**Dynamic rescue clauses**

rescue clauses are similar in appearance and operation to another Ruby construct: the case statement. Just as with a case clause, you can supply a class or list of classes to be matched, followed by the code to be executed in case of a match.

```ruby
# case
case obj
  when Numeric, String, NilClass, FalseClass, TrueClass
    puts "scalar"
  # ...
end

# rescue
rescue SystemCallError, IOError, SignalException
  # handle exception...
end
```

*Listing 10: Comparing case and rescue.*

rescue clauses share something else in common with case statements: the list of classes or modules to be matched doesn’t have to be fixed. Here’s an example of a method that suppresses all exceptions that match a given list of types. The list is “splatted” with the * operator before being passed to rescue:
def ignore_exceptions(*exceptions)
    yield
    rescue *exceptions => e
        puts "IGNORED: '#{e}'"
    end

    puts "Doing risky operation"
    ignore_exceptions(IOError, SystemCallError) do
        open("NONEXISTENT_FILE")
    end
    puts "Carrying on..."
end

puts "Doing risky operation"
ignore_exceptions(IOError, SystemCallError) do
    open("NONEXISTENT_FILE")
end
puts "Carrying on..."

Listing 11: Dynamic exception lists for rescue.

But that’s not the end of the resemblance. As you may know, case works by calling the “threequals” (===) operator on each potential match. rescue works exactly the same way, but with an extra, somewhat arbitrary limitation:

```ruby
# define a custom matcher that matches classes starting with "A"
def starts_with_a = Object.new
    starts_with_a.===(e) = /^A/ =~ e.name
end

begin
    raise ArgumentError, "Bad argument"
    rescue starts_with_a => e
        puts "#{e} starts with a; ignored"
    end
end
```

Listing 12: Limitations on exception matchers.

The sole difference between case matching semantics and rescue matching semantics is that the arguments to rescue must all be classes or modules.
But so long as we satisfy that requirement, we can define the match conditions to be anything we want. Here’s an example that matches on the exception message using a regular expression:

```
def errors_with_message(pattern)
    # Generate an anonymous "matcher module" with a custom threeequals
    m = Module.new
    (class << m; self; end).instance_eval do
        define_method(:===) do |e|
            pattern === e.message
        end
    end
    m
end
```

```
puts "About to raise"
begin
    raise "Timeout while reading from socket"
rescue errors_with_message(/socket/)  
    puts "Ignoring socket error"
end
puts "Continuing...
```

Output

```
About to raise
Ignoring socket error
Continuing...
```

*Listing 13:* A custom exception matcher.

We can generalize that to create exception matchers based on an arbitrary block predicate:
def errors_matching(&block)
  m = Module.new
  (class << m; self; end).instance_eval do
    define_method(:===, &block)
  end
  m
end

class RetryableError < StandardError
  attr_reader :num_tries
  def initialize(message, num_tries)
    @num_tries = num_tries
    super("#{message} (#{num_tries})")
  end
end

puts "About to raise"
begin
  raise RetryableError.new("Connection timeout", 2)
  rescue errors_matching{|e| e.num_tries < 3} => e
    puts "Ignoring #{e.message}"
  end
  puts "Continuing..."
end

Output
About to raise
Ignoring Connection timeout (#2)
Continuing...

Listing 14: An exception matcher generator.

The rescue clause is powerful, and surprisingly dynamic. With dynamic type lists and custom matchers, you can be as specific as you need to be in matching exceptions to handlers.

rescue as a statement modifier

There is one final way to use rescue. In the same way that you can append an if or unless modifier to a Ruby statement, you can also append a rescue. A somewhat infamous example is the rescue nil modifier, which is sometimes used to ignore failures:

```ruby
f = open("nonesuch.txt") rescue nil
```