Error handling

Outline

1. Introduction to Error handling
2. Demonstration
3. “begin … rescue” block
4. Flow of handling

1. Introduction to Error handling

- No matter how carefully you code your script, your program is prone to failure for reasons beyond your control. A website that your script scrapes may suddenly be down. Or someone sharing the same hard drive may delete a file your program is supposed to read from.
- Circumstances such as these will crash your program. For any kind of long continuous task that you don't want to baby-sit and manually restart, you will need to write some exception-handling code to tell the program how to carry on when things go wrong.

2. Demonstration

- The Exception class handles nearly every kind of hiccup that might occur during runtime, including syntax screwups and incorrect type handling.
- We learned early on that adding numbers and strings with no type conversion would crash a program:

```
a = 10
b = "42"
a + b
```

➔ The attempted arithmetic results in this error

2. Demonstration

The **begin/rescue** block is typically used on code in which you anticipate errors. There's only one line here for us to worry about:
=> Executing the revised code gets us error

```
a = 10
b = "42"

begin
  a + b
rescue
  puts "Could not add variables a (#{a.class}) and b (#{b.class})"
else
  puts "a + b is #{a + b}"
end

# Result
Could not add variables a (Integer) and b (String)
```

2. Demonstration

Let's feed this simple operation with an array of values of different types to see how the else clause comes into play:

```
values = [42, 'a', 'r', 9, 5, 10022, 8.7, "sharon", "Libya", "Mars", "12", 98, rand + rand,
  {:dog=>'cat'}, 100, nil,     200.0000, Object, 680, 3.14, "Steve", 78, "Argo"].shuffle

while values.length > 0
  a = values.pop
  b = values.pop
  begin
     a + b
  rescue
     puts "Could not add variables a (#{a.class}) and b (#{b.class})"
  else
     puts "a + b is #{a + b}"
  end
end
```

2. Demonstration

```
#Result
Could not add variables a (String) and b (Float)
Could not add variables a (Integer) and b (String)
a + b is 108.7
a + b is 689
a + b is 47
Could not add variables a (NilClass) and b (String)
Could not add variables a (Hash) and b (Class)
Could not add variables a (Integer) and b (String)
a + b is 12Libya
a + b is 4.231714118025822
Could not add variables a (Integer) and b (String)
Could not add variables a (String) and b (NilClass)
```

3. “begin...rescue” block

- This is the most basic error handling technique. It starts off with the keyword begin and acts in similar fashion to an if statement in that it your program flows to an alternate branch if an error is encountered.
- The main idea is to wrap any part of the program that could fail in this block. Commands that work with outside input, such as downloading a webpage or making calculation something based from user input, are points of failure. Something like puts "hello world" or 1 + 1 is not.

3. “begin...rescue” block

```ruby
#Handle error
a = 10
b = "34"
begin
  a + b
rescue
  puts " Could not add variables a (#{a.class})
and b (#{b.class})"
else
  puts " a + b is #{a + b}"
ensure
  puts " I'm ensure !!!"
end
```

```
# Result
Could not add variables a (Integer) and b
(String)
 I'm ensure !!!
```

```ruby
#Handle error with specify rescue
begin
  get_name
rescue NameError => e
  puts e
else
  puts " Execute get_name method success"
ensure
  puts " I'm ensure !!!"
end
```

```
# Result
undefined local variable or method `name' for
main:Object
 I'm ensure !!!
```

3. “begin...rescue” block

- begin: This starts off the exception-handling block. Put in the operation(s) that is at risk of failing in this clause.
- rescue: This is the branch that executes if an exception or error is raised. Possible exceptions include: the website is down, or that it times out during a request.
- else: If all goes well, this is where the program branches to.
- ensure: This branch will execute whether an error/exception was rescued or not.

4. Flow of handling

The retry statement redirects the program back to the begin statement. This is helpful if your begin/rescue block is inside a loop and you want to retry the same command and parameters that previously resulted in failure.

4. Flow of handling

```ruby
#Using retry
for i in "A".."C"
  retries = 2
  begin
    puts "Executing command #{i}"
    raise "Exception: #{i}"
  rescue Exception=>e
    puts "\tCaught: #{e}"
    if retries > 0
      puts "\tTrying #{retries} more times\n"
      retries -= 1
      sleep 2
      retry
    end
  end
end
```

4. Flow of handling

```
#Output:
Executing command A
   Caught: Exception: A
   Trying 2 more times
Executing command A
   Caught: Exception: A
   Trying 1 more times
Executing command A
   Caught: Exception: A
Executing command B
   Caught: Exception: B
   Trying 2 more times
Executing command B
   Caught: Exception: B
   Trying 1 more times
Executing command B
   Caught: Exception: B
```

```
Executing command C
   Caught: Exception: C
   Trying 2 more times
Executing command C
   Caught: Exception: C
   Trying 1 more times
Executing command C
   Caught: Exception: C
```

References

- http://ruby-doc.org/
- http://ruby.bastardsbook.com/chapters/exception-handling/
- http://culttt.com/2015/07/22/using-ruby-exceptions/
- http://phocode.com/ruby/ruby-module-va-exception/
- https://github.com/awesome-academy/RubyExample_TFW

Question & Answer?