### 1. Play With an Array

### Arrays are a fundamental data structure that you'll use all the time in programs. Basically any time you need to hold on to an ordered collection of objects, you'll create an array. Then, when you need to fetch one of the objects from the array, you'll ask the array for the object at a given position. We often refer to this as *indexing into the array*. And (just to keep everyone on their toes), array indices always start at zero.

### An array in Ruby is an object in its own right, which means it has state (the object references it contains) and behavior (methods to manipulate the array). A great way to get comfortable with what an array can do is to play with one in irb. So before you add arrays to the game, spend a few minutes just messin' around with arrays in irb.

### If you need a little guidance, try this:

1. Start by creating a literal array that's pre-filled with book titles, perhaps "Blink", "Freakonomics", and "Outliers".

**>> books = %w(Blink Freakonomics Outliers)  
=> [“Blink”, “Freakonomics”, “Outliers”]**

1. Now ask your array some questions:
   * How many books (elements) does it contain?
   * Which book is second in the list?
   * Which book is in the fourth slot?

**>> books.size  
=>3  
>>books[1]  
=>”Freakonomics”  
>>books[3]  
=>nil**

1. Now, start with an empty array and use array methods to fill the array with the same book titles in the same order.

**>>books = []  
=> []  
>> books.push(“Blink”)  
=> [“Blink”]  
>> books.push(“Freakonimics”)  
=>[“Blink”, “Freakonomics”]  
>> books.push(“Outliers”)  
=>[“Blink”, “Freakonomics”, “Outliers”]**

1. Then use array methods to remove each book title until the array is empty.

**>>books.pop  
=> “Outliers”  
>>books  
=> [“Blink”, “Freakonomics”]  
>> books.pop  
=>”Freakonomics”  
>>books  
=>[“Blink”]  
>>books.pop  
=>”Blink”  
>>books  
=>[]**

1. The Array class has a lot of handy methods for manipulating arrays. Knowing what you get for free can save you a lot of time, and help keep your code clean. For example, let's say you wanted to print out your list of books so that it looks like this: "Blink and Freakonomics and Outliers". That's a perfect task for the join method. Use ri to learn how to use the join method.

**ri Array.join**

1. Then use it in irb to print "Blink and Freakonomics and Outliers".

**>>books.join(“ and “)  
=>”Blink and Freakonomics and Outliers”**

1. Finally, suppose you wanted to randomly reorder all the elements in your array. Use ri to find a method that does all that in one fell swoop.

**ri Array  
ri Array.shuffle**

1. Then use it in irb to mix up the array.

**>> books = [“Blink”, “Freakonomics”, “Outliers”]  
=> [“Blink”, “Freakonomics”, “Outliers”]  
=> books.shuffle  
=> [“Outliers”, “Freakonomics”, “Blink”]**

### 

### 

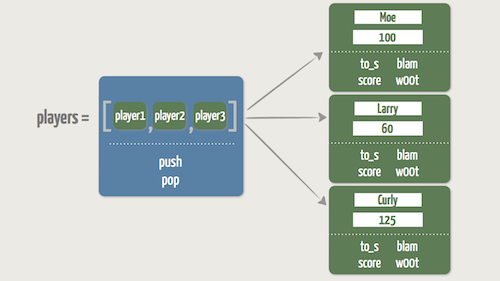
### 

### 2. Create a Array of Players and Iterate

### Now that you're a bit more comfortable with arrays, let's put one to work in our game. We currently have three Player objects that we created this way:

player1 = Player.new("moe")  
player2 = Player.new("larry", 60)  
player3 = Player.new("curly", 125)

And we currently have code that either w00ts or blams each player and prints them out. But we're doing this one player at a time. Instead, we'd like to put all the players in an array, like this (click to enlarge):



Then, once we have our player objects in an array, we can iterate through them and do the same thing to each player. We'll start by iterating through them and printing out each player's information like this:

There are 3 players in the game:  
I'm Moe with a health of 100 and a score of 103.  
I'm Larry with a health of 60 and a score of 65.  
I'm Curly with a health of 125 and a score of 130.

1. At the bottom of your studio\_game.rb file, create an array named players that contains all three Player objects (not just their name):

**players = [player1, player2, player3]**

1. Now print out "There are 3 players in the game:" where the number 3 is returned from an array method that computes the array size (don't just type in '3').

**puts “There are #{players.size} players in the game:”**

1. Finally, use the each method to iterate through all the players. In the block associated with the each method, call putsand pass in the player object that each handed to the block (the *block parameter*)

**INSERT ANSWER HERE**

Here's a tip for your next office party: Refer to the vertical bars (||) surrounding the block parameters as "goal posts". It makes you sound like a hip Ruby programmer, and sports metaphors always go over well around the office!

### 3. Create More Blocks

### Methods such as each that invoke a block of code repeatedly are often called *iterators*. A block is simply a chunk of code between do and end that's associated with a method invocation. For example, the block we just wrote was associated with a call to the each method.

### Now let's create a few more blocks to make sure you have the hang of it. We'll learn more about blocks in a future exercise, as well.

1. In the previous exercise on *Attributes*, the Health Commission asked for a report on each player's health. To produce that report, you wrote these three lines of code:
2. puts player1.health  
   puts player2.health  
   puts player3.health
3. Now that you have all the players in an array, you can do the same thing in a slightly different way. Iterate through the array of players and run a block that prints out just the health of each player.

**players.each do |player|  
 puts player.health  
end**

1. Next, iterate through the players again, but this time w00t or blam each player (you decide how many w00ts or blams) and then print out their information. For now, the same thing will happen to each player. Later on we'll make it more random. Here's an example of the output you're aiming for:

Moe got blammed!  
Moe got w00ted!  
Moe got w00ted!  
I'm Moe with a health of 120 and a score of 123.  
Larry got blammed!  
Larry got w00ted!  
Larry got w00ted!  
I'm Larry with a health of 80 and a score of 85.  
Curly got blammed!  
Curly got w00ted!  
Curly got w00ted!  
I'm Curly with a health of 145 and a score of 150.

**players.each do |player|  
 player.blam  
 player.w00t  
 player.w00t  
 puts player  
end**

1. Now suppose Curly gets sick and can't come out to play today. (You probably figured out that health was an important aspect of this game.) We need to remove him and substitute in a new player. So pop Curly out of the array and push on a new player named "Shemp" with an initial health of 90. Then when you iterate through the array of players, Shemp should be on the roster.

**players.pop  
player4 = Player.new(“Shemp”, 90)  
players.push(player4)**

1. At this point you should feel confident enough to start experimenting a little. Perhaps you want to customize your players, add a few more, or change how they're printed out. Go for it! Again, you'll use arrays and iterators all over the place in Ruby programs, so it's important that you're comfortable with them before moving on.

### Bonus Round

Fundraising Program

You're now ready to put your projects in an array. What will this allow you to do? Well, what would you like to do across all your projects? Here are a few ideas:

* Print out the number of projects in your array.
* Iterate through your projects and print each project's specific information.
* Iterate through your projects and print out the target funding amount of each project.
* Iterate through your projects adding or removing funds from each project and then print out their revised information.
* Let's say that if a project hasn't accumulated enough funding, then that project should be removed from the array. So remove one project from your list, add a new project, and print out an updated list of project information.

### class Project

### attr\_accessor :name attr\_reader :funding, :target

### def initialize(name, target\_funding\_amount, funding=0) @name = name @target = target\_funding\_amount @funding = funding end

### def to\_s "#{@name} has $#{@funding} in funding towards a goal of $#{@target}." end

### def remove\_funds @funding -= 15 puts "#{@name} lost some funds!" end

### def add\_funds @funding += 25 puts "#{@name} got more funds!" end

### def total\_funding\_outstanding @target - @funding end

### end

### project1 = Project.new("Project ABC", 5000, 1000) project2 = Project.new("Project LMN", 3000, 500) project3 = Project.new("Project XYZ", 75, 25) projects = [project1, project2, project3]

### puts "There are #{projects.size} projects that you could fund:"

### projects.each do |project| puts project end

### puts "\*\*\*" puts "Here are the target funding amounts of each project:" projects.each do |project| puts project.target end puts "\*\*\*" puts "Let's go through a round of funding requests and see what happens:"

### projects.each do |project| project.add\_funds project.remove\_funds project.add\_funds puts project end

### puts "\*\*\*" puts "Project ABC is removed and Project TBD is added. Now we have:" projects.delete(project1) project4 = Project.new("Project TBD", 10000) projects.push(project4) puts projects

### Wrap Up

### This exercise was absolutely, un-abashedly, all about arrays and iterators. We also had a quick peek at blocks as they relate to iterators. You learned how to:

* create an array
* add elements to an array
* fetch objects by indexing into an array
* iterate through an array
* define a simple block
* call methods to manipulate an array
* use the documentation to learn more array methods

Hip, hip, array! In the next exercise, we'll create a container class for managing all the players (a Game class) and see how multiple objects interact to form more complex behavior.