**Exceptions**

**Goal**

* Know when to use exceptions.
* Know how to raise an exception. Know how to handle one.
* Know how to run some code regardless of an exception being thrown.

**Basics**

Things don't always work out the way you plan. Sometimes your code will experience an error. Exceptions are the means of telling the caller that your code can't do what was asked.

def sqrt(num)

unless num >= 0

raise ArgumentError.new "Cannot take sqrt of negative number"

end

# code to calculate square root...

end

Since we can't take the square root of a negative number, we **raise** an **exception** instead of returning an answer. When an exception is raised, the method stops executing. Instead of returning, an exception is thrown. The method's caller then gets a chance to handle the exception:

def main

# run the main program in a loop

while true

# get an integer from the user

puts "Please input a number"

num = gets.to\_i

begin

sqrt(num)

rescue ArgumentError => e

puts "Couldn't take the square root of #{num}"

puts "Error was: #{e.message}"

end

end

end

If the user feeds in a negative number, sqrt will raise an exception. Because main has wrapped this code in a begin/rescue/end block, the exception will be **caught**. The code will jump to the rescue block that anticipates an ArgumentError. It will save the exception in the variable e, then run the error handling code.

If the calling method doesn't rescue (we could also say **catch** or **handle**) an exception, it continues to **bubble up** the **call stack**. So the caller's caller gets a chance, then their caller, then...

If no method throughout the entire call stack catches the exception, the exception is printed to the user and the program exits.

**Ensure**

Sometimes there is important code that must be executed whether an exception is raised or otherwise. In this case, we can use ensure.

begin

a\_dangerous\_operation

rescue StandardError => e

puts "Something went wrong: #{e.message}"

ensure

puts "No matter what, make sure to execute this!"

end

A common example is closing files:

f = File.open

begin

f << a\_dangerous\_operation

ensure

# must. close. file.

f.close

end

**Retry**

A common response to an error is to try, try again.

def prompt\_name

puts "Please input a name:"

# split name on spaces

name\_parts = gets.chomp.split

if name\_parts.count != 2

raise "Uh-oh, finnicky parsing!"

end

name\_parts

end

def echo\_name

begin

fname, lname = prompt\_name

puts "Hello #{fname} of #{lname}"

rescue

puts "Please only use two names."

retry

end

end

The retry keyword will cause Ruby to repeat the begin block from the beginning. It is useful for "looping" until an operation completes successfuly.

**Implicit Begin Blocks**

Method and class definitions are implicitly wrapped in a begin/end block, so if your error handling applies to the whole method, all you have to add is rescue.

def slope(pos1, pos2)

(pos2.y - pos1.y) / (pos2.x - pos1.x)

rescue ZeroDivisionError

nil

end

The method from the retry example could also be written this way.

def echo\_name

fname, lname = prompt\_name

puts "Hello #{fname} of #{lname}"

rescue

puts "Please only use two names."

retry

end

**Exception Hierarchy**

There are a number of predefined exception classes in Ruby. You can find them [here](http://blog.nicksieger.com/articles/2006/09/06/rubys-exception-hierarchy). You should try to choose an appropriate class. One of the more common exceptions to use is ArgumentError, which signals that an argument passed to a method is invalid. RuntimeError is used for generic errors; this is probably your other goto.

When creating an exception, you can add an error message so the user knows what went wrong:

raise RuntimeError.new("Didn't try hard enough")

If you want your user to be able to distinguish different failure types (perhaps to handle them differently), you can extend StandardError and write your own:

class EngineStalledError < StandardError

end

class CollisionOccurredError < StandardError

end

def drive\_car

# engine may stall, collision may occur

end

begin

drive\_car

rescue EngineStalledError => e

puts "Rescued from engine stalled!"

rescue CollisionOccurredError => e

puts "Rescued from collision!"

ensure

puts "Car stopped."

end

**Don't go crazy**

Exceptions are a great tool for handling unexpected errors. But once you have a hammer, you may find yourself starting to look for nails.

Writing durable, "hardened" code means thinking of everything that could go wrong, watching out for those issues, and throwing an exception in that case. For instance, when writing sqrt, we can think ahead and recognize that negative numbers are a problem. So we add code to check for this and throw an exception.

Likewise, durable code anticipates exceptions being thrown. It makes sure that exceptions are handled properly. It avoids the program crashing; it does everything possible so that the program may carry on in spite of the exception.

However, writing hardened code like this is tedious and slow. There are always many, many things that could go wrong, and you could spend a ton of time writing exception classes, raising errors, making sure to catch them, etc.

For this reason, raise exceptions sparingly until you are hardening a project. Focus on driving out the functionality first. And don't waste your time imagining perverse scenarios; assume for the moment that the universe doesn't hate you. Just consider the things that could reasonably go wrong. You can always add more exception-handling code later.

Remember the maxim: *you ain't gonna need it*. Wait to implement functionality until you need it, not when you anticipate it. Features that aren't required take time away from more important requirements. More importantly, they are often poorly conceived, because until you have a practical need for a feature, you're just trying to imagine how that feature should work.