Chapter 10

Exceptions, Catch, and Throw

So far we've been developing code in Pleasantville, a wonderful place where nothing ever, ever goes wrong. Every library call succeeds, users never enter incorrect data, and resources are plentiful and cheap. Well, that's about to change. Welcome to the real world!

In the real world, errors happen. Good programs (and programmers) anticipate them and arrange to handle them gracefully. This isn't always as easy as it may sound. Often the code that detects an error does not have the context to know what to do about it. For example, attempting to open a file that doesn't exist is acceptable in some circumstances and is a fatal error at other times. What's your file-handling module to do?

The traditional approach is to use return codes. The open method could return some specific value to say it failed. This value is then propagated back through the layers of calling routines until someone wants to take responsibility for it. The problem with this approach is that managing all these error codes can be a pain. If a function calls open, then read, and finally close and each can return an error indication, how can the function distinguish these error codes in the value it returns to *its* caller?

To a large extent, *exceptions* solve this problem. 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.

The 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, shown in Figure 10.1 on page 169. As we'll see later, this hierarchy makes handling exceptions considerably easier.

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. Make your own exceptions subclasses of StandardError or one of its children. If you don't, your exceptions won't be caught by default.

Every Exception has associated with it a message string and a stack backtrace. If you define your own exceptions, you can add extra information.

Handling Exceptions

Here's some simple code that uses the open-uri library to download the contents of a web page and write it to a file, line by line:

Download **samples/tutexceptions_1.rb**

```
require 'open-uri'
web_page = open("http://pragprog.com/podcasts")
output = File.open("podcasts.html", "w")
while line = web_page.gets
  output.puts line
end
output.close
```

What happens if we get a fatal error halfway through? We certainly don't want to store an incomplete page to the output file.

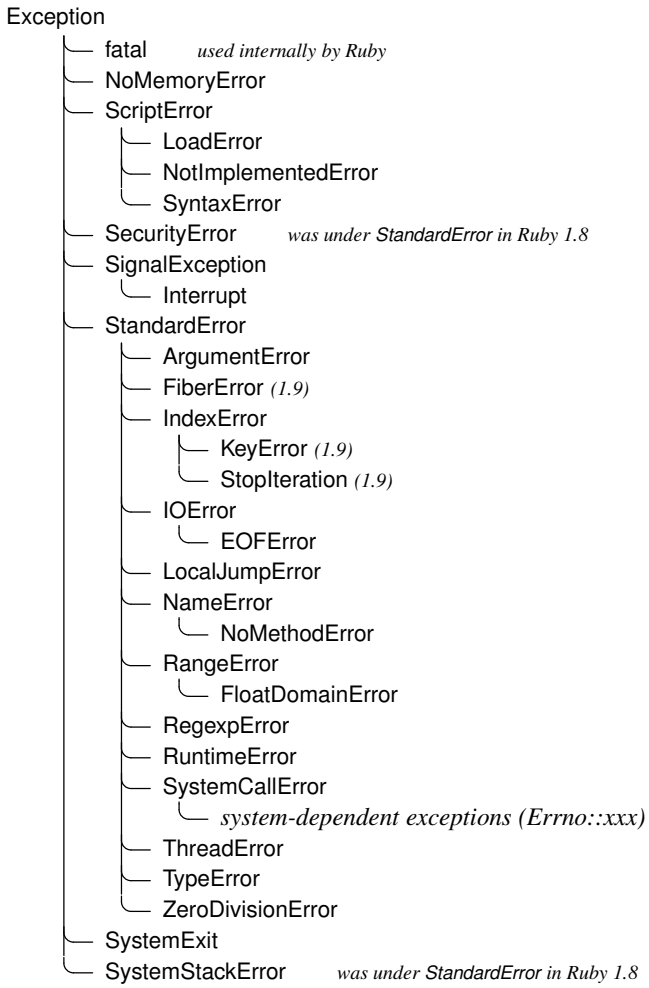
Let's add some exception-handling code and see how it helps. To do exception handling, we enclose the code that could raise an exception in a begin/end block and use one or more rescue clauses to tell Ruby the types of exceptions we want to handle. Because we specified Exception in the rescue line, we'll handle all exceptions of class Exception and all of its subclasses (which covers all Ruby exceptions). In the error-handling block, we report the error, close and delete the output file, and then reraise the exception:

Download **samples/tutexceptions_2.rb**

```
require 'open-uri'
page = "podcasts"
file_name = "#{page}.html"
web_page = open("http://pragprog.com/#{page}")
output = File.open(file_name, "w")
begin
  while line = web_page.gets
    output.puts line
  end
  output.close
rescue Exception
  STDERR.puts "Failed to download #{page}: #{$!}"
  output.close
  File.delete(file_name)
  raise
end
```

When an exception is raised, and independent of any subsequent exception handling, Ruby places a reference to the associated Exception object into the global variable $! (the exclamation point presumably mirroring our surprise that any of *our* code could cause errors). In the previous example, we used the $! variable to format our error message.

Figure 10.1. Ruby Exception Hierarchy



After closing and deleting the file, we call raise with no parameters, which reraises the exception in $!. This is a useful technique, because it allows you to write code that filters exceptions, passing on those you can't handle to higher levels. It's almost like implementing an inheritance hierarchy for error processing.

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. Most 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
```

How does Ruby decide which rescue clause to execute? It turns out that the processing is pretty similar to that used by the case statement. 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. The match is made using *parameter*===$!. For most exceptions, this means that the match will succeed if the exception named in the rescue clause is the same as the type of the currently thrown exception or is a superclass of that exception.[1] If you write a rescue clause with no parameter list, the parameter defaults to StandardError.

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

Although the parameters to the rescue clause are typically the names of Exception classes, they can actually be arbitrary expressions (including method calls) that return an Exception class.

System Errors

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. (If you're on a Unix box, you could type **man errno** to get a list of these errors.)

Ruby takes these errors and wraps them each in a specific exception object. Each is a subclass of SystemCallError, and each is defined in a module called Errno. This means you'll find exceptions with class names such as Errno::EAGAIN, Errno::EIO, and Errno::EPERM. If you want to get to the underlying system error code, Errno exception objects each have a class constant called (somewhat confusingly) Errno that contains the value.

1. This comparison happens because exceptions are classes, and classes in turn are kinds of Module. The === method is defined for modules, returning true if the class of the operand is the same as or is a descendant of the receiver.

```
Errno::EAGAIN::Errno        # =>  35
Errno::EPERM::Errno         # =>  1
Errno::EIO::Errno           # =>  5
Errno::EWOULDBLOCK::Errno   # =>  35
```

Note that EWOULDBLOCK and EAGAIN have the same error number. This is a feature of the operating system of the computer used to produce this book—the two constants map to the same error number. To deal with this, Ruby arranges things so that Errno::EAGAIN and Errno::EWOULDBLOCK are treated identically in a rescue clause. If you ask to rescue one, you'll rescue either. It does this by redefining SystemCallError#=== so that if two subclasses of SystemCallError are compared, the comparison is done on their error number and not on their position in the hierarchy.

Tidying Up

Sometimes you need to guarantee that some processing is done at the end of a block of code, regardless of whether an exception was raised. For example, you may have a file open on entry to the block, and you need to make sure it gets closed as the block exits.

The ensure clause does just this. ensure goes after the last rescue clause and contains a chunk of code that will always be executed as the block terminates. It doesn't matter if the block exits normally, if it raises and rescues an exception, or if it is terminated by an uncaught exception—the ensure block will get run:

```
f = File.open("testfile")
begin
  # .. process
rescue
  # .. handle error
ensure
  f.close
end
```

Beginners commonly make the mistake of putting the File.open inside the begin block. In this case, that would be incorrect, because open can itself raise an exception. If that were to happen, you wouldn't want to run the code in the ensure block, because there'd be no file to close.

The else clause is a similar, although less useful, construct. If present, it goes after the rescue clauses and before any ensure. The body of an else clause is executed only if no exceptions are raised by the main body of code.

```
f = File.open("testfile")
begin
  # .. process
rescue
  # .. handle error
else
  puts "Congratulations-- no errors!"
ensure
  f.close
end
```

Play It Again

Sometimes you may be able to correct the cause of an exception. In those cases, you can use the retry statement within a rescue clause to repeat the entire begin/end block. Clearly, tremendous scope exists for infinite loops here, so this is a feature to use with caution (and with a finger resting lightly on the interrupt key).

As an example of code that retries on exceptions, take a look at the following, adapted from Minero Aoki's net/smtp.rb library:

```
@esmtp = true
begin
  # First try an extended login. If it fails because the
  # server doesn't support it, fall back to a normal login
  if @esmtp then
    @command.ehlo(helodom)
  else
    @command.helo(helodom)
  end
rescue ProtocolError
  if @esmtp then
    @esmtp = false
    retry
  else
    raise
  end
end
```

This code tries first to connect to an SMTP server using the EHLO command, which is not universally supported. If the connection attempt fails, the code sets the @esmtp variable to false and retries the connection. If this fails a second time, the exception is raised up to the caller.

Raising Exceptions

So far we've been on the defensive, handling exceptions raised by others. It's time to turn the tables and go on the offensive. (Some say your gentle authors are always offensive, but that's a different book.)

You can raise exceptions in your code with the Kernel.raise method (or its somewhat judgmental synonym, Kernel.fail):

```
raise
raise "bad mp3 encoding"
raise InterfaceException, "Keyboard failure", caller
```

The first form simply reraises the current exception (or a RuntimeError if there is no current exception). This is used in exception handlers that need to intercept an exception before passing it on.

The second form creates a new RuntimeError exception, setting its message to the given string. This exception is then raised up the call stack.

The third form uses the first argument to create an exception and then sets the associated message to the second argument and the stack trace to the third argument. Typically the first argument will be either the name of a class in the Exception hierarchy or a reference to an object instance of one of these classes.[2] The stack trace is normally produced using the Kernel.caller method.

Here are some typical examples of raise 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
```

In the last example, we remove the current routine from the stack backtrace, which is often useful in library modules. We do this using the caller method, which returns the current stack trace. We can take this further; the following code removes two routines from the backtrace by passing only a subset of the call stack to the new exception:

```
raise ArgumentError, "Name too big", caller[1..-1]
```

Adding Information to Exceptions

You can define your own exceptions to hold any information that you need to pass out from the site of an error. For example, certain types of network errors may be transient depending on the circumstances. If such an error occurs and the circumstances are right, you could set a flag in the exception to tell the handler that it may be worth retrying the operation:

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

Somewhere down in the depths of the code, a transient error occurs:

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

2. Technically, this argument can be any object that responds to the message exception by returning an object such that object.kind_of?(Exception) is true.

Higher up the call stack, we handle the exception:

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

Catch and Throw

Although the exception mechanism of raise and rescue is great for abandoning execution when things go wrong, it's sometimes nice to be able to jump out of some deeply nested construct during normal processing. This is where catch and throw come in handy. Here's a trivial example—this code reads a list of words one at a time and adds them to an array. When done, it prints the array in reverse order. However, if any of the lines in the file doesn't contain a valid word, we want to abandon the whole process.

Download **samples/tutexceptions_14.rb**

```
word_list = File.open("wordlist")
catch (:done)  do
  result = []
  while line = word_list.gets
    word = line.chomp
    throw :done unless word =~ /^\w+$/
    result << word
  end
  puts result.reverse
end
```

catch defines a block that is labeled with the given name (which may be a Symbol or a String). The block is executed normally until a throw is encountered.

When Ruby encounters a throw, it zips back up the call stack looking for a catch block with a matching symbol. When it finds it, Ruby unwinds the stack to that point and terminates the block. So, in the previous example, if the input does not contain correctly formatted lines, the throw will skip to the end of the corresponding catch, not only terminating the while loop but also skipping the code that writes the reversed list. If the throw is called with the optional second parameter, that value is returned as the value of the catch. In this example, our word list incorrectly contains the line "*wow*." Without the second parameter to throw, the corresponding catch returns nil.

Download **samples/tutexceptions_15.rb**

```
word_list = File.open("wordlist")
word_in_error = catch(:done)  do
  result = []
  while line = word_list.gets
    word = line.chomp
```

```
      throw(:done, word) unless word =~ /^\w+$/
      result << word
    end
    puts result.reverse
  end
  if word_in_error
    puts "Failed: '#{word_in_error}' found, but a word was expected"
  end
```

produces:

```
Failed: '*wow*' found, but a word was expected
```

The following example uses a throw to terminate interaction with the user if ! is typed in response to any prompt:

Download **samples/tutexceptions_16.rb**

```
def prompt_and_get(prompt)
  print prompt
  res = readline.chomp
  throw :quit_requested if res == "!"
  res
end
catch :quit_requested do
  name = prompt_and_get("Name: ")
  age  = prompt_and_get("Age:  ")
  sex  = prompt_and_get("Sex:  ")
  # ..
  # process information
end
```

As this example illustrates, the throw does not have to appear within the static scope of the catch.