Ruby master - Feature #15554
warn/error passing a block to a method which never use a block
01/22/2019 04:48 AM - ko1 (Koichi Sasada)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td>matz (Yukihiro Matsumoto)</td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
</tbody>
</table>

Description

Abstract
Warn or raise an ArgumentError if block is passed to a method which does not use a block. In other words, detect "block user methods" implicitly and only "block user methods" can accept a block.

Background
Sometimes, we pass a block to a method which ignores the passed block accidentally.

```ruby
def my_open(name)
  open(name)
end

# user hopes it works as Kernel#open which invokes a block with opened file.
my_open(name){|f| important_work_with f }
# but simply ignored...
```

To solve this issue, this feature request propose showing warnings or raising an exception on such case.

Last developer's meeting, matz proposed &nil which declares this method never receive a block. It is explicit, but it is tough to add this &nil parameter declaration to all of methods (do you want to add it to def []=(i, e, &nil)?).
(I agree &nil is valuable on some situations)

Spec

Define "use a block" methods

We need to define which method accepts a block and which method does not.

- (1) method has a block parameter (&b)
- (2) method body has `yield`
- (3) method body has super (ZSUPER in internal terminology) or super(...)  # Optional
- (4) method body has singleton method (optional)

(1) and (2) is very clear. I need to explain about (3) and (4).

(3). super (ZSUPER) passes all parameters as arguments. So there is no surprise that which can accept block. However super(...) also passes a block if no explicit block passing (like super(){}) or super(&b)) are written. I'm not sure we need to continue this strange specification, but to keep compatibility depending this spec, I add this rule.

(4). surprisingly, the following code invoke a block:

```ruby
def foo
  class << Object.new
    yield
  end
end

foo( p :ok ) #=> :ok
```

I'm also not sure we need to keep this spec, but to allow this spec, I added (4) rule.
Strictly speaking, it is not required, but we don't keep the link from singleton class ISeq to lexical parent iseq now, so I added it.

**Exceptional cases**

A method called by super doesn't warn warning even if this method doesn't use a block. The rule (3) can pass blocks easily and there are many methods don't use a block.

So my patch ignores callings by super.

**corner cases**

There are several cases to use block without (1)-(4) rules.

**Proc.new/proc/lambda without a block**

Now it was deprecated in r66772 ([9f1fb0a17f6ebc59356d58ce5e98db61a3c03550](9f1fb0a17f6ebc59356d58ce5e98db61a3c03550)). Related discussion: [Bug #15539](bug15539)

**block_given?**

block_given? expects block, but I believe we use it with yield or a block parameter. If you know the usecase without them, please tell us.

**yield in eval**

We can't know yield (or (3), (4) rule) in an eval evaluating string at calling time.

```ruby
def foo
  eval('yield')
end
```

```
foo() # at calling time,
      # we can't know the method foo can accept a block or not.
```

So I added a warning to use yield in eval like that: test.rb:4: warning: use yield in eval will not be supported in Ruby 3.

Workaround is use a block parameter explicitly.

```ruby
def foo &b
  eval('b.call')
end
```

foo( p :ok )

**Implementation**

Strategy is:

- [compile time] introduce iseq::has_yield field and check it if the iseq (or child iseq) contains yield (or something)
- [calling time] if block is given, check iseq::has_yield flag and show warning (or raise an exception)

[https://gist.github.com/ko1/c9148ad0224bf5be033c76ed2220c0b](https://gist.github.com/ko1/c9148ad0224bf5be033c76ed2220c0b)

On this patch, now it raises an error to make it easy to detect. It is easy to switch to show the warning.

**Evaluation and discussion**

I tried to avoid ruby's tests.

[https://gist.github.com/ko1/37483e7940cdc4390b8eb00018837b86](https://gist.github.com/ko1/37483e7940cdc4390b8eb00018837b86)

Here is a patch.
There are several patterns to avoid warnings.

### tests for block_given?, Proc.new (and similar) without block

Add a dummy block parameter.
It is test-specific issue.

### empty each

Some tests add each methods do not yield, like: def each; end.
Maybe test-specific issue, and adding a dummy block parameter.

### Subtyping / duck typing

[https://github.com/ruby/ruby/blob/c01a5ee85e2d6a7128ccc1fb143bfa694284ca87/lib/optparse.rb#L698](https://github.com/ruby/ruby/blob/c01a5ee85e2d6a7128ccc1fb143bfa694284ca87/lib/optparse.rb#L698)

This parse method doesn’t use yield, but other sub-type's parse methods use.

### super with new method

[https://gist.github.com/ko1/37483e7940cdc4390bf8eb0001883786#file-tests-patch-L61](https://gist.github.com/ko1/37483e7940cdc4390bf8eb0001883786#file-tests-patch-L61)

This method override Class#new method and introduce a hook with block (yield a block in this hook code).


In this method, call super and it also passing a block. However, called initialize doesn’t use a block.

### Change robustness

This change reduce robustness for API change.

Delegator requires to support __getobj__ for client classes.
Now __getobj__ should accept block but most of __getobj__ clients do not call given block.

[https://github.com/ruby/ruby/blob/trunk/lib/delegate.rb#L80](https://github.com/ruby/ruby/blob/trunk/lib/delegate.rb#L80)

This is because of delegator.rb's API change.

[https://gist.github.com/ko1/37483e7940cdc4390bf8eb0001883786#file-tests-patch-L86](https://gist.github.com/ko1/37483e7940cdc4390bf8eb0001883786#file-tests-patch-L86)

Nobu says calling block is not required (ignoring a block is no problem) so it is not a bug for delegator client classes.

### Found issues.

```ruby
[2945/20449] Rinda::TestRingServer#test_do_reply = 0.00 s
  1) Error:
Rinda::TestRingServer#test_do_reply:
ArgumentError: passing block to the method "with_timeout" (defined at /home/ko1/src/ruby/trunk/test/rinda/test_rinda.rb:787) is never used.
  /home/ko1/src/ruby/trunk/test/rinda/test_rinda.rb:635:in `test_do_reply'

[2946/20449] Rinda::TestRingServer#test_do_reply_local = 0.00 s
  2) Error:
Rinda::TestRingServer#test_do_reply_local:
ArgumentError: passing block to the method "with_timeout" (defined at /home/ko1/src/ruby/trunk/test/rinda/test_rinda.rb:787) is never used.
  /home/ko1/src/ruby/trunk/test/rinda/test_rinda.rb:657:in `test_do_reply_local'

[10024/20449] TestGemRequestSetGemDependencyAPI#test_platform_mswin = 0.01 s
  3) Error:
TestGemRequestSetGemDependencyAPI#test_platform_mswin:
ArgumentError: passing block to the method "util_set_arch" (defined at /home/ko1/src/ruby/trunk/lib/rubygems/test_case.rb:1053) is never used.
```
These 4 detection show the problem. with_timeout method (used in Rinda test) and util_set_arch method (used in Rubygems test) simply ignore the given block.
So these tests are simply ignored.

I reported them. (https://github.com/rubygems/rubygems/issues/2601)

raise an error or show a warning?

At least, Ruby 2.7 should show warning for this kind of violation with -w.
How about for Ruby3?

Related issues:
Copied from Ruby master - Feature #10499: Eliminate implicit magic in Proc.ne...

Associated revisions
Revision 8da74ab - 01/04/2021 06:54 AM - nobu (Nobuyoshi Nakada)
[DOC] Update Proc.new without a block [ci skip]

[Feature #10499]
[Feature #15554]

History
#1 - 01/22/2019 04:51 AM - ko1 (Koichi Sasada)
- Description updated

#2 - 01/22/2019 06:36 AM - hsbt (Hiroshi SHIBATA)
I fixed the issues of TestGemRequestSetGemDependencyAPI on upstream repository and merged them at r66904.

#3 - 02/10/2019 02:24 PM - alanwu (Alan Wu)
Related: Feature #10499

#4 - 02/12/2019 08:52 PM - decuplet (Nikita Shilnikov)
I have a nice example where I use calls like super {...} even if the super method doesn't yield a block. From my understanding, this behavior won't be broken by the changes, but I still want to add it to the context.
I'm the author of the dry-monads gem, it defines, you might have guessed, a bunch of monads. For instance, there is the Result monad which represents possibly-unsuccesful computation. A typical use case:

def call(params)
  validate(params).bind { |values|
    create_account(values[:account]).bind { |account|
      create_owner(account, values[:owner]).fmap { |owner|
        [account, owner]
      }
    }
  }
end

Here validate, create_account, and create_owner all return a Result value, binds compose the results together. That's rather ugly, that's why the gem also adds so-called do notation (the name has stolen from Haskell), it's a mixin you add to a class which prepends every method you define and passes a block which in order tries to unwrap a Result value. Better see the code and read the comments:

# this line also prepends the current class with a dynamically created module
include Dry::Monads::Do

# this method will be overridden in the prepended module
def call(params)
    # yield will halt the execution if validate returns Failure(...)
    # or unwraps the result if it returns Success(...) so that it'll
    # be assigned to values. This way we avoid the chain of .bind calls
    values = yield validate(params)
    account = yield create_account(values[:account])
    owner = yield create_owner(account, values[:owner])

    Success([account, owner])
end

Obviously, the do version is way cleaner yet has the same semantics. Dry::Monads::Do wraps all the methods as they are defined so that you don't
need to worry whether a method unwraps Results or not. If it doesn't, then it just doesn't call the block, that simple. Also Dry::Monads::Do is smart
enough to discard its block if another block is given to a method:

include Dry::Monads::Result::Mixin
include Dry::Monads::Do

def foo
    bar { 5 }
    baz # baz will return 6, see below
end

def bar
    # here the block comes from foo, not from Do
    yield + 1
end

def baz
    # no block from foo given, Do in action
    yield Success(6)
end

Now here's the problem.

include Dry::Monads::Do

def call
    foo
end

def foo
    # foo gets the block from Do but doesn't use it
    5
end

In the last example, I have no means to detect that foo won't use the block. I could analyze parameters, I guess it would be slower, but if a method
uses yield I just cannot detect it without analyzing the source code, that'd be dead slow and not reliable either.

See what Do does here and there for more details.

I should add that dry-monads is quite popular, it's not only used by me :) (see it on rubygems https://rubygems.org/gems/dry-monads)

Gem's docs

#5 - 03/28/2019 11:44 AM - localhostdotdev (localhost .dev)
To detect if a block is used, binding would also need to be detected, e.g.: def b(arg); arg.eval("yield"); end; def a; b(binding); end

#6 - 03/28/2019 12:28 PM - matthewd (Matthew Draper)
This is great! Ignored blocks can be very confusing.

    A method called by super doesn't warn warning even if this method doesn't use a block.
    The rule (3) can pass blocks easily and there are many methods dont use a block.

    So my patch ignores callings by super.

Would it be possible for a method called by super to only ignore an unexpected block if the calling method (that contained super) also contained
something that explicitly uses/consumes the block (&b / yield)?

I think that would still allow existing uses where a method consumes a block itself, and then (accidentally) implicitly passes it to a super that doesn't
want it -- but that it would also keep the warning in the other situation, where an entire super-chain does not use the block.
Otherwise I am worried that super becomes a bad thing to use, because it completely disables this new safety feature.

#7 - 05/28/2019 08:39 PM - k0kubun (Takashi Kokubun)
- Description updated

#8 - 05/28/2019 08:41 PM - k0kubun (Takashi Kokubun)
- Copied from Feature #10499: Eliminate implicit magic in Proc.new and Kernel#proc added

#9 - 01/04/2021 06:45 AM - nobu (Nobuyoshi Nakada)
- Status changed from Open to Closed