### Objective

### Now that we can play multiple rounds, it's time to introduce another facet of the game. Think of it as a treasure hunt of sorts!

### We'll arrange things so that the game has six treasures that can be found. Each treasure will have a name and a point value. At the beginning of the game we'll print out all the treasures and their respective point values, like this:

There are 6 treasures to be found:  
A pie is worth 5 points  
A bottle is worth 25 points  
A hammer is worth 50 points  
A skillet is worth 100 points  
A broomstick is worth 200 points  
A crowbar is worth 400 points

Then, every time a player takes a turn, they'll randomly "find" one of the treasures. In the next exercise, we'll actually have the player accumulate treasures. For now, we'll simply print out what each player found. So for the purposes of this exercise, our objective is to print something like the following for each round:

Round 1:  
Curly got w00ted!  
Curly found a broomstick worth 200 points.  
Larry was skipped.  
Larry found a hammer worth 50 points.  
Moe got w00ted!  
Moe found a pie worth 5 points.

Remember, each player finds a *random* treasure, so the results will vary from round to round.

### 

### 

### 1. Play with Symbols

### We're going to name the treasures in our game using symbols, not strings. In fact, this isn't the first time we've used symbols. We used them before when we created attributes for classes. Here's what that looked like for the Player class:

**attr\_accessor** **:name**  
**attr\_reader** **:health**

Symbols start with a colon, thus :name and :health are symbols. When you see a symbol think "named" because that's all a symbol really is: a convenient way to name or identify something in your code. For example, in the code above we're saying we want attributes named "name" and "health". Notice that we don't have to assign anything to the symbol—it's a constant name. Ruby makes sure there's only one value for the :name symbol, for example, in the entire program.

Now, folks often ask: what's the difference between a symbol and a string literal? Well, they're different objects entirely, and you use them in different situations. You use a symbol when you simply want to name or identify something in your program. For example, if we were writing a program to manage books, we might refer to the [Programming Ruby](https://pragprog.com/book/ruby4/programming-ruby-1-9-2-0) book as:

**:programming\_ruby**  
or

**:pickaxe**

You use a string when you need an object with string-like behavior. For example, to represent the text of the book (where the text is something we painstakingly edit), we would use a string:

"This book is a new version of the PickAxe..."

Let that sink in a bit as you dig a bit deeper into symbols from the comfort of an irb session. This is a bit of a diversion from the game, but it's instructive. And lots of people ask. So go with us for a minute.

1. Fire up an irb session (you should probably just keep one running in a separate window all the time, eh?). Then call the class method on the symbol :crowbar like this:
2. >> **:crowbar**.class  
   => Symbol
3. OK, so symbols are (simple) objects that Ruby creates from the Symbol class. To learn more about the class, have a quick peek at its documentation. To do that, open a *seperate* console window, *outside* of the irb session, and use the ricommand to view the documentation:
4. ri Symbol
5. In particular, notice how few methods you can call on a symbol. Contrast that with the number of methods in the Stringclass. Strings have more behavior.
6. As the documentation says, the same Symbol object will be created for a given name for the duration of the program. In other words, a particular symbol always references the same underlying Symbol object. To prove that, back in the window with the irb session call the object\_id method on the :crowbar symbol multiple times.

>> :crowbar.object\_id  
=> 460328  
>> :crowbar.object\_id  
=> 460328

1. Interesting! Notice that the symbol :crowbar always has the same object identifier. A :crowbar is a :crowbar is a :crowbar. That is, there's only one :crowbar in the entire program—it's a constant name. So far, so good.
2. Now create a string literal "crowbar" and call its class method, like so
3. >> "crowbar".class  
   => String
4. OK, so using a string literal creates a new String object. That makes sense. Now call the object\_id method on the string literal "crowbar" multiple times.

>> "crowbar".object\_id  
=> 70365078699620  
>> "crowbar".object\_id  
=> 70365078694040

1. Ah! Notice that string literals create *new* String objects—they each have unique object identifiers.
2. So are strings and symbols the same thing? Certainly not. They are different objects. Try this: Use the == comparison operator to compare the symbol :crowbar to the string literal "crowbar", like so:
3. >> **:crowbar** == "crowbar"  
   => false
4. Ruby, of course, knows that these are different objects.
5. Now, in reality, anywhere you could use a symbol you generally could use a string instead. (The opposite is not always true.) And converting between the two is easy to do, and you may find you need to do the conversion yourself at times. Convert the string literal "crowbar" to its equivalent symbol :crowbar using the to\_sym method.

>> “crowbar”.to\_sym  
=> :crowbar

1. Then convert the symbol :crowbar back to its string equivalent using the to\_s method.

>> “crowbar”.to\_s  
=> “crowbar”

1. Also worth noting is that a symbol can't be changed (it's immutable) whereas a string can be changed (it's mutable). Let's find out! Try this:
2. >> **:crowbar**[0] = "k"
3. You should get an error. This, however, should work:
4. >> "crowbar"[0] = "k"  
   => "k"

So if you can get away with using strings in place of symbols, why not just always use strings? It boils down to a matter of intent and convenience. It's simply more idiomatic in Ruby to use symbols to name things in code. Strings are used when you're dealing with text because string objects have a boatload of methods for doing string manipulation and text processing.

All that being said, it's easy to over think symbols. We hope this irb diversion answered more questions than it raised. For now, just think of a symbol as a constant name. With more experience you'll totally "get" symbols.

### 

### 

### 

### 2. Experiment with a Struct

### In the game we'll need to model the concept of a treasure. The simplest way to represent a treasure is using two attributes: a name and a point value. We could create an explicit class with just two attributes, but that's overkill.

### Instead, we'll use a Struct, which is a convenient way to bundle related chunks of data (attributes) together without having to write an explicit class. Calling Struct.new with a list of attributes (symbols) returns a Class object that has accessor methods defined for each of the attributes. The class object can then be used to create specific instances (objects) of the Struct.

### If that explanation sounds fairly abstract, that's ok. It's easier just to show a Struct in action, but to be honest the Structsyntax looks a bit strange the first couple times you see it. Again, it's good to play in irb first. Try this example we showed you in the video for yourself:

>> Snack = Struct.new(**:name**, **:carbs**)  
=> Snack

At this point, it's as if we explicitly wrote a class called Snack with accessor methods called name and carbs. Given the Snackclass object, we can carry on creating a specific snack object using Snack.new like so

>> tasty\_snack = Snack.new(**:nachos**, 40)  
=> #<struct Snack name=:nachos, carbs=40>

This tasty snack has the name :nachos and 40 carbs. Notice we used a symbol for the name of the snack (:nachos) simply because we're naming the snack and we don't need a full-blown string object. Again, symbols name things.

Remember, the symbols you pass to Struct.new—in this case :name and :carbs—act like regular attributes. So once we have the nachos snack object, we can call the accessor methods as follows:

>> tasty\_snack.name  
=> :nachos  
>> tasty\_snack.carbs  
=> 40

Although it's not relevant to this exercise, it's worth pointing out that the attributes of a Struct can also be modified by assigning values like so:

>> tasty\_snack.name = **:totopos**  
>> tasty\_snack.carbs = 30  
  
>> tasty\_snack.name  
=> :totopos  
>> tasty\_snack.carbs  
=> 30

Again, we could have gone to the trouble of explicitly writing a Snack class with name and carbs attributes, but the Structeffectively does that for us. Plus, it gave us an opportunity to learn something new about Ruby.

### 3. Create the Treasure Trove

### Returning our attention back to the game (whew!), we need to create a Treasure struct with name and points attributes. Then we'll use it to create six Treasure objects in an array called TREASURES (a constant) in a TreasureTrove module.

### That's a mouthful, so let's express it in code examples. Click the link below to see the code examples for this objective. Read through them so you understand where we're headed. Then copy and paste the code examples into a new spec file named treasure\_trove\_spec.rb so you can check your work:

require\_relative 'treasure\_trove'  
   
describe Treasure **do**  
   
 before **do**  
 @treasure = Treasure.new(**:hammer**, 50)  
 **end**  
   
 it "has a name attribute" **do**  
 @treasure.name.should == **:hammer**  
 **end**  
   
 it "has a points attribute" **do**  
 @treasure.points.should == 50  
 **end**  
   
**end**  
  
describe TreasureTrove **do**  
   
 it "has six treasures" **do**  
 TreasureTrove::TREASURES.size.should == 6  
 **end**  
   
 it "has a pie worth 5 points" **do**  
 TreasureTrove::TREASURES[0].should == Treasure.new(**:pie**, 5)  
 **end**  
   
 it "has a bottle worth 25 points" **do**  
 TreasureTrove::TREASURES[1].should == Treasure.new(**:bottle**, 25)  
 **end**  
   
 it "has a hammer worth 50 points" **do**  
 TreasureTrove::TREASURES[2].should == Treasure.new(**:hammer**, 50)  
 **end**  
   
 it "has a skillet worth 100 points" **do**  
 TreasureTrove::TREASURES[3].should == Treasure.new(**:skillet**, 100)  
 **end**  
  
 it "has a broomstick worth 200 points" **do**  
 TreasureTrove::TREASURES[4].should == Treasure.new(**:broomstick**, 200)  
 **end**  
   
 it "has a crowbar worth 400 points" **do**  
 TreasureTrove::TREASURES[5].should == Treasure.new(**:crowbar**, 400)  
 **end**  
  
**end**

Depending on if this is your first, second, or third time through the course, you may decide to try to get the spec to pass on your own, or you may prefer to follow the steps below...

In a new file called treasure\_trove.rb, create a new Struct with attributes named name and points. Assign the returned class object to a variable called Treasure (it starts with an uppercase letter because it's a class object).

Treasure = Struct.new(:name, :points)

In the same file, define a TreasureTrove module.

module TreasureTrove  
end

Next we want to create a collection of specific treasures that are always available in the treasure trove. Inside the TreasureTrove module, define a constant called TREASURES that's an array. Fill the array with six Treasure objects. The treasures for this game are as follows:

|  |  |
| --- | --- |
| **name** | **points** |
| pie | 5 |
| bottle | 25 |
| hammer | 50 |
| skillet | 100 |
| broomstick | 200 |
| crowbar | 400 |

module TreasureTrove

TREASURES = [  
 Treasure.new(:pie, 5),  
 Treasure.new(:bottle, 25),   
 Treasure.new(:hammer, 50),  
 Treasure.new(:skillet, 100),  
 Treasure.new(:broomstick, 200),  
 Treasure.new(:crowbar, 400)  
]  
end

Finally, run the treasure\_trove\_spec.rb spec to make sure all the code examples pass.

### 4. Print the Treasure Trove

### Now that we have a trove of treasures that a player can find in the game, let's print them out at the beginning of the game. Here's what we want the output to look like:

There are 6 treasures to be found:  
A pie is worth 5 points  
A bottle is worth 25 points  
A hammer is worth 50 points  
A skillet is worth 100 points  
A broomstick is worth 200 points  
A crowbar is worth 400 points

To do that, we'll need to access the TREASURES constant from within the Game class. And because the TREASURES constant lives inside of the TreasureTrove module, we'll need to use the scope resolution operator (::). In other words, we need to refer to the constant as TreasureTrove::TREASURES.

Keep in mind that while TreasureTrove::TREASURES is a constant, it's also an array. And we already know that we can call all sorts of methods on an array (such as getting its size). We also already know that we can iterate through an array (think "block"). In this case, iterating through the TreasureTrove::TREASURES array would pass each Treasure object as a block parameter. Then we can turn around and use accessors to ask each treasure for its name and point value.

1. First, in the play method of the Game class, after printing all the players, assign the TreasureTrove::TREASURES array to a local variable named treasures:

treasures = TreasureTrove::TREASURES

1. Although we don't necessarily need a local variable, it makes using the array a bit easier.
2. Now we can treat it just like arrays we've used previously. First print the number of treasures in the array. Then iterate through the treasures and print out each treasure's name and point value as shown above.

puts "\nThere are #{treasures.size} treasures to be found:"  
 treasures.each do |treasure|  
 puts "A #{treasure.name} is worth #{treasure.points} points"  
 end

1. Run the game to make sure it now prints out the treasure items. Keep in mind, game.rb now has a dependency on the TreasureTrove module, so you'll need to require the treasure\_trove.rb file.

### 5. Find a Random Treasure

### Now that we have all our treasures in a trove, we want to randomly "find" a treasure every time a player takes a turn. For the purposes of this exercise, we'll just print out the treasure's name and point value. Here's the output we're aiming to print out, from a random round of play:

Round 2:  
Curly got blammed!  
Curly found a crowbar worth 400 points.  
Larry got w00ted!  
Larry found a pie worth 5 points.  
Moe got w00ted!  
Moe found a pie worth 5 points.

1. In the TreasureTrove module, define a module method called random that returns a random treasure. Remember that module methods are defined on the self variable, unlike regular instance methods. The method needs to return a random treasure from the TREASURES constant. Again, it's a constant that references an *array*, which means we can call any array methods. For example, getting a random element from an array is as easy as calling the sample method.
2. To check your work, add the following code example to the TreasureTrove description in your treasure\_trove\_spec.rb file.

module TreasureTrove  
 def self.random  
 TREASURES.sample  
 end  
end

1. it "returns a random treasure" **do**  
    treasure = TreasureTrove.random  
     
    TreasureTrove::TREASURES.should **include**(treasure)  
      
    *# or use alternate expectation syntax:*  
    *# expect(TreasureTrove::TREASURES).to include(treasure)*  
   **end**

module TreasureTrove  
 def self.random  
 TREASURES.sample  
 end  
end

1. Now, every time a player takes a turn, we need to call the random method and print out the random treasure that was found, like this
2. Curly found a crowbar worth 400 points.
3. Where should the call to the random method live? Think about it for a minute. Go look at your code. Deciding where to put code is an important part of object-oriented programming.
4. At first blush, we might be tempted to put the call in the play method of the Game class, right after the call to GameTurn.take\_turn(player). But finding a treasure is an intrinsic part of taking a turn, so it makes more sense to put the code as the last step in the take\_turn method of the GameTurn class. Go ahead, make that work.

module GameTurn  
 def self.take\_turn(player)  
 die = Die.new  
 case die.roll  
 when 1..2  
 player.blam  
 when 3..4  
 puts “#{player.name} was skipped.”  
 else  
 player.w00t  
 end  
 treasure = TreasureTrove.random  
 puts “#{player.name} found a #{treasure.name} worth #{treasure.points} points.”  
end  
end

1. Putting that logic in the take\_turn method encapsulates all the logic related to a player taking a turn in a single method, rather than spreading it out in multiple places. And it means the play method has less responsibility which makes it read better:
2. 1.upto(rounds) **do** |round|  
    puts "\nRound #{round}:"  
    @players.each **do** |player|  
    GameTurn.take\_turn(player)  
    **end**  
   **end**
3. Run the game and make sure each player finds a treasure when he takes a turn. (Did you remember to addtreasure\_trove as a required file in game\_turn.rb?)
4. Also make sure to run all your specs:
5. rspec . --color
6. Green dots all around? Excellent!

### Bonus Round

### Fundraising Program

### So far in the fundraising program, you've been able to add funds (in $25 increments) and remove funds (in $15 increments) to a project. But what if you have some really generous friends who want to contribute more than $25 per funding round? Let them have at it, we say! In your fundraising app, layer in the concept of pledges of varying amounts using symbols and structs.

### Here's one idea of how it could work:

1. Each pledge has a name and a pledge amount, say $50, $75, and $100. When your app begins, you print out how many projects you have (you've already done this part), the initial status of each project (you've already done this part too), and the possible pledges that could be given (this is the new part). Something like:
2. There are 3 possible pledge amounts:  
    A bronze pledge is worth $50.  
    A silver pledge is worth $75.  
    A gold pledge is worth $100.
3. Then, each time a project goes through a round of funding, two things happen. First, funds are randomly added or removed from each project based on the roll of a die (you've already written this part). Secondly, each project will randomly receive a bronze, silver, or gold pledge (this is the new part). Something like:
4. 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.

Hints: Model the concept of a pledge as a struct with name and amount attributes. Then use the struct to create three Pledgeobjects in an array called PLEDGES (a constant) in a PledgePool module.

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\_and\_funding(project)

puts "#{project.name} ($#{project.funding})"

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\_and\_funding(project)

end

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

under\_funded\_projects.each do |project|

print\_name\_and\_funding(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

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

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

def fully\_funded?

total\_funding\_outstanding <= 0

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("Project ABC", 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("Project ABC", 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("Project ABC", 5000, 1000)

end

it "is under-funded" do

@project.should\_not be\_fully\_funded

end

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

crowdfund.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

fuding\_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

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

end

end