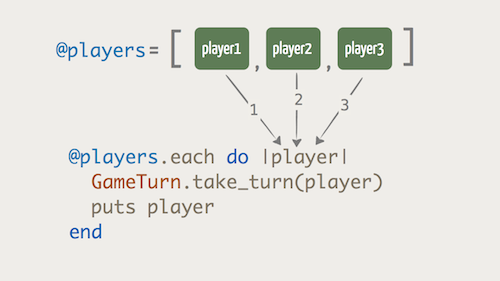
### 1. Play Multiple Rounds

Inside the play method of the Game class, we're already calling the each method with a block to iterate through all the players and let them take their one turn. It looks like this:

@players.each **do** |player|  
 GameTurn.take\_turn(player)  
 puts player  
**End**

Remember that blocks in Ruby are simply chunks of code between braces (single-line blocks) or between do and end (multi-line blocks). You can't execute a block of code directly. Instead, you associate a block with a method invocation, in this case the each method. Note, however, that the block isn't immediately executed. Rather, when the associated method runs it turns around and executes the block one or more times. For example, the each method executes the block once for each element in the array.

So in the code above, if you have three players in the @players array, the block will get executed three times. For each player in the array, the player object is assigned to the player block parameter and the block is executed like this:



Methods such as each that act like loops—methods that execute a block repeatedly—are called *iterators*.

As it turns out, you can nest iterators inside of other iterators (blocks inside of blocks). And that's a fairly common thing to want to do. Think of it like a clock: for every one iteration of the minute hand there are 60 iterations of the second hand. Our game has similar mechanics. We want to iterate through a number of "rounds" where each round then iterates through all the players to give them a turn.

To do that, we'll change the play method to take a parameter called rounds that represents the number of rounds to play. Then we'll iterate through each round and give each player a turn. For example, the output for a two-round game (subject to randomness, of course) would look something like this:

Round 1:  
Moe got blammed.  
I'm Moe with a health of 93 and a score of 96.  
Larry got blammed.  
I'm Larry with a health of 55 and a score of 60.  
Curly was skipped.  
I'm Curly with a health of 130 and a score of 135.  
  
Round 2:  
Moe got blammed.  
I'm Moe with a health of 86 and a score of 89.  
Larry was skipped.  
I'm Larry with a health of 60 and a score of 65.  
Curly got blammed.  
I'm Curly with a health of 125 and a score of 130.

So, that's our first objective, which again we can (and should!) express in the specs to easily check our work. Recall that the game\_spec.rb file already has code examples for the play method. However, you'll need to change those examples to account for this new behavior. Specifically, when the specs call play they need to pass a parameter for the number of rounds.

Now let's write some blocks to make our specs pass...

1. Start by changing the definition of the play method to take a parameter called rounds which represents the number of rounds to play.

def play(rounds)  
 ….  
end

1. Then, inside the play method, use the upto method to iterate from 1 to the number of rounds specified in the roundsparameter. For each round, print the round number on a new line and give each player a turn.

def play(rounds)  
 puts “There are #{@players.size} players in #{@title} :”  
  
 @players.each do |player|  
 puts player  
 end  
  
 1.upto(rounds) do |round|  
 puts “\nRound #{round}:”  
 @players.each do |player|  
 GameTurn.take\_turn(player)  
 puts player  
 end  
 end  
end

1. Run the revised game\_spec.rb file to check your work!
2. Even if the specs pass, you can no longer run a game because the code in the main program is now broken. It's still calling play without a parameter. (That's an example of why unit tests aren't always enough.) In the main studio\_game.rbprogram file, change how you call the play method to include the number of rounds you want to play.

knuckleheads.play(3)

1. Give it a whirl, and experiment a little! Try a game with lots of rounds. Try a game with lots of rounds *and* lots of players. Don't worry, Ruby can handle it. Go crazy!

### 2. Experiment with More Iterators

### Our next objective is to use blocks to tally up statistics for the game. But before jumping straight into that, let's spend a few minutes in irb just getting comfortable with a few more iterator methods. We can also practice a bit with single-line blocks.

### Seeing a pattern here? We almost always practice with new methods in irb first. It's faster (and more fun) than plunking the methods directly into our program and then having to run the entire thing to get any output. With irb, the feedback is nearly instantaneous, and that's a huge win when you're learning something new. So we suggest always keeping an irb session open and using it frequently.

1. Spark up an irb session and create an array with these 4 lucky lottery numbers: 24, 13, 8, 65. (Fine print: If you win big on these numbers, you agree to share the bounty!)

>> numbers = [24,13,8,65]

1. Using a block, iterate through the numbers and select (wink, wink) the numbers greater than 20.

>> numbers.select { |number| number > 20}  
=> [24, 65]

1. Now do the opposite: iterate through the array and *reject* the numbers greater than 20.

>> numbers.reject { |number| number > 20}  
=> [13,8]

1. OK, now sort the numbers from lowest to highest.

>> numbers.sort  
=> [8,13,24,65]

1. And then sort them from highest to lowest.

>> numbers.sort { |a,b| b ⇔ a}  
=> [65,24,13,8]

1. You might be wondering if these four small lucky numbers add up to one big lucky number. See for yourself by using an array method to sum up all the numbers.

>> numbers.reduce {|sum, n| sum + n}  
=> 110

1. Finally, in one deft move, partition the numbers into two variables called evens and odds.

>>evens, odds = numbers.partition { |n| n.even?}  
=> [[24, 8], [13,65]]  
>>evens   
=> [24,8]  
>>odds  
=> [13,65]

If you're still a little hazy on the block syntax, now would be a good time to play with a few more collections of numbers. Try your own lucky numbers. Try using the rand method to generate random (possibly-lucky) numbers. Or if you're bored of numbers, try an array of strings. For example, select and reject all the strings that start with a particular letter. Sort them alphabetically and then by their length. And if you're feeling particularly fond of specs, try codifying what you learned in a new spec file.

Practice, practice, practice!

### 3. Print Game Statistics

### Turning our attention back to the game, let's put what we've learned about iterators to good use. When the game is over, we want to print out some simple statistics. For example, after all those rounds of play, it would be handy to know the number of strong and wimpy players and partition them out.

### The objective is to print out something like this at the end of the game:

Knuckleheads Statistics:  
  
1 strong players:  
Curly (130)  
  
2 wimpy players:  
Moe (70)  
Larry (65)

Think about how you might do that. Obviously, the actual numbers will be unique for every game played.

1. In the Game class, define an instance method called print\_stats. Inside that method, use the iterator methods you tried in irb in the previous steps to create two arrays: one containing only the strong players and the other containing only the wimpy players. (Recall that your Player class already has a strong? method that returns true or false.)

def print\_stats  
 strong\_players = @players.select { |player| player.strong?}  
 wimpy\_players = @players.reject { |player| player.strong?}  
end

1. Now, still in the print\_stats method, print out the statistics in the format shown above. You'll need to iterate through each of the strong players and each of the wimpy players. Also, note that you only need to print the player's name and health, not all the player's information.

def print\_stats  
 strong\_players = @players.select {|player| player.strong?}  
 wimpy\_players = @players.reject{ |player| player.strong?}

puts “\n#{@title} Statistics:”  
puts “\n#{strong\_players.size} strong players:”  
strong\_players.each do |player|  
 puts “#{player.name} (#{player.health})”  
end  
  
puts “\n#{wimpy\_players.size) wimpy players: “  
wimpy\_players.each do |player|  
 puts “#{player.name} (#player.health})”  
end  
end

1. Then, back over in the main program (studio\_game.rb), call the method to print out the statistics after playing all the rounds.

knuckleheads.print\_stats

1. Run the game, and make sure you're seeing the output we set as the objective above before moving on.

### 4. Print High Scores

### Moving onward and upward, we desperately need a way to find out which player actually won the game! How do we know who's the winner? Well, the winner is the player with the highest score, of course. And a player already knows his score because we defined a score method on the Player class:

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

So it would be really groovy if at the end of the game we printed a list of high scores just like your favorite old-school arcade game. You know, something like this:

Knuckleheads High Scores:  
Curly............... 125  
Moe................. 93  
Larry............... 65

You've already learned everything you need to pull this off, so give it a go.

1. Adding to what's already in the print\_stats method, use a method you learned in irb to sort the players by their score, high to low. Assign the outcome to a variable called sorted\_players.

sorted\_players = @players.sort { |a,b| b.score ⇔ a.score}

1. Now, print out the high score list in the format shown above. You'll need to use yet another block to iterate through each of the sorted players and print out their name and score.

sorted\_players = @players.sort { |a,b| b.score ⇔ a.score}  
  
puts “\n#{@title} High Scores:”  
sorted\_players.each do |player|  
 formatted\_name = player.name.ljust(20, ‘.’)  
 puts “#formatted\_name} #{player.score}”

end

1. Now that you have that working, let's do a small refactoring. Somewhere else in the game (stay tuned), we'll likely need to sort the players. Override the <=> method (the general comparison, or spaceship, operator) in the Player class so that any time you call sort on an array of players it always returns them sorted by descending score. Here's a spec you can add to your player\_spec.rb file to use as a guide:
2. context "in a collection of players" **do**  
    before **do**  
    @player1 = Player.new("moe", 100)  
    @player2 = Player.new("larry", 200)  
    @player3 = Player.new("curly", 300)  
     
    @players = [@player1, @player2, @player3]  
    **end**  
     
    it "is sorted by decreasing score" **do**  
    @players.sort.should == [@player3, @player2, @player1]  
    **end**  
   **end**

class Player  
 def ⇔ (other)  
 other.score ⇔ score  
 end  
end

1. Finally, back in the print\_stats method in the Game class, remove the sorted\_players variable (we know, you just added it!). Instead, use the default sort method on the array of players. Remember that if you call sort on an array, without giving it a block, behind the scenes sort will automatically call the <=> method to determine how to sort the objects.

puts “\n#{@title} High Scores:”  
@players.sort.each do |player|  
 formatted\_name = player.name.ljust(20, ‘.’)  
 puts “#{formatted\_name} #{player.score}”  
end

1. Run your code (which you've been doing all along, right?) to make sure everything shakes out and to see who won. Run it a few more times because we know you want your favorite player to win.

Groovy, indeed!

### 5. Refactor

### Now that you have the code working, let's take a moment to consider if the print\_stats could benefit from any refactoring. Are you picking up the scent of any "code smells"? There's no right or wrong answer here—it's a design question that's often worth asking.

### Sometimes rephrasing the question helps. So here's a better question: What if you needed to change how the strong and wimpy players were printed? Suppose, for example, instead of printing the name followed by the health, you wanted to print the health first. How many places in the code would need to be changed? Well, currently that would be two places. And that's one too many.

### Think about how you might eliminate this duplication. Try it on your own.

### Here's one approach called the *extract method* refactoring: In the Game class, define a print\_name\_and\_health method that takes a player object. Move the code that prints the player's name and health into this method. Then call the print\_name\_and\_health method from the print\_stats method wherever you need to print a strong or wimpy player.

def print\_name\_and\_health(player)  
 puts “#{player.name} (#{player.health})”  
end  
  
def print\_stats  
 puts “\n#{@title} Statistics:”  
  
 strong\_players, wimpy\_players = @players.partition { |player| player.strong?}  
 puts “#{strong\_player.size} strong players:”  
 strong\_players.each do |player|  
 print\_name\_and\_health(player)  
 end  
  
 puts “#{wimpy\_players.size} wimpy players:”  
 wimpy\_players.each do |player|  
 print\_name\_and\_health(player)  
 end  
end

How far do you go when it comes to refactoring? Unfortunately, there are no hard and fast rules. It's a judgement call that gets easier to make as you gain more experience. For now, focus on making sure classes and methods aren't doing too much and removing any obvious duplication.

More With Blocks

If you want to flex your block muscles, return to the lottery numbers and try these practice steps:

1. Start with these numbers in irb:
2. => numbers = [24, 13, 8, 65]
3. Suppose you'd like to know if the array has any lottery numbers that are greater than 20. To do that, look up the documentation for the [any?](http://ruby-doc.org/core-2.0.0/Enumerable.html#method-i-any-3F) method. Call that method with a block to see if any lottery numbers are greater than 20.

>> numbers.any? { |number| number > 20}  
=> true

1. Now that you know the array has at least one number greater than 20, use the Array.detect method to find and return the first number in the array that is greater than 20.

>> numbers.detect {|numbers| number > 20}  
=> 24

1. Next find a method that maps (hint) the items in your array into a new array containing the values returned by the given block. Use this method to create a new array that contains all the lottery numbers multiplied by 2. Check that your original array is unchanged.

>> numbers\_doubled = numbers.map {|number| number \* 2}  
=> [48,26,16,130]  
=> numbers  
=> [24,13,8,65]

1. Spend some time reviewing all the methods in the [Enumerable](http://ruby-doc.org/core-2.0.0/Enumerable.html) module. This is where a lot of this magic happens.Enumerable is included in the Array class as well as other collection classes in Ruby. (We'll talk about how that happens a bit later.) It's a great example of the power of blocks. Learning the ins and outs of Enumerable is time very well spent.

Fundraising Program

How could you use blocks in your fundraising program? There are lots of options. To get you started, here are a few ideas:

* Instead of a one-time request for funding, change your app to allow for multiple rounds of funding requests (or at least until your friends start to avoid you because you're asking them for money a little too often).
* When the funding rounds are over, print out the number of projects that are fully-funded and under-funded.
* Print off a list of projects that still need contributions, sorted by amount outstanding.

### 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

### funding\_round.rb

require\_relative 'project'

require\_relative 'die'

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

end

end

### fund\_request.rb

require\_relative 'project'

require\_relative 'die'

require\_relative 'funding\_round'

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

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

### 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

### 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

### Wrap Up

### Now you really know your way around the ol' Ruby block! (Sorry, we couldn't help ourselves.) This was a meaty exercise and you learned a lot along the way, including how to:

* use { }, | |, and <=> when writing blocks
* call methods that take blocks (such as each, times, upto)
* write blocks that take parameters (for the number of rounds in the game)
* use blocks for list processing (for our list of players)
* and to really challenge your brain cells, code up blocks inside of blocks (in order for each player to take a turn each round)