You might want a Data object if...

Kelly Popko | 9 July 2025
https://www.linkedin.com/in/kelly-popko/

Hello! My name is Kelly Popko and I am here to talk about when you might want a Data object.

What is `Data` class?

Kelly Popko | 9 July 2025
https://www.linkedin.com/in/kelly-popko/

`Data` is a relatively new Ruby class, introduced in Ruby 3.2.

Use Data when the intent is to store immutable atomic values and when you want to define simple classes for value-alike objects.

So, how do we use it?

Syntax

Kelly Popko | 9 July 2025
https://www.linkedin.com/in/kelly-popko/

Define the Data Object

- Data.define
- Keyword arguments

```
Book = Data.define(:title, :author, :year)
```

Kelly Popko | 9 July 2025
https://www.linkedin.com/in/kelly-popko/

Define with Optional Block

```ruby
Book = Data.define(:title, :author, :year) do
  SUMMARY = '%<title>s was written by %<author>s in %<year>i.'

  def to_s
    SUMMARY % {title:, author:, year:}
  end
end
```

```ruby
hobbit = Book.new("The Hobbit", "J.R.R. Tolkien", 1937)
puts hobbit
# => nil
'The Hobbit was written by J.R.R. Tolkien in 1937.'
```

Instantiating `Book`

- Call `.new` or `[]` on the object
- Pass either keyword or positional arguments

```ruby
hobbit = Book.new(title: "The Hobbit", author: "J. R. R. Tolkien", year: 1937)

hobbit = Book[title: "The Hobbit", author: "J. R. R. Tolkien", year: 1937]

hobbit = Book.new("The Hobbit", "J.R.R. Tolkien", 1937)

hobbit = Book["The Hobbit", "J.R.R. Tolkien", 1937]

# => <data Book title="The Hobbit", author="J. R. R. Tolkien", year=1937>

hobbit.author
```

Data vs. Struct

Kelly Popko | 9 July 2025
https://www.linkedin.com/in/kelly-popko/

Data vs Struct

Data

```
House = Data.define(:rooms, :area, :floors)
ranch = House.new(rooms: 5, area: 1200, floors: 1)
# => #<data House rooms=5, area=1200, floors=1>
```

Struct

```
House = Struct.new(:rooms, :area, :floors)
ranch = House.new(rooms: 5, area: 1200, floors: 1)
# => #<struct House rooms=5, area=1200, floors=1>
```

Data vs Struct: Mutability

Struct renovations

```ruby
House = Struct.new(:rooms, :area, :floors)
ranch = House.new(rooms: 5, area: 1200, floors: 1)
# => #<struct House rooms=5, area=1200, floors=1>

ranch.floors += 1
# => 2

ranch.floors
# => 2

ranch.frozen?
# => false
```

Struct's contents can be changed and/or iterated upon.

Using the house example, let's try to update the 1-story ranch house created from Struct.
We can add 1 floor to the house, and now the ranch has 2 floors.

We note too that ranch is not `frozen?`

Data vs Struct: Mutability

Data renovations

```ruby
House = Data.define(:rooms, :area, :floors)
ranch = House.new(rooms: 5, area: 1200, floors: 1)
# => #<data House rooms=5, area=1200, floors=1>

ranch.floors += 1
# (irb):3:in `<main>': undefined method `floors=' for an instance of House (NoMethodError)
```

NoMethodError is raised

Data's contents cannot be changed.

We cannot make the same renovation to the ranch House built by Data that we could to the ranch House built by Struct.

We note too that there is no `floors=` writer method for `ranch` so `NoMethodError` is raised.

What would happen if we define a writer?

Data vs Struct: Mutability

Defining a writer: Data

```ruby
House = Data.define(:rooms, :area, :floors) do
  def floors=(quantity)
    @floors = quantity
  end
end
# => House

ranch = House.new(rooms: 5, area: 700, floors: 1)
# => #<data House rooms=5, area=700, floors=1>

ranch.floors = 2
# (irb):21:in `floors=': can't modify frozen House: #<data House rooms=5, area=700, floors=1> (FrozenError)
ranch.frozen?
# => true
```

FrozenError is raised

We can create a writer method on `House`.

But when we try to use it `FrozenError` is raised.

And we note that `ranch` is frozen.

Data vs Struct: Built-in Methods

Data

```
>> Data.methods(false)
=> [:define]
>> Data.instance_methods(false)
=>
[:deconstruct_keys,
 :pretty_print,
 :pretty_print_cycle,
 :deconstruct,
 :hash,
 :==,
 :inspect,
 :members,
 :to_s,
 :with,
 :eql?,
 :to_h]
```

Struct

```
>> Struct.methods(false)
=> [:new]
>> Struct.instance_methods(false)
=>
[:deconstruct_keys,
 :==,
 :members,
 :to_a,
 :to_s,
 :[],
 :[]=,
 :values_at,
 :eql?,
 :to_h,
 :select,
 :filter,
 :pretty_print,
 :pretty_print_cycle,
 :deconstruct,
 :hash,
 :inspect,
 :each_pair,
 :values,
 :length,
 :dig,
 :size,
 :each]
```

Comparing built in methods - check out how many methods that come with Struct!

Data vs Struct:Built-in Methods (differences)

Data (not Struct)

```
>> data_methods - struct_methods
=> [:with]
```

```
>> struct_methods - data_methods
=>
[:to_a,
 :[],
 :[]=,
 :values_at,
 :select,
 :filter,
 :each_pair,
 :values,
 :length,
 :dig,
 :size,
 :each]
```

Struct (not Data)

When we compare differences in methods - there is only one that uses `.with`

`Data#with`

```ruby
House = Data.define(:rooms, :area, :floors)
cottage = House.new(rooms: 5, area: 1200, floors: 1)
# => #<data House rooms=5, area=1200, floors=1>

two_story_cottage = cottage.with(floors: 2)
# => #<data House rooms=5, area=1200, floors=2>
```

✅ Copies the instance with new value(s)

We can use `Data`'s with method to create a copy of the instance with any new attribute values we choose.

To Data or Not to Data?

- Mutability
- Behavior
- Communication

So, when do we want to use Data? When do we want to use something else?

When we want a lot of methods, we want to be able to change values, we don’t want Data, we probably want to use Class.

Use `Data` when:
- you want to create a simple class for a value object
- You want a well-encapsulated object - I.e. prevent changes to attributes, values, behavior
- In other words: securing the values so that the values cannot be mutated once created

But is that fully true?

Data Chaos and Immutability Workarounds

Kelly Popko | 9 July 2025
https://www.linkedin.com/in/kelly-popko/

An *instance* of Data is frozen - but if its attribute values are mutable, those values are still mutable.

So we can, in a sense, work against the intent of Data.

Immutability Workarounds

Example: updating a value of a Hash

```
ColorCounts = Data.define(:color_tally)
# => Colorcounts

screen_colors = ColorCounts.new({red: 0, blue: 0, green: 0})
# => #<data ColorCounts color_tally={:red=>0, :blue=>0, :green=>0}>

# Update a value in `color_tally`
screen_colors.color_tally[:red] += 5
# => 5

# Note that the value for  `:red` has been updated
screen_colors.color_tally
# => {:red=>5, :blue=>0, :green=>0}
```

We can work (arguably) against the intent of Data in some ways.

For example, when an attribute is set to a Hash, we can update values in the hash.

Immutability Workarounds

Example: updating a value of a Array

```
Cities = Data.define(:names)
# => Cities
wisconsin_cities = Cities.new([:milwaukee, :madison])
# => #<data Cities names=[:milwaukee, :madison]>

wisconsin_cities.names.pop
# => :madison

wisconsin_cities.names
# => [:milwaukee]
```

We can also work around this with an Array, because an Array is mutable.

Conclusions

You might want a Data object if...

Mutability | Behavior | Communication

Kelly Popko | 9 July 2025
https://www.linkedin.com/in/kelly-popko/
https://github.com/kellyky

Data is a great choice when you want to avoid a mutated value on the object coming back to haunt you.

That said, there are workarounds to know about, so it is not foolproof.

Using a Data object also communicates the intent of the object to current and future collaborators.

—

And that is a fly-by of Ruby's Data class!

Feel free to find me to discuss / share feedback.

Scan the QR code for links to the talk, my sources, and to find me LinkedIn.

Thank you so much!