The high-level objective of this exercise is to use a hash to keep track of a player's treasures and their respective point totals. We'll print out the hash after each round like so:

Round 3:  
Curly got blammed!  
Curly found a skillet worth 100 points.  
Curly's treasures: {:crowbar=>400, :bottle=>25, :skillet=>100}  
Larry got w00ted!  
Larry found a bottle worth 25 points.  
Larry's treasures: {:broomstick=>200, :hammer=>50, :bottle=>25}  
Moe got blammed!  
Moe found a hammer worth 50 points.  
Moe's treasures: {:bottle=>25, :hammer=>100}

And at the end of the game we'll print out each player's total treasure points:

Moe's point totals:  
125 grand total points  
  
Curly's point totals:  
525 grand total points  
  
Larry's point totals:  
275 grand total points

As always, we'll work through this incrementally and use specs as a guide...

### 1. Play With a Hash

### Along with arrays, you'll use hashes all the time in Ruby programming. Indeed, mastery of arrays and hashes is key (get it?) to being an effective Ruby programmer. So before adding hashes to the game, spend a few minutes just goofing around with hashes in irb to get comfortable.

### Let's try an example to illustrate what we're going to do with treasures in the game...

1. Start by creating a literal hash called moes\_treasures with a hammer worth 50 points.

>> moes\_treasures = { :hammer => 50}

1. Next, suppose Moe finds a crowbar worth 400 points. Add that key-value pair to the hash.

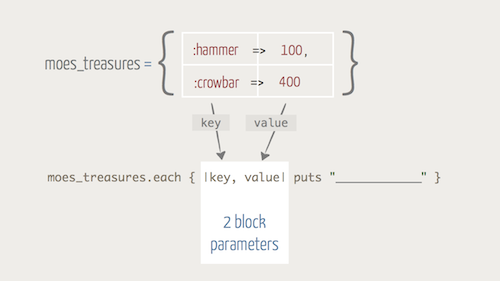
>> moes\_treasures[:crowbar] = 400

1. Now let's ask the moes\_treasures hash some basic questions:
   * How many items does it contain?
   * What's the point value for the hammer?
   * What's the point value for the crowbar?
   * What's the point value for the balloon?

>> moes\_treasures.size  
=> 2  
>> moes\_treasures[:hammer]  
=> 50  
>> moes\_treasures[:crowbar]  
=> 400  
>> moes\_treasures[:balloon]  
=> nil

1. So far, so good. Now suppose Moe finds another hammer. The new value associated with the :hammer key should be the sum of the existing hammer points (50) and the new hammer points (50) for a total of 100 hammer points.
2. Add 50 points to the value of the :hammer key and then print the total value for that key.

>> moes\_treasures[:hammer] += 50  
>> moes\_treasures[:hammer]  
=> 100

1. Now iterate through the hash and print out each key-value pair, such as "hammer adds up to 100 points." Note that calling each on a hash calls the associated block once for each key in the hash, and passes the key-value pair via *two* block parameters: the key and the value.
2. 

>> moes\_treasures.each { |key, value| puts "#{key} adds up to #{value} points" }

1. Notice that iterating over the entries in a hash returns them in the order in which they were added to the hash.
2. Next, find and use a hash method that returns an array of all the *keys* in the hash.

>> moes\_treasures.keys  
=> [:hammer, :crowbar]

1. Then find and use a method that returns an array of all the *values* in the hash.

>> moes\_treasures.values  
=> [100, 400]

1. OK, now it's time to tally up the total points. To do that, you'll need to *reduce* (hint) the hash values down to a single number.

>> moes\_treasures.values.reduce(0, :+)  
=> 500

1. Not too shabby, Moe!

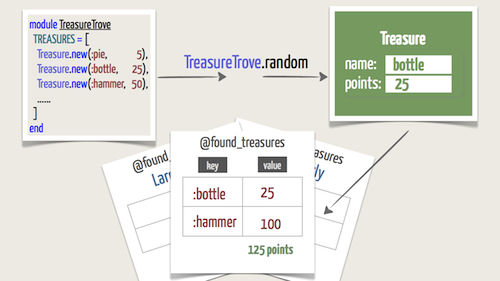
2. Store a Player's Treasures

Now that you're a bit more comfortable with hashes, let's apply what you've learned to the game.

Currently, on each player's turn the game picks a random treasure and prints it out, for example: "Moe found a broomstick worth 200 points." We now want to store the treasure with each player as they take their turns.

Furthermore, we want to store the total points accumulated for any given treasure. We also need a way to tally up a player's grand total treasure points. After all, the obvious measure of a resourceful player is how many treasure points they've accumulated!

Here's a high-level visual view of where we're going:



Here's another way to think about our objective. Each of our players currently has the state (instance variables) and behaviors (methods) in the left column in the table below. In this exercise, we want to add in the state and behaviors in the right column.

|  |  |
| --- | --- |
| **Current State** | **State To Be Added** |
| @name | @found\_treasures hash |
| @health |  |
|  |  |
| **Current Methods** | **Methods To Be Added** |
| to\_s | found\_treasure |
| blam | points |
| w00t |  |
| score |  |
| strong? |  |
| <=> |  |

For a high-level testing view of where we're going, check out the spec below. Read through the code examples. Then copy and paste the examples into your existing player\_spec.rb file, and require treasure\_trove.rb, so you can check your work as you go through the steps below.

Let's get started....

1. To keep track of the treasures that a player finds, each player will need it's own hash. The hash keys are the treasure names and the values are the accumulated points for that treasure (just like we did in irb). Because this is part of a player's state, we'll store the hash in an instance variable called @found\_treasures. In other words, each player will have its own distinct hash of found treasures and their point values.
2. Start by adding an instance variable called @found\_treasures to the Player class. Initialize it to an empty hash with a default value of 0.

class Player  
 def initialize(name, health=100)  
 @name = name.capitalize  
 @health = health  
 @found\_treasures = Hash.new(0)  
 end  
end

1. Then, still in the Player class, define a method called found\_treasure that takes one parameter: a Treasure object. Even though a Treasure object is a Struct with two attributes, we want to pass in the entire Treasure object. Therefore, the method only needs one parameter.

def found\_treasure(treasure)  
end

1. Implement the found\_treasure method so that it adds a key-value pair to the @found\_treasures hash. The *key* is the name of the treasure in the Treasure object and the *value* is the sum of the points of that treasure. Remember, we want to use the hash to accumulate the total points for each found treasure, so add the points in the Treasure object to any existing points. Hint: make sure to use += like you did in irb.

def found\_treasure(treasure)  
 @found\_treasures[treasure.name] += treasure.points  
end

1. After updating the @found\_treasures hash, print out what was found. For example: "Larry found a hammer worth 50 points." You're already printing this in the game\_turn.rb file, but it makes more sense to put it in found\_treasure now.

def found\_treasure(treasure)  
 @found\_treasures[treasure.name] += treasure.points  
 puts "#{@name} found a #{treasure.name} worth #{treasure.points} points."  
end

1. Also print out the contents of the @found\_treasures hash so you can see how the hash changes throughout the game. For example, "Curly's treasures: {:hammer=>50, :skillet=>200}".

def found\_treasure(treasure)  
 @found\_treasures[treasure.name] += treasure.points  
 puts "#{@name} found a #{treasure.name} worth #{treasure.points} points."  
 puts "#{@name}'s treasures: #{@found\_treasures}"  
end

1. Next, define a method in the Player class called points that returns the sum of all the player's treasure points. This is another example of a *virtual accessor* that returns a value derived from an instance variable (@found\_treasures in this case).

def points  
 @found\_treasures.values.reduce(0, :+)  
end

1. Run the updated player\_spec.rb spec to make sure all the specs pass.

3. Assign a Treasure Each Turn

At this point, we can assign a treasure to a player by calling the found\_treasure method. However, nowhere in the game are we calling that method (it's just in our spec).

So, when should we assign a treasure to a player? Well, we want it to happen when he takes a turn. Remember that all the code related to taking a turn is encapsulated in the take\_turn method of the GameTurn module. Currently that method prints out a random treasure, but doesn't assign it to the player taking a turn.

Let's fix that. Add the following code example to your game\_spec.rb file that expects a player to have points (from treasures) after his turn:

it "assigns a treasure for points during a player's turn" **do**  
 game = Game.new("Knuckleheads")  
 player = Player.new("moe")  
  
 game.add\_player(player)  
  
 game.play(1)  
  
 player.points.should\_not be\_zero  
  
 *# or use alternate expectation syntax:*  
 *# expect(player.points).not\_to be\_zero*  
**end**

The goal, of course, is to get it to pass...

1. Change the take\_turn method to call the found\_treasure method on the player object taking a turn, passing in the randomly-found treasure to the method. (Remove the extra print out of what was found since it's now in the found\_treasure method.)

def self.take\_turn(player)  
 treasure = TreasureTrove.random  
 player.found\_treasure(treasure)  
end

1. Run your specs and they should pass.
2. Just as a sanity check, go ahead and run the entire game program. Make sure each player prints out what they found and what they've already accumulated. For example, here's some sample output of two rounds of play:

Round 2:  
Curly was skipped.  
Curly found a bottle worth 25 points.  
Curly's treasures: {:crowbar=>400, :bottle=>25}  
Larry got blammed!  
Larry found a hammer worth 50 points.  
Larry's treasures: {:broomstick=>200, :hammer=>50}  
Moe got w00ted!  
Moe found a hammer worth 50 points.  
Moe's treasures: {:bottle=>25, :hammer=>50}  
  
Round 3:  
Curly got blammed!  
Curly found a skillet worth 100 points.  
Curly's treasures: {:crowbar=>400, :bottle=>25, :skillet=>100}  
Larry got w00ted!  
Larry found a bottle worth 25 points.  
Larry's treasures: {:broomstick=>200, :hammer=>50, :bottle=>25}  
Moe got blammed!  
Moe found a hammer worth 50 points.  
Moe's treasures: {:bottle=>25, :hammer=>100}

1. There's a lot going on here. If you get any errors, take the time to read them *very* carefully.

4. Include Points in a Player's Score

### Currently a player's overall score is derived from their health and the length of their name. Here's the score method we wrote in an earlier exercise:

**class** Player  
 **def** **score**  
 @health + @name.length  
 **end**  
**end**

And when we print out a player, we get something like this:

I'm Larry with a health of 150 and a score of 155.

We said we'd change the score method at a later time. Well, we finally have a good reason to change it! Since a player now accumulates treasure points, we want to change a player's score to be the sum of their health and their points. For example, if Larry has a health of 150 and found treasures adding up to 100 total points, then he should have a score of 250. Also, we want to see the following when we print him:

I'm Larry with health = 150, points = 100, and score = 250.

Let's work through these two objectives in a test-first style...

1. Start by changing your existing score example in the player\_spec.rb spec to the following:
2. it "computes a score as the sum of its health and points" **do**  
    @player.found\_treasure(Treasure.new(**:hammer**, 50))  
    @player.found\_treasure(Treasure.new(**:hammer**, 50))  
     
    @player.score.should == 250  
   **end**
3. Run the player\_spec.rb file and the example above should fail. Remember, the code still computes the score based on the player's health and the length of their name.
4. To make the spec pass, start by changing the score method to return the sum of the player's health and points.

def score  
 @health + points  
end

1. Re-run the spec and now the score example should pass, but the example for the player's string representation should fail. The computed score at this point is 250, but the string representation example is expecting it to be 155 per our old scoring strategy.
2. To fix this, update your existing to\_s example in the player\_spec.rb spec to the following:
3. it "has a string representation" **do**  
    @player.found\_treasure(Treasure.new(**:hammer**, 50))  
    @player.found\_treasure(Treasure.new(**:hammer**, 50))  
     
    @player.to\_s.should == "I'm Larry with a health of 150 and a score of 250."  
   **end**
4. Re-run the player\_spec.rb and all the examples should now pass.
5. That takes care of the scoring. But remember that we also wanted to change the string representation for a player so that it includes the points and is formatted like so:
6. I'm Larry with health = 150, points = 100, and score = 250.
7. So update the to\_s example to expect that string.

it "has a string representation" do  
 @player.found\_treasure(Treasure.new(:hammer, 50))  
 @player.found\_treasure(Treasure.new(:hammer, 50))  
  
 @player.to\_s.should == "I'm Larry with health = 150, points = 100, and score = 250."  
end

1. Run the player\_spec.rb file and the example above should fail.
2. Make the spec pass by changing the to\_s method in the Player class so that it includes the points and score in the proper format.

def to\_s  
 "I'm #{@name} with health = #{@health}, points = #{points}, and score = #{score}."  
end

1. Re-run the spec and if everything passes, then you're done!

### 5. Update Game Stats

Finally, at the end of the game we'd like to print out the total points accumulated by each player throughout the game. In the next exercise we'll change this to include a breakdown of the points on a *per-treasure* basis. For now, we're just interested in total points for each player, so the stats at the end of the game should look something like this:

Moe's point totals:  
125 grand total points  
  
Curly's point totals:  
525 grand total points  
  
Larry's point totals:  
275 grand total points

1. In the print\_stats method of the Game class, iterate through each player and print out their total points.

@players.each do |player|  
 puts "\n#{player.name}'s point totals:"  
 puts "#{player.points} grand total points"  
end

1. Run the program and make sure you get the output you expect.
2. At the end of the game you should also see the high score list we created a few exercises back. So what changes are needed now that each player is accumulating points? Thankfully, none. Check your code and you'll recall that the high score list calls the score method for each player. And because we changed the score method to include points, that's now reflected in the high score list (the numbers should be bigger). Voila!

### 

### Bonus Round

### Grand Total Points

### Feeling confident, are you? We're glad you're up for a small challenge. This one will help reinforce arrays and iterators, and it dovetails on the code we just added.

### Suppose Moe has found 100 points worth of treasures and Larry has found 400 points worth of treasures. How would you go about adding a game statistic that prints out the grand total of *all* the points for *all* the players, like so?

### 500 total points from treasures found

### Here's a code example that you can add to your game\_spec.rb file that expresses how it should work:

### it "computes total points as the sum of all player points" **do** game = Game.new("Knuckleheads") player1 = Player.new("moe") player2 = Player.new("larry") game.add\_player(player1) game.add\_player(player2) player1.found\_treasure(Treasure.new(**:hammer**, 50)) player1.found\_treasure(Treasure.new(**:hammer**, 50)) player2.found\_treasure(Treasure.new(**:crowbar**, 400)) game.total\_points.should == 500 **end**

### Define a total\_points method on the Game class that iterates through each player and sums up the total points. This is another example of a *virtual attribute* in that the total\_points method doesn't return the value of a @total\_pointsinstance variable. Instead, the total points are computed when the method is called.

### def total\_points @players.reduce(0) { |sum, player| sum + player.points } end

### Call that method in the print\_stats method and print out the total treasure points found during the game.

puts "#{total\_points} total points from treasures found"

### Run the specs to make sure all the code examples pass.

#### Scrabble

### Here's some extra practice with hashes and blocks that has nothing to do with our game, but it's fun nonetheless. Suppose we wanted to simulate a game of [Scrabble](http://en.wikipedia.org/wiki/Scrabble_letter_distributions). In Scrabble, each letter has an associated point value and the score of a word is the sum of the points of all the letters in the word. In our abbreviated version of the game, we'll use the following set of letters and their associated point values:

|  |  |
| --- | --- |
| **Letter** | **Points** |
| c | 3 |
| e | 1 |
| l | 1 |
| n | 1 |
| t | 1 |
| x | 8 |
| y | 4 |

### 

### The goal is to write a program that calculates the score for the word "excellently". It would also be handy to know the total points accumulated for any given letter. For example, "excellently" has three "e" letters, each worth 1 point. So the total points for the letter "e" is 3 (1 + 1 + 1). To pull this off, you'll need to use a hash to represent the distribution of letters and their point values. Then you can iterate through the characters in the word and look up each character's point value in the hash.

### In irb, start by creating a literal hash called letters with the letters (as strings) and their associated point values shown in the table above.

>> letters = {"c" => 3, "e" => 1, "l" => 1, "n" => 1, "t" => 1, "x" => 8, "y" => 4}

### Now iterate through the characters in the word "excellently" and calculating the word score by looking up each character's point value in the letters hash and accumulating the values in a variable called score. Here's a hint: The String class has an each\_char method that calls a block, passing in each character at a time. You might want to look up that method in the documentation as a starting point.

>> score = 0  
=> 0  
>> "excellently".each\_char { |c| score += letters[c] }  
=> "excellently"  
>> score  
=> 23

### Now that we have the word score, we'd like to know the total points accumulated for each letter in the word. For example, given the word "excellently", we want to create a hash with the following keys and values:

### {"e" => 3, "x" => 8, "c" => 3, "l" => 3, "n" => 1, "t" => 1, "y" => 4}

### To do that, you'll need to iterate through the characters in the word and look up each character's point value in the letters hash, just as before. But in this case you'll need to use a new hash (call it point\_totals, for example) to accumulate the total points for each letter.

>> point\_totals = Hash.new(0)  
=> {}  
>> "excellently".each\_char { |c| point\_totals[c] += letters[c] }  
=> "excellently"  
>> point\_totals  
=> {"e"=>3, "x"=>8, "c"=>3, "l"=>3, "n"=>1, "t"=>1, "y"=>4}

### Finally, now that you have the point\_totals hash, you can use it to calculate the word score a different way. This time, calculate the word score by reducing (hint!) all the point values down to a single number.

>> point\_totals.values.reduce(0, :+)  
=> 23

#### Fundraising Program

### If you've been working on the fundraising program (and we hope you have) then you know that each time a project goes through a round of funding, two things happen:

### Funds are randomly added or removed from each project based on the roll of a die.

### Each project randomly receives a bronze, silver, or gold pledge.

### The output looks something like this:

### Funding Round 1: Project ABC lost some funds! Project ABC received a silver pledge worth $75. Project LMN got more funds! Project LMN received a gold pledge worth $100. Project XYZ lost some funds! Project XYZ received a bronze pledge worth $50.

### But what we aren't doing (yet) is actually accumulating received pledges for each project. Project ABC may get a silver pledge in round 1 and a gold pledge in round 2, but we aren't holding on to these pledges. The money is simply slipping through our fingers. Yikes! We gotta fix that!

### Using hashes, keep track of a project's pledges and the respective amounts. Then, after each round print out a hash something like this:

### Funding Round 1: Project ABC lost some funds! Project ABC received a silver pledge worth $75. Project ABC's pledges: {:silver=>75} Funding Round 2: Project ABC got more funds! Project ABC received a gold pledge worth $100. Project ABC's pledges: {:silver=>75, :gold=>100} Funding Round 3: Project ABC lost some funds! Project ABC received a gold pledge worth $100. Project ABC's pledges: {:silver=>75, :gold=>200}

### Next, you need to change the way funding is calculated. The funding amount is currently only impacted by the remove\_fundsand add\_funds methods. Now that we're accumulating the pledge amounts, these should also be included in the project's funding calculation. Here's an example of what you could do: If ProjectABC's initial funding is $100 and more funds (+$25) have been randomly added, and then it receives a gold pledge (worth $100), the total funds should be $225.

### Be sure to change all the appropriate stats to reflect your newly calculated funding amount.

### 

### fund\_request.rb

require\_relative 'project'

require\_relative 'funding\_round'

require\_relative 'pledge\_pool'

class FundRequest

attr\_reader :title

def initialize(title)

@title = title

@projects = []

end

def add\_project(a\_project)

@projects.push(a\_project)

end

def request\_funding(rounds)

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

@projects.each do |project|

puts project

end

pledges = PledgePool::PLEDGES

puts "\nThere are #{pledges.size} possible pledge amounts:"

pledges.each do |pledge|

puts "A #{pledge.name} pledge is worth $#{pledge.amount}."

end

1.upto(rounds) do |round|

puts "\nFunding Round #{round}:"

@projects.each do |project|

FundingRound.one\_round(project)

puts project

end

end

end

def print\_name(project)

puts "#{project.name}"

end

def print\_results

fully\_funded\_projects = @projects.select { |project| project.fully\_funded? }

under\_funded\_projects = @projects.reject { |project| project.fully\_funded? }

puts "\n#{fully\_funded\_projects.size} Fully Funded Projects:"

fully\_funded\_projects.each do |project|

print\_name(project)

end

puts "\n#{under\_funded\_projects.size} Under Funded Projects:"

under\_funded\_projects.each do |project|

print\_name(project)

end

sorted\_projects = under\_funded\_projects.sort { |a, b| b.total\_funding\_outstanding <=> a.total\_funding\_outstanding}

puts "\n#{under\_funded\_projects.size} projects still need your help:"

sorted\_projects.each do |project|

formatted\_name = project.name.ljust(20, '.')

puts "#{formatted\_name} $#{project.total\_funding\_outstanding} under"

end

end

end

### fund\_request\_specifications.rb

require\_relative 'fundrequest'

describe FundRequest do

before do

@fundrequest = FundRequest.new("VC-Friendly Start-up Projects")

@initial\_funds = 1000

@project = Project.new("Project ABC", 5000, @initial\_funds)

@fundrequest.add\_project(@project)

end

it "adds funds to a project if an even number is rolled" do

Die.any\_instance.stub(:roll).and\_return(4)

@fundrequest.request\_funding(2)

@project.funding.should == @initial\_funds + (25 \* 2)

end

it "removes funds to a project if an odd number is rolled" do

Die.any\_instance.stub(:roll).and\_return(3)

@fundrequest.request\_funding(2)

@project.funding.should == @initial\_funds - (15 \* 2)

end

it "assigns a pledge for amount during a project's funding round" do

fundrequest = FundRequest.new("VC-Friendly Start-up Projects")

project = Project.new("Project ABC", 5000)

fundrequest.add\_project(project)

fundrequest.request\_funding(1)

project.pledges.should\_not be\_zero

end

end

### pledge\_pool.rb

Pledge = Struct.new(:name, :amount)

module PledgePool

PLEDGES = [

Pledge.new(:bronze, 50),

Pledge.new(:silver, 75),

Pledge.new(:gold, 100)

]

def self.random

PLEDGES.sample

end

end

### pledge\_pool\_specifications.rb

require\_relative 'pledge\_pool'

describe Pledge do

before do

@pledge = Pledge.new(:bronze, 50)

end

it "has a name attribute" do

@pledge.name.should == :bronze

end

it "has an amount attribute" do

@pledge.amount.should == 50

end

end

describe PledgePool do

it "has three pledges" do

PledgePool::PLEDGES.size.should == 3

end

it "has a bronze pledge worth $50" do

PledgePool::PLEDGES[0].should == Pledge.new(:bronze, 50)

end

it "has a silver pledge worth 75 points" do

PledgePool::PLEDGES[1].should == Pledge.new(:silver, 75)

end

it "has a gold pledge worth 100 points" do

PledgePool::PLEDGES[2].should == Pledge.new(:gold, 100)

end

it "returns a random pledge" do

pledge = PledgePool.random

PledgePool::PLEDGES.should include(pledge)

end

end

### project.rb

require\_relative 'pledge\_pool'

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

@received\_pledge = Hash.new(0)

end

def to\_s

"#{@name} has $#{total\_funds} 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 - total\_funds

end

def fully\_funded?

total\_funding\_outstanding <= 0

end

def received\_pledge(pledge)

@received\_pledge[pledge.name] += pledge.amount

puts "#{@name} received a #{pledge.name} pledge worth $#{pledge.amount}."

puts "#{@name}'s pledges: #{@received\_pledge}"

end

def pledges

@received\_pledge.values.reduce(0, :+)

end

def total\_funds

@funding + pledges

end

end

if \_\_FILE\_\_ == $0

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

puts project.name

puts project.funding

project.remove\_funds

puts project.funding

project.add\_funds

puts project.funding

end

### project\_specifications.rb

require\_relative 'project'

describe Project do

before do

@initial\_funds = 1000

@project = Project.new("Project ABC", 5000, @initial\_funds)

$stdout = StringIO.new

end

it "has an initial target funding amount" do

@project.target.should == 5000

end

it "computes the total funds outstanding as the target funding amount minus the funding amount" do

@project.total\_funding\_outstanding.should == (5000 - 1000)

end

it "increases funds by 25 when funds are added" do

@project.add\_funds

@project.funding.should == @initial\_funds + 25

end

it "decreases funds by 15 when funds are removed" do

@project.remove\_funds

@project.funding.should == @initial\_funds - 15

end

context "created without a funding amount" do

before do

@project = Project.new("ProjectABC", 5000)

end

it "has a default funding amount of 0" do

@project.funding.should == 0

end

end

context "when total funding outstanding is less than or equal to 0" do

before do

@project = Project.new("ProjectABC", 5000, 5000)

end

it "is fully-funded" do

@project.should be\_fully\_funded

end

end

context "when total funding outstanding is greater than 0" do

before do

@project = Project.new("ProjectABC", 5000, 1000)

end

it "is under-funded" do

@project.should\_not be\_fully\_funded

end

end

it "computes pledges as the sum of all pledges" do

@project.pledges.should == 0

@project.received\_pledge(Pledge.new(:silver, 75))

@project.pledges.should == 75

@project.received\_pledge(Pledge.new(:gold, 100))

@project.pledges.should == 175

@project.received\_pledge(Pledge.new(:gold, 100))

@project.pledges.should == 275

end

it "computes total funds as the sum of a projects funding and pledges" do

@project.received\_pledge(Pledge.new(:gold, 100))

@project.received\_pledge(Pledge.new(:gold, 100))

@project.total\_funds.should == 1200

end

end

### crowdfund.rb

require\_relative 'project'

require\_relative 'fundrequest'

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

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

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

projects = FundRequest.new("VC-Friendly Start-up Projects")

puts projects.title

projects.add\_project(project1)

projects.add\_project(project2)

projects.add\_project(project3)

projects.request\_funding(3)

projects.print\_results

### die.rb

class Die

attr\_reader :number

def initialize

roll

end

def roll

@number = rand(1..6)

end

end

if \_\_FILE\_\_ == $0

die = Die.new

puts die.roll

puts die.roll

puts die.roll

end

### funding\_round.rb

require\_relative 'project'

require\_relative 'die'

require\_relative 'pledge\_pool'

module FundingRound

def self.one\_round(project)

die = Die.new

number\_rolled = die.roll

if number\_rolled.odd?

project.remove\_funds

else

project.add\_funds

end

pledge = PledgePool.random

project.received\_pledge(pledge)

end

end

### Wrap Up

### Whew, we made a lot of changes in this exercise which required putting a whole bunch of concepts together! We started by getting a feel for how hashes worked in irb. Then we applied that toward accumulating found treasures for each player. Using a hash let us keep a running total of points for each treasure that a player found. This is a fairly typical use of hashes.

### You learned how to:

### create a hash

### add key-value pairs to a hash

### look up values by indexing into a hash

### iterate through a hash

### call methods to manipulate a hash

### write another virtual accessor

### In the next section we'll arrange things so the game can print out a breakdown of a player's points on a per-treasure basis. To do that, we'll need to write our own custom iterator.

### 