Add Kernel#must! to the standard library

We should add a method Kernel#must! (name TBD) which raises if self is nil and returns self otherwise.

Ruby 3 introduces type annotations for the standard library. Type checkers consume these annotations, and report errors for type mismatches. One of the most common and most valuable type errors is whether nil is allowed as an argument or return value. Sorbet's type system tracks this, and RBS files have syntax for annotating whether nil is allowed or not.

Since Sorbet checks proper usage of nil, it requires code that looks like this:

```
if thing.nil?
  raise "The thing was nil"
end
thing.do_something
```

This is good because it forces the programmer to acknowledge that the thing might be nil, and declare that they'd rather raise an exception in that case than handle the nil (of course, there are many other times where nil is both possible and valid, which is why Sorbet forces at least considering in all cases).

It is annoying and repetitive to have to write these if .nil? checks everywhere to ignore the type error, so Sorbet provides it as a library function, called T.must:

```
T.must(thing).do_something
```

Sorbet knows that the call to T.must raises if thing is nil. To make this very concrete, here's a Sorbet playground where you can see this in action:

→ View on sorbet.run

You can read more about T.must in the [Sorbet documentation](#).

Problem

While T.must works, it is not ideal for a couple reasons:

1. It leads to a weird outward spiral of flow control, which disrupts method chains:

```
#           T.must(T.must(task).mailing_params).fetch('template_context')
#               #         #
# ▼         ▼
# compare that control flow with this:
#
```

03/16/2022 1/8
2. It is not a method, so you can't map it over a list using Symbol#to_proc. Instead, you have to expand the block:

```ruby
array_of_integers = array_of_nilable_integers.map { |x| T.must(x) }
```

Compare that with this:

```ruby
array_of_integers = array_of_nilable_integers.map(&:must!)
```

3. It is in a Sorbet-specific gem. We do not intend for Sorbet to be the only type checker. It would be nice to have such a method in the Ruby standard library so that it can be shared by all type checkers.

4. This method can make Ruby codebases that don't use type checkers more robust! Kernel#must! could be an easy way to assert invariants early. Failing early makes it more likely that a test will fail, rather than getting TypeError's and NoMethodError's in production. This makes all Ruby code better, not just the Ruby code using types.

**Proposal**

We should extend the Ruby standard library with something like this:

```ruby
module Kernel
  def must!; self; end
end
class NilClass
  def must!
    raise TypeError.new("nil.must!")
  end
end
```

These methods would get type annotations that look like this:

(using Sorbet's RBI syntax, because I don't know RBS well yet)

```ruby
module Kernel
  sig {returns(T.self_type)}
  def must!; end
end
class NilClass
  sig {returns(T.noreturn)}
  def must!; end
end
```

What these annotations say:

- In Kernel#must!, the return value is T.self_type, or "whatever the type of the receiver was." That means that 0.must! will have type Integer, "".must! will have type String, etc.

- In NilClass#must!, there is an override of Kernel#must! with return type T.noreturn. This is a fancy type that says "this code either infinitely loops or raises an exception." This is the name for Sorbet's bottom type, if you are familiar with that terminology.

Here is a Sorbet example where you can see how these annotations behave:

→ View on sorbet.run

**Alternatives considered**

There was some discussion of this feature at the Feb 2020 Ruby Types discussion:
Summarizing:

- Sorbet team frequently recommends people to use `xs.fetch(0)` instead of `T.must(xs[0])` on Array's and Hash's because it chains and reads better. `.fetch` not available on other classes.
- It's intentional that `T.must` requires as many characters as it does. Making it slightly annoying to type encourages developers to refactor their code so that nil never occurs.
- There was a proposal to introduce new syntax like `thing.!!`. This is currently a syntax error.

**Rebuttal:** There is burden to introducing new syntax. Tools like Rubocop, Sorbet, and syntax highlighting plugins have to be updated. Also: it is hard to search for on Google (as a new Ruby developer). Also: it is very short—having something slightly shorter makes people think about whether they want to type it out instead of changing the code so that nil can't occur.

Another alternative would be to dismiss this as "not useful / common enough". I don't think that's true. Here are some statistics from Stripe's Ruby monolith (~10 million lines of code):

<table>
<thead>
<tr>
<th>method</th>
<th>percentage of files mentioning method</th>
<th>number of occurrences of method</th>
</tr>
</thead>
<tbody>
<tr>
<td>.nil?</td>
<td>16.69%</td>
<td>31340</td>
</tr>
<tr>
<td>T.must</td>
<td>23.89%</td>
<td>74742</td>
</tr>
</tbody>
</table>

From this, we see that

- T.must is in 1.43x more files than .nil?
- T.must occurs 2.38x more often than .nil?

**Naming**

I prefer `must!` because it is what the method in Sorbet is already called.

I am open to naming suggestions. Please provide reasoning.

**Discussion**

In the above example, I used `T.must` twice. An alternative way to have written that would have been using save navigation:

```
T.must(task&.mailing_params).fetch('template_context')
```

This works as well. The proposed `.must!` method works just as well when chaining methods with safe navigation:

```
task&.mailing_params.must!.fetch('template_context')
```

However, there is still merit in using `T.must` (or `.must!`) twice—it calls out that the programmer intended neither location to be nil. In fact, if this method had been chained across multiple lines, the backtrace would include line numbers saying specifically `which .must! failed`:

```
task.must!
  .mailing_params.must!
    .fetch('template_context')
```

**History**

#1 - 11/14/2020 11:18 PM - jez (Jake Zimmerman)
- Description updated

#2 - 11/16/2020 12:31 AM - shyouhei (Shyouhei Urabe)
- Yes I think we need this feature (except the name)
• Re: naming, the rule of a bang method is:

    A bang method cannot exist alone. If there is #must!, there shall also be #must which is a "safer" variant of it.

However I can hardly think of a safer behaviour than what is proposed.

#3 - 11/16/2020 06:25 AM - nrodriguez (Nicolas Rodriguez)
There is not_nil! in Crystal https://github.com/crystal-lang/crystal/blob/master/src/object.cr#L223

#4 - 11/16/2020 07:33 AM - mame (Yusuke Endoh)
I like this feature as a tiny tool for debugging and assertion.

When I encounter undefined method 'foo' for nil:NilClass on x.foo, I insert Kernel#p to many places, such as x = expr to x = p(expr), to try identifying where the nil comes from. It would be good to have alternative way like x = (expr).must!.

And I sometimes add an assertion to except nil when I wrote a complex application:

```ruby
class Foo
  def initialize(arg)
    raise unless arg
    @arg = arg
  end
end

@arg = arg.must! is useful in this case.
```

I understand Austin's opinion. I agree that type annotations like must! is definitely ugly, and in fact, you don't have to insert tons of must! to your gem if you don't use the type checkers. But I think this feature itself is still useful for those who are not interested in static type checking.

The name issue is very very difficult, though... As shyouhei said, must! and not-nil! have no safer variants that I could come up with. Thus I like a new syntax for it, but I agree that a new syntax is painful.

#5 - 11/16/2020 12:25 PM - ufuk (Ufuk Kayserililoglu)
Given that Ruby 3 will ship with RBS which will be supplanted with a type-checker based on RBS (Steep) soon, I expect more people to experiment with types adoption in their codebases. Our early adoption of type checking at Shopify has shown that type annotations are not enough to precisely define types in some cases, and inline type assertions are necessary to further make developer intent explicit.

For that reason, we, at Shopify, would love for this addition to be considered for Ruby 3. Kernel#must! would make inline type assertions much much more readable and we would love to switch to using them instead of T.must. As we noted above, it would also allow Rubyists to experiment with type checking in a more concrete manner.

As for the matching must method, I think it could be an alias to Kernel#tap which allows the user to introspect the value and decide what to do with it, as opposed to must! which introspects and always raises on nil.

#6 - 11/16/2020 12:53 PM - sawa (Tsuyoshi Sawada)
Since this feature is reminiscent of the safe navigation operator &., I think it would be good if we can make the notation similar to the latter notation or use some variant of it. I suggest to make the following two changes to the current syntax.

1. Introduce a |. operator, which works complementarily to the &. operator, that is, it executes the next method call if the receiver is nil, and skips the next method call otherwise.

   ```ruby
   l|.inspect.*(2) # => 2
   nil|.inspect.*(2) # => "nilnil"
   ```

2. Let raise be a public method.

Then we can do:

```ruby
task|.raise.mailing_params|.raise.fetch('template_context')
```

Using such syntax, we can specify the details of the error to be raised, or even use a custom error handler:

```ruby
task|.log("Task is nil at line #(__LINE__)")
 .mailing_params|.handle_error("There may be a bug in mailing_params method")
 .fetch('template_context')
```

#7 - 11/17/2020 06:35 AM - nrodriguez (Nicolas Rodriguez)
I like this syntax:
```
task.raise.mailing_params.raise.fetch('template_context')
```
As you said it's consistent with &.

#8 - 11/17/2020 03:20 PM - Dan0042 (Daniel DeLorme)
I don't think I can agree with the | operator, but having the raise in a block would make a lot of sense to me.
```
task.or{raise}.mailing_params.or{raise}.fetch('template_context')
```
This could also be used to return a default value instead of raising.
The naming remains tricky though. One might expect or to operate on falsy values rather than just nil. Same thing with else. Is there another good name for this?

#9 - 11/17/2020 03:44 PM - zverok (Victor Shepelev)
Just an aside note: not saying something against, or for the proposal, I can't help noticing that abandoned "method reference" idea solved at least (1) and (2) of original ticket, removing some of the necessity of constantly extending core objects:
```
# 1. "Inverted flow":
T.must(T.must(task).mailing_params).fetch('template_context')
# "Forward flow" with then and &method
task.then(&T.method(:must)).mailing_params.then(&T.method(:must)).fetch('template_context')
# ...with abandoned method reference:
T.task.then(T.must).mailing_params.then(T.must).fetch('template_context')

# 2. "Blocks"
array_of_integers = array_of_nilable_integers.map{|x| T.must(x) }
array_of_integers = array_of_nilable_integers.map&T.must
```
(In fact, maybe it is just me, but having it as an external "assertions library" used the way I show, seems more preferrable for me than Object.must)

#10 - 11/17/2020 07:46 PM - jez (Jake Zimmerman)
I really don't want to add new syntax for something that can already be expressed in normal Ruby code.

I wanted to bump this suggestion from Ufuk:
```
uufuk (Ufuk Kayserilioglu) wrote in #note-5:

As for the matching must method, I think it could be an alias to Kernel#tap which allows the user to introspect the value and decide what to do with it, as opposed to must! which always raises on nil.
```
I like this idea.

Another idea: we could make Kernel#must take a block that runs only if self is nil. Like this:
```
module Kernel
  def must(&blk); self; end
  def must!; self; end
end
class NilClass
  def must(&blk)
    yield
  end
  def must!
    raise TypeError.new("nil.must!")
  end
end

nil.must { 'default' }.size #=> 7
'x'.must { 'default' }.size #=> 1

nil.must! # => TypeError
'x'.must! # => 'x'
```
I want to re-iterate: I very much do not want to introduce new syntax. Syntax changes are much more work for downstream Ruby tooling like Sorbet and the parser gem. Syntax changes are almost always looked at with confusion by one half of the Ruby community or another. Etc.

Standard library changes on the other hand are very easy to support by downstream tooling, can be monkey patched into projects stuck on old Ruby
versions, etc.

I am fine choosing another name, like not Nil! something else. But for what it's worth there are tens of thousands of call sites to T.must in Stripe's codebase, and I think people generally agree on that name and what it means. Also using the same name for T.must and this method will make it easy for people to move between codebases using Sorbet and not using Sorbet, without having to learn a new word.

#11 - 11/17/2020 08:50 PM - jeremyevans0 (Jeremy Evans)

This method is trivial to write in Ruby. While it can be useful even in codebases that do not use static typing, I think the benefit of adding it is smaller than the cost of adding another method to Kernel. I think it should probably be in a separate gem, or an optional part of Sorbet. Doing so makes even easier to support in older Ruby versions, and doesn't break backwards compatibility for users that are currently using the method name.

If we do consider this feature for inclusion in Kernel, I think that outside of Stripe/Sorbet users (a small fraction of Ruby programmers), .must! meaning raise if nil? is not intuitive. I think a better name is needed that would be more intuitive to the average Ruby programmer if we do want this method in Kernel.

#12 - 11/17/2020 09:12 PM - jez (Jake Zimmerman)

jeremyevans0 (Jeremy Evans) wrote in #note-12:

the benefit of adding it is smaller than the cost of adding another method to Kernel

Can you speak more on the cost of adding a method to Kernel? While I understand the costs of something new syntax would bring, I am less familiar with the cost to adding something to the standard library.

The benefit to having something in the standard library instead of a gem is specifically to make it a common idiom shared across typed and non-typed Ruby codebases. T.must already exists in sorbet-runtime for the people who want to only use Sorbet. I am of the belief that third party gems should refrain from monkey patching new methods into standard library classes, which is why I proposed a change to the standard library directly.

I also want to reiterate: in Stripe's codebase, T.must is more than twice as frequent than .nil?, and no one contends that .nil? doesn't belong in the standard library.

jeremyevans0 (Jeremy Evans) wrote in #note-12:

I think that outside of Stripe/Sorbet users (a small fraction of Ruby programmers), .must! meaning raise if nil? is not intuitive.

Again, I'm 100% fine to change the name to make it more intuitive. .not_nil, .not_nil!, .unwrap_nil, .drop_nil, etc. If the only blocker is the name, that is great: I will agree to any name. I am very open to further suggestions of what the name should be.

#13 - 11/17/2020 09:46 PM - jeremyevans0 (Jeremy Evans)

jez (Jake Zimmerman) wrote in #note-12:

jeremyevans0 (Jeremy Evans) wrote in #note-12:

the benefit of adding it is smaller than the cost of adding another method to Kernel

Can you speak more on the cost of adding a method to Kernel? While I understand the costs of something new syntax would bring, I am less familiar with the cost to adding something to the standard library.

The benefit to having something in the standard library instead of a gem is specifically to make it a common idiom shared across typed and non-typed Ruby codebases. T.must already exists in sorbet-runtime for the people who want to only use Sorbet. I am of the belief that third party gems should refrain from monkey patching new methods into standard library classes, which is why I proposed a change to the standard library directly.

I understand that advantage. However, for non-typed Ruby, the advantage is minimal in my opinion, and not enough to clear the high bar. I understand for Sorbet users, the advantage is quite large, but I don't think we should add a method to Kernel unless there is a significant advantage even for non-Sorbet users, considering that Sorbet users can easily add such a method themselves.

I agree with you that third party gems should refrain from monkey patching by default, unless that is the sole purpose of the gem. Which is why I said that making this an optional part of Sorbet (e.g. require 'sorbet/core_ext/must') or an external gem may be best. Looks like the gem name must is already taken. The must gem adds Object#must, but not Object#must!
Jeremy Evans wrote in #note-14:

But the fact that there should be a high bar is something I would hope we all understand.

I hope that something I wrote didn't make you think I disagree there! I was just genuinely curious to have the costs stated explicitly, just in case I was missing something (e.g., you did not mention runtime performance).

Another idea which would work equally well for the purposes of type checking, but which has more wide appeal:

```ruby
module Kernel
  def or_else(&blk); self; end
end

class NilClass
  def or_else(&blk)
    if block_given?
      yield
    else
      raise TypeError.new("nil.must!")
    end
  end
end

nil.or_else {'default'}.size #=> 7
'x'.or_else {'default'}.size #=> 1

nil.or_else        #=> TypeError
'x'.or_else        #=> 'x'

nil.or_else {raise "Custom error"}.size #=> TypeError
'x'.or_else {raise "Custom error"}.size #=> 1
```

Maybe this name is more intuitive to people writing Ruby, and would more obviously have value in typed and untyped codebases alike?

This was what Dan0042 wrote in #note-9 above, but I have made it explicit what the behavior would be with and without a block.

**#15 - 11/18/2020 12:06 AM - jez (Jake Zimmerman)**

A colleague pointed out that `or_else` has the nice property that it could replace the `||=` for default initializing instance variables:

```ruby
@foo ||= compute_initial_value_slow_returns_true_or_false(...)  #^ this logic will run every time, because the `||=` sees the 'false' and executes the slow method every time.
# compare:
@foo = @foo.or_else { compute_initial_value_slow_returns_true_or_false(...) }  #^ this runs the slow computation only once, even if the first run returns 'false'.
```

(which I hope is one more reason to suspect that this has value outside of typed codebases)

**#16 - 11/18/2020 12:32 AM - phluid61 (Matthew Kerwin)**

Jez (Jake Zimmerman) wrote in #note-16:

A colleague pointed out that `or_else` has the nice property that it could replace the `||=` for default initializing instance variables:

```ruby
@foo ||= compute_initial_value_slow_returns_true_or_false(...)  #^ this logic will run every time, because the `||=` sees the 'false' and executes the slow method every time.
# compare:
@foo = @foo.or_else { compute_initial_value_slow_returns_true_or_false(...) }  #^ this runs the slow computation only once, even if the first run returns 'false'.
```

Except that `@foo ||= x` is `@foo ||= @foo = x`), but your `or_else` example always does the redundant assignment. You probably want `@foo.or_else { @foo = x }` which is really hard to grok.

Personally I think these are more readable, and the second one can be generalised to the case where there's also a valid nil
Incidentally: [https://phluid61.github.io/mug/#gem-and-or](https://phluid61.github.io/mug/#gem-and-or)

data_store.get_value.or(default_value).do_something

The differences are that I used falsey (not just nil) tests, and my or_then returns the object rather than the result of the block.

#17 - 11/26/2020 05:14 AM - matz (Yukihiro Matsumoto)
I strongly oppose the name must. must assumes coercing something but no relation to types nor nil. With a different name, there may be a chance for the method.

I think it's too late for 3.0. Maybe 3.1.

Matz.

#18 - 11/30/2020 03:26 PM - Dan0042 (Daniel DeLorme)
I'm also not fond of the name must, so I'm relieved that Matz is of the same opinion.

In the end I think that notnil is the best option. It's clear. It's explicit. It's short. There's no rule that says an error-raising method should have a bang (cf. Hash#fetch)

I prefer notnil to not_nil just based on gut feeling, because looking at that code feels more like the special shortcircuit operation that it is, whereas not_nil feels more like a regular method that any class might define. Maybe also because notnil reads a bit like nonzero?. And chaining methods after a bang looks weird.

    task.notnil.mailing_params.notnil.fetch('template_context')
    task.notnil.mailing_params.notnil.fetch('template_context')
    task.notnil.mailing_params.notnil.fetch('template_context')
    task.notnil(DEFAULT_TASK).mailing_params.notnil{raise "other message"}.fetch('template_context')