Section Review

Learn to Code with Ruby

Intro to Modules

- A **module** is a Ruby "container" for storing related functionality. A module can store methods, classes, constants, and even other modules.
- Declare a module with the **module** keyword and a name (in **PascalCase**). Finish the module with the **end** keyword.
- If using the module's methods independently, prefix each module method with **self**.
- Use dot syntax to reference a method inside a module.
- Different modules can nest identical names; the module creates a boundary/namespace for the name.

Built-in Modules

- Ruby has modules built into the core language. Some modules are automatically available. Other modules have to be imported.
- The **Math** module stores various mathematical methods.
- The **URI** and **Net** modules store functionality for making network requests.
- There are additional modules for manipulating files and directories, working with datetimes, parsing data (JSON, YAML, CSV, etc) and more

Mixins

- A **mixin** is a module that we "inject" into a class to add additional behavior. When defining module methods for a mixin, remove the **self** keyword.
- Use the **include** keyword to mixin a module's methods as instance methods.
- Use the **extend** keyword to mixin a module's methods as class methods.
- Use the **prepend** keyword to mixin a module's methods as instance methods and prioritize them over instance methods with the same name in the class definition.

Ruby Modules as Mixins

- We add mixins into classes to share behavior with forcing a subclass hierarchy.
- Ruby's **Enumerable** module can be mixed in to enable iteration on a custom object. Define an **each** method and gain access to dozens of iteration methods (**map**, **select**, **any**?, **all**?, etc).
- Ruby's **Comparable** modules can be mixed in to enable comparison between custom objects. Define an <=> (spaceship operator) method to gain access to dozens of comparison methods (**==**, **!=**, **<**, **>=**).