Exception Handling

What if your program is trying to connect to a database or a website that happens to be offline? What will the user see if this happens? You want to try to avoid the user knowing that your code has blown up (is not working). What you want to do is detect that the code has blown up and contain it, so the user does not have a bad experience. Exception handling is when you detect the blow up and prevent it from blowing up for the user.

```
~/Documents/wdi3/ruby/lectures ✗ master ∴ touch 2013-02-01.rb

~/Documents/wdi3/ruby/lectures ✗ master ∴ subl 2013-02-01.rb
```

```ruby
require 'pry'

puts "the result of 3 divided by 0 is #{3/0}"
```

```
~/Documents/wdi3/ruby/lectures ✗ master ∴ ruby 2013-02-01.rb

2013-02-01.rb:3:in `/': divided by 0 (ZeroDivisionError)

   from 2013-02-01.rb:3:in `<main>'
```

The code blew up. We want to put any code that is volatile in begin...rescue...end syntax.

```ruby
begin
  puts "the result of 3 divided by 0 is #{3/0}"
rescue
  puts "wow, your code just exploded! "
end
```

This means try to do the first part and if it doesn't work, use the rescue code.

```
~/Documents/wdi3/ruby/lectures ✗ master ∴ ruby 2013-02-01.rb

wow, your code just exploded!
```

Now add some code to let the user input the number to divide by. If you put this within the begin...rescue syntax, you have the option to add retry if the code does blow up.

```ruby
begin
  print "what do you want to divide by? "
  number = gets.to_i
  puts "the result of 3 divided by 0 is #{3/number}"
rescue
  puts "wow, your code just exploded!"
  retry
end
```

```
~/Documents/wdi3/ruby/lectures ✗ master ∴ ruby 2013-02-01.rb

what do you want to divide by? 0

wow, your code just exploded!

what do you want to divide by? 3

the result of 3 divided by 0 is 1
```

The **raise** command causes a code explosion to happen. You will use this rarely, but it is used when you detect something that makes you want to stop running the program.

```ruby
puts "this code is highly explosive!"
raise 'boom!'
```

```
~/Documents/wdi3/ruby/lectures ✗ master ∴ ruby 2013-02-01.rb

this code is highly explosive!

2013-02-01.rb:13:in `<main>': boom! (RuntimeError)
```

The **ensure** keyword cause code to run regardless of whether or not something blows up. For example, if a file is opened, it needs to be closed. If the code blows up while the file is being opened, **ensure** will make sure that the file does get closed.

```ruby
begin
  print "what do you want to divide by?"
  number = gets.to_i
  puts "the result of 3 divided by 0 is #{3/number}"
rescue
  puts "wow, your code just exploded!"
  retry
ensure
  puts "i need to make sure this is run... "
end
```

File I/O (Input/Output)

Rather than creating a data.rb file, wouldn't it be cool if your program could write some data to a file, sort of like a database? File I/O lets you do this.

```ruby
f = File.new('database.txt', 'a+')
...
f.close
```

This opens a new file and saves it as database.txt. The **a+** means the file will be appended, rather than overwritten.

`f.close` closes the file.

When the data is input, we'll enter it as comma-separated. `f.puts` means that a string will be put into the file.

```
f = File.new('database.txt', 'a+')

print "write name to file (y/n) "
response = gets.chomp.downcase
while response == 'y'
  print "Enter name, age, gender: "
  f.puts(gets.chomp)

  print "Write name to file?"
  response = gets.chomp.downcase
end

f.close
```

```
~/Documents/wdi3/ruby/lectures ✗ master ∴ ruby 2013-02-01.rb

write info to file (y/n)y

enter name, age, gender: bob, 21, male

write name to file (y/n) y

enter name, age, gender: mary, 22, female

write name to file (y/n) y

enter name, age, gender: sally, 30, female

write name to file (y/n) n
```

Now if you look at the directory listing, you'll see the database.txt file.

```
~/Documents/wdi3/ruby/lectures ✗ master ∴ l

-rw-r--r--  1 user  wheel    49 Feb  1 11:43 database.txt
```

Open the file using `cat file_name` and check it out:

```
~/Documents/wdi3/ruby/lectures ✗ master ∴ cat database.txt

bob, 21, male

mary, 22, female

sally, 30, female
```

If you run the program again, it will add to the end of the file.

```
~/Documents/wdi3/ruby/lectures ✗ master ∴ ruby 2013-02-01.rb

write info to file (y/n) y

enter name, age, gender: bob, 33, male

write name to file (y/n) n

~/Documents/wdi3/ruby/lectures ✗ master ∴ cat database.txt

bob, 21, male

mary, 22, female

sally, 30, female

bob, 33, male
```

In pry, `ls -c` returns constants (anything in Ruby that begins with a capital letter; this includes classes). Items in red in this list are error messages.

Now let's create a Person class so we can make person objects out of the people we added to the file.

```ruby
class Person
  attr_accessor :name, :age, :gender
  def initialize(name, age, gender)
    @name = name
    @age = age
    @gender = gender
  end

  def to_s
    "#{@name} is a #{@age} year old #{@gender}"
  end
end
```

Now let's access the database.txt file. Use `r` to read the file. The block will initially just display each line of the file.

```ruby
f = File.new('database.txt', 'r')

f.each do |line|
  puts "the line is the #{line}"
end

f.close
```

```
~/Documents/wdi3/ruby/lectures ✗ master ∴ ruby 2013-02-01.rb
```

```
write info to file (y/n): n

the line is the bob, 21, male

the line is the mary, 22, female

the line is the sally, 30, female

the line is the bob, 33, male
```

The output is a string with commas. We want to split this into an array. Use `.split(', ')` to get the parameters out. Now our Person.new can use the index positions of this array to set data. Add a people array before the block, then add each new person into it.

```ruby
people = []

f.each do |line|
  person_array = line.chomp.split(', ')
  person = Person.new(person_array[0], person_array[1], person_array[2])
  people << person
end
```

Now after the file closes, let's print out the people in the people array by looping over it. This will call the `to_s` method in the Person class.

```ruby
people.each do |person|
  puts "the person is: #{person}"
end
```

```
~/Documents/wdi3/ruby/lectures ✗ master ∴ ruby 2013-02-01.rb

write info to file (y/n): n

the person is: bob is a 21 year old male

the person is: mary is a 22 year old female

the person is: sally is a 30 year old female

the person is: bob is a 33 year old male
```

Test-Driven Development (TDD) 1:45

In test-driven development, also called red/green development, you write a test, called test code, before writing any code. The test will initially fail, and drives how you will write your actual code, called your implementation, until the test passes. This is an iterative process. Each iteration could be a story, like the stories in Pivotal Tracker, or the iteration could be a few stories grouped together.

The tests are available throughout your development process, so changes later in development break something, you'll easily be able to identify the problem.

Watch the Code School Testing with RSpec video for more on testing.

In pair programming, there is one computer/screen with two mice and two keyboard. In TDD, one person writes the test code; the other writes the implementation code. In an interview, you might be given some test code and be asked to write the implementation code.

We're going to build a bank account application using TDD. We'll create a bank, add accounts, then deposit and withdraw money.

Create a bank_account directory. Create a spec directory in it. Then install rspec.

```
~/Documents/wdi3/ruby/labs ✗ master ∴ mkdir bank_account

~/Documents/wdi3/ruby/labs/bank_account ✗ master ∴ mkdir spec

~/Documents/wdi3/ruby/labs/bank_account ✗ master ∴ gem install rspec
```

Go into the spec directory and create a blank file called spec_helper.rb.

```
~/Documents/wdi3/ruby/labs/bank_account/spec ✗ master ∴ touch spec_helper.rb
```

You have to create the spec directory and spec_helper.rb file for this to work. Open the file and cut and paste in the contents from https://gist.github.com/4689587

```ruby
RSpec.configure do |config|
  config.color_enabled = true
  config.tty = true
  config.formatter = :documentation
end
```

Close the file, then open the bank_account directory in Sublime.

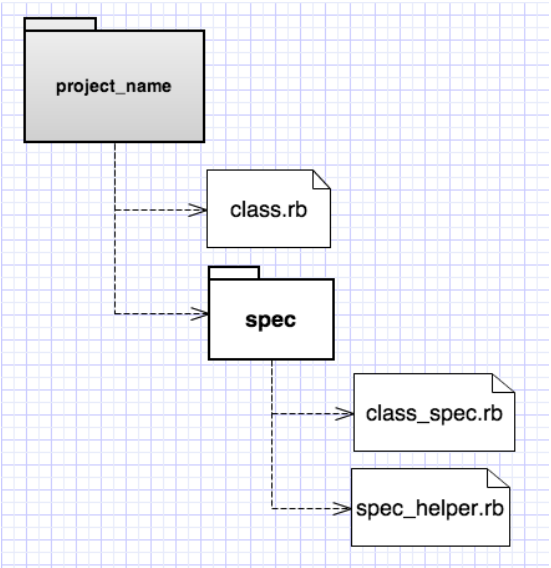
What kind of models (classes) do you need for this app? Maybe we should start with a Bank. Create a bank.rb file.

```
~/Documents/wdi3/ruby/labs/bank_account ✗ master ∴ touch bank.rb
```

There's nothing in this file at the moment. Let's write a test for our bank. To test a file, create a test within /spec called bank_spec.rb.

```
~/Documents/wdi3/ruby/labs/bank_account ✗ master ∴ touch spec/bank_spec.rb
```

General structure of a project with rspec testing



Any spec file needs to have access to your spec_helper file, so you need to require it. Because it's a file in the same directory, use **`require_relative`**. You also need to include the file that is being tested. Notice that the file name is preceded by **`../`**. This means that it is accessing the directory above the one you are currently in.

```
require_relative 'spec_helper'
require_relative '../bank'
```

Now try to run the test. Use **`rspec spec`**.

```
~/Documents/wdi3/ruby/labs/bank_account ✗ master ∴ rspec spec

No examples found.


Finished in 0.00005 seconds

0 examples, 0 failures
```

RSpec is a DSL (domain-specific language). This means that it is written specifically for use with Ruby, but is not Ruby.

What are we testing? We want to create a bank. What method would this use? New.

A describe at the top of the test is for the class, then another describe is for the method being testing, then the it line is the test.

```
describe Bank do
  describe "#new" do
    it "creates a Bank object" do
    end
  end
```

```
end
```

Run the test. We get an error. This is good.

```
~/Documents/wdi3/ruby/labs/bank_account ✗ master ∴ rspec spec

/Users/user/Documents/wdi3/ruby/labs/bank_account/spec/bank_spec.rb:4:in
`<top (required)>': uninitialized constant Bank (NameError)...
```

Go into the bank.rb file and write some implementation code until the error goes away. The solution is to create a class called Bank.

```
class Bank
end
```

```
~/Documents/wdi3/ruby/labs/bank_account ✗ master ∴ rspec spec


Bank

  #new

    creates a Bank object


Finished in 0.00036 seconds

1 example, 0 failures
```

What are parameters we will want to include when creating a bank object? A name. Now we can write some more test code within the existing block

```
...
    it "creates a Bank object" do
      bank = Bank.new('TD Bank')
      expect(bank).to_not eq nil
...
```

Run the test again.

```
~/Documents/wdi3/ruby/labs/bank_account ✗ master ∴ rspec spec


Bank

  #new
```

```
    creates a Bank object


Finished in 0.00036 seconds
1 example, 0 failures
~/Documents/wdi3/ruby/labs/bank_account ✗ master ∴ rspec spec


Bank
  #new
    creates a Bank object (FAILED - 1)


Failures:


  1) Bank#new creates a Bank object
     Failure/Error: bank = Bank.new('TD Bank')
     ArgumentError:
       wrong number of arguments(1 for 0)
     # ./spec/bank_spec.rb:7:in `initialize'
     # ./spec/bank_spec.rb:7:in `new'
     # ./spec/bank_spec.rb:7:in `block (3 levels) in <top (required)>'


Finished in 0.00043 seconds
1 example, 1 failure


Failed examples:


rspec ./spec/bank_spec.rb:6 # Bank#new creates a Bank object
```

What does the “wrong number of arguments” mean? We haven’t created an initialize method yet, but the program is finding one in the ancestors that doesn’t ask for parameters like our test does. Fix the problem.

```ruby
def initialize(name)
end
```

If you included attr_accessor, your test will not pass. Only write the minimum amount of code needed to meet the requirements.

Let's create another test. This tests the name method and expects it to return the name 'TD Bank'.

```ruby
describe ".new" do
  it "has a name" do
    bank = Bank.new('TD Bank')
    expect(bank.name).to eq 'TD Bank'
  end
end
```

```
~/Documents/wdi3/ruby/labs/bank_account ✗ master ∴ rspec spec


Bank
  #new
    creates a Bank object
  #new
    has a name (FAILED - 1)


Failures:


  1) Bank#new has a name
     Failure/Error: expect(bank.name).to eq 'TD Bank'
     NoMethodError:
       undefined method `name' for #<Bank:0x007f7f8bbeeeb8 @name="TD Bank">
     # ./spec/bank_spec.rb:15:in `block (3 levels) in <top (required)>'


Finished in 0.0011 seconds
2 examples, 1 failure
```

```
Failed examples:


rspec ./spec/bank_spec.rb:13 # Bank#new has a name
```

Now fix the “undefined method ‘name’” error so the bank name test passes.

```ruby
class Bank
  attr_accessor :name

  def initialize(name)
    @name = name
  end
end
```

Remember, there are two types of method—class methods, like new (Person.new), and instance methods, like name, age, gender. When you call these, use .class_method and #instance_method. The test is looking for a method called name. You could define a name method, but you can also use attr_accessor :name, which creates the name and name= methods.

This test passed. Now let’s add a test for creating a bank account with a name and an initial deposit.

```ruby
describe "#create_account" do
  it "create an account" do
    bank = Bank.new('TD Bank')
    bank.create_accounts('Bob', 200)
    expect(bank.accounts['Bob']).to eq 200
  end
end
```

Now write the code to make this test pass. This one was more difficult for the class. We need to create a create_account function that takes a name and an initial deposit. Look at the error message to see what needs to be done.

```
     1) Bank#create_account create an account

        Failure/Error: bank.create_accounts('Bob', 200.00)

        NoMethodError:

          undefined method `create_accounts' for #<Bank:0x007fb6523ea778
   @name="TD Bank">
```

```ruby
def create_account(account, init_deposit)
end
```

Now run the test again.

```
  1) Bank#create_account create an account

     Failure/Error: expect(bank.accounts['Bob']).to eq 200

     NoMethodError:

       undefined method `accounts' for #<Bank:0x007fce65be7f18 @name="TD
Bank">

     # ./spec/bank_spec.rb:23:in `block (3 levels) in <top (required)>'
```

Now the error is on line 23 and the 'accounts' method is not recognized. What is accounts? We know it is a hash because the the brackets in the test. We can fix this, but we need to add an accounts hash, and this should be done earlier in the process.

Let's add something to the new class method so that a new bank has no active accounts. While doing this, notice that we have repeated the creating of a bank object in all of our tests. Let's consolidate this code.

```
let(:bank) {Bank.new('TD Bank')}
```

Now run the test to see if this refactoring of the test works. OK, now complete the new test. Accounts should be a hash that is initialized to zero.

```
describe ".new" do
  it "creates a Bank object" do
    expect(bank).to_not eq nil
  end
  it "has no accounts" do
    expect(bank.accounts.count).to eq 0
  end
end
```

This will pass the test.

```
attr_accessor :name, :accounts

def initialize(name)
  @name = name
  @accounts = {}
end
```

Now we can go back to the create_accounts test we started earlier:

```
describe "#create_account" do
  it "create an account" do
    bank.create_accounts('Bob', 200)
    expect(bank.accounts['Bob']).to eq 200
  end
end
```

and write some code to pass the test.

```
def create_account(name, deposit)
  @accounts[name] = deposit
end
```

Now we'll write a test to handle a deposit into an account.

```
describe "#deposit" do
  it "deposits money from client into account" do
    bank.create_account('Bob', 200)
    bank.deposit('Bob', 300)
    expect(bank.accounts['Bob']).to eq 500
  end
end
```

Now write some code to define the deposit method.

Using the graph, how can you tell what the balance of Bob's account is?

```
bank → Bank

       accts

         →

{'Bob', 'Sue'}

  →      →

 200     100
```

It is bank.accounts['Bob']. Now add the $300 deposit to it.

```
def deposit(name, amount)
  @accounts[name] = @accounts[name] + amount
end
```

This works. Tighter syntax:

```
def deposit(name, amount)
  @accounts[name] += amount
end
```

Let's add a feature to check the balance on an account. Start with the test.

```
describe "#balance" do
```

```
  it "returns the balance for the client" do
    bank.create_account('Bob', 200)
    expect(bank.balance('Bob')).to eq 200
  end
end
```

And write the code. The balance is simply @accounts[name].

```
def balance(name)
  @accounts[name]
end
```

Now add a test for a withdrawal.

```
describe "#withdraw" do
  it "subtracts money from the account" do
    bank.create_account('Bob', 200)
    bank.withdraw('Bob', 50)
    expect(bank.balance('Bob')).to eq 150
  end
end
```

And write the code to pass the test.

```
def withdraw(name, amount)
  @accounts[name] -= amount
end
```

Now add one more test to ignore withdrawals that are greater than the account balance.

```
describe "#withdraw" do
...
  it "ignores requests for withdrawals greater than account balance" do
    bank.create_account('Bob', 200)
    bank.withdraw('Bob', 5000)
    expect(bank.balance('Bob')).to eq 200
  end
end
```

And write the code to pass the test.

```
def withdraw(name, amount)
  @accounts[name] -= amount if amount <= @accounts[name]
end
```

Congratulations! You just created an app with lots of features using TDD!

Is the test always right? Humans write the tests too, so this is not infallible.