第七章　例外處理

OOP with Ruby

本章內容

- 何謂例外
- 例外類別
- 例外處理
- 如何觸發例外
- 如何捕捉例外

OOP with Ruby

何謂例外

- Sometimes the best outcome can be when nothing unusual happens
  - However, the case where exceptional things happen must also be prepared for
- Ruby exception handling facilities are used when the invocation of a method may cause something exceptional to occur

Ruby 的例外機制

- Ruby defines a mechanism that signals when something unusual happens
  - This is called *raising an exception*
- In another place in the program, the programmer must provide code that deals with the exceptional case
  - This is called *handling the exception*
- In Ruby, exceptions let you package information about an error into an object
  - That exception object is then propagated back up the calling stack automatically until the runtime system finds code that explicitly declares that it knows how to handle that type of exception

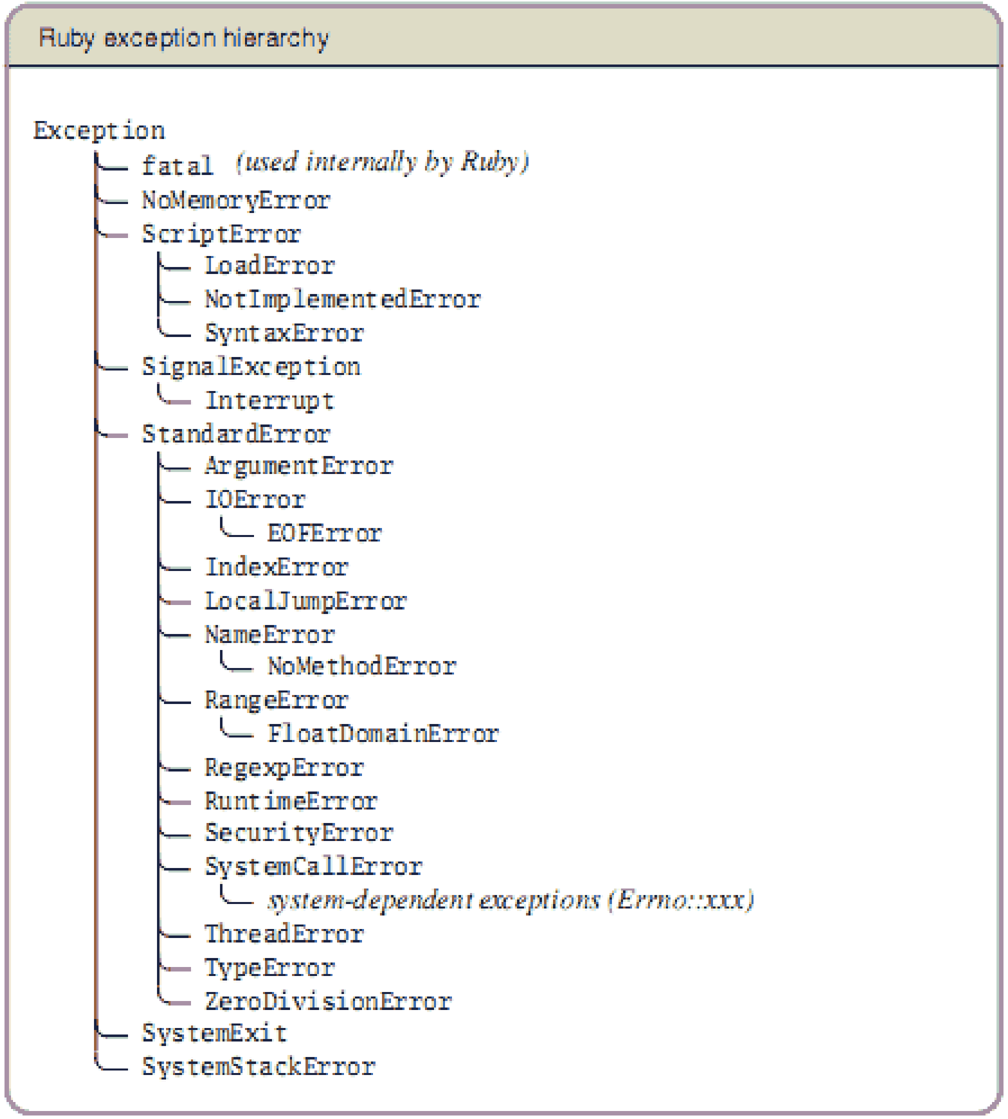
OOP with Ruby

例外類別 (Exception Class)

- The package that contains the information about an exception is an object of class *Exception* or one of class *Exception’s* children
- Ruby predefines a tidy hierarchy of exceptions
  - When you need to raise an exception, you can use one of the built-in Exception classes, or you can create one of your own
  - If you create your own, you may want to make it a subclass of *StandardError* or one of its children
- Every exception has associated with it a message string and a stack backtrace
  - If you define your own exceptions, you can add additional information

OOP with Ruby

例外類別 (2)



Ruby exception hierarchy

```
Exception
    ├── fatal   (used internally by Ruby)
    ├── NoMemoryError
    ├── ScriptError
    │     ├── LoadError
    │     ├── NotImplementedError
    │     └── SyntaxError
    ├── SignalException
    │     └── Interrupt
    ├── StandardError
    │     ├── ArgumentError
    │     ├── IOError
    │     │     └── EOFError
    │     ├── IndexError
    │     ├── LocalJumpError
    │     ├── NameError
    │     │     └── NoMethodError
    │     ├── RangeError
    │     │     └── FloatDomainError
    │     ├── RegexpError
    │     ├── RuntimeError
    │     ├── SecurityError
    │     ├── SystemCallError
    │     │     └── system-dependent exceptions (Errno::xxx)
    │     ├── ThreadError
    │     ├── TypeError
    │     └── ZeroDivisionError
    ├── SystemExit
    └── SystemStackError
```

例外處理

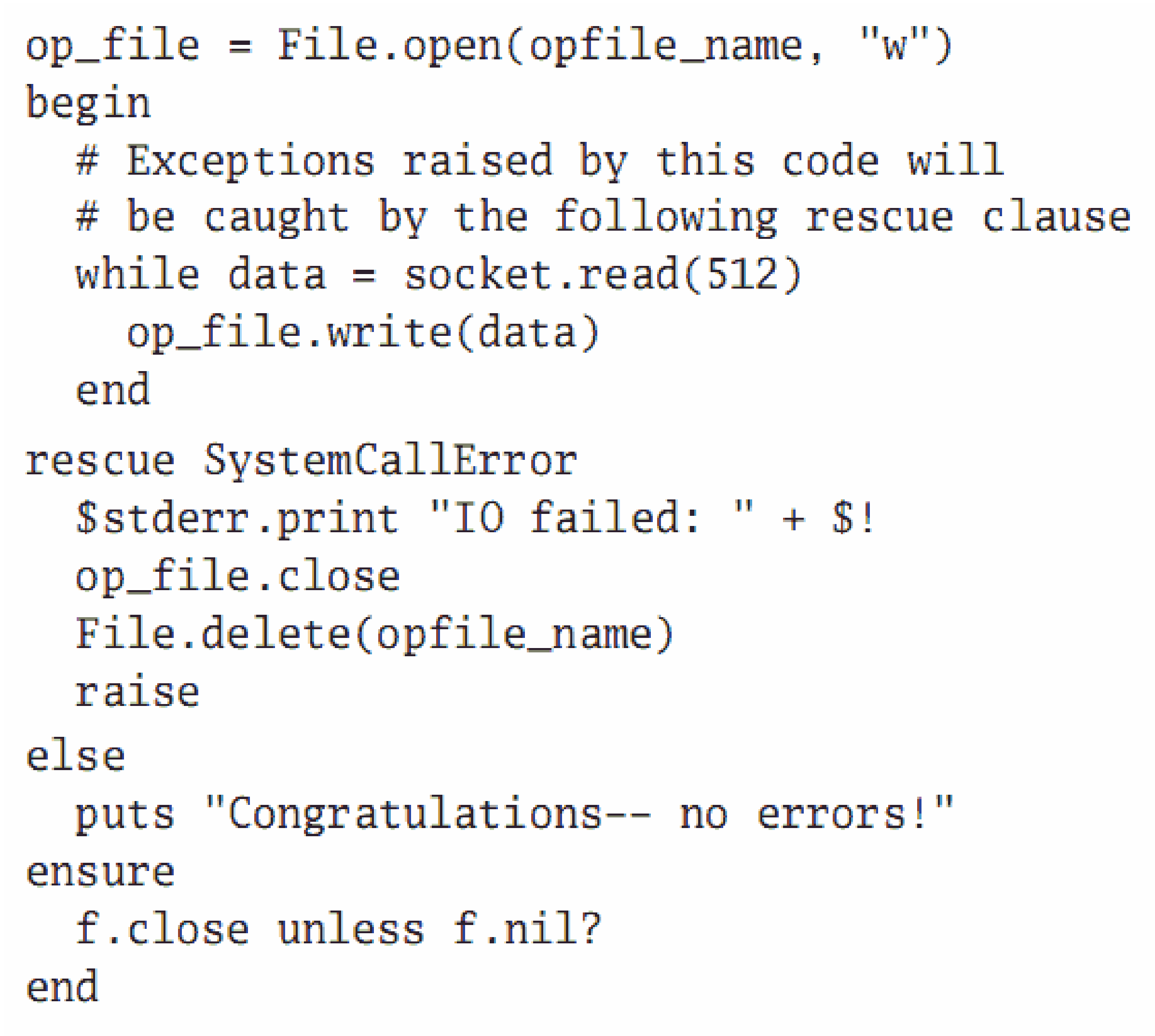
- The basic way of handling exceptions in Java consists of the *begin-rescue-else-ensure* trio
- The *begin* block contains the code for the basic algorithm
  - It tells what to do when everything goes smoothly
  - It can also contain code that throws an exception if something unusual happens
- The *rescue* block contains the code for exception handling

else 與 ensure 區塊

- The *else* block is executed only if no exceptions are raised by the main body of code
- The *ensure* block goes after the last rescue clause, which contains code to be executed whether or not an exception is raised in the begin block

例外處理形式

- In the form of “*begin-rescue-else-ensure*”



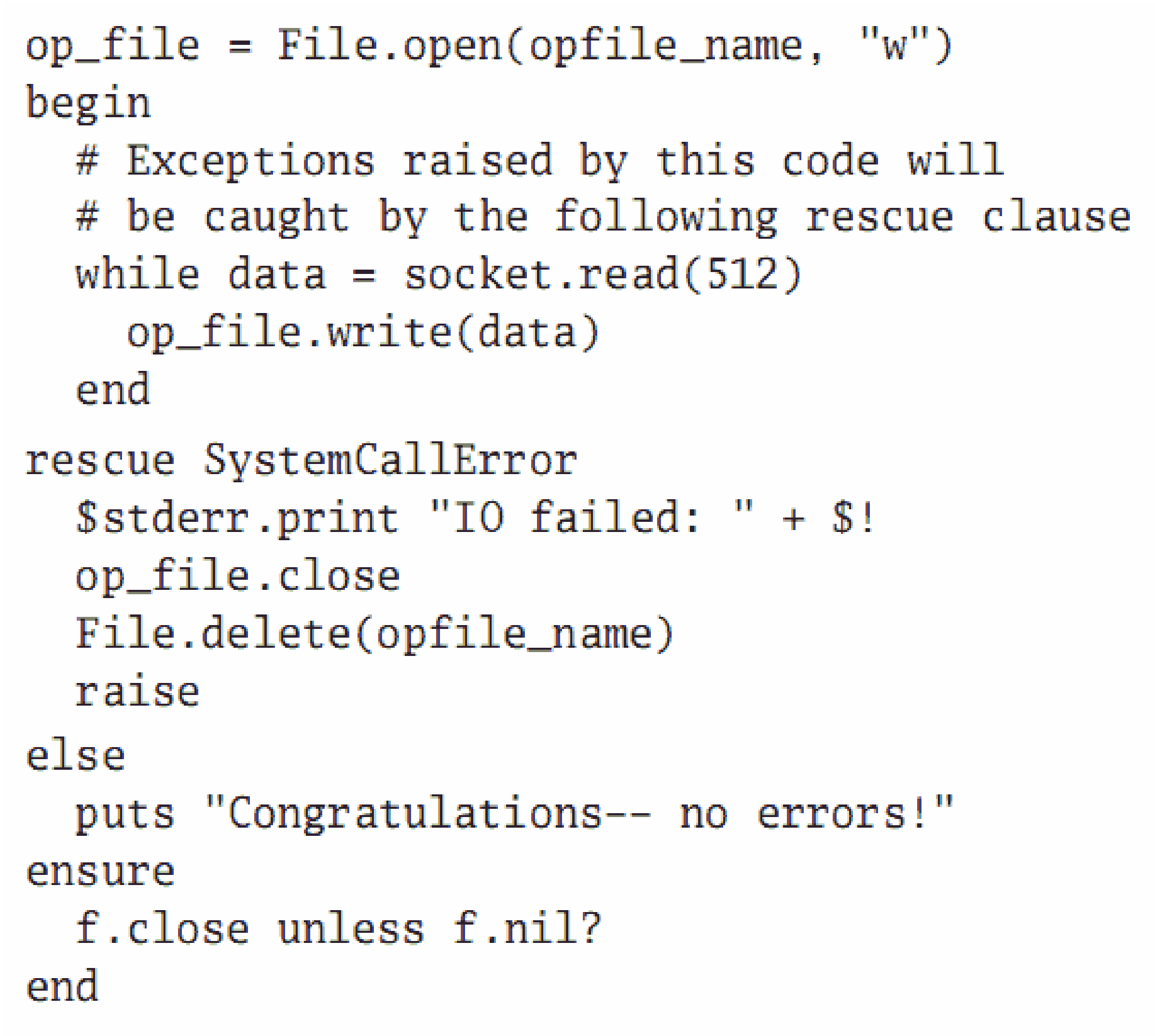
```
op_file = File.open(opfile_name, "w")
begin
  # Exceptions raised by this code will
  # be caught by the following rescue clause
  while data = socket.read(512)
    op_file.write(data)
  end

rescue SystemCallError
  $stderr.print "IO failed: " + $!
  op_file.close
  File.delete(opfile_name)
  raise
else
  puts "Congratulations-- no errors!"
ensure
  f.close unless f.nil?
end
```

如何捕捉例外 (Rescue)

- When an exception is raised, the *rescue* block begins execution
  - The rescue contains one or more parameters to represent the exception objects
  - The exception object thrown is plugged in for the rescue block parameter

例外處理範例

- Review the last example



```
op_file = File.open(opfile_name, "w")
begin
  # Exceptions raised by this code will
  # be caught by the following rescue clause
  while data = socket.read(512)
    op_file.write(data)
  end
rescue SystemCallError
  $stderr.print "IO failed: " + $!
  op_file.close
  File.delete(opfile_name)
  raise
else
  puts "Congratulations-- no errors!"
ensure
  f.close unless f.nil?
end
```

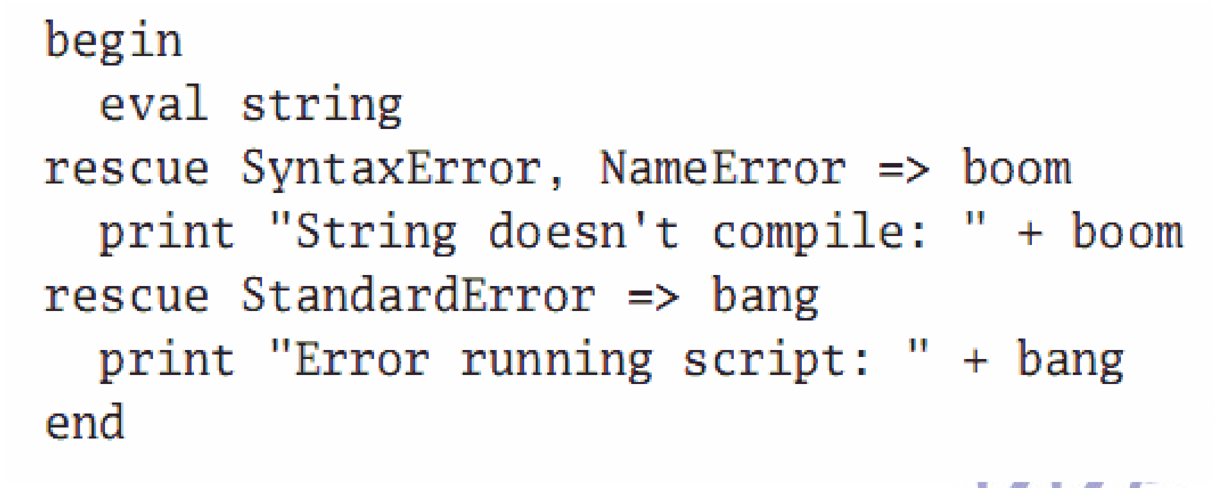
例外處理範例 (2)

- The rescue block in this example catches the exception “SystemCallError”, which means the read system operation failed
- Ruby places a reference to the associated Exception object into the global variable $!
  - In this example, the $! variable is used to format error message
- Additional: the *retry* statement in the rescue clause will cause the entire begin/end block to be repeated

OOP with Ruby

多重 rescue 區塊

- You can have multiple rescue clauses in a begin block, and each rescue clause can specify multiple exceptions to catch
  - At the end of each rescue clause you can give Ruby the name of a local variable to receive the matched exception
  - Many people find this more readable than using $! all over the place



```
begin
  eval string
rescue SyntaxError, NameError => boom
  print "String doesn't compile: " + boom
rescue StandardError => bang
  print "Error running script: " + bang
end
```

多重 rescue 區塊 (2)

- For each rescue clause in the begin block, Ruby compares the raised exception against each of the parameters in turn
  - If the raised exception matches a parameter, Ruby executes the body of the rescue and *stops looking*
- If you write a rescue clause with no parameter list, the parameter defaults to *StandardError*

沒捕捉到的例外？

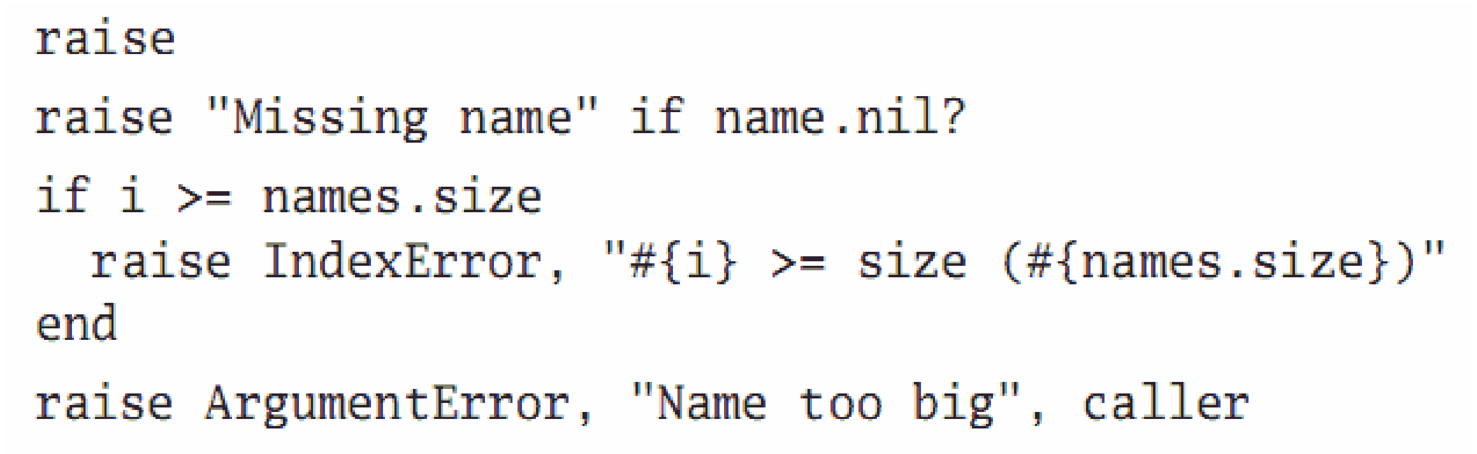
- If no rescue clause matches, or if an exception is raised outside a begin/end block, then
  - Ruby moves up the stack and looks for an exception handler in the caller, then in the caller’s caller, and so on

系統呼叫所引發的錯誤

- System errors are raised when a call to the operating system returns an error code
- On POSIX systems, these errors have names such as EAGAIN and EPERM
  - Ruby use these names
- Ruby takes these errors and wraps them each in a specific exception object
  - Each is a subclass of *SystemCallError*
  - Each is defined in a module called *Errno*
  - You'll find exceptions with class names such a Errno::EAGAIN, Errno::EIO, and Errno::EPERM

如何觸發例外 (Raising Exception)

- You can raise exceptions in your code with the *Kernel.raise* method
  - Some typical examples in action:



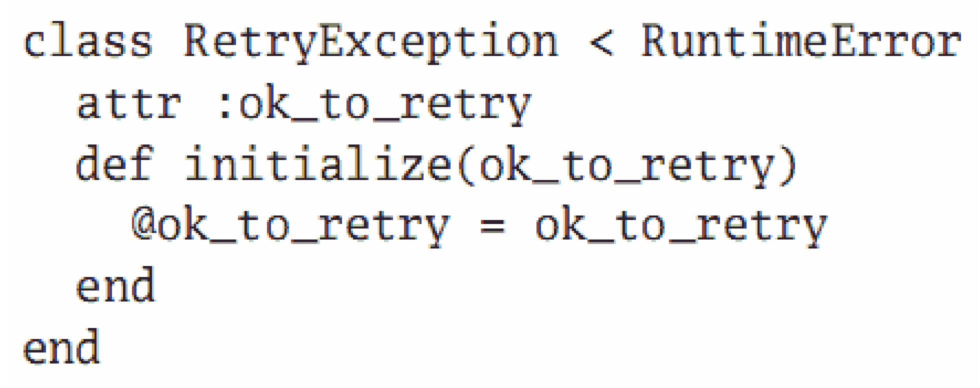
```
raise

raise "Missing name" if name.nil?

if i >= names.size
  raise IndexError, "#{i} >= size (#{names.size})"
end

raise ArgumentError, "Name too big", caller
```

如何觸發例外 (2)

- The first form simply re-raises the current exception (or a *RuntimeError* if there is no current exception)
- The second form creates a new *RuntimeError* exception, setting its message to the given string
  - This exception is then raised up the call stack
  - With a conditional decision
- The third form just like the second one but creates a *IndexError* exception
- The forth form creates an *ArgumentError* exception and then sets the associated message to the second argument and the stack trace to the third argument
  - The stack trace is normally produced using the *Kernel.caller* method

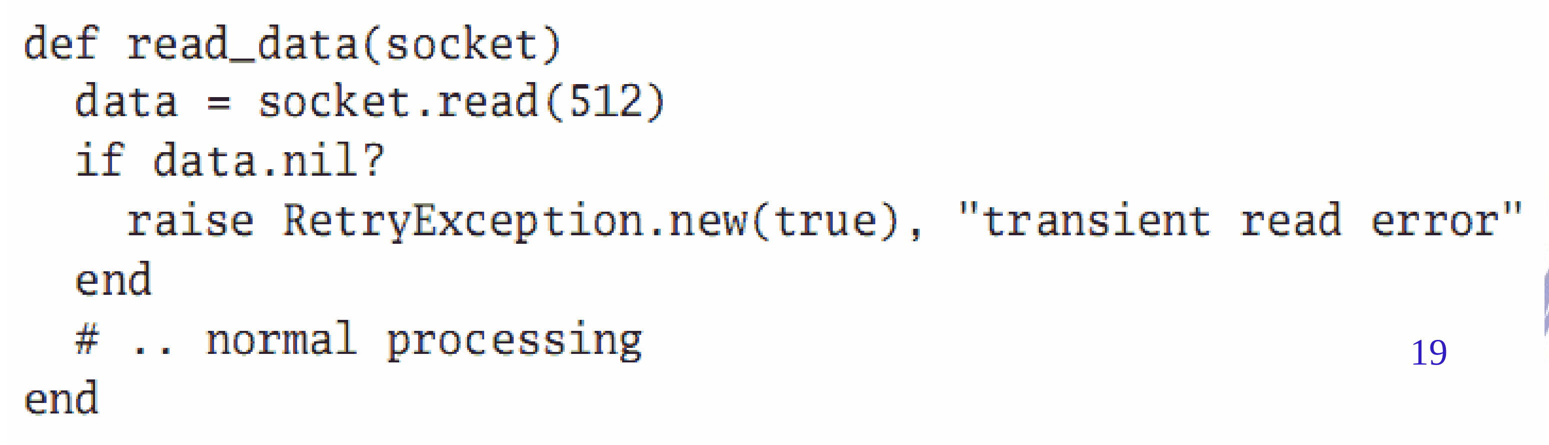
自訂例外類別

- You can define your own exceptions to hold any information that you need to pass out from the site of an error



```
class RetryException < RuntimeError
  attr :ok_to_retry
  def initialize(ok_to_retry)
    @ok_to_retry = ok_to_retry
  end
end
```

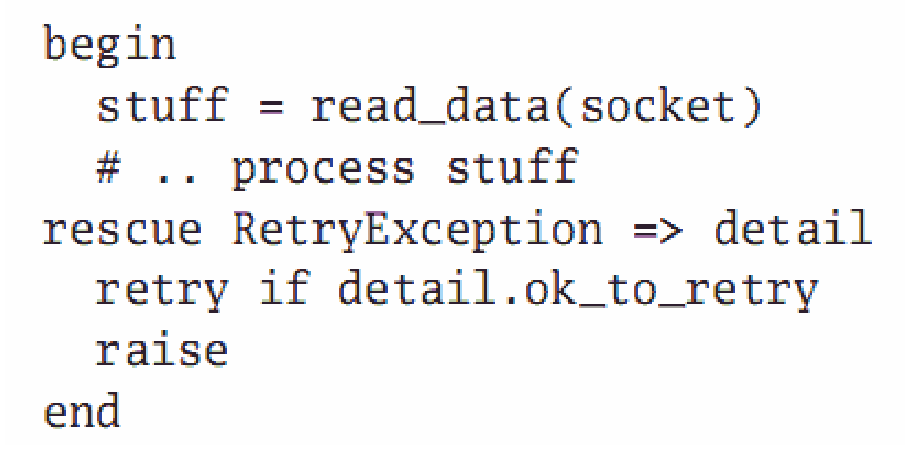
- Code which raises such exception:



```
def read_data(socket)
  data = socket.read(512)
  if data.nil?
    raise RetryException.new(true), "transient read error"
  end
  # .. normal processing
end
```

自訂例外類別 (2)

- And up the call stack:



```
begin
  stuff = read_data(socket)
  # .. process stuff
rescue RetryException => detail
  retry if detail.ok_to_retry
  raise
end
```

- The *retry* statement in the rescue clause will cause the entire begin/end block to be repeated

本章回顧