Value Object is a useful concept, introduced by Martin Fowler (his post, Wikipedia Entry) with the following properties (simplifying the idea):

- representing some relatively simple data;
- immutable;
- compared by type & value;
- nicely represented.

Value objects are super-useful especially for defining APIs, their input/return values. Recently, there were some movement towards using more immutability-friendly approach in Ruby programming, leading to creating several discussions/libraries with value objects. For example, Tom Dalling's gem, Good Ruby Value object convention (disclaimer: the latter is maintained by yours truly).

I propose to introduce native value objects to Ruby as a core class.

Why not a gem?

- I believe that concept is that simple, that nobody will even try to use a gem for representing it with, unless the framework/library used already provides one.
- Potentially, a lot of standard library (and probably even core) APIs could benefit from the concept.

Why Struct is not enough

Core Struct class is "somewhat alike" value-object, and frequently used instead of one: it is compared by value and consists of simple attributes. On the other hand, Struct is:

- mutable;
- collection-alike (defines to_a and is Enumerable);
- dictionary-alike (has [] and .values methods).

The above traits somehow erodes the semantics, making code less clear, especially when duck-typing is used.

For example, this code snippet shows why to_a is problematic:

```ruby
Result = Struct.new(:success, :content)

# Now, imagine that other code assumes 'data' could be either Result, or [Result, Result, Result]
# So, ...

data = Result.new(true, 'it is awesome')

Array(data) # => expected [Result(true, 'it is awesome')], got [true, 'it is awesome']

# or...
def foo(arg1, arg2 = nil)
p arg1, arg2
end

foo(*data) # => expected [Result(true, 'it is awesome'), nil], got [true, 'it is awesome']
```

Having [] and each defined on something that is thought as "just value" can also lead to subtle bugs, when some method checks "if the received argument is collection-alike", and value object's author doesn't thought of it as a collection.

Concrete proposal
Class name: Struct::Value: lot of Rubyists are used to have Struct as a quick "something-like-value" drop-in, so alternative, more strict implementation, being part of Struct API, will be quite discoverable; alternative: just Value

Class API is copying Structs one (most of the time -- even reuses the implementation), with the following exceptions (note: the immutability is not the only difference):
- Not Enumerable;
- Immutable;
- Doesn't think of itself as "almost hash" (doesn't have to_a, values and [] methods);
- Can have empty members list (fun fact: Struct.new('Foo') creating member-less Struct::Foo, is allowed, but Struct.new() is not) to allow usage patterns like:

```ruby
class MyService
  Success = Struct::Value.new(:results)
  NotFound = Struct::Value.new
end
```

NotFound here, unlike, say, Object.new.freeze (another pattern for creating "empty typed value object"), has nice inspect #<value NotFound>, and created consistently with the Success, making the code more readable. And if it will evolve to have some attributes, the code change would be easy.

**Patch is provided**

[Sample rendered RDoc documentation](https://zverok.github.io/ruby-rdoc/Struct-Value.html)

**Related issues:**

<table>
<thead>
<tr>
<th>Related to Ruby master - Feature #16769: Struct.new(..., immutable: true)</th>
<th>Rejected</th>
</tr>
</thead>
</table>

**History**

**#1 - 08/24/2019 06:55 AM - Eregon (Benoit Daloze)**

This sounds interesting to me.
What would a simple implementation of Struct::Value.new look like in Ruby code?
I'm not quite sure what the available API is since it's all described as Struct - some methods.

**#2 - 08/24/2019 10:17 AM - zverok (Victor Shepelev)**

[@Eregon (Benoit Daloze), here is rendered version of class' docs:](https://zverok.github.io/ruby-rdoc/Struct-Value.html)

Basically, it is what is said on the tin: like Struct, just leaner.

**#3 - 08/24/2019 10:37 AM - zverok (Victor Shepelev)**

What would a simple implementation of Struct::Value.new look like in Ruby code?

Oh, I've probably answered the wrong question... But I am not quite sure I understand yours.

Theoretically, it is just something like this (ignoring the fact that Structs implementation has optimized storage and other tricks, and any input validation):

```ruby
class Struct::Value
  def self.new(*args, keyword_init: false)
    name, *members = args.first.is_a?(String) ? args : [nil, *args]
    Class.new(self) do
      @members = members

      def self.new(*args)
        allocate.tap do |o|
          __send__(:initialize, *args)
        end

        members.each { |m| define_method(m) { instance_variable_get("#{@(m)}") } }
        end.tap { |cls| const_set(name, cls) if name }
      end
    end
  end
end
```

So, (if that's what you've asking) it produces object of different class, Struct::Value, unrelated to Struct, but sharing most of the implementation.

**#4 - 08/29/2019 08:01 AM - matz (Yukihiro Matsumoto)**

- Status changed from Open to Feedback
The typical solution is Struct.new(...).freeze. This doesn't require any enhancement. The other option is Struct.new(..., immutable: false). It looks simpler than the proposed Struct::Value.

Matz.

#5 - 08/29/2019 09:35 AM - zverok (Victor Shepelev)
@matz (Yukihiro Matsumoto) Sorry for not sharing more detailed reasoning which led to the current proposal (I explained the "final reasons" in its text, but it is too terse).

So, it went as following:

1. First, I really wanted just Struct.new(..., immutable: false) (and even experimented for some time with a private monkey-patch, doing just that)
2. But in fact, to be a proper convenient "value object", it is also bad for container to mimic Enumerable, and especially bad to implement to_a. Simple example:

```ruby
Result = Struct.new(:success, :content)
# Now, imagine that other code assumes 'data' could be either Result, or [Result, Result, Result]
# So, ...
data = Result.new(true, 'it is awesome')
Array(data) #=> expected [Result(true, 'it is awesome')], got [true, 'it is awesome']
# or...
def foo(arg1, arg2 = nil)
  p arg1, arg2
end
foo('data') #=> expected [Result(true, 'it is awesome'), nil], got [true, 'it is awesome']
```

3. And generally, some random value object "duck typing" itself as a collection seems not really appropriate.
4. The same, I believe, is related to supporting [:foo] and ['foo'] accessors: convenient for "general content object" that Struct is, but for "just value" it could seem an unnecessary expansion of the interface.
5. Finally, empty-member Value is allowed, while empty-member Struct somehow does not (I don't know if it is by design or just a bug, as I am mentioning above, Struct.new('Name') IS allowed, but Struct.new is NOT).

So, considering all the points above, it could be either multiple settings: immutable: true, enumerable: false, hash_accessors: false (the (5) probably could be just fixed for Struct, too) -- which is not that convenient if you are defining 3-5 types in a row, and requires some cognitive efforts both from writer (errrm, what options did I used last time to set it as a "good" value object?) and reader (ugh, what's this struct with so many settings?)

So I eventually decided to propose going another way.

#6 - 08/29/2019 12:50 PM - Dan0042 (Daniel DeLorme)
If I understand correctly, the idea is to have X=Struct::Value.new(:x,:y,:z) which is strictly equivalent to

```ruby
class X
  def initialize(x=nil, y=nil, z=nil)
    @x, @y, @z = x, y, z
  end
  attr_reader :x, :y, :z
  #and other methods based on x,y,z attributes:
  #def ==(other)
  #def eql?(other)
  #def hash
end
```

Or was there some nuance I didn't catch?

#7 - 08/29/2019 12:55 PM - zverok (Victor Shepelev)
@Dan0042 (Daniel DeLorme) you are (probably) missing #inspect, #==, #eql?, #hash, #to_h and a bunch of other methods that are pretty trivial, but also important for the "value object".

#8 - 08/29/2019 02:14 PM - mame (Yusuke Endoh)
I couldn't understand what is "value object", and I found: https://martinfowler.com/bliki/ValueObject.html
Please do not assume that everybody knows such an essay :)

No one pointed out the article during the developer's meeting, so we cannot understand what you want.
I have some comments:

- Why don't you start it with a gem? It may be useful for your case, but I'm not sure if it is useful for many people so that it deserves a built-in feature. And the design of Struct::Value is not clear to me (e.g., non-Enumerable is trade off; is it really useful for many cases?). If your gem become so popular, we can import it as a built-in feature.
- The behavior of Struct::Value is too different from Struct. Another class name (like ValueClass or NamedTuple or what not) looks more suitable.
- What you (first) want is called “structural equality” in other languages (OCaml, F#, C#, TypeScript, Kotlin, as far as I know). Also it resembles “namedtuple” in Python. You may want to study them.

BTW, I understand the motivation of the proposal. I want “structural equality” in Ruby. Personally, I often write:

```ruby
class Point3D
  include StructuralEquality
  def initialize(x, y, z)
    @x, @y, @z = x, y, z
  end
end

foo1 = Point3D.new(1, 2, 3)
foo2 = Point3D.new(1, 2, 3)
p foo1 == foo2 #=> true
h = { foo1 => "ok" }
p h[foo2] #=> "ok"
```

(The definition of StructuralEquality is here: https://github.com/mame/ruby-type-profiler/blob/436a10787fc74db47a8b2e9db995aa6ef7c16311/lib/type-profiler/utils.rb#L8-L31)

But, I'm unsure if it deserves a built-in feature.

@zverok (Victor Shepelev) I understand your concerns. I'll update the description today or tomorrow to include all the terminology and detailed rationale behind the proposal.

@zverok (Victor Shepelev) Of course, there are several good gems with more-or-less similar functionality. But, from the hard experience, large codebases tend to look with great caution on the “small utility” gems to avoid dependency bloat and tend to depend only on large non-trivial functionality. But if it is a part of the language core, it is beneficial for everyone.

@Dan0042 (Daniel DeLorme) Of course, there are several good gems with more-or-less similar functionality. But, from the hard experience, large codebases tend to look with great caution on the “small utility” gems to avoid dependency bloat and tend to depend only on large non-trivial functionality. But if it is a part of the language core, it is beneficial for everyone.

@Dan0042 (Daniel DeLorme) Question: you say "Doesn't think of itself as almost hash" but at the same time you say it should have to_h. Isn't that a contradiction? What exactly are you looking for?

Naming suggestion: BasicStruct (in parallel to Object and BasicObject)

@zverok (Victor Shepelev) Question: you say "Doesn't think of itself as almost hash" but at the same time you say it should have to_h. Isn't that a contradiction? But "mimicking" some of the Hash API (with [] and values and values_at) makes the object responsibility less focused.
Ok I see what you meant. BTW Struct#values_at follows the Array rather than Hash API, because Struct also thinks of itself as a tuple :-

```ruby
Struct.new(:x).new(42).values_at(0)  #=> [42]
Struct.new(:x).new(42).values_at(:x) #=> TypeError
```

zverok (Victor Shepelev) wrote:

**Why not a gem?**

- I believe that concept is that simple, that nobody will even try to use a gem for representing it with, unless the framework/library used already provides one.

If a concept is popular and there is a well-designed gem that implements it then people use it. For example, a lot of people use dry-initializer, which is also dead-simple and provides the functionality that could easily be implemented from scratch (and even could be useful as a part of the standard library).

If there is still no such a gem then there is no enough demand for the feature itself.

So, why pushing it to the core?

@palkan (Vladimir Dementyev) I have a strong feeling of "value object notion should be a part of the language, not an externally implemented optional thingy", but it is not easy to rationalize it.

Maybe the thing is that "value object" is a notion most useful at API borders (and it is not just utility usability, but conceptual one, "our API accepts this and that type of value objects and return this and that type of them"). And I believe "this is a concept of the language" makes a huge difference in using, documenting and explaining your APIs, compared to "well, we use that external gem, developed by some random dude, to bloat our dependencies, because it is tinsy bit more convenient."

In other words, I am proposing to introduce the concept, not implementation.

So, considering all the points above, it could be either multiple settings: immutable: true, enumerable: false, hash_accessors: false

I think that's a great idea. That way it's possible for everyone to mix and match the behavior they want in their structs. For example let say I want a struct to be mutable but not enumerable (because of the Array(mystruct) bug shown above), the Struct::Value approach doesn't work. If you find yourself always repeating the same options, it's trivial to write your own ValueStruct helper function.

Or maybe Struct could include a few built-in helpers

- Struct::Value => immutable: true, enumerable: false, hash_accessors: false
- Struct::Basic => immutable: false, enumerable: false, hash_accessors: false
- Struct::Tuple => immutable: false, enumerable: true, hash_accessors: false

I like the idea of helpers in https://bugs.ruby-lang.org/issues/16122#note-18. We need to discuss further the combination of attributes (immutable, enumerable, etc.).

Matz.

- Related to Feature #16769: Struct.new(..., immutable: true) added

We already have Struct.new(..., keyword_init: true). I think having other variants like immutable: true, enumerable: false, hash_accessors: false is consistent and flexible.

Having only the helpers like Struct.Value would restrict to a few combinations, and still need to handle keyword_init:

I think Struct::Value.new could be a nice helper for immutable: true, enumerable: false, hash_accessors: false. The others seem more specific, less common to use, and I would rather let people choose the configuration they want with keyword arguments for Struct.new().
Implementation-wise and conceptually, I think it's also nicer if Struct::Value.new(...) is implemented as as Struct.new(..., immutable: true, enumerable: false, hash_accessors: false).

#22 - 04/10/2020 12:13 PM - Eregon (Benoit Daloze)
In my view, Struct.new is the perfect example to generate a custom class in Ruby.
I think making it customizable with new keyword arguments is both elegant and simple.
OTOH I think having N "subclasses" with different behaviors invites to confusion about what differs between them and enforces duplication in implementation code.

#23 - 01/11/2022 07:29 AM - ko1 (Koichi Sasada)
I don't use Enumerable features of Struct classes, but I don't have any trouble by having Enumerable.
Why do you want to remove Enumerable features?
I can not find any benefits.

#24 - 01/11/2022 07:43 AM - zverok (Victor Shepelev)
@ko1 (Koichi Sasada), the initial ticket provides some explanations:

For example, this code snippet shows why to_a is problematic:

```
Result = Struct.new(:success, :content)

# Now, imagine that other code assumes 'data' could be either Result, or [Result, Result, Result]
# So, ...

data = Result.new(true, 'it is awesome')

Array[data] # => expected [Result(true, 'it is awesome')], got [true, 'it is awesome']

# or...
def foo(arg1, arg2 = nil)
  p arg1, arg2
end

foo(*data) # => expected [Result(true, 'it is awesome'), nil], got [true, 'it is awesome']
```

That's about just #to_a method, but I think that in general, considering duck typing, it is undesirable that the object that the developer thinks of as an "atomic" will be duck-typed as a collection (#respond_to?(:each)). In general, you never know when "is it one thing, or is it an enumeration of things" will be crucial in code, and I think it is important to underline Struct::Value is one thing.

I believe there are good reasons why #each was removed from String, for example.

#25 - 01/11/2022 07:47 AM - ko1 (Koichi Sasada)
zverok (Victor Shepelev) wrote in #note-24:

@ko1 (Koichi Sasada), the initial ticket provides some explanations:

Sorry I found it just after commented.
It seems not related to "immutability".

#26 - 01/11/2022 07:52 AM - zverok (Victor Shepelev)
@ko1 (Koichi Sasada)

It seems not related to "immutability".

Yes, I covered this, too (I know it is a large wall of text, sorry!), in Concrete proposal section:

```
Class API is copying Structs one (most of the time -- even reuses the implementation), with the following exceptions (note: the immutability is not the only difference)
```

#27 - 01/11/2022 04:16 PM - Dan0042 (Daniel DeLorme)
matz (Yukihiro Matsumoto) wrote in #note-19:
I like the idea of helpers in https://bugs.ruby-lang.org/issues/16122#note-18. We need to discuss further the combination of attributes (immutable, enumerable, etc.)

Having helpers would definitely provide a nice easy experience. But since the important thing here is the optional settings, disagreement/bikeshed on the helpers should not prevent or delay a decision on the immutable/enumerable/hash_accessors settings. It should be ok to first decide on those settings, and in a second step decide on the helpers. After all once the settings are available, it's trivial for anyone to define their own helpers.

So regarding those helpers I was thinking of something like Struct::Value(x, y) but there's also the Struct::Value.new(x, y) syntax that simulates a subclass. Having a Value is the main topic of this ticket, but personally I'm more interested in Basic that behaves more like a simple C struct. It's easier to use and reason about if you don't have to worry about accidental conversion to array and auto-splatting bugs. I'm not particularly attached to Tuple but I thought it was a good name to make it explicit when we want a splattable struct where the ordering of the fields is important, like x,y,z = *tuple.

#28 - 01/29/2022 08:31 AM - mame (Yusuke Endoh)
Discussed on the dev-meeting.

@matz (Yukihiro Matsumoto) is now negative to allow settings. Having various semantics in one Struct class will bring confusion rather than usability. keyword_init settings will be no longer needed after Ruby 3.2. (See #16806 and c956f979e5d05900315d2753d5c3b1389aaf8dase4)

Instead, he seems positive to provide one strict version of Struct. His current preference is:

- Has: field reader methods, deconstruct_keys, deconstruct, ==, eql?, hash
- Does not have: field writer methods like writer=, each and Enumerable, to_a, each_pair, values, [], []=, dig, members, values_at, select, filter, size, to_h

But he couldn't seem to decide on a name. Struct::Value seems acceptable to him, but he wanted to seek a better name. Devs suggested Tuple, NamedTuple, and Record, but none of them seemed to fit for him.

#29 - 01/30/2022 12:53 PM - Eregon (Benoit Daloze)
ValueStruct or ImmutableStruct or FrozenStruct maybe?
ImmutableStruct would probably only make sense if values are made immutable too, which doesn't seem proposed here.

I think the nesting of Struct::Value feels a bit weird, especially with the existing behavior of Struct.new("Foo", :a) defining Struct::Foo. But not really against it.

#30 - 01/30/2022 08:39 PM - matheusrich (Matheus Richard)
Some more alternatives to get the ideas rolling: Unit and Item (might be paired with Struct)
I also like Box.

#31 - 01/30/2022 08:57 PM - Dan0042 (Daniel DeLorme)
Eregon (Benoit Daloze) wrote in #note-29:

ValueStruct or ImmutableStruct or FrozenStruct maybe?

Those are good ideas. Or to highlight the " pared-down" aspect of this strict version of Struct: SimpleStruct / PlainStruct / BasicStruct (parallel to Object vs BasicObject).

#32 - 01/30/2022 09:48 PM - myronmarston (Myron Marston)
I'm quite fond of this proposal--I basically never use Struct unless I specifically need mutability and have been using the values gem for years, which has a simple implementation of about 100 lines:

https://github.com/tcrayford/Values/blob/master/lib/values.rb

It offers a number of core features that I'd hope any stdlib equivalent would also provide:

- Instantiation via positional arguments (ValueClass.new(1, 2))
- Instantiation via keyword arguments (ValueClass.with(foo: 1, bar: 2))
- Ability to make a copy with one or more attributes updated: value.with(foo: 1)
- ==/eql?/hash defined for value-based equality semantics
- Readable to_s/inspect/pretty_print
- Easy conversion to a hash with to_h

Most engineers I've worked with have referred to this category of objects as "value objects" so I think "Value" in the name is good...but I don't care a whole lot about the name. Kotlin (another language I use) offers a similar feature and calls them data classes:

https://kotlinlang.org/docs/data-classes.html
If this is adopted, it'd also be great to see what stdlib types can be safely ported to build on this type--things like Date/Time/URI, etc. (It may of course be hard or impossible to port these to use the new feature while retaining backwards compatibility.)

#33 - 02/12/2022 09:54 PM - dsisnero (Dominic Sisneros)
+1 -
Also, is there plans to have a flag in C or a different shape so that the VM's can make this fast

#34 - 08/16/2022 03:18 PM - mame (Yusuke Endoh)
@nobu (Nobuyoshi Nakada) proposed Data, which used to be a class for extension library authors, but deprecated since ruby 2.5 and removed since 3.0. We might reuse it now.

Summarise the proposed name candidates:

- Struct::Value
- ImmutableStrudct
- FrozenStruct
- Unit
- Item
- Box
- SimpleStruct
- PlainStruct
- BasicStruct
- ValueClass (provided by values gem?)
- Value (provided by values gem)
- Data

#35 - 08/16/2022 03:31 PM - mame (Yusuke Endoh)
BTW, I personally wanted Struct to store the field values simply in instance variables rather than the hidden storage. For example:

```ruby
FooBar = Struct::Value.new(:foo, :bar)
class FooBar
  def foo_plus_bar
    # These bare "foo" and "bar" are not visually obvious
    # whether they are a method call or local variable access
    foo + bar

    # We can write it as follows, but it is a bit verbose
    self.foo + self.bar

    # If they are stored in instance variables,
    # it is obvious that they are field access
    @foo + @bar
  end
end
```

I know it is impossible to change Struct for compatibility reason, but if we introduce a new Struct-like class, I wonder if we can change this too?

#36 - 08/16/2022 03:54 PM - Eregon (Benoit Daloze)
mame (Yusuke Endoh) wrote in #note-35:

BTW, I personally wanted Struct to store the field values simply in instance variables rather than the hidden storage.
I know it is impossible to change Struct for compatibility reason, but if we introduce a new Struct-like class, I wonder if we can change this too?

FWIW, TruffleRuby used to use ivars for Struct but changed to "hidden ivars" for compatibility. Hidden ivars probably adds a bit more flexibility in the implementation but also means e.g. attr_reader can't be used directly to implement Struct::Value.
I'd think it'd be nice if we can share implementation code between Struct and Struct::Value, so it seems best to use the same representation from that pov.

A problem with ivars is Struct allows members which are not valid ivar names (IIRC), so ivars can't be used internally, or Kernel#instance_variables will not necessarily be all Struct::Value attributes.

#37 - 08/18/2022 06:38 AM - k0kubun (Takashi Kokubun)
My enthusiastic +1 for Data.
I've used Kotlin and its Data classes like @myronmarston (Myron Marston), and I feel calling it a Data class is somewhat accepted by the community. On the other hand, calling it Struct::Value feels like a workaround to avoid a conflict with existing names. I'm not sure if @zverok (Victor Shepelev) likes Data over his own proposal, but note that data appears in his local variable name as well.

#38 - 08/18/2022 06:59 AM - baweaver (Brandon Weaver)

k0kubun (Takashi Kokubun) wrote in #note-37:

My enthusiastic +1 for Data.

I've used Kotlin and its Data Classes like @myronmarston (Myron Marston), and I feel calling it a Data class is somewhat accepted by the community. On the other hand, calling it Struct::Value feels like a workaround to avoid a conflict with existing names. I'm not sure if @zverok (Victor Shepelev) likes Data over his own proposal, but note that data appears in his local variable name as well.

+1 as well. It's similar to the idea of Case Class in Scala as well, and I think the name Data is reasonable. Happy to see that Struct is looking to deprecate keyword_init in favor of accepting both styles as well, both are welcome changes.

These will be especially useful with pattern matching features

#39 - 08/18/2022 02:14 PM - zverok (Victor Shepelev)

I'm not sure if @zverok (Victor Shepelev) likes Data over his own proposal, but note that data appears in his local variable name as well.

It is OK, I think, save for some clumsiness when you try to speak in plurals (wrong "datas" or right-yet-not-obvious "datum").

I was never too sure about the name anyway.

If the rest is OK, I'll rebase my PR and update naming on the weekend.

#40 - 08/18/2022 05:05 PM - myronmarston (Myron Marston)

If “data” is the naming direction folks like, I think the class name should be DataClass. This aligns with kotlin (where data is a keyword before the class keyword) and reads better, IMO: DataClass.new gives you a new class whose purpose is to hold data. Data.new sounds like it gives you a new data which sounds weird.

#41 - 08/18/2022 05:59 PM - k0kubun (Takashi Kokubun)

I can live with DataClass too, but I still can't forget the beautiful shortness of Data. DataClass.new feels like you wanted to be so right that the name ended up being a bit verbose. To address that point, I thought of Data.class, which looks cool, but I guess such an interface doesn't exist in Ruby yet and DataClass.new is more "correct" in the OOP world.

#42 - 08/19/2022 08:23 AM - mame (Yusuke Endoh)

At the dev meeting, @matz (Yukihiro Matsumoto) rejected all name candidates except Struct::Value and Data.

- He wants to avoid the names already used by gems: ImmutableStruct, ValueClass, Value
- Short common nouns would be conflicting: Unit, Item, Box
- The main purpose of the new class is immutability, not "frozen", not "plain", not simplicity: FrozenStruct, SimpleStruct, PlainStruct
- He doesn't plan to make the old Struct inherit from the new one, so BasicStruct like BasicObject is not suitable

Incidentally, my proposal in #note-35 was rejected because an instance variable is weak against a typo. (A misspelled reader method raises NameError, but a misspelled instance variable returns nil implicitly.)

#43 - 08/19/2022 08:30 AM - mame (Yusuke Endoh)

This is my personal opinion. I think Data is a good choice since there are few compatibility issues at this time despite the short and simple name. If we were to use a slightly different word for this, such as DataClass, I don't see much point in choosing this word.

#44 - 08/19/2022 10:13 AM - zverok (Victor Shepelev)

At the dev meeting, @matz (Yukihiro Matsumoto) rejected all name candidates except Struct::Value and Data.

So, as far as I can understand, we only should choose one of two now, right?
I like Struct::Value slightly more, but not to the point of spending one more year discussing it :)

Let's stick with Data then, and I prepare the final PR.

#45 - 08/19/2022 10:19 AM - matz (Yukihiro Matsumoto)
I am not 100% satisfied with any of the candidates, but Struct::Value and Data are better than others. Struct::Value can cause conflict when someone is using Struct.new("Value", :foo, :bar) (this old-style code creates Struct::Value class). Data is a little ambiguous, but probably we can get used.

Matz.

#46 - 08/19/2022 04:24 PM - zverok (Victor Shepelev)

Umm wait.

Data is actually a plural form. While using it as singular is acceptable in modern English, in this case we don't have a plural for it.

I believe it will be a problem while writing docs, tutorials and discussing things. "Let's define a data here. Now, let's define some more ???

#47 - 08/19/2022 07:09 PM - Kokubun (Takashi Kokubun)

It consists of multiple members, so calling it data itself doesn't seem like a problem to me. For documentation, you could say a data class or data classes.

#48 - 08/20/2022 10:31 AM - Eregon (Benoit Daloze)

The main purpose of the new class is immutability, not "frozen", not "plain", not simplicity: FrozenStruct, SimpleStruct, PlainStruct

Immutable AFAIK means "deeply frozen", while frozen means "shallow frozen" (= Kernel#freeze/frozen?). This new struct-like class will not ensure every value is immutable, so it won't guarantee the Struct::Value instance is immutable. So in terms of documentation and semantics the new class will create frozen but not immutable instances.

Regarding the name, I like Data as well. If we want to avoid the confusion of Data.new returning a class and not an instance of Data, we could have another name for "create a Data subclass with these fields", maybe Data.for(:a, :b) or Data.new_class(a, :b)/Data.new_subclass(a, :b) or so.

I have seen many people being confused with Struct.new returning a subclass, so I think it is something worth considering for the new struct-like class.

#49 - 08/25/2022 10:40 AM - Matz (Yukihiro Matsumoto)

We are going to implement Data class in the following spec:

- D = Data.def(:foo, :bar) to define a data class (subclass of Data)
- Data.new raises exception (unlike Struct)
- d = D.new(1,2) to create Data instance
- Or d = D.new(:foo:1, :foo:2)
- D.new(1) or D.new(1,2,3) raises ArgumentError
- D.new(:foo:1) or D.new(:foo:1,:bar:2,:baz:3) raises ArgumentError
- Instead of D.new(...) you may want to use D[...]

We need further discussion regarding the following:

- default value to initialize Data
- how to call initialize method for D class
- whether we will introduce Struct.def

Matz.

#50 - 08/25/2022 11:25 AM - zverok (Victor Shepelev)

@matz (Yukihiro Matsumoto) Thanks for the decisions!

A few questions:

1. I am a bit concerned about using def, which is strongly associated with defining methods. I wouldn't want it to be a blocker (it would be really cool to have Data by the 3.2), but can we consider our options here? From the top of my head, I can think of define (used in methods context too, but less strong association with def method), setup, create or generate.
2. "default value to initialize Data" I am not sure what do you mean by that, can you please elaborate?
3. "how to call initialize method for D class". What options do we have here? Is there a necessity for deviation from how other classes work?..
When we allow mixing positional and keyword initializers, implementing initialize may be a bit complex. But we may unify 2 kinds of initializers to keyword initializer in D#new. But it's implementation detail. Need to be discussed later.

```ruby
D = Data.def(:foo, :bar)
class D
  # if arguments to `new` is passed directly
  def initialize(*args,**kwd)
    # ... checking positional and keyword initializers
  end
  # instead
  def initialize(**kwd) # only takes keyword arguments
    # ... initialization become much simpler
  end
end
```

Oh, I forgot. During the discussion, someone came up with an idea of Anonymous Data, data instance without creating a subclass (kinda like (Named)Tuples in other languages). This is another topic.

Matz.

**#52 - 08/25/2022 08:02 PM - k0kubun (Takashi Kokubun)**
- Subject changed from Struct::Value: simple immutable value object to Data: simple immutable value object
- Status changed from Feedback to Assigned
- Assignee set to zverok (Victor Shepelev)

**#53 - 08/25/2022 10:48 PM - k0kubun (Takashi Kokubun)**

Data.new aside, Data.def and Data.define are my current top-2 preferences. I'd be happy with either of these at this point.

But thinking about this further, I might like Data.define slightly more. It's a trade-off with shortness, but Data.define sounds more natural and it's still short enough, thanks to Data being chosen instead of Struct::Value. Even if it were to be ported to Struct, Struct.define doesn't seem too long either.

**Files**

| struct_value.patch | 18.6 KB | 08/23/2019 | zverok (Victor Shepelev) |