### Objective

### In this exercise, we want to liven up our players and make them a little more classy! By defining a class, we'll be able to easily and consistently create new objects that represent game players. We'll also use the class to encapsulate a player's behavior in methods that change the player's state.

### Our goal is to write a Player class with w00t and blam methods, as well as a to\_s method to easily print a player's information. In the end, we want to be able to run this code:

player1 = Player.new("moe")  
puts player1  
  
player2 = Player.new("larry", 60)  
puts player2  
  
player3 = Player.new("curly", 125)  
puts player3  
player3.blam  
puts player3  
player3.w00t  
puts player3

And get this result:

I'm Moe with a health of 100.  
I'm Larry with a health of 60.  
I'm Curly with a health of 125.  
Curly got blammed!  
I'm Curly with a health of 115.  
Curly got w00ted!  
I'm Curly with a health of 130.

### 

### 1. Create a Player Object

### First, we need to create a simple object that represents a player.

### To create objects in Ruby, you ask a *class* to generate one. You can think of a class as being like a factory or a blueprint for creating objects. A String class creates string objects, a Player class creates player objects, and so on. Ruby already has a built-in String class. We'll need to define our own Player class for our game.

1. Delete the code currently in your studio\_game.rb file (or save it in a different file) so you have a nice clean slate to work on. We'll end up creating the same players we had before, but this time using a class.
2. Then inside the file define a class called Player. Class names always start with an uppercase letter, and multi-word class names have each word capitalized (e.g., MagicEightBall).

**class Player  
end**

1. To create a new object, you call the new method of the associated class. Use the Player class to create a new player object and assign it to a variable named player1.

**Player1 = Player.new**

2. Initialize the Object's State

At this point we basically have an empty object. It doesn't have any state or behavior. In terms of state, we know that each player object needs to hold on to the player's name and health. That's what distinguishes one player object from another. So when we create a player object, we need to initialize its state. We do that by defining an initialize method in the class.

The initialize method is a special "constructor" method. You never call it directly. Instead, Ruby will automatically call initialize when you call the new method on the class. Any parameters you pass in to new are passed along to initialize. In short, you define an initialize method to initialize your object's state and let Ruby take care of the rest.

Inside the initialize method we store an object's state in *instance variables*. The values assigned to those instance variables are unique to each object. For example, each player object has a different name and health. Instance variables always start with the @ sign, and they spring into existence the first time you assign to them. Unlike a method's local variables which evaporate when the method is finished, instance variables live for the life of the object, so we can use them throughout our app.

1. Write an initialize method inside your Player class that takes two parameters: name and health. Save each parameter value in an instance variable.

**def initialize(name, health)  
 @name = name  
 @health = health  
end**

1. Remember, spelling counts in programming, and if you misspell initialize then the method won't get called.
2. Now that you have an initialize method that takes two parameters, you'll get an error if you try to create a player object by calling new with no parameters. So change how you create the player by passing in the name "moe" (lowercase) and a health of 100.

**player1 = Player.new(“moe”, 100)**

1. Check your work by calling the built-in inspect method on the player object like so:
2. puts player1.inspect
3. The inspect method prints out what's inside an object, including the name of its instance variables and their values. Check to make sure that the parameters you passed to new got transferred into the correct instance variables. When you run the program, you should see something like this
4. *#<Player:0x007fd2238badb0 @name="moe", @health=100>*
5. Next, we want a player's name to always be capitalized. But we don't want to have to remember to always pass in a capitalized name when we create a player object. The intent of the initialize method is to put objects in a good usable state, so it can take care of automatically capitalizing the name.
6. In initialize, capitalize the name parameter value before assigning it to its respective instance variable.

**def initialize(name, health)  
 @name = name.capitalize  
 @health = health  
end**

1. Run the program again, and check that the inspected name is indeed capitalized, like so
2. *#<Player:0x007fd2238badb0 @name="Moe", @health=100>*
3. Finally, let's make creating player objects a tad easier. Let's suppose most players will start off with a health value of 100. That being the case, if a health is not specified for a player then it should default to a value of 100.
4. Change the initialize method so that the health parameter has this default value.

**def initialize(name, health=100)  
 @name = name.capitalize  
 @health = health  
end**

1. Hey, Moe's health is the same as the default value! Go figure. Now you can change how you create Moe by not passing in an initial health value.

**Player1 = Player.new(“moe”)**

### 3. Define the Object's Behavior

### Now that we have Moe's information stored in the Player object's state, we'd like to be able to tell the player to introduce himself to the game. For example, when we run the following code:

player1 = Player.new("moe")  
puts player1.say\_hello

we'd like it to print out:

I'm Moe with a health of 100.

In addition to holding on to state, objects can also have methods (behavior) that operate on an object's state. To add behavior to an object, you define methods inside the class. These methods are shared by all objects of the same class

Then, to tell an object to do something, you invoke a method by sending a message to an object. The message includes the method's name and any required parameters. We refer to an object's methods as its *instance methods* because they must be called with the object (instance) as the receiver. Instance methods have access to the object's instance variables, which means the object's methods can act upon the object's state.

1. In the previous exercise, you wrote a stand-alone say\_hello method (not in a class) that took two parameters like this
2. **def** **say\_hello**(name, health=100)  
    "I'm #{name.capitalize} with a health of #{health}."  
   **end**
3. We want to move this method into the Player class. Inside the class we can use the instance variables for the player's name and health rather than passing them in to the say\_hello method.
4. Define a say\_hello instance method in the Player class that returns the same string as the stand-alone method, but doesn't take parameters.

**class Player  
 def initialize (name, health=100)  
 @name= name.capitalize  
 @health = health  
  
  
 def say\_hello  
 “I’m #{@name} with a health of #{@health}.”  
 end  
end**

1. Now make sure it works by calling the say\_hello instance method on the player1 object. Remember that the say\_hello method simply returns a string but doesn't print it out, so you'll need to specifically print out the result of calling the method. Here's the output you're aiming for: I'm Moe with a health of 100.

**player1 = Player.new(“Moe”)  
puts player1.say\_hello**

1. Excellent! Now create two more players: larry and curly. When you ask them to introduce themselves (by calling the say\_hello method), they should print:
2. I'm Larry with a health of 60.  
   I'm Curly with a health of 125.

**player2 = Player.new(“Larry”, 60)  
puts player2.say\_hello  
  
player3 = Player.new(“Curly”, 125)  
puts player3.say\_hello**

1. Now we have *three* player objects, each with distinct state and a shared set of behavior!

### 4. Add State-Changing Behavior

### OK, we have to admit it—the game is pretty darn boring so far! So our next task is to add a bit more behavior to the player objects to make them more interesting.

### Here's what we want to have happen: A player should start with an initial health value. If a player gets "w00ted", their health value should increase by 15. But if a player gets "blammed", their health value should decrease by 10. In other words, a player's health (state) should change depending on which method (behavior) is invoked.

### For example, when we run this code:

player3 = Player.new("curly", 125)  
puts player3.say\_hello  
player3.blam  
puts player3.say\_hello  
player3.w00t  
puts player3.say\_hello

We should get the following output:

I'm Curly with a health of 125.  
Curly got blammed!  
I'm Curly with a health of 115.  
Curly got w00ted!  
I'm Curly with a health of 130.

To do that, we'll need to add two more instance methods to the Player class: blam and w00t.

1. Add a blam instance method to the Player class that decreases a player's health by 10 and prints out a message saying that the player got blammed.

**def blam  
 @helath -=10  
 puts “#{@name} got blammed!”  
end**

1. Now blam the Curly player and call say\_hello to check that his health decreased by 10.

**player3.blam  
puts player3.say\_hello**

1. Next add a w00t instance method to the Player class that increases a player's health by 15 and prints out a message saying that the player got w00ted.

**def w00t  
 @health +=15  
 puts “#{@name} got w00ted!”  
end**

1. Now w00t the Curly player and call say\_hello to check that his health increased by 15.

**player3.w00t  
puts player3.say\_hello**

1. Go ahead, experiment a little! w00t and blam another player object and notice how its state changes independent of the other players. That's the beauty of objects: they manage their own internal state. Kinda like objects in the real world...

It's important to remember that methods encapsulate behavior in one place in our program. Suppose one day we decide to change the definition of what it means to w00t a player. For example, suppose we need to adjust the game mechanics by increasing the w00t value to 20. How many places would we need to update that number? Exactly one: in the w00t method.

The w00t method is the single point in the program that determines what happens when a player is w00ted. It encapsulates that behavior. By defining the w00t method, we avoid introducing duplication in our program. And unnecessary duplication is something we try very hard to eliminate in programs. Because if there's one thing true about programs, it's that change is inevitable.

### 5. Print the Object

### Currently, when we want to print a player's information we have to call the say\_hello method. Throughout the game we'll likely need to print a player's information many times, so let's make that a bit easier to do.

### If you define an instance method called to\_s in a class, Ruby will automatically call to\_s when the object needs to be rendered as a string. When do you need to render an object as a string? Typically when you pass the object to puts.

### All classes inherit a default to\_s method (more on inheritance later), but the default implementation returns a fairly generic (and cryptic) string representation of the object. To get a classier string representation of players, we'll need to define a custom to\_s method in the Player class.

1. Our Player class doesn't currently define a to\_s method. So when you pass a player object to puts, the result isn't very helpful. Go ahead, try this:
2. puts player1
3. You should see output like this:
4. *#<Player:0x007ff27a8bade8>*
5. (We warned you it wasn't very helpful or classy.)
6. Instead, we want the output to include the player's name and health. For example, Moe should be printed as "I'm Moe with a health of 100." That would definitely be more helpful. And, as it turns out we already have an instance method called say\_hello that does exactly that.
7. Rename say\_hello to to\_s. (We won't use say\_hello any more.)

**def to\_s  
 “I’m #{@name} with a health of #{health}.”  
end**

1. **Gotcha:** A common pitfall when defining a to\_s method is to have it actually print the object using puts. Instead, the to\_s method must simply return a string representing the object. Ruby will automatically call the to\_s method whenever you call puts player1, for example. The string returned from the to\_s method is then printed out to the console by the puts method.
2. Now change your program so that rather than calling say\_hello on each player object you instead pass the player object to the puts method to print the player's information.

**puts player1  
puts player2  
puts player3**

1. Going forward, any time you need to print a player's information you can simply pass the player object to puts, which turns around and calls the to\_s method to get the player's string representation.

6. Vocabulary Quiz

### Learning about classes and objects introduces a bunch of new vocabulary. To make things even more challenging, some of the new words have synonyms. So, if you're new to OO programming, take a minute to review the vocabulary and let it sink in before moving on. (Fill in the blanks!)

1. A **class** is a factory (blueprint) that instantiates **objects**.
2. Objects have unique **state** and a common set of **behaviors**.
3. An **instance** variable exists for the life of an **object**.
4. **Instance** methods have access to their object's **instance** variables.
5. Class names start with an **uppercase** letter.
6. Method names start with a **lowercase** letter.
7. Variable names start with a **lowercase** letter.
8. Instances variables start with an **@ sign**.

### Bonus Round

#### Calculator Class

If you'd like some extra practice with classes and objects, give this a whirl. Create another class called Calculator in a separate Ruby program file (not related to our game). Implement the class so that when you run this code...

calc = Calculator.new(20, 11)  
puts calc.add  
puts calc.subtract

...you get this output:

31  
9

class Calculator

def initialize(a, b)

@a = a

@b = b

end

def add

@a + @b

end

def subtract

@a - @b

end

end

#### Fundraising Program

The fundraising program is all about raising funds, so let's get to it! Create a Project class. Within this class, each project should have, at a minimum, a name, an initial funding amount, and a target funding amount. Then, add methods for adding funds to a project, removing funds (if a project doesn't meet its deliverables, for example), and displaying a project's status. So, if you call the method to add funds, a project's funding should increase by 25, for example. And if you call the method to remove funds, a project's funding should decrease by 15. If you want to get tricky, pass a value into these methods and either increase or decrease by that value. The output should be something like:

Project LMN has $500 in funding towards a goal of $3000.  
Project XYZ has $25 in funding towards a goal of $75.  
Project LMN lost some funds!  
Project XYZ got more funds!  
Project LMN has $485 in funding towards a goal of $3000.  
Project XYZ has $50 in funding towards a goal of $75.

class Project

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

end

project1 = Project.new("Project ABC", 5000, 1000)

puts project1

project2 = Project.new("Project LMN", 3000, 500)

puts project2

project3 = Project.new("Project XYZ", 75, 25)

puts project3

project4 = Project.new("Project TBD", 10000)

puts project4

puts "\*\*\*"

project1.remove\_funds

project2.remove\_funds

project3.add\_funds

project4.add\_funds

puts "\*\*\*"

puts project1

puts project2

puts project3

puts project4