thing you may be wondering about some of these helper methods is why we had to supply either the `card` or `value` as a parameter to the methods like we did in these last two methods here when both card and value, both existed already as variables inside in the router handlers.

Well, this is because of the scope of variables, which is the places in which the code where variables are accessible. A variable can only be accessed inside a method if it has been created in that method, or if it's been entered as an argument. So the variables `card` and `value` are not available in any of the helper methods, even if they are mentioned inside the route handler that calls the helper method unless we supply that as an argument to the method. A method doesn't have access to any variable created outside of it.

In a similar way, a variable created inside a method is only available inside that method scope.