**Requiring Files**

As our programs grow more and more complex, we'll want to separate code into many files. This will help our code stay organized and easy to maintain. We've been using this organization a bit so far in the course: we often separate our test code (spec files) from our actual implementation files. In the previous RSPEC based projects, you may have noticed the /spec and /lib folders found in each project root. Let's step through an example of how we can divide our code into files!

**Enter Pet Hotel**

Using the previous concept of classes, let's design a PetHotel that will house Cats as guests. Let's begin by writing both classes in the same file:

# pet\_hotel.rb

class PetHotel

def initialize(name)

@name = name

@guests = []

end

def check\_in(guest)

@guests << guest

end

end

class Cat

def initialize(name)

@name = name

end

end

hotel = PetHotel.new("Animal Inn")

cat\_1 = Cat.new("Sennacy")

cat\_2 = Cat.new("Whiskers")

hotel.check\_in(cat\_1)

hotel.check\_in(cat\_2)

p hotel

# <PetHotel:0x007fb1ce1e91f0

# @name="Animal Inn",

# @guests=[

# #<Cat:0x007fb1ce1e9060 @name="Sennacy">,

# #<Cat:0x007fb1ce1e8f48 @name="Whiskers">

# ]

# >

The code above seems acceptable, but what if our PetHotel and Cat classes were more complex? Grouping these two classes in the same file may no longer be viable. Even further, what if our PetHotel should be able to house other animals like Dogs, Birds, etc.? We should avoid creating one massive file for all of these classes and logic. A great design principle to follow is *Separation of Concerns*. One file should focus on implementing one class. After all, our Cat class should be able to stand on its own without interaction from PetHotel. Let's decouple Cat from PetHotel to increase reusability of our classes.

**A Better Hotel design**

Below is one way we could organize our different files. project\_root is our outermost folder that will contain everything. Names that end in .rb are files, names that don't end in .rb are folders:

project\_root

├── pet\_hotel.rb

├── cat.rb

└── other\_animals

└── dog.rb

However, if PetHotel and Cat exist in separate files, how can we connect the two? We need to "import" cat.rb into pet\_hotel.rb. In other words, we need to require cat.rb in pet\_hotel.rb. Let's take a look into how we can do this along with dog.rb, using the require\_relative method next.

**Require Relative**

Keep in mind the folder structure above and take a look at the code we could have in each file:

# project\_root/cat.rb

class Cat

def initialize(name)

@name = name

end

end

# project\_root/other\_animals/dog.rb

class Dog

def initialize(name)

@name = name

end

end

# project\_root/pet\_hotel.rb

# Let's require the last two files, by specifying their path's relative to this pet\_hotel.rb file

require\_relative "./cat.rb"

require\_relative "./other\_animals/dog.rb"

class PetHotel

def initialize(name)

@name = name

@guests = []

end

def check\_in(guest)

@guests << guest

end

end

hotel = PetHotel.new("Animal Inn")

cat = Cat.new("Sennacy")

dog = Dog.new("Fido")

hotel.check\_in(cat)

hotel.check\_in(dog)

p hotel

# <PetHotel:0x007ffe7f01af60

# @name="Animal Inn",

# @guests=[

# #<Cat:0x007ffe7f01aee8 @name="Sennacy">,

# #<Dog:0x007ffe7f01ae98 @name="Fido">

# ]

# >

To run our hotel, we would just need to execute ruby pet\_hotel.rb. Because pet\_hotel.rb requires cat.rb and dog.rb, the code in those files will also execute.

require\_relative is a method we can use to specify a path to another ruby file. As its name suggests, we need to specify a path that is relative to our current location. So if we are at pet\_hotel.rb a relative path to cat.rb is ./cat.rb. A single dot (./) denotes the current location of our file.

dog.rb is not immediately found in our current location; it is one folder away. So the relative path from pet\_hotel.rb to dog.rb is ./other\_animals/dog.rb.

**require vs require\_relative**

You may have noticed that in the course projects we have used require instead of require\_relative in our /spec files. We use the plain require because we run the code using RSPEC via bundle exec rspec. RSPEC will automatically run the code in a certain configuration where it will automatically know to look inside of the /lib folder. In fact, RSPEC is designed to be used where you have a separate /spec and /lib folders. To get into the nitty gritty details, RSPEC will configure the $LOAD\_PATH for us. But don't worry about this for now.

As a rule of thumb, we'll use the plain require where gems are involved. In the previous sections that used the byebug gem, we had to write require "byebug" to access the debugger. It's obvious that we don't have an explicit byebug.rb file in those old projects. This is because ruby already knows where to find byebug through the $LOAD\_PATH.