Scopes and Binding

The scope you are currently in determines what methods and variables are available.

Application/Class Scope

Every Sinatra application corresponds to a subclass of Sinatra::Base. If you are using the top-level DSL (require 'sinatra'), then this class is Sinatra::Application, otherwise it is the subclass you created explicitly. At class level you have methods like get or before, but you cannot access the request or session objects, as there is only a single application class for all requests.

Options created via set are methods at class level:

```
class MyApp < Sinatra::Base
  # Hey, I'm in the application scope!
  set :foo, 42
  foo # => 42

  get '/foo' do
    # Hey, I'm no longer in the application scope!
  end
end
```

You have the application scope binding inside:

- Your application class body
- Methods defined by extensions
- The block passed to helpers
- Procs/blocks used as value for set
- The block passed to Sinatra.new

You can reach the scope object (the class) like this:

- Via the object passed to configure blocks (configure { |c| ... })
- settings from within the request scope

Request/Instance Scope

For every incoming request, a new instance of your application class is created, and all handler blocks run in that scope. From within this scope you can access the request and session objects or call rendering methods like erb or haml. You can access the application scope from within the request scope via the settings helper:

```
class MyApp < Sinatra::Base
  # Hey, I'm in the application scope!
  get '/define_route/:name' do
    # Request scope for '/define_route/:name'
    @value = 42
    settings.get("/#{params['name']}") do
      # Request scope for "/#{params['name']}"
      @value # => nil (not the same request)
    end

    "Route defined!"
  end
end
```

You have the request scope binding inside:

- get, head, post, put, delete, options, patch, link, and unlink blocks
- before and after filters
- helper methods
- templates/views

Delegation Scope

The delegation scope just forwards methods to the class scope. However, it does not behave exactly like the class scope, as you do not have the class binding. Only methods explicitly marked for delegation are available, and you do not share variables/state with the class scope (read: you have a different `self`). You can explicitly add method delegations by calling `Sinatra::Delegator.delegate :method_name`.

You have the delegate scope binding inside:

- The top level binding, if you did `require "sinatra"`
- An object extended with the `Sinatra::Delegator` mixin

Have a look at the code for yourself: here's the Sinatra::Delegator mixin being extending the main object.