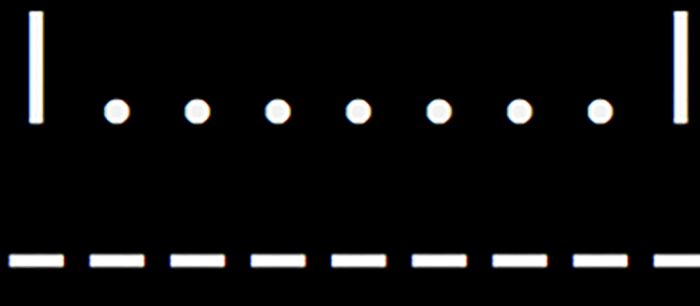
Write Yourself a Roguelike

by Matt Mongeau

thoughtbot



Write Yourself a Roguelike

Matt Mongeau

September 25, 2015

Contents

Introduction

Chapter i - What is a roguelike?

You are about to embark on a journey. This journey will be plagued with orcs, gnomes, algorithms, data structures, and kittens. You, valiant developer, will be writing a Roguelike.

If you are reading this book, then bets are you already know what a roguelike is, but let's go over some quick ground rules for what a roguelike should have:

- Proceduraly generated environments
- Turn-based gameplay
- Permanent death
- Tile-based maps

So where do these rules come from? Roguelike games get their rules and name from the game Rogue. Rogue was developed in the 1980s, but if you have ever played it, then it's clear that its influence has spread far and wide as evident in games like Dwarf Fortress and Diablo.

In Rogue, you play as an adventurer who descends into a dungeon for loot and fame. There is a catch though. Until you make it to the final level and retrieve the Amulet of Yendor, you are unable to return to the previous level. If you retrieve the Amulet of Yendor and make it back to the surface then you journey home, sell all of your loot, and get admitted to the fighters guild. Huzzah!

Now, let's get back to the rules because they are important for the task at hand. A huge part of writing a good roguelike is the procedurally generated environments. These can be anything from dungeons to entire worlds. The importance of the procedurally generated environments is replayability. You're unlikely to have a similar gaming experience between plays. It creates an incentive to play over and over. Our roguelike will be far more exciting if one time you play you put on cursed boots and starve to death because you can't reach the floor and the next time you get assaulted by a feral kitten early in the game.

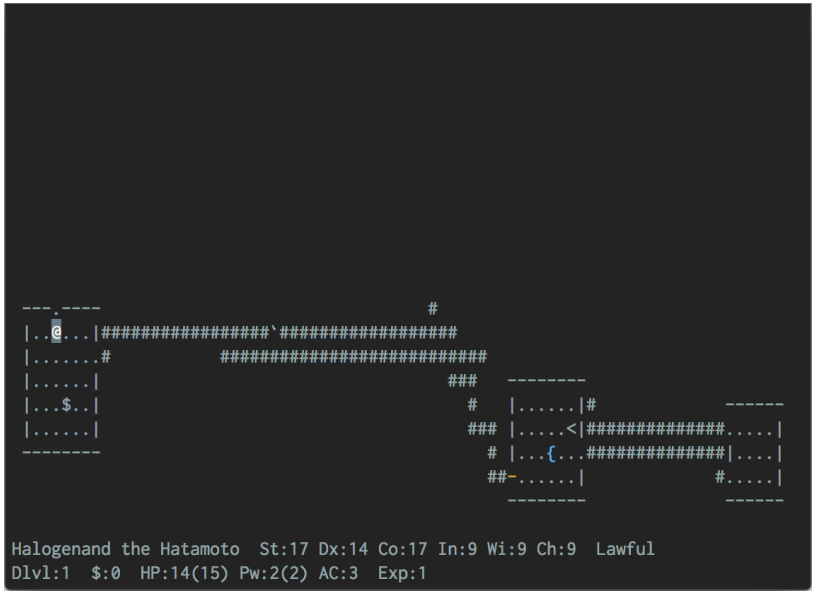
An important part of a roguelike game is having to think very hard about your next move. Quick! You're about to die! Do you pray, do you write "Elbereth" on the ground hopping the never ending assult of ants will leave your pitiful tourist alone! Each keypress could be your last. For this reason most roguelikes are turn-based. As the player, you'll issue a single command and the entire world will update. This

gives you a chance to examine all the changes and carefully plan how you'll deal with the situation at hand.

Which brings us to the next rule... permanent death! This is the crux of a roguelike. If you die, you lose EVERYTHING and have to start over. All your hard work! That amazing wand you found! All gone! This means your choices have serious consequences. This makes success very rewarding and failure very frustrating. Do you read a scroll that could destroy your armor making you defenseless, but also could detect food you so desperately need? This tension is rarely found in games of other ilk.

And now, for the final roguelike rule. Our entire world will be tile-based. A character can move orthogonally and diagonally between tiles. This perspective and the use of ASCII to represent our game will allow it to run in terminals all across the world.

Chapter ii - What is NetHack?



NetHack is an ASCII based roguelike originally released in 1987, but it is currently still being further developed. It is a sort of evolution from Rogue. You're still descending down through levels and levels of dungeons to retrieve the Amulet of Yendor, but your goal now is to escape the dungeon, ascend through the elemental planes until you get to the astral plane in which you offer the amulet to your assigned god and are granted demigodhood (never thought I'd type that word).

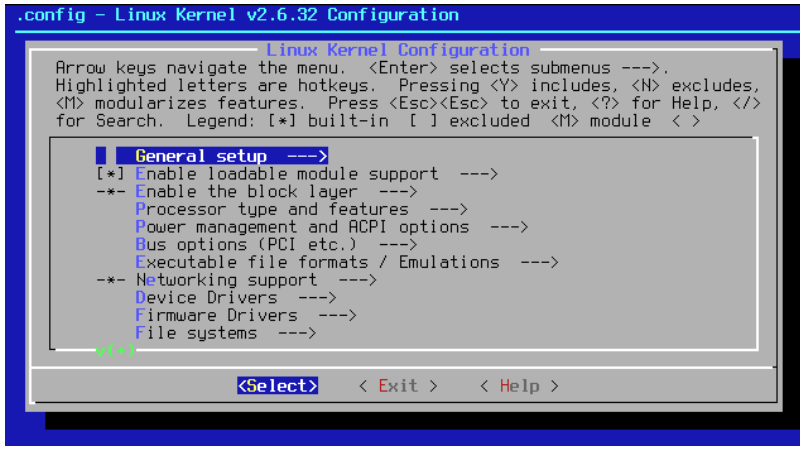
There are also a number of other aspects that make NetHack more interesting. There is considerable depth added on top of the original Rogue-formula. Instead of being a generic adventurer you can be a samurai, valkyrie, tourist, or other role each with their own specific quest. You have a race and alignment which will affect how monsters interact with you and even what you can do in the game (at the cost of angering your god). There are multiple paths in the dungeon and you can return to earlier levels (usually necessary to win).

As we build our roguelike, we will continously reference NetHack for both implementation and inspiration. We can do this because NetHack is open source, and while the code-base is almost 30 years old, I've gone ahead and spent countless hours using lldb to figure out what's going on for you.

Chapter iii - Tooling

In order to make our roguelike we'll be using a few different tools. First off, we'll be using Ruby. I've chosen Ruby for this initial edition for a couple of reasons. First and foremost, I find Ruby to be fairly easy to understand and when I want to build something quickly it's my goto. Second, I feel like object-oriented programming lends itself fairly well for programming games. Lastly, it has Ncurses bindings available through the gem `curses`.

What is Ncurses you ask? Ncurses stands for New Curses, it's a freeware reimplementation of the original curses library. Its purpose is to make managing screen state easier and more portable. If you've ever installed some flavor of linux on a computer you might have seen something like this:



This screen was created using Ncurses. So why do **we** need Ncurses? After all, we're just writing a simple game. Using Ruby let's try to do something simple like clearing the screen. If you're on OSX you might have written something like:

```
system("clear")
```

Everything works great until you try to run this code on Windows. It will fail on windows because there is no `clear.exe`. On Windows we need to run:

```
system("CLS")
```

Now we'd need to run code to detect the OS. We'll need to import `rbconfig` and write some code to detect the OS and choose the correct code to run. We can avoid this by using Ncurses which will do all the heavy lifting for us. In order to use it, you'll need to make sure you have ncurses installed on your system. For Windows, you'll most likely want to use a version of Ruby that comes with ncurses in its standard library. In my case, I've gotten it running by using ruby 1.9.3p484. If you're using a Unix-based system you can most likely run `man ncurses` and if a man page comes up you should be set. If nothing comes up you'll want to install `ncurses-dev` via whatever package manager your system utilizes.

In addition to using `curses`, we'll also be making heavy use of YAML. You could really use any data language you want (XML, JSON, etc.), but YAML seems to be a

defacto choice for Ruby. We'll be using YAML to store the large amounts of data that is needed to write a game of this nature.

Chapter iv - Why write this book?

The reason I decided to write this book is because game development was one of the things that first attracted me to developing software. I started programming in highschool with QBasic. QBasic made it pretty easy to enter into a graphics mode and start drawing on the screen. It wasn't long before I had written a very primitive RPG. Nowadays, it's much harder to get started due to the complexities of graphical hardware and complex operating system interactions. Roguelikes allow us to simplfy things one again.

There is a certain purity in an ASCII based game - there is very little overhead and very little math required to get started. Imagination also plays a large role. Most games these days have taken imagination out of the equation. I know what everything looks like because the game's artists have fleshed it all out. However, games like NetHack allow me to imagine what's going on and to in some ways weave my own story.

Because I've found a lot of enjoyment in playing games like NetHack, I've developed a natural curiosity for the internal workings. How would one go about creating the same kind of game? I've spent considerable time diving in and out of the C code to answer that question. I hope this book will answer that question for you as well.

Generating a Character

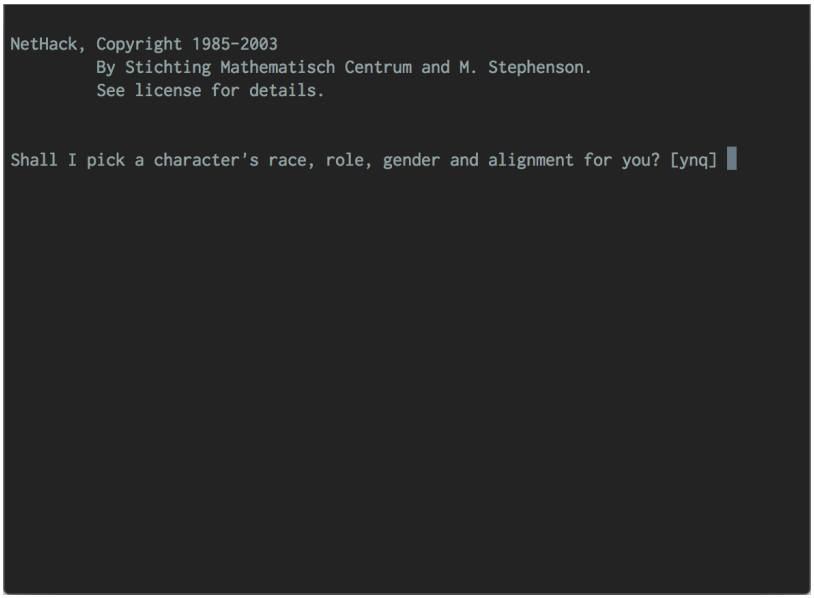
Chapter 1 - The Title Screen

Most gaming journeys begin with the fabled title screen - where we get to see the title of the game once again before we can begin. We're going to begin our journey the same way. To implement our title screen, and the rest of our game, we're going to need to make use of the `curses` gem. If you don't have ncurses installed on your computer, please go back and read the tooling chapter in the introduction. If you have `ncurses` installed, but don't already have the `curses` gem, you can install it via:

```
gem install curses
```

If this fails with "Failed to build gem native extension." you might not have `ncurses` properly installed and should reference the tooling chapter for installation instructions.

Now that we have the curses gem installed, we can start working on the title screen. We're going to base this on the NetHack title screen. The NetHack title screen is relativly simple as you can see here:



Let's start our game by writing the simplest curses example we can come up with. The program will initialize curses, read a single character, and then quit. To do this, create a file named `main.rb` and add the following:

```
require "curses" # require the curses gem
include Curses   # mixin curses

# The next three methods are provided by including the Curses module.

init_screen      # starts curses visual mode
getch            # reads a single character from stdin
close_screen     # closes the ncurses screen
```

If you run this program, you will see the terminal go black and upon pressing a character it will return back to normal.

Now that we've got a simple curses example running, let's work on our title screen. We're going to break our code up into three files. The first file we'll create is named `ui.rb`. This will hold all of our interface routines. We're breaking the UI into its own

class for a few reasons. First, in game development, it's easy to produce code that is difficult to understand. We want to avoid this by trying to employ the single-responsibility pattern as much as possible. Tangentally, if we decide to replace our UI implementation with a different one, the isolation here makes doing that far easier. The implementation for our `UI` class will look like this:

```
class UI
  include Curses

  def initialize
    noecho # do not print characters the user types
    init_screen
  end

  def close
    close_screen
  end

  def message(y, x, string)
    setpos(y, x) # place the cursor at our position
    addstr(string) # prints a string at cursor position
  end

  def choice_prompt(y, x, string, choices)
    message(y, x, string + " ")

    loop do
      choice = getch
      return choice if choices.include?(choice)
    end
  end
end
```

You may be wondering why we're passing values in (y, x) order instead of (x, y). We're passing the values as (y, x) because that's the order ncurses will expect to see them in. Ncurses wants the coordinates in (y, x) order because of how it renders the screen. Essentially, ncurses will start in the top left of the screen, y = 0, and

then write out the entire line before moving on to the next line, y = 1. By storing the y coordinate first it can handle this process more optimally.

Now let's create the file `game.rb` which will hold our `Game` class. The responsibility of the `Game` class is to execute the main run loop as well as manage setup and global state. The implementation for the `Game` class will look like this:

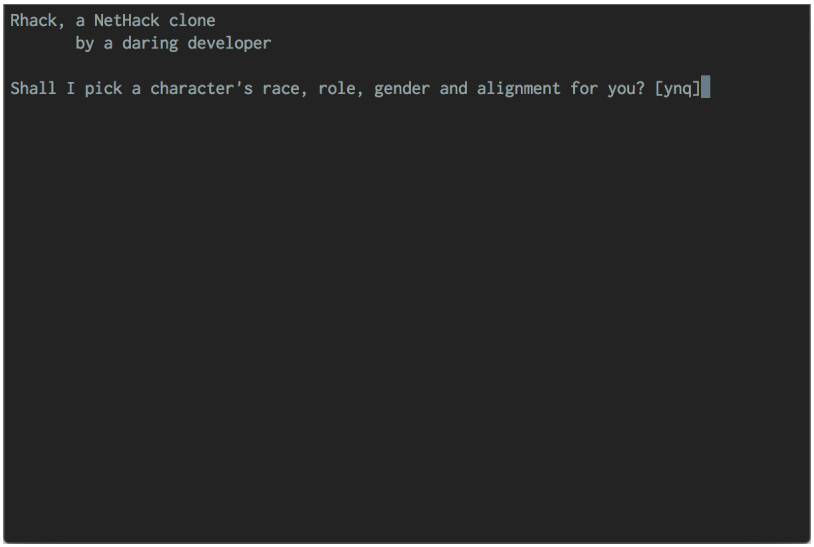
```
class Game
  def initialize
    @ui = UI.new
    at_exit { ui.close } # runs at program exit
  end

  def run
    title_screen
  end

  private

  attr_reader :ui

  def title_screen
    ui.message(0, 0, "Rhack, a NetHack clone")
    ui.message(1, 7, "by a daring developer")
    ui.choice_prompt(3, 0, "Shall I pick a character's race, role, gender and " +
      "alignment for you? [ynq]", "ynq")
  end
end
```

Finally, change the `main.rb` file to use our new classes:

```
$LOAD_PATH.unshift "." # makes requiring files easier

require "pp"
require "curses"
require "ui"
require "game"
```

```
Game.new.run
```

If you run the program now, it will look very much like the initial NetHack screen.



Moving forward, we're going to want to show more than a title screen. Let's start by refactoring our current code into something more adaptable. Refactor `game.rb` to the following:

```
class Game
  def initialize
    @ui = UI.new
    @options = { quit: false, randall: false } # variable for options
    at_exit { ui.close; pp options } # See selected options at exit
  end

  def run
    title_screen
  end
```

```
  private

  attr_reader :ui, :options # Add attr_reader for options

  def title_screen
    TitleScreen.new(ui, options).render
    quit?
  end

  def quit?
    exit if options[:quit]
  end
end
```

Now we'll create a `title_screen.rb` file with the following:

```
class TitleScreen
  def initialize(ui, options)
    @ui = ui
    @options = options
  end

  def render
    ui.message(0, 0, "Rhack, a NetHack clone")
    ui.message(1, 7, "by a daring developer")
    handle_choice prompt
  end

  private

  attr_reader :ui, :options

  def prompt
    ui.choice_prompt(3, 0, "Shall I pick a character's race, role, gender and " +
      "alignment for you? [ynq]", "ynq")
  end
```

```
  def handle_choice(choice)
    case choice
    when "q" then options[:quit] = true
    when "y" then options[:randall] = true
    end
  end
end
```

You can see here how we'll be making use of the `options` variable created in `game.rb`. It will store the selections the user makes during setup. We need to do this in order to communicate between the different selection screens about what choices the player has made. If the user selects "q" we'll store that we need to quit, if they choose "y" then we'll randomly assign the rest of the traits. In order to keep our application working you'll also need to add:

```
require "title_screen"
```

to `main.rb`. Now when running the program and choose an option you'll see set in the output that is printed. For instance, if I select yes, then I'll see the following:

```
{:quit=>false, :randall=>true}
```

Chapter 2 - Messages

There are going to be a lot of in-game messages and to make things more fluid we should extact them into a yaml file. This makes them far easier to change (or to internationalize) later. Let's start by creating a `data` directory. This directory will hold some yaml files that will contain in game text and other data. In that directory, let's create a file for holding our in-game messages. Name the file `messages.yaml` and add the following to it:

```
---
title:
  name: Rhack, a NetHack clone
  by: by a daring developer
  pick_random: "Shall I pick a character's race, role, gender and alignment for you? [ynq]"
```

Next, we'll want to update our `TitleScreen` class to make use of these messages. In order to do that, we'll first need some way to load the yaml file. In this situation, it's a good idea to isolate an external dependency like YAML in order to make it easier to replace or modify in the future. We took this exact approach with Curses by extracting the UI into its own class. Let's extract a `DataLoader` class that knows how to load our data for us. Create a `data_loader.rb` file with the following:

```
class DataLoader
  def self.load_file(file)
    new.load_file(file)
  end

  def load_file(file)
    symbolize_keys YAML.load_file("data/#{file}.yaml")
  end

  private

  def symbolize_keys(object)
    case object
    when Hash
      object.each_with_object({}) do |(key, value), hash|
        hash[key.to_sym] = symbolize_keys(value)
      end
    when Array
      object.map { |element| symbolize_keys(element) }
    else
      object
    end
  end
end
```

The reason behind `symbolize_keys` is that YAML will parse all the keys as strings and I prefer symbols for this. Even though `ActiveSupport` has a similar method, we're going to leave it out because it won't work directly with arrays. Our implementation will symbolize the keys correctly for hashes or arrays even if they are nested.

Now we'll create a global way to access these messages. Create a file called `messages.rb` with the following:

```
module Messages
  def self.messages
    @messages ||= DataLoader.load_file("messages")
  end

  def self.[](key)
    messages[key]
  end
end
```

It's evident here that our Messages module knows nothing about the YAML backend, instead it simply asks our DataLoader to load the messages. Now that we have a way to get our messages let's change our `title_screen.rb` to make use of it. In initialize add the following:

```
@messages = Messages[:title]
```

Make sure to add `:messages` to the `attr_reader` line, like so:

```
attr_reader :ui, :options, :messages
```

and then change `render` to the following:

```
def render
  ui.message(0, 0, messages[:name])
  ui.message(1, 7, messages[:by])
  handle_choice prompt
end
```

And change `prompt` to:

```
def prompt
  ui.choice_prompt(3, 0, messages[:pick_random], "ynq")
end
```

Now to finish up, add `require`s in `main.rb` for `yaml`, `data_loader`, and `messages`. When you run the program again it should still function like our previous implementation.

Chapter 3 - Role call

For a game like NetHack, there is a lot of information that goes in to creating a character. From a top level, a character will have a role, race, gender and alignment. Each of these traits will determine how a game session will play.

We'll start by allowing the player to choose their role. In NetHack, these are the roles a player can select:

We will implement all of these. Looking at this list, "data" should immediately come to mind. We're going to create another data file to hold the information for our roles. To start with, we're going to give each role a `name` and a `hotkey`. Create `data/roles.yaml` with the following:

```yaml
---
- name: Archeologist
  hotkey: a
- name: Barbarian
  hotkey: b
```

```
- name: Caveman
  hotkey: c
- name: Healer
  hotkey: h
- name: Knight
  hotkey: k
- name: Monk
  hotkey: m
- name: Priest
  hotkey: p
- name: Rogue
  hotkey: r
- name: Ranger
  hotkey: R
- name: Samurai
  hotkey: s
- name: Tourist
  hotkey: t
- name: Valkyrie
  hotkey: v
- name: Wizard
  hotkey: w
```

Now we're going to create a `Role` class that can load all of this data. Create a file named `role.rb` with the following:

```
class Role
  def self.for_options(_)
    all
  end

  def self.all
    DataLoader.load_file("roles").map do |data|
      new(data)
    end
  end
```

```
  attr_reader :name, :hotkey

  def initialize(data)
    data.each do |key, value|
      instance_variable_set("@#{key}", value)
    end
  end

  def to_s
    name
  end
end
```

We're using for_options here to unify the interface across all of our characteristics, since race and alignment will be dependent on role. We'll see shortly why this abstraction makes sense.

Now we're going to write a generic `SelectionScreen` class. It's job will be to print two messages and display a list of options that can be selected by a hotkey. Create the file `selection_screen.rb` with:

```
class SelectionScreen
end
```

Now let's add some methods one by one. First we'll add our `initialize` and some `attr_reader`s:

```
def initialize(trait, ui, options)
  @items = trait.for_options(options)

  @ui = ui
  @options = options

  @key = trait.name.downcase.to_sym
  @messages = Messages[key]
end
```

```
private

attr_reader :items, :ui, :options, :key, :messages
```

When we create a our selection screen we'll call it from `game.rb` with:

```
SelectionScreen.new(Role, ui, options).render
```

So in this case, `trait` will be the class `Role`. On the first line we fetch all the relevant roles by calling `for_options`. If you recall, `for_options` just reads the yaml file of roles and returns all of them. Next we assign the `ui` and `options` variables. Then, we determine a key that we'll use for a couple of things. If `Role` is our trait, then we want `:role` to be our key. Finally, we grab a hash of messages related to our key (:role in this case).

Now we'll implement our only **public** method `render` (make sure this goes above the `private` line):

```
def render
  if random?
    options[key] = random_item
  else
    render_screen
  end
end
```

In this method we check to see if we need to randomly select an item. If we do we don't want to render the screen, so it simply sets the option and returns. Otherwise we'll render the screen. The implementation for `random?` and `random_item` look like this:

```
def random?
  options[:randall]
end

def random_item
  items.sample
end
```

For now, `random?` simply checks if `randall` was set and `random_item` just chooses a random element form our items array. Now we can implement `render_screen` and `instructions`:

```
def render_screen
  ui.clear
  ui.message(0, 0, messages[:choosing])
  ui.message(0, right_offset, instructions)
  render_choices
  handle_choice prompt
end

# instructions has been pulled out into it's own method for a reason
# you will see later

def instructions
  messages[:instructions]
end
```

Here we clear the screen, display the message on the left - “Choosing Role”, display the message on the right - “Pick the role of your character”, display the choices, and then prompt and handle the player’s selection. For convenience, I’ve pulled out `right_offset` into a method since we’ll use it a few times:

```
def right_offset
  @right_offset ||= (instructions.length + 2) * -1
end
```

This method returns a negative number representing how far left from the right side we should be when printing the right half of our screen. We’ll need to update our `UI` class to handle negative numbers, but let’s finish our `SelectionScreen` class first.

Now we’ll write our method for rendering our choices

```
def render_choices
  items.each_with_index do |item, index|
```

```
    ui.message(index + 2, right_offset, "#{item.hotkey} - #{item}")
  end

  ui.message(items.length + 2, right_offset, "* - Random")
  ui.message(items.length + 3, right_offset, "q - Quit")
end
```

This method is relatively straight forward. We loop through each item and print out the hotkey and the name of the role (we're cheating here by not printing "a" or "an" in front of the name, but it's not really important).

Now let's implement `handle_choice` and `item_for_hotkey`:

```
def handle_choice(choice)
  case choice
  when "q" then options[:quit] = true
  when "*" then options[key] = random_item
  else options[key] = item_for_hotkey(choice)
  end
end

def item_for_hotkey(hotkey)
  items.find { |item| item.hotkey == hotkey }
end
```

Here we have 3 choices. If the user presses "q" then we want to quit. If they press "*" then we want to randomly choose an item. If they press any other valid option we want to assign the corresponding role.

Finally let's implement `prompt` and `hotkeys`:

```
def prompt
  ui.choice_prompt(items.length + 4, right_offset, "(end)", hotkeys)
end

def hotkeys
  items.map(&:hotkey).join + "*q"
end
```

The `hotkeys` represent our valid choices, but we need to make sure to add "*" and "q" as valid hotkeys.

Now we're ready to initialize this screen in `game.rb`. Add the following constant:

```
TRAITS = [Role]
```

Then change the `run` method to look like this:

```
def run
  title_screen
  setup_character
end
```

And then add `setup_character` and `get_traits` as a private methods:

```
def setup_character
  get_traits
end

def get_traits
  TRAITS.each do |trait|
    SelectionScreen.new(trait, ui, options).render
    quit?
  end
end
```

There are a few things left to do in order to get this working. First, in `main.rb` add:

```
require "role"
require "selection_screen"
```

Above the `require "game"` line. Next, we'll need to modify our `UI` class to have a `clear` method. Curses provides this method, but it's private, so we'll need to add the following to `ui.rb`:

```
def clear
  super # call curses's clear method
end
```

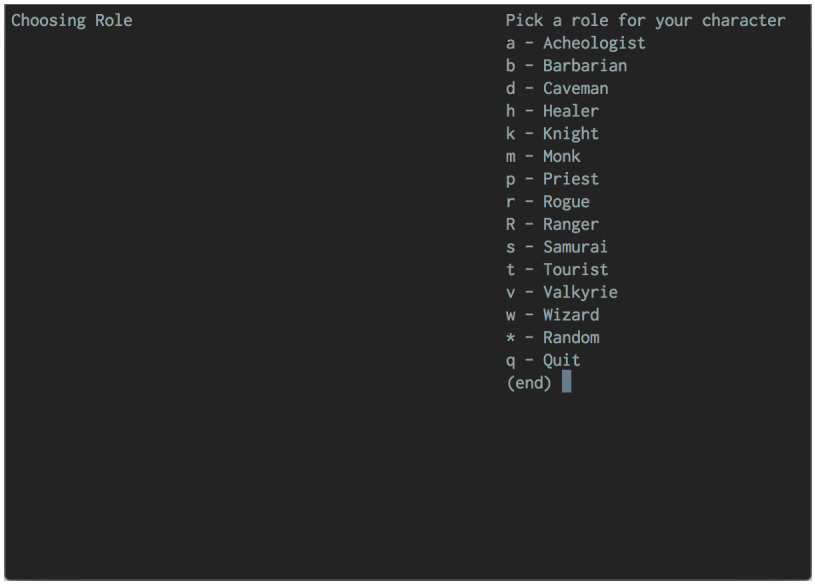
While we have the `ui.rb` file open we should handle our `right_offset` issue we described before. Change the implementation of `message` to the following:

```
def message(x, y, string)
  x = x + cols if x < 0
  y = y + lines if y < 0

  setpos(y, x)
  addstr(string)
end
```

Finally, we'll need to add some messages to our `data/messages.yaml` file:

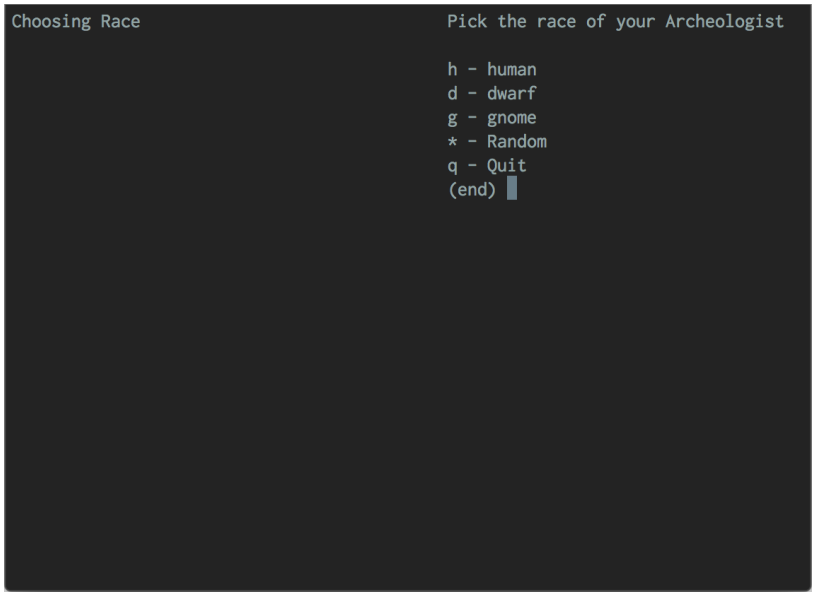
```
role:
  choosing: Choosing Role
  instructions: Pick a role for your character
```

If you run the program and choose "n" for the first choice then you should see:



Choosing any role will print out the options again, but this time it will display the selected role as well. If you choose "y" at the title screen a random role will appear here. Now that we've laid down the framework for setting traits it should be fairly easy to implement the remaining ones.

Chapter 4 - Off to the races



In our NetHack implementation, race will determine which alignments we can choose as well as some starting stat bonuses. Race will also determine your starting alignment. Dwarves are lawful, gnomes are neutral, elves and orcs are chaotic, and humans can be any alignment (but this may be restricted by the role chosen e.g. samurai are lawful). In terms of stats, dwarves are typically stronger, gnomes and elves are generally smarter, and humans are generally balanced across the stats.

We'll start implementing race by creating a `data/races.yaml` file and filling it in with the following:

```
---
- name: human
  hotkey: h
- name: dwarf
  hotkey: d
- name: gnome
```

```
  hotkey: g
- name: orc
  hotkey: o
- name: elf
  hotkey: e
```

Now let's add a `race.rb` file for loading up our races:

```
class Race
  def self.for_options(options)
    role = options[:role]

    all.select { |race| role.races.include? race.hotkey }
  end

  def self.all
    DataLoader.load_file("races").map do |data|
      new(data)
    end
  end

  attr_reader :name, :hotkey

  def initialize(data)
    data.each do |key, value|
      instance_variable_set("@#{key}", value)
    end
  end

  def to_s
    name
  end
end
```

Here we wan't to limit the selectable races to those allowed by the role. We'll need to modify our existing `data/roles.yaml` to specify which races can be selected for each role. You'll want to add `races` as a key to each role like so

```
- name: Archeologist
  hotkey: a
  races: hdg
```

The race values for the roles are as follows:

- Archeologist: hdg
- Barbarian: ho
- Caveman: hdg
- Healer: hg
- Knight: h
- Monk: h
- Priest: he
- Rogue: ho
- Ranger: hego
- Samurai: h
- Tourist: h
- Valkyrie: hd
- Wizard: hego

In terms of data, we'll also need to add the `:choosing` and `:instructions` messages for `:race`. Open `data/messages.yaml` and add the following:

```
race:
  choosing: Choosing Race
  instructions: Pick the race of your %role
```

We'll be using `%role` as a placeholder for the actual role text. In order to make this work, we'll need to change `instructions` in `selection_screen.rb` to:

```
def instructions
  @instructions ||= interpolate(messages[:instructions])
end
```

and then add the `interpolate` method:

```
def interpolate(message)
  message.gsub(/%(\w+)/) { options[$1.to_sym] }
end
```

To wrap this up we'll want to update the `attr_reader` in `role.rb` to include `:races`. Then we'll also need to change `TRAITS` in `game.rb` to include `Race` **after** `Role`. Finally add a `require` for `race` in `main.rb` before the `require` for `game`.

Now when we run the program we can select the race after the role. However there is one small annoyance. When there is only one possible race, the game should select it for us. An easy way to solve this problem is to change `random?` in our selection_screen class to:

```
def random?
  options[:randall] || items.length == 1
end
```

That way if there is only one element in the array, we'll randomly select it.

Chapter 5 - Genders

For our purposes, gender will determine which pronoun your character will be addressed with. In actual NetHack gender does affect some interactions in the game, but we won't be going that in depth with our implementation.

We're going to follow the same pattern for genders as we did with roles and races. First create a `data/genders.yaml` file with the following:

```
---
- name: male
  hotkey: m
- name: female
  hotkey: f
```

Valkyries are the only role that cannot choose between both genders. All Valkyries are female. Rather than making an exception, we'll implement this in the same manner we handled race by specifying which gender you can choose in our role file. For each of the roles aside from Valkyrie add:

```
genders: mf
```

For Valkyrie add:

```
genders: f
```

Now add `:genders` to the list of `attr_reader`s in `role.rb` and create a `gender.rb` with the following:

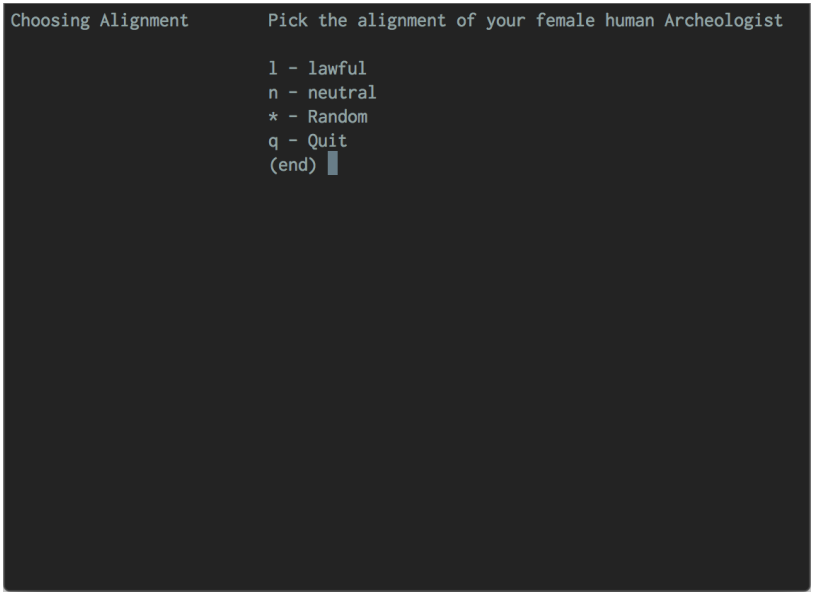
```
class Gender
  def self.for_options(options)
    role = options[:role]
    all.select { |gender| role.genders.include? gender.hotkey }
  end

  def self.all
    DataLoader.load_file("genders").map do |data|
      new(data)
    end
  end

  attr_reader :name, :hotkey

  def initialize(data)
    data.each do |key, value|
      instance_variable_set("@#{key}", value)
    end
  end

  def to_s
    name
  end
end
```

Then once again we'll need to change `data/messages.yaml` to include the `:gender` messages

```
gender:
  choosing: Choosing Gender
  instructions: Pick the gender of your %race %role
```

Finally you'll want to add `Gender` to the end of `TRAITS` in `game.rb` and add `gender` to the list of requires before `game`.

Chapter 6 - Properly aligned



Now for the final trait. Alignment determines how the actions you take in game will affect you. If you do things that contrast your alignment your god will be angry with you and the game will become more difficult.

First let's follow suit and create a `data/alignments.yaml` file with the following:

```
---
- name: lawful
```

```
  hotkey: l
- name: neutral
  hotkey: n
- name: chaotic
  hotkey: c
```

The alignments you are allowed to choose from depend on your race and role. If your role is anything other than human, your alignment is predetermined, whereas humans can choose any alignment available to the role. So for each race we need to add an `alignments` key to `data/races.yaml` with the following values:

- Human: lnc
- Dwarf: l
- Gnome: n
- Orc: c
- Elf: c

You'll need to do the same thing in `data/roles.yaml`:

- Archeologist: ln
- Barbarian: nc
- Caveman ln
- Healer: n
- Knight: l
- Monk: lnc
- Priest: lnc
- Rogue: c
- Ranger: nc
- Samurai: l
- Tourist: n
- Valkyrie: ln
- Wizard: nc

To finish off editing our data let's add the following to the end of `data/messages.yaml`:

```
alignment:
  choosing: Choosing Alignment
  instructions: Pick the alignment of your %gender %race %role
```

Now since our available alignments depend on both our role and our race we'll need to create the following `alignment.rb`:

```
class Alignment
  def self.for_options(options)
    role = options[:role]
    race = options[:race]
    possible = role.alignments.chars & race.alignments.chars

    all.select { |alignment| possible.include? alignment.hotkey }
  end

  def self.all
    DataLoader.load_file("alignments").map do |data|
      new(data)
    end
  end

  attr_reader :name, :hotkey

  def initialize(data)
    data.each do |key, value|
      instance_variable_set("@#{key}", value)
    end
  end

  def to_s
    name
  end
end
```

Finally you'll want to add `Alignment` to the end of `TRAITS` in `game.rb`, add the appropriate require before requiring `game` in `main.rb`, and add `:alignments` as an `attr_reader` to `Role` and `Race`.

Running the program now you should see it output all of chosen traits.

Chapter 7 - Generating Abilities

So once a player has chosen their role, race, gender, and alignment, we'll need to calculate their character's base abilities. In NetHack you have values, ranging between 3 and 25, for strength, wisdom, intelligence, dexterity, charisma, and constitution. We will allocate a total of 75 points to these abilities. Based on a player's role, we'll allocate a specific amount of those points to each ability. The leftover points are then allocated randomly according to rules set by the player's selected role.

Let's briefly go over each stat and how it should affect the game.

Strength will determine how much weight we can handle in our inventory. It will also determine how much melee damage we do as well as how far we can throw things. In NetHack, there are a number of other things that strength affects but we will ignore those for the purpose of out implementation. For display, stength is a bit odd because for a value between 18 and 19 it is shown as a percentage. A value of 18/35 would mean that you are 35% of the way between 18 and 19.

Dexterity will determine your chance of hitting monsters, either by melee combat, missles, or spells.

Constitution increases your healing rate and also attributes to how much weight you can carry. This is useful for roles with low strength like Tourist.

Intelligence is used for reading books and spellcasting for roles other than healers, knights, monks, priests, and valkyries.

Wisdom is used for spellcasting for healers, knights, monks, priests, and valkyries. It also determines how fast your power regenerates and how much power you gain when levelling up.

Charisma is used for getting better prices in shops.

Now for each role there's a specific assignment of points. We're going to add starting attributes to `data/roles.yaml` an entry would look something like this:

```
- name: Archeologist
  hotkey: a
```

```
  races: hdg
  genders: mf
  alignments: ln
  starting_attributes:
    strength: 7
    intelligence: 10
    wisdom: 10
    dexterity: 7
    constitution: 7
    charisma: 7
```

Add all the starting attributes according to this chart:

	strength	intelligence	wisdom	dexterity	constitution	charisma
Archeologist	7	10	10	7	7	7
Barbarian	16	7	7	15	16	6
Caveman	10	7	7	7	8	6
Healer	7	7	13	7	11	16
Knight	13	7	14	8	10	17
Monk	10	7	8	8	7	7
Priest	7	7	10	7	7	7
Rogue	7	7	7	10	7	6
Ranger	13	13	13	9	13	7
Samurai	10	8	7	10	17	6
Tourist	7	10	6	7	7	10
Valkyrie	10	7	7	7	10	7
Wizard	7	10	7	7	7	7

With the role abilities set we now need to distribute the remaining points. In order to do this we'll need to define for each role the probability that a point will be assigned to that ability. This is different for each class, so for each role we'll add the correct probability - much like we did with the `starting_attributes`. Here is an example of what you'd add after `starting_probabilities`:

```
attribute_probabilities:
  strength: 20
  intelligence: 20
```

```
  wisdom: 20
  dexterity: 10
  constitution: 20
  charisma: 10
```

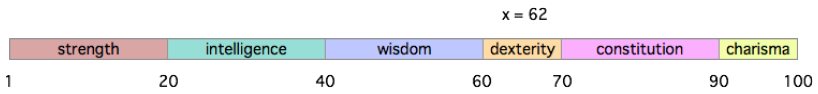
Add all the attribute probabilities in `data/roles.yaml` according to the following chart:

	strength	intelligence	wisdom	dexterity	constitution	charisma
Archeologist	20%	20%	20%	10%	20%	10%
Barbarian	30%	6%	7%	20%	30%	7%
Caveman	30%	6%	7%	20%	30%	7%
Healer	15%	20%	20%	15%	25%	5%
Knight	30%	15%	15%	10%	20%	10%
Monk	25%	10%	20%	20%	15%	10%
Priest	15%	10%	30%	15%	20%	10%
Rogue	20%	10%	10%	30%	20%	10%
Ranger	30%	10%	10%	20%	20%	10%
Samurai	30%	10%	8%	30%	14%	8%
Tourist	15%	10%	10%	15%	30%	20%
Valkyrie	30%	6%	7%	20%	30%	7%
Wizard	10%	30%	10%	20%	20%	10%

Now we're ready to create a player class. Add `player.rb` with the following:

```
class Player
  attr_reader :role, :race, :gender, :alignment, :attributes

  def initialize(options)
    @role = options[:role]
    @race = options[:race]
    @gender = options[:gender]
    @alignment = options[:alignment]

    @attributes = AttributeGenerator.new(role).attributes
  end
end
```

And now for the `AttributeGenerator`. With the `AttributeGenerator` we need to make sure we distribute the points according to the probabilities we've defined in our `roles.yaml` file. One way to solve this problem is to generate a number between 0 and 99 and then for each attribute subtract the probability from that number. Once we are less than or equal to zero we choose that attribute. In essence what we're doing is mapping the attributes on a number line and then randomly choosing a number from that line. Whichever attribute our number falls in is the one we want:



Now we can implement the `AttributeGenerator` according to this algorithm:

```
class AttributeGenerator
  def initialize(role, total = 75)
    @role = role
    @base_attributes = role.starting_attributes.dup
    @total = total
  end

  def attributes
    @attributes ||= assign_remaining_points
  end

  private

  attr_reader :role, :base_attributes, :total

  def remaining_points
    total - base_attributes.values.reduce(:+)
  end

  def assign_remaining_points
    remaining_points.times do
      increment_random_attribute
```

```
    end

    base_attributes
  end

  def increment_random_attribute
    base_attributes[next_random_attribute] += 1
  end

  def next_random_attribute
    x = rand(100)

    base_attributes.keys.find do |key|
      (x -= role.attribute_probabilities[key]) <= 0
    end
  end
end
```

Make sure you have `starting_attributes` and `attribute_probabilities` as `attr_readers` in `Role`. Also add `require`s for `Player` and `AttributeGenerator` to `main.rb`. Now in `game.rb` we can instantiate our player:

```
def setup_character
  get_traits
  options[:player] = make_player
end

def make_player
  Player.new(options).tap do
    %i(role race gender alignment).each { |key| options.delete(key) }
  end
end
```

Now for the last set of attributes we'll want to assign hitpoints and power. These operate a little differently because you need to store the current amount you have as well as your maximum. Hitpoints are determined by adding together your role's hitpoints and your race's hitpoints. Your power is determined by adding your

role's power plus some random number (usually zero) and your race's power. In `roles.yaml` and `races.yaml` add `hitpoints` and `power` and `rand_power` according to the following charts:

	hitpoints	power	rand_power
Archeologist	11	1	0
Barbarian	14	1	0
Caveman	14	1	0
Healer	11	1	4
Knight	14	1	4
Monk	12	2	2
Priest	12	4	3
Rogue	10	1	0
Ranger	13	1	0
Samurai	13	1	0
Tourist	8	1	0
Valkyrie	14	1	0
Wizard	10	4	3

---

	hitpoints	power
Human	2	1
Dwarf	4	0
Gnome	1	2
Orc	1	1
Elf	1	2

Once you've added `attr_accessor`s for `hitpoints` and `power` in both `Role` and `Race` as well as `rand_power` in `Race` you just have to add the following to your `Player` `initialize` method:

```
@hitpoints = role.hitpoints + race.hitpoints
@max_hitpoints = hitpoints
@power = role.power + rand(role.rand_power + 1) + race.power
@max_power = power
```

And then add `attr_accessors` for `hitpoints`, `max_hitpoints`, `power`, and `max_power`. There are plenty of other things we could add to our characters at this point, but this is enough to move us onward to the challenge of procedurally generated dungeons.

Dungeon Generation

Chapter 8 - Generating a Room

Rooms in NetHack are all rectangular with `.` representing the floor and walls represented by `|` or `-`.

```
----
|..|
|..|
----
```

the smallest possible room

Each room is randomly assigned a width and a height and then possibly added to the dungeon. Due to randomness, sometimes the room won't fit in the given dungeon and therefore it is discarded. This is but a single step in the process of generating a dungeon.

We're going to start our dungeon generation process by creating a single random room. To generate this room we'll need to follow a few guidelines. We'll want this room to fit entirely within our dungeon and we won't want it to be too big or too small. For instance, the minimum size of a room should be 2x2.

We'll also want our height to be more constrained than our width due to the nature of characters in a terminal taking up more vertical space than horizontal. In our case, our dungeon will be 80 characters wide by 20 characters tall. Let's

start by generating a height between 2 and 6. Add the following to a file named `dungeon_generator.rb`:

```
height = 2 + rand(4)
```

For our width, let's generate a value between 2 and 14:

```
width = 2 + rand(12)
```

In order to prevent rooms, that are too big, we'll want to set one more constraint here. Let's ensure that the area of a room is never greater than 50. We'll do this by decreasing the width if our room is too big:

```
if width * height > 50
  width = 50 / height
end
```

Now in order to place our room, let's assume that we're trying to fit it into a rectangular subsection of our dungeon. At the moment that subsection is exactly the same size as our dungeon. Later when we add more rooms, we'll create smaller subsections. Here is a `Rect` class and instance to represent that subsection:

```
Rect = Struct.new(:left, :top, :right, :bottom)

rect = Rect.new(0, 0, 80, 21)
```

Now in order to fit our room into this rectangle the left side of our room must be at least one more character to the right than the left side of our subsection. This is necessary in order to have room for the left wall. In order to randomly place the room inside our subsection, we'll want to find a suitable position for the left side of our room. The total space needed for our room is it's width plus 2 (for both walls). If we went with the equation `left = rect.left + rand(rect.right - width - 2)` we'd run into a problem because rand will generate a value between 0 and n. If it were to generate 0 then we would not have room for our wall, so the equation we actually need is:

```
left = rect.left + 1 + rand(rect.right - width - 2)
```

For the top value, we'll do the same thing:

```
top = rect.top + 1 + rand(rect.bottom - height - 2)
```

Now we can calculate the `right` and `bottom` values of our room:

```
right = left + width
bottom = top + height
```

With those room coordinates, we'll want to draw this rectangle in our "dungeon." Let's start by initializing our dungeon with stone. We'll represent stone with a space character:

```
dungeon = Array.new(21) { Array.new(80) { " " } }
```

Now let's draw the floor of our room. To do this, we'll just use a nested loop going from left to right and then top to bottom:

```
left.upto(right) do |x|
  top.upto(bottom) do |y|
    dungeon[y][x] = "."
  end
end
```

Now let's see what our dungeon looks like:

```
puts dungeon.map(&:join)
```

```
..........
..........
..........
..........
..........
..........
```

To draw the vertical walls we'll want to iterate from top to bottom adding a wall one space to the left of our room and a wall one space to the right of our room:

```
top.upto(bottom) do |y|
  dungeon[y][left - 1] = "|"
  dungeon[y][right + 1] = "|"
end
```

For the horizontal walls we'll do something similar:

```
(left - 1).upto(right + 1) do |x|
  dungeon[top - 1][x] = "-"
  dungeon[bottom + 1][x] = "-"
end
```

Now let's print our dungeon again, but this time we'll see walls:

```
---------
|.......|
|.......|
|.......|
---------
```

Because our room should be random it will be located in a new spot with different dimensions. Now that we have an idea of how to generate our random rooms, let's be more object-oriented with our code and start converting this code into objects:

```
Rect = Struct.new(:left, :top, :right, :bottom)

class DungeonGenerator
  WIDTH = 80
  HEIGHT = 21

  STONE = " "
  FLOOR = "."
  HWALL = "-"
  VWALL = "|"

  def initialize
    @dungeon = Array.new(HEIGHT) { Array.new(WIDTH) { STONE } }
    @rects = [ Rect.new(0, 0, WIDTH, HEIGHT) ]
  end
```

```
def generate
  room = create_room
  render_room(room)
  print_dungeon
end

private

attr_reader :dungeon, :rects

def create_room
  rect = rects.first

  height = 2 + rand(4)
  width = 2 + rand(12)

  if width * height > 50
    width = 50 / height
  end

  left = rect.left + 1 + rand(rect.right - width - 2)
  top = rect.top + 1 + rand(rect.bottom - height - 2)

  right = left + width
  bottom = top + height

  Rect.new(left, top, right, bottom)
end

def render_room(room)
  render_floor(room)
  render_vertical_walls(room)
  render_horizontal_walls(room)
end

def render_floor(room)
  room.left.upto(room.right) do |x|
```

```
      room.top.upto(room.bottom) do |y|
        dungeon[y][x] = FLOOR
      end
    end
  end

  def render_vertical_walls(room)
    room.top.upto(room.bottom) do |y|
      dungeon[y][room.left - 1] = VWALL
      dungeon[y][room.right + 1] = VWALL
    end
  end

  def render_horizontal_walls(room)
    (room.left - 1).upto(room.right + 1) do |x|
      dungeon[room.top - 1][x] = HWALL
      dungeon[room.bottom + 1][x] = HWALL
    end
  end

  def print_dungeon
    puts dungeon.map(&:join)
  end
end

dungeon_generator = DungeonGenerator.new
dungeon_generator.generate
```

While this code is decent, there still an opportunity to refactor it a bit further by reducing the responsibilities of our `DungeonGenerator`. Currently it knows how to create rooms and render a dungeon. Since games often have a tendency to quickly get out of control we'll want to follow the Single Responsibility Principle as closely as we can. Let's pull out a few classes with single responsibilities. For starters, let's make a class that can generate rooms

```
class RoomGenerator
  MIN_WIDTH = 2
```

```
  MIN_HEIGHT = 2
  MAX_WIDTH_MODIFIER = 12
  MAX_HEIGHT_MODIFIER = 4
  MAX_FLOOR_AREA = 50

  def initialize(rect)
    @rect = rect
  end

  def generate
    constrain_floor_area
    build_room
  end

  private

  attr_reader :rect

  def constrain_floor_area
    if floor_area > MAX_FLOOR_AREA
      @width = 50 / height
    end
  end

  def build_room
    Rect.new(left, top, right, bottom)
  end

  def floor_area
    width * height
  end

  def height
    @height ||= MIN_HEIGHT + rand(MAX_HEIGHT_MODIFIER)
  end

  def width
    @width ||= MIN_WIDTH + rand(MAX_WIDTH_MODIFIER)
```

```
  end

  def left
    @left ||= rect.left + 1 + rand(rect.right - width - 2)
  end

  def top
    @top ||= rect.top + 1 + rand(rect.bottom - height - 2)
  end

  def right
    @right ||= left + width
  end

  def bottom
    @bottom ||= top + height
  end
end
```

The great part about the class we've extracted is that we now have a great place for storing the relevant constants without polluting our `DungeonGenerator` class. We can also easily extract fairly short methods that all deal with the concept of generating a room. Had we have done this inside our `DungeonGenerator` class it would be a rather messy and confusing class.

Since we could possibly want to change the way we generate rooms in a dungeon, let's use dependency injection to allow us to change the generator. While we're at it, let's make width and height options you can pass in. First rename the constants `WIDTH` and `HEIGHT` from our `DungeonGenerator` to `DEFAULT_WIDTH` and `DEFAULT_HEIGHT` and then modify its `initialize` method to take in `options`:

```
def initialize(options = {})
  @options = options
  @dungeon = Array.new(height) { Array.new(width) { STONE }
  @rects = [ Rect.new(0, 0, width, height) ]
end
```

Then we can change the `create_room` function of our `DungeonGenerator` to utilize this like so:

```
def create_room
  RoomGenerator.new(rects.first).generate
end
```

Similarly, we'll add methods for `width` and `height`:

```
def width
  options.fetch(:width, DEFAULT_WIDTH)
end

def height
  options.fetch(:height, DEFAULT_HEIGHT)
end
```

You'll also need to make sure to add `:options` to our `attr_reader`s:

```
attr_reader :dungeon, :rects, :options
```

Another class we can now pull out is `Dungeon` itself. It seems odd that the `DungeonGenerator` knows what to render for stone, floors, and walls. In order to do this, let's first extract a `Tileset` class:

```
require_relative "data_loader"

class Tileset
  def self.load(name, loader: DataLoader)
    data = loader.load_file("tilesets/#{name}")
    new(data)
  end

  def initialize(tiles)
    @tiles = tiles
  end

  def [](key)
    tiles[key]
```

```
  end

  private

  attr_reader :tiles
end
```

Then we'll need our "default" tileset. Create the file `data/tilesets/default.yaml` with:

```
---
stone: " "
floor: "."
vertical_wall: "|"
horizontal_wall: "-"
```

Now that we have a way of loading our tilesets, let's create our `Dungeon` class:

```
class Dungeon
  attr_reader :rows

  def initialize(width, height, tileset:)
    @tileset = tileset
    @rows = Array.new(height) { Array.new(width) { tileset[:stone] } }
  end

  def build(type, x, y)
    rows[y][x] = tileset[type]
  end

  private

  attr_reader :tileset
end
```

We'll also need to update `DungeonGenerator` to load and pass along our tileset. Change the `initialize` method to:

```
def initialize(options = {})
  @options = options
  @dungeon = Dungeon.new(width, height, tileset: tileset)
  @rects = [ Rect.new(0, 0, width, height) ]
end
```

Then add the following `private` methods:

```
def tileset
  @_tileset ||= Tileset.load(tileset_name)
end

def tileset_name
  options.fetch(:tileset_name, DEFAULT_TILESET_NAME)
end
```

Then add the contant `DEFAULT_TILESET_NAME` to `DungeonGenerator`:

```
DEFAULT_TILESET_NAME = "default"
```

Now that the tileset is handling the characters of our dungeon, let's remove the constants `STONE`, `FLOOR`, `HWALL`, and `VWALL` from `DungeonGenerator`.

Next let's remove the methods used for rendering a room from our `DungeonGenerator` and extract it into a `RoomRenderer` class. To do this you'll want to remove `render_floor`, `render_vertical_walls`, and `render_horizontal_walls`. Then create the following `RoomRenderer` class:

```
class RoomRenderer
  def initialize(room, dungeon)
    @left = room.left
    @right = room.right
    @top = room.top
    @bottom = room.bottom
    @dungeon = dungeon
  end
```

```
  def render
    render_floor
    render_vertical_walls
    render_horizontal_walls
  end

  private

  attr_reader :top, :left, :right, :bottom, :dungeon

  def render_floor
    left.upto(right) do |x|
      top.upto(bottom) do |y|
        dungeon.build(:floor, x, y)
      end
    end
  end

  def render_vertical_walls
    top.upto(bottom) do |y|
      dungeon.build(:vertical_wall, left - 1, y)
      dungeon.build(:vertical_wall, right + 1, y)
    end
  end

  def render_horizontal_walls
    (left - 1).upto(right + 1) do |x|
      dungeon.build(:horizontal_wall, x, top - 1)
      dungeon.build(:horizontal_wall, x, bottom + 1)
    end
  end
end
```

Now that we have a separate class for rendering our room, we can have our `DungeonGenerator` utilize it by replacing `render_room` like so:

```
def render_room(room)
  room_renderer.new(room, dungeon).render
```

```
end

def room_renderer
  options.fetch(:room_renderer, RoomRenderer)
end
```

To be consistent, let's pull out a `DungeonPrinter` class that will print our dungeon. To make this work, we'll delete `DungeonGenerator#print_dungeon` and change the `DungeonGenerator#generate` method to look like:

```
def generate
  room = create_room
  render_room(room)
  dungeon
end
```

and add the following `DungeonPrinter` class:

```
class DungeonPrinter
  def initialize(dungeon, io = STDOUT)
    @dungeon = dungeon
    @io = io
  end

  def print
    io.puts dungeon.rows.map(&:join)
  end

  private

  attr_reader :io, :dungeon
end
```

Finally, let's change:

```
dungeon_generator = DungeonGenerator.new
dungeon_generator.generate
```

to:

```
dungeon = DungeonGenerator.new.generate
DungeonPrinter.new(dungeon).print
```

At this point, we're doing a fairly good job of adhering to the Single Responsibility Principle and it should make evolving our code go a lot more smoothly than if we hadn't. We have a lot ahead of us in order to get our DungeonGenerator to where we want. Next, we'll be tackling one of the hardest parts of generating our dungeon - adding more rooms.