Ruby master - Feature #10481

Add "if" and "unless" clauses to rescue statements

11/05/2014 07:20 PM - javawizard (Alex Boyd)

Status:	Assigned	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:		

Description

I'd like to propose a syntax change: allow boolean "if" and "unless" clauses to follow a rescue statement.

Consider the following:

```
begin
  ...
rescue SomeError => e
  if e.error_code == 1
    ...handle error...
  else
    raise
  end
end
```

This is a fairly common way of dealing with exceptions where some condition above and beyond the exception's type determines whether the exception should be rescued. It's verbose, though, and it's not obvious at first glance exactly what conditions are being rescued, especially if "...handle error..." is more than a few lines long. I propose that the following be allowed:

```
begin
  ...
rescue SomeError => e if e.error_code == 1
  ...handle error...
end
```

"unless" would, of course, be allowed as well:

```
begin
  ...
rescue SomeError => e unless e.error_code == 2
  ...handle error...
end
```

A rescue statement whose boolean condition failed would be treated the same as if the exception being raised didn't match the exception being rescued, and move on to the next rescue statement:

```
begin
  ...
rescue SomeError => e if e.error_code == 1
  ...handle error code 1...
rescue SomeError => e if e.error_code == 2
  ...handle error code 2...
end
```

And finally, catch-all rescue statements would be allowed as well:

```
begin
  ...
rescue => e if e.message == "some error"
  ...handle error...
end
```

History

#1 - 11/20/2014 12:23 PM - javawizard (Alex Boyd)

- File rescue-conditions.diff added

A patch.

I've only tested the Ripper bits cursorily, so I could very well have missed something - feedback in that particular direction would be appreciated.

I'm also not sold on the format of NODE_RESCOND in node.c, but I can't think of anything better at the moment.

No tests, yet - I'll write them up if there's a halfway-decent chance of this patch getting accepted.

#2 - 11/20/2014 12:30 PM - javawizard (Alex Boyd)

- File rescue-conditions.diff added

A second patch, to fix an uninitialized variable I noticed just after I'd uploaded the first.

#3 - 11/20/2014 03:09 PM - rklemme (Robert Klemme)

Alex Boyd wrote:

> I'd like to propose a syntax change: allow boolean "if" and "unless" clauses to follow a rescue statement.
>
> I propose that the following be allowed:
>
> ```
> begin
>   ...
> rescue SomeError => e if e.error_code == 1
>   ...handle error...
> end
> ```

Do you have an idea of the runtime performance impact? I mean, a type check (as done today) is fast and effort cannot change. But with your proposal arbitrary code can be executed as part of the test. If you do that on multiple levels of the call stack the effect on performance may be noticeable.

Also it must be defined how you treat exceptions that are raised inside the test expression. It may be hard to hunt down errors because these exceptions would likely shadow the original exception and it may be hard to find out what content in the original exception triggered the exception in the testing code.

Also it must be made sure that code in "ensure" block is executed under all conditions. That may become more difficult with your change.

#4 - 11/20/2014 06:52 PM - javawizard (Alex Boyd)

Robert Klemme wrote:

> Do you have an idea of the runtime performance impact? I mean, a type check (as done today) is fast and effort cannot change. But with your proposal arbitrary code can be executed as part of the test. If you do that on multiple levels of the call stack the effect on performance may be noticeable.

For code that doesn't make use of this feature, the performance impact is 0 - no additional bytecodes are added to the compiled result. For code that does, the impact is 4 bytecodes *less* than, but otherwise identical to, adding an equivalent raise unless e.error_code == 1 check to the top of the rescue body (and 6 bytecodes less than a full if/then/else raise end/ check):

```
irb(main):072:0> puts RubyVM::InstructionSequence.disassemble proc { begin; do_something; rescue => e if foo;
handle_failure; end }
== disasm: <RubyVM::InstructionSequence:block in irb_binding@(irb)>=====
== catch table
| catch type: rescue st: 0004 ed: 0009 sp: 0000 cont: 0010
== disasm: <RubyVM::InstructionSequence:rescue in block in irb_binding@(irb)>
local table (size: 2, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, keyword: 0@3] s0)
[ 2] "\#$!"
0000 getlocal_OP__WC__0 2                                 (  72)
0002 putobject        StandardError
0004 checkmatch       3
0006 branchunless     23
0008 getlocal_OP__WC__0 2
0010 setlocal_OP__WC__1 2
0012 putself
0013 opt_send_simple  <callinfo!mid:foo, argc:0, FCALL|VCALL|ARGS_SKIP>
0015 branchunless     23
0017 trace            1
```

```
0019 putself
0020 opt_send_simple  <callinfo!mid:handle_failure, argc:0, FCALL|VCALL|ARGS_SKIP>
0022 leave
0023 getlocal_OP__WC__0 2
0025 throw            0
| catch type: retry  st: 0009 ed: 0010 sp: 0000 cont: 0004
| catch type: redo   st: 0002 ed: 0010 sp: 0000 cont: 0002
| catch type: next   st: 0002 ed: 0010 sp: 0000 cont: 0010
|------------------------------------------------------------------------
local table (size: 2, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, keyword: 0@3] s1)
[ 2] e
0000 trace            256                                           (  72)
0002 trace            1
0004 trace            1
0006 putself
0007 opt_send_simple  <callinfo!mid:do_something, argc:0, FCALL|VCALL|ARGS_SKIP>
0009 nop
0010 trace            512
0012 leave
=> nil

irb(main):069:0> puts RubyVM::InstructionSequence.disassemble proc { begin; do_something; rescue => e; raise u
nless foo; handle_failure; end }
== disasm: <RubyVM::InstructionSequence:block in irb_binding@(irb)>=====
== catch table
| catch type: rescue st: 0004 ed: 0009 sp: 0000 cont: 0010
== disasm: <RubyVM::InstructionSequence:rescue in block in irb_binding@(irb)>
local table (size: 2, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, keyword: 0@3] s0)
[ 2] "\#$!"
0000 getlocal_OP__WC__0 2                                           (  69)
0002 putobject        StandardError
0004 checkmatch       3
0006 branchunless     29
0008 getlocal_OP__WC__0 2
0010 setlocal_OP__WC__1 2
0012 trace            1
0014 putself
0015 opt_send_simple  <callinfo!mid:foo, argc:0, FCALL|VCALL|ARGS_SKIP>
0017 branchif         23
0019 putself
0020 opt_send_simple  <callinfo!mid:raise, argc:0, FCALL|VCALL|ARGS_SKIP>
0022 pop
0023 trace            1
0025 putself
0026 opt_send_simple  <callinfo!mid:handle_failure, argc:0, FCALL|VCALL|ARGS_SKIP>
0028 leave
0029 getlocal_OP__WC__0 2
0031 throw            0
| catch type: retry  st: 0009 ed: 0010 sp: 0000 cont: 0004
| catch type: redo   st: 0002 ed: 0010 sp: 0000 cont: 0002
| catch type: next   st: 0002 ed: 0010 sp: 0000 cont: 0010
|------------------------------------------------------------------------
local table (size: 2, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, keyword: 0@3] s1)
[ 2] e
0000 trace            256                                           (  69)
0002 trace            1
0004 trace            1
0006 putself
0007 opt_send_simple  <callinfo!mid:do_something, argc:0, FCALL|VCALL|ARGS_SKIP>
0009 nop
0010 trace            512
0012 leave
=> nil
```

> Also it must be defined how you treat exceptions that are raised inside the test expression. It may be hard to hunt down errors because these exceptions would likely shadow the original exception and it may be hard to find out what content in the original exception triggered the exception in the testing code.

They're treated otherwise as if they had occurred within the rescue body, but with the line number set to the rescue clause itself:

```
irb(main):074:0> begin
irb(main):075:1*  raise 'something'
irb(main):076:1> rescue if raise 'something else'
irb(main):077:1>  puts "shouldn't get here"
```

```
irb(main):078:1> end
RuntimeError: something else
    from (irb):76:in `rescue in irb_binding'
    from (irb):74
    from bin/irb:11:in `<main>'
```

I think this seems fairly clear. It would perhaps be nice if it said in `rescue condition in irb_binding` instead of in `rescue in irb_binding`, but that's beyond the scope of my knowledge (and I think the line number makes it fairly hard to miss what's going on). Thoughts?

> Also it must be made sure that code in "ensure" block is executed under all conditions. That may become more difficult with your change.

The additional bytecodes to check the condition and branch to label_miss are tacked on just before the rescue body itself, so they receive all the same protections. My changes are confined entirely to the case NODE_RESBODY: block (all of whose generated bytecodes are protected by the ensure clause), so I imagine it would be fairly hard to break this interaction without deliberately intending to do so - but feel free to contradict me if I'm wrong. At any rate, this does work correctly as-is:

```
irb(main):086:0> begin
irb(main):087:1*  raise 'something'
irb(main):088:1> rescue if raise 'something else'
irb(main):089:1>  puts "shouldn't get here"
irb(main):090:1> ensure
irb(main):091:1*  puts "but we should get here"
irb(main):092:1> end
but we should get here
RuntimeError: something else
    from (irb):88:in `rescue in irb_binding'
    from (irb):92
    from bin/irb:11:in `<main>'
```

#5 - 11/21/2014 05:23 PM - nobu (Nobuyoshi Nakada)

Intersting.

With your patch, the condition is checked *after* matching against the exception classes, not a part of the matching, right?
Which behavior is preferred?

#6 - 11/21/2014 05:51 PM - javawizard (Alex Boyd)

Nobuyoshi Nakada wrote:

> With your patch, the condition is checked *after* matching against the exception classes, not a part of the matching, right?
> Which behavior is preferred?

They're checked after checking for a match against the exception classes, so the conditions won't be run at all if the exception class doesn't match, but they'll cause the next rescue clause to be checked if they fail instead of skipping all remaining rescue clauses like raise unless some_condition would. Is that what you're asking?

(If so, I, for one, prefer this behavior.)

#7 - 11/22/2014 08:48 AM - nobu (Nobuyoshi Nakada)

- Description updated

- Status changed from Open to Assigned

Your patch seems based on pretty old revision.
I made updated version: https://github.com/nobu/ruby/compare/Feature%2310481-conditional-rescue

#8 - 11/22/2014 08:59 AM - javawizard (Alex Boyd)

Awesome, thanks! I was having trouble building trunk on OS X, so (I believe) I based it off of 2.1.5. Quite a bit more's changed than I had anticipated.

Thanks for writing up tests as well!

#9 - 12/04/2014 08:24 AM - javawizard (Alex Boyd)

Any update on this? Is this just waiting for 2.2 to make it out the door?

#10 - 12/04/2014 04:38 PM - brianhempel (Brian Hempel)

- File smime.p7s added

You're right there may be some risk for anti-patterns, but from my experience this example in the original proposal is practical:

```
begin
  ...
rescue => e if e.message == "some error"
  ...handle error...
end
```

Sometimes code libraries forget to make a separate object for every error and you are forced to match on the error message. Right now, you have to write code like:

```
begin
…
rescue => e
  raise unless e.message =~ /card declined/i
  …handle error…
end
```

Granted, the best solution is to fix the code library.

#11 - 12/04/2014 06:35 PM - javawizard (Alex Boyd)

Brian Hempel wrote:

> Sometimes code libraries forget to make a separate object for every error

This was, in fact, one of the reasons that motivated me to propose this in the first place - I've been doing some work with a web service whose documentation is less than thorough about all of the various error codes it can issue when it fails, and that makes having a separate exception for each and every error in a client library for said web service difficult.

bruka wrote:

> I'm trying to think of use cases for this

In addition to the above, here's another one: a block of code that could fail for reasons beyond your control, where you want to catch errors in a production environment and handle them as gracefully as possible while allowing them to pass through in development and test. In Rails, this could look like:

```
begin
  ...critical code here...
rescue SomeError if Rails.env.production?
  ...try to recover as best we can...
end
```

This was also a motivating reason for my proposing this.

> You read the entire loop and at the end you realize that it only runs in certain cases. Similar with the conditional rescues. You're jumping back and forth in your method to try to figure out what happens when. Not fun.

I agree that jumping back and forth can be confusing, but I would argue that this is no worse than rescue clauses as they currently are, and even *better* when the alternative is to have a raise unless inside the rescue body where it could be missed.

> Moreover, as another person noted, skipping the rescue clause can deprive you of valuable debugging information. Why not just send the error to another object to be processed:

That seems like a separate issue entirely - insofar as there's already a way to limit the scope of what's being rescued (re-raiseing and only rescuing certain exception types in the first place), rescue conditionals won't make this any different. I would also argue that having a separate class or method to encapsulate what would otherwise be a line or two of logic is overkill, and just serves to obfuscate what's really going on further.

I do agree that (like most language features) there's a potential for abuse here, but (in my opinion) the benefits outweigh said potential. That's my 4 cents. :-)

#12 - 12/28/2014 10:28 AM - javawizard (Alex Boyd)

Any update on this? (Or is this just a matter of more people getting around to giving it due consideration before merging?)

#13 - 01/17/2015 07:37 PM - lehm (Lenna Hammer)

Might 'case-when' follow this syntax where 'if' can be used...

or maybe just add some new method like 'raise' but try next recuse clause?

#14 - 01/17/2015 10:50 PM - sawa (Tsuyoshi Sawada)

I think this is a terrible idea. Error messages should be used only for creating outputs, and not for internal conditioning. They should be able to be revisited independently of other parts of the code. Hard coding conditions based on the error messages would be fragile and crappy. Assigning flags to exception instances as in the example indicates that exception subclassing was not done properly. You should have made exception subclass with enough fineness so that it would be sufficient to condition just by the exception class.

#15 - 01/18/2015 02:46 PM - radan (Radan Skorić)

Alex Boyd wrote:

> In addition to the above, here's another one: a block of code that could fail for reasons beyond your control, where you want to catch errors in a production environment and handle them as gracefully as possible while allowing them to pass through in development and test. In Rails, this could look like:
>
> ```
> begin
>   ...critical code here...
> rescue SomeError if Rails.env.production?
>   ...try to recover as best we can...
> end
> ```

It seems to me that this is not a very good example. From a high level perspective there are two decisions to be made about an exception:

1. Deciding whether to raise or not to raise the exception and
2. Deciding how to handle the exception

Raising the exception only in production clearly falls within "should I raise?" decision but here it is being performed as part of "how will I handle it?" decision. It seems to me that this decision should be done by the code that is doing the raising. If I look at the place in code where raising is happening I don't want to have to look for all the catch clauses to see that it will effectively not be raised at all in development.

I also agree with Tsuyoshi that error messages should be only for formatting. Instead, a child exception should have been raised.

That leaves the error_code based examples. That looks like it could be a legitimate use case but it seems to me it would be just as readable, if not more, as a case statement inside the rescue block.

The feature it self looks very cool but looking at the code bases I'm working on I can't really see places where using it would be a better long term solution than refactoring the code that is doing the raising.

Files

rescue-conditions.diff	6.76 KB	11/20/2014	javawizard (Alex Boyd)
rescue-conditions.diff	6.57 KB	11/20/2014	javawizard (Alex Boyd)
smime.p7s	4.78 KB	12/04/2014	brianhempel (Brian Hempel)