**Exceptions in Ruby**

**Exceptions** are what Ruby uses to deal with errors or other unexpected events. For now it's okay to use *exception* and *error* synonymously. If you have made it this far in the course, chances are you have run into many exceptions. Here we are referring to exceptions in the code that will literally crash your program and stop execution immediately.

Here are two examples of classic exceptions or errors that will halt our execution:

def my\_method(arg\_1, arg\_2)

puts arg\_1

puts arg\_2

end

my\_method("a") # ArgumentError: wrong number of arguments (given 1, expected 2)

5 + nil # TypeError: nil can't be coerced into Integer

Upon reaching an exception, the default behavior in ruby is to terminate the program. However, we can also define our own behavior to handle exceptions.

**Handling Exceptions**

Let's explore how we can implement our own custom exception handling in Ruby. Exception handling can be very involved, but let's just focus on the fundamentals. Let's say we had some isolated code that is vulnerable to an exception:

10 / num

Do you see a potential problem in the code above? That code seems innocuous, right? ...Wrong! What if num is 0:

num = 0

10 / num # ZeroDivisionError: divided by 0

puts "finished"

Reaching the ZeroDivisionError exception would halt execution of the program and we would never get to reach the final puts "finished". This could be deadly if there is some critical code after the exception that we *still* want to execute, regardless of the fact we got an exception.

With that in mind, let's redesign this code to handle division by zero more gracefully. We'll need to use a new structure that is specific to handling exceptions, begin...rescue.

num = 0

begin

puts "dividing 10 by #{num}"

ans = 10 / num

puts "the answer is #{ans}"

rescue

puts "There was an error with that division."

end

puts "--------"

puts "finished"

The output of the above code is:

dividing 10 by 0

There was an error with that division.

--------

finished

The behavior of begin...rescue is this: The code in the begin block will execute until an exception is reached. Once an exception is reached, the execution will immediately jump to rescue. This behavior is evident by the fact that the code above doesn't print "the answer is ", because the exception is reached on the line before.

In the event that an exception is never hit in the begin block, then execution will never go to rescue. This is the case in the following code:

num = 5

begin

puts "dividing 10 by #{num}"

ans = 10 / num

puts "the answer is #{ans}"

rescue

puts "There was an error with that division."

end

puts "--------"

puts "finished"

The output of the above code is:

dividing 10 by 5

the answer is 2

--------

finished

Cool! Now we know the basics of exception handling in ruby. begin running some code, and if something goes horribly wrong, we jump to rescue it. begin...rescue has a somewhat similar control flow to an if...else in that both structures have branching logic. We explored how to handle ZeroDivisionError, but other errors can be rescued too. Here are a few more common error types that are native to ruby:

* ArgumentError
* NameError
* NoMethodError
* IndexError
* TypeError

This is by no means an exhaustive list, but these are the common ones.

**Raising Exceptions**

Previously we explored how to handle native exceptions like ZeroDivisionError, but what if we wanted to implement our own exceptions? The point of an *exception* is to flag an *exceptional* scenario in the code that should be handled in a specific way. As a programmer, you can decide what those exceptional scenarios are!

Say we wrote this method:

def format\_name(first, last)

first.capitalize + " " + last.capitalize

end

format\_name("grace", "HOPPER") # => "Grace Hopper"

It's obvious how this method *should* be used. That is, we ought to pass in two strings when calling format\_name. But this is still prone to misuse:

format\_name(42, true) # => NoMethodError: undefined method `capitalize' for 42:Integer

This code will reach a native exception, but it could be better. We want the exception message to really describe how our format\_name method was misused. Right now the error message really pertains to misuse of capitalize. The current exception message is also uncomfortable because it exposes the inner implementation details of our format\_name method. For someone that is calling our method, maybe we don't want them to know what operations we do inside. If they have intelligence of the inner workings of our method, they could use that knowledge to deduce how to break our code. We know, we know, format\_name is a harmless method, but if it was an absolutely critical method, we would want to protect it.

We'll let you be the judge, which exception do you think is more appropriate:

format\_name(42, true) # => NoMethodError: undefined method `capitalize' for 42:Integer

# OR

format\_name(42, true) # => RuntimeError: arguments must be strings

Let's implement the second exception! Since we want to **raise** an exception when the arguments are not strings, we'll need a quick aside on how to check data type:

"hello".instance\_of?(String) # => true

42.instance\_of?(String) # => false

Simple enough! Let's use this to rewrite format\_name:

def format\_name(first, last)

if !(first.instance\_of?(String) && last.instance\_of?(String))

raise "arguments must be strings"

end

first.capitalize + " " + last.capitalize

end

format\_name("grace", "hopper") # => "Grace Hopper"

format\_name(42, true) # => RuntimeError: arguments must be strings

In the code above we see raise, this is how we can make exceptions manually. You'll commonly hear the phrase "raise an error" or "raise an exception" to refer to this.

**Bring it all together**

Since our format\_name method can raise an exception, we can also handle it with begin...rescue.

def format\_name(first, last)

if !(first.instance\_of?(String) && last.instance\_of?(String))

raise "arguments must be strings"

end

first.capitalize + " " + last.capitalize

end

first\_name = 42

last\_name = true

begin

puts format\_name(first\_name, last\_name)

rescue

# do stuff to rescue the "arguments must be strings" exception...

puts "there was an exception :("

end

An important distinction to note is that raise is how we bring up an exception, whereas begin...rescue is how we react to that exception.

There is much more to exceptions in ruby, but these are the fundamentals that you'll need throughout the course. In the course, you'll run into scenarios where a spec may say "should raise an error when...". Solving these specs can be as simple as using raise with an if statement, just like we explored in our format\_name method!