### 1. Interact with the Game

First, we want to be able to interact with the game through keyboard input. So when you run the game, we want it to prompt users like this:

How many game rounds? ('quit' to exit)

1. To start with, change the studio\_game.rb main program to include a loop that prompts (think puts) for user input.

loop do  
 puts "\nHow many game rounds? ('quit' to exit)"  
end

1. Then, get the user's answer and make sure it's in a format we can use.

loop do  
 puts "\nHow many game rounds? ('quit' to exit)"  
 answer = gets.chomp.downcase  
end

1. Finally, use a case statement to direct the flow through the possible paths:
   * If the user enters a number, play the specified number of rounds and then print the prompt again. The regular expression for matching a number can be difficult to remember, so go ahead and use /^\d+$/.
   * If the user enters the words "quit" or "exit", print out the game statistics and break out of the loop.
   * Otherwise, if any other input is given, print a helpful message, such as "Please enter a number or 'quit'", followed by the prompt.
2. You'll end up moving the calls to the play and print\_stats methods inside of the case statement.
3. Take your time and work through each possible path incrementally.

loop do  
 puts "\nHow many game rounds? ('quit' to exit)"  
 answer = gets.chomp.downcase  
case answer  
when /^\d+$/  
 knuckleheads.play(answer.to\_i)  
when 'quit', 'exit'  
 knuckleheads.print\_stats  
 break  
else  
 puts "Please enter a number or 'quit'"  
end  
end

1. You'll need to run the program from the command line, rather than from inside of your editor, so that you can enter the number of rounds. TextMate, for example, doesn't have a way to prompt for user input so you'll get an error if you try to run the program in TextMate.

Nicely done! That makes it a lot easier to play the game.

### 2. Read Players From a File

### Now suppose folks playing our game (we affectionately call them *users*) would like to define their own players. Unfortunately, the only way to do that currently is by writing Ruby code like this:

player4 = Player.new("Alvin", 100)  
player5 = Player.new("Simon", 60)  
player6 = Player.new("Theo", 125)  
game.add\_player(player4)  
game.add\_player(player5)  
game.add\_player(player6)

As a seasoned Ruby programmer, that code is now intuitively obvious to you. But our users haven't taken this fine course (yet!), so it's a tad intimidating for them to go hacking around in our Ruby code to change user names and such.

Instead, we'd like to offer users an easier way to make and play their *own* game. To do that, we'll ask them to list the player names and healths in a comma-separated text file called players.csv. Here's an example of what that file contains:

Alvin,100  
Simon,60  
Theo,125

1. Start by using your text editor to create a new file named players.csv in your studio\_game directory. In the file, list each player's name and health separated by a comma as shown above. We'll use this file as a test.

Alvin,100  
Simon,60  
Theo,125

1. Then define a load\_players method in the Game class that takes a filename as a parameter. Start off with an easy method implementation. For now, simply read each line of the given CSV file into an array and iterate through it to print out each line.

def load\_players(from\_file)  
 File.readlines(from\_file).each do |line|  
 puts line  
 end  
end

1. Next, call the load\_players method after you create a game in the studio\_game.rb file. Be sure to pass in the filename "players.csv".

knuckleheads = Game.new("Knuckleheads")  
knuckleheads.load\_players("players.csv")

1. Run the program and check the output. For example, when the load\_players method is called with the CSV file above, it should produce the following output:
2. Alvin,100  
   Simon,60  
   Theo,125
3. Now that you know the file is being correctly read, change the load\_players method to add each player to the game. Don't forget you need to *safely* convert the health value from a string to an integer.

def load\_players(from\_file)  
 File.readlines(from\_file).each do |line|  
 name, health = line.split(',')  
 player = Player.new(name, Integer(health))  
 add\_player(player)  
 end  
end

1. Now, back in the studio\_game.rb file, make sure you aren't explicitly adding any players with the add\_players method. Then run the game and all the new players in the CSV file should be playing!
2. Finally, it would be handy to run the game with a specific CSV file of players. For example, you might put all your favorite players in a file called "my\_favorite\_players.csv". You could then pass in your filename as a command-line argument when you run the game, like so:
3. ruby studio\_game.rb my\_favorite\_players.csv
4. Change the studio\_game.rb file so that it loads the players from the CSV file given as a command-line argument, if one exists. Otherwise, default to loading the players in the "players.csv" file.

knuckleheads.load\_players(ARGV.shift || "players.csv")

### 3. Save High Scores to a File

### Finally, to put a finishing touch on a game, we want to save off the high score list to a text file called high\_scores.txt. That way, a proud user can easily attach the file to an email and brag to all of his friends. Ruby makes this a trivial task, but there's an important lesson here: Always offer an convenient way for users to get *their* data out of *your* program.

### Here's an example file to give you something to aim for:

Knuckleheads High Scores:  
Larry............... 925  
Moe................. 590  
Curly............... 445

Notice that the players are sorted and formatted, just like they are when the high scores are printed to the console.

1. Define a save\_high\_scores method in the Game class that takes a filename as a parameter. If no parameter is given, set the default filename to "high\_scores.txt". Again, start off with an easy method implementation. For now, simply open the file for writing and print a header string such "Knuckleheads High Scores:" to the file.

def save\_high\_scores(to\_file="high\_scores.txt")  
 File.open(to\_file, "w") do |file|  
 file.puts "#{@title} High Scores:"  
 end  
end

1. Now call the save\_high\_scores method at the very end of the game (after the loop) in the studio\_game.rb file.

knuckleheads.save\_high\_scores

1. Run the program to check your work. When the game exits, open the high\_scores.txt file and make sure it has the header string, "Knuckleheads High Scores:".
2. Now that you know you're writing to the file, change the save\_high\_scores method to iterate through each player (in sorted order) and print their name and score in the format shown above. (Again, it's the same format as is being printed to the console, so reuse some of that code if you need to.)

def save\_high\_scores(to\_file="high\_scores.txt")  
 File.open(to\_file, "w") do |file|  
 file.puts "#{@title} High Scores:"  
 @players.sort.each do |player|  
 formatted\_name = player.name.ljust(20, '.')  
 file.puts "#{formatted\_name} #{player.score}"  
 end  
 end  
end

1. For the grand finale, run the game and check out that respectable high score file!

### 

### 4. Refactor

### Now that the code is working, it's time to step back and see if any refactoring is in order.

### Duplication

### The first thing to look for is unnecessary duplication. Do we have any? Well, depending on how you solved the exercises, it's quite possible. And to be honest we kinda set you up for it. Remember when we said that the formatting of the high score list in the file was the same as that printed to the console? That implied some duplication. And it would be really easy to fall into the trap of doing a copy/paste of the code that did that formatting.

Check out the formatting code in the print\_stats method:

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

And compare it to the formatting code in the save\_high\_scores method:

**def** **save\_high\_scores**(to\_file="high\_scores.txt")  
 File.open(to\_file, "w") **do** |file|  
 file.puts "#{@title} High Scores:"  
 @players.sort.each **do** |player|  
 formatted\_name = player.name.ljust(20, '.')  
 file.puts "#{formatted\_name} #{player.score}"  
 **end**  
 **end**  
**end**

Now, there is a slight difference: The version that writes to the console uses puts and the one that writes to the file uses file.puts. But both chunks of code have these duplicated statements:

formatted\_name = player.name.ljust(20, '.')  
"#{formatted\_name} #{player.score}"

Remember, unnecessary duplication is what makes programs hard to change. Suppose, for example, that we decide to change how high score entries are formatted. To do that successfully, we have to remember to change the code in two places: in the print\_stats method *and* in the save\_high\_scores method. Implementing the formatting code in a single point in the program would make it easier to change down the road.

Think about how you might clean up this duplication. Try it on your own first. Take your time; we're not going anywhere…

Gave it an earnest try? Great. Just for comparison, here's one approach using the *extract method* refactoring again:

1. In the Game class, define a high\_score\_entry method that takes a player object as the parameter.

def high\_score\_entry(player)  
end

1. Move the code that generates the formatted high-score string into the high\_score\_entry method. Have it return the formatted string.

def high\_score\_entry(player)  
 formatted\_name = player.name.ljust(20, '.')  
 "#{formatted\_name} #{player.score}"  
end

1. Then call the high\_score\_entry method from the print\_stats and save\_high\_scores methods wherever you need a string formatted with a player's name and high score.

def print\_stats  
 …  
puts "\n#{@title} High Scores:"   
@players.sort.each do |player|  
 puts high\_score\_entry(player)  
end  
end

def save\_high\_scores(to\_file="high\_scores.txt")  
 File.open(to\_file, "w") do |file|  
 file.puts "#{@title} High Scores:"  
 @players.sort.each do |player|  
 file.puts high\_score\_entry(player)  
 end  
 end  
end

Smaller Methods

The second refactoring opportunity to look for is methods (or classes) that do too much work. The way we ended up implementing the load\_players method serves as an example:

**def** **load\_players**(from\_file)  
 File.readlines(from\_file).each **do** |line|  
 name, health = line.split(',')  
 player = Player.new(name, Integer(health))  
 add\_player(player)  
 **end**  
**end**

Notice the different levels of abstraction in this method. It starts off with a single call that reads in all the lines from a file. That's a fairly high-level abstraction. The subsequent two lines quickly dive into the lower-level details of parsing each line to create a Player object. Then the method pops back up to a high-level method that adds the newly-created player.

In general, we strive to make everything inside of a method be at the same general level of abstraction. That makes the method easier to read and understand. It also means we end up with smaller methods that can be composed in various ways.

Think about how you might refactor the load\_players method. Specifically focus on abstracting the concept of turning a CSV string into a Player object. There's no rush; we'll wait right here…

Gave it your best shot? Bravo. Just for posterity, here's one approach:

1. In the Player class, define a from\_csv class-level method that takes a string formatted as CSV. Remember that class-level methods are defined on self, similar to how we defined module methods in a previous exercise. Move the code that creates a Player object from a CSV string into this method.

def self.from\_csv(string)  
 name, health = string.split(',')  
 Player.new(name, Integer(health))  
end

1. Before moving on, write a test for that in the player\_spec.rb file and make sure it passes. :-)

it "can be created from a CSV string" do  
 player = Player.from\_csv("larry,150")  
 player.name.should == "Larry"  
 player.health.should == 150  
end

1. Then call the from\_csv method from the load\_players method to create a Player object. Because you don't have a player object to call the method on, you need to call it on the Player class itself.

def load\_players(from\_file)  
 File.readlines(from\_file).each do |line|  
 add\_player(Player.from\_csv(line))  
end  
end

Bonus Round

CSV Library

Now that you know how to roll your own CSV solution, you'll appreciate that Ruby can do some of that heavy lifting for you.

In the Ruby Standard Library, you'll find a CSV class that makes working with comma-separated data files even easier. Check out the documentation and then change the load\_players method to apply what you found. Note that to use the CSV library you'll need to require it like so:

require 'csv'

def load\_players(from\_file)  
 CSV.foreach(from\_file) do |row|  
 player = Player.new(row[0], row[1].to\_i)  
 add\_player(player)  
 end  
end

Fundraising Program

There are a variety of inputs and outputs you could incorporate into your fundraising program. Here are a few possible objectives:

* Limiting your fundraising to a certain number of rounds seems restrictive. Allow users to input a certain number of rounds from the command line.
* Allow users to add their own projects from a file, such as a comma-separated text file like so:
* Project1,100,0  
  Project2,500,50  
  Project3,1000,100
* In order to quickly send an update to their friends and family, allow users to save off a list of projects that still need funding.

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")  
projects.load\_projects(ARGV.shift || "projects.csv")

loop do  
 puts "\nHow many rounds of funding? ('quit' to exit)"  
 answer = gets.chomp.downcase  
 case answer  
when /^\d+$/  
 projects.request\_funding(answer.to\_i)  
 when 'quit', 'exit'  
 projects.print\_results  
 break  
 else  
 puts "Please enter a number or 'quit'"  
 end  
end

projects.save\_under\_funded\_projects

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

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

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

def print\_results  
 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  
  
puts "\n#{under\_funded\_projects.size} projects still need your help:"  
sorted\_under\_funded\_projects.each do |project|  
 formatted\_name = project.name.ljust(20, '.')  
 puts "#{formatted\_name} $#{project.total\_funding\_outstanding} under"  
end

@projects.each do |project|  
puts "\n#{project.name}'s pledges:"  
project.each\_received\_pledge do |pledge|  
 puts "$#{pledge.amount} in #{pledge.name} pledges"  
end  
  
puts "$#{project.pledges} in total pledges"  
end  
end

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

def load\_projects(from\_file)  
 File.readlines(from\_file).each do |line|  
 name, target\_funding\_amount, funding = line.split(',')  
 project = Project.new(name, Integer(target\_funding\_amount), Integer(funding))  
 add\_project(project)  
 end  
end  
  
def save\_under\_funded\_projects(to\_file="needmoremoney.txt")  
 File.open(to\_file, "w") do |file|  
 file.puts "These projects still need your help:"  
 sorted\_under\_funded\_projects.each do |project|  
 formatted\_name = project.name.ljust(20, '.')  
 file.puts "#{formatted\_name} $#{project.total\_funding\_outstanding} under"  
 end  
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\_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

def each\_received\_pledge  
 @received\_pledge.each do |name, amount|  
 yield Pledge.new(name, amount)  
 end  
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("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

it "yields each received pledge and its total pledge amount" do  
 @project.received\_pledge(Pledge.new(:bronze, 50))  
 @project.received\_pledge(Pledge.new(:silver, 75))  
 @project.received\_pledge(Pledge.new(:silver, 75))  
 @project.received\_pledge(Pledge.new(:gold, 100))  
 @project.received\_pledge(Pledge.new(:gold, 100))  
 @project.received\_pledge(Pledge.new(:gold, 100))  
  
 yielded = []  
 @project.each\_received\_pledge do |pledge|  
 yielded << pledge  
end

yielded.should == [  
Pledge.new(:bronze, 50),  
Pledge.new(:silver, 150),  
Pledge.new(:gold, 300)  
]   
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

Wrap Up

Now you're getting the ins and outs of Ruby programming! In this exercise we focused mostly on file I/O. Ruby has built-in support for other types of I/O, as well, including talking to resources on the web and communicating with external processes. Combine that with a versatile String class and great regular expression support, and Ruby makes interfacing with the rest of the world fun and easy. And you now have a great start on all these fun possibilities, including how to:

* handle keyboard input
* read from files
* write to files
* keep your code well factored