Extracted from:

Programming Crystal

Create High-Performance, Safe, Concurrent Apps

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming Crystal

Create High-Performance, Safe, Concurrent Apps

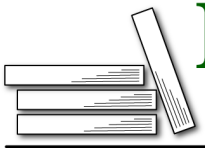
Ivo Balbaert
Simon St. Laurent
edited by Andrea Stewart

Programming Crystal

Create High-Performance, Safe, Concurrent Apps

Ivo Balbaert
Simon St. Laurent

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Andrea Stewart
Copy Editor: Nancy Rapoport
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-286-2
Book version: P1.0—February 2019

Structuring a Class

In the previous section, and in *Organizing Code in Classes and Modules*, on page ?, you saw a simple Mineral class. Here's the class code by itself without any additional logic:

```
class Mineral
  getter name : String
  getter hardness : Float64
  getter crystal_struct : String

  def initialize(@name, @hardness, @crystal_struct)  # constructor
  end
end
```

This class has three read-only instance variables: name, hardness, and crystal_struct. Giving them a type is imposed by the Crystal compiler. But you can also do this in the initialize method:

classes_and_structs/classes.cr

```
class Mineral
  getter name, hardness, crystal_struct

  def initialize(@name : String,
                 @hardness : Float64,
                 @crystal_struct : String)
  end
end
```

Default values can be assigned like this:

```
def initialize(@name : String = "unknown", ...)
end
```

Some people use symbols like :hardness for the property name, but it isn't required. A property without a type must have a default value. Or you could give it a value in initialize (try it!). You don't need to define variables at the start of the class.

The new method creates a Mineral object:

```
min1 = Mineral.new("gold", 1.0, "cubic")
min1 # => #<Mineral:0x271cf00 @crystal_struct="cubic",
# =>                                    @hardness=1.0, @name="gold">
min1.object_id # => 41012992 == 0x271cf00
typeof(min1) # => Mineral   # compile-time type
min1.class   # => Mineral   # run-time type
Mineral.class # => Class    # all classes have type Class
```

new is a class method that's created automatically for every class. It allocates memory, calls initialize, and then returns the newly created object. An object

is created on the heap and it has an object_id: its memory address. When it gets a new name or when it's passed to a method, only the reference is passed. This means the object is changed when it's changed in the method.

When you're not sure which types your initialize method will accept, you can also use generic types like T, as in this class Mineralg:

classes_and_structs/classes.cr

```
class Mineralg(T)
  getter name

  def initialize(@name : T)
  end
end

min = Mineralg.new("gold")
min2 = Mineralg.new(42)
min3 = Mineralg(String).new(42)

# => Error: no overload matches 'Mineralg(String).new' with type Int32
```

When naming instance variables, prefix them with @. For *class variables*, use @@, like the @@planet our mineral species comes from. All objects built using this class will share this variable, and its value will be the same to all of them. (However, subclasses, which you'll see in the next section, all get their own copy with the value shared across the subclass.)

To name properties that can change, such as quantity in the code that follows, prefix them with property. For write-only properties that can't be read, use the prefix setter, like id in the following code. Trying to show them is an error:

classes_and_structs/classes.cr

```
class Mineral
  @@planet = "Earth"

  getter name, hardness, crystal_struct
  setter id
  property quantity : Float32

  def initialize(@id : Int32, @name : String, @hardness : Float64,
                 @crystal_struct : String)
    @quantity = 0f32
  end

  def self.planet
    @@planet
  end
end

min1 = Mineral.new(101, "gold", 1.0, "cubic")
min1.quantity = 453.0f32 # => 453.0
min1.id                  # => Error: undefined method 'id' for Mineral
Mineral.planet           # => "Earth"

min2 = min1.dup
min1 == min2 # => false
```

You must make sure that properties are always initialized, either in the initialize method or when calling new. The names of methods called on the class itself are prefixed with self., like the planet method.

Use the dup method to create a "shallow" copy of the object: the copy min2 is a different object, but if the original contains fields that are objects themselves, these are not copied. If you need a "deep" copy, you have to define a clone method.

You can also optionally write a finalize method for a class, which is automatically invoked when an object is garbage collected:

```
def finalize
  puts "Bye bye from this #{self}!"
end
```

But this creates a burden for the garbage collection process. You should use it only if you want to free resources taken by external libraries that the Crystal garbage collector won't free for you. Add this code snippet to see finalization at work, but be warned: you'll exhaust your machine's memory by digging so much gold. So save anything you need before running it.

```
loop do
  Mineral.new(101, "gold", 1.0, "cubic")
end
```

As in Ruby or C#, you can *reopen a class*, which means making additional definitions of a class: they're all combined into a single class. This even works for built-in classes. How cool is it to define your own new methods on existing classes, such as String or Array? (Yes, this is sometimes derisively called "monkey patching," and it's not always a good idea.)

Your Turn 1

➤ a. Employee: Create a class Employee with a getter name and a property age. Make an Employee object and try to change its name.

➤ b. Increment: Create a class Increment with a property amount and two versions of a method increment: one that adds 1 to amount, and another that adds a value, inc_amount.

Applying Inheritance

As in all object-oriented languages and much the same as in Ruby, Crystal provides for single *inheritance*, indicated by: subclass < superclass. Putting properties and methods common to several classes into a superclass lets them all share functionality. That way, you can use all instance variables and all methods of the superclass in the subclass, including the constructors. You can see this in the following example where PDFDocument inherits initialize, name, and print from Document:

classes_and_structs/inheritance.cr

```
class Document
  property name

  def initialize(@name : String)
  end

  def print
    puts "Hi, I'm printing #{@name}"
  end
end

class PDFDocument < Document
end

doc = PDFDocument.new("Salary Report Q4 2018")
doc.print # => Hi, I'm printing Salary Report Q4 2018
```

You can also override any inherited method in the subclass. If the subclass defines its own initialize method(s), they aren't inherited anymore. If you want to use the superclass functionality after you overrode it, you can call any method of the superclass with super:

classes_and_structs/inheritance.cr

```
class PDFDocument < Document
  def initialize(@name : String, @company : String)
  end

  def print
    super
    puts "From company #{@company}"
  end
end

# doc = PDFDocument.new("Salary Report Q4 2018")
# => Error: wrong number of arguments for 'PDFDocument.new' (given 1,
# => expected 2)
doc = PDFDocument.new("Salary Report Q4 2018", "ACME")
doc.print

# => Hi, I'm printing Salary Report Q4 2018
#    From company ACME
```

Crystal's type system gives you more options here. Instead of overriding, you can define specialized methods by using type restrictions, such as `print` in `PDFDocument`:

classes_and_structs/inheritance.cr

```
class PDFDocument < Document
  def initialize(@name : String, @company : String)
  end

  def print(date : Time)
    puts "Printing #{@name}"
    puts "From company #{@company} at date #{date}"
  end
end

doc = PDFDocument.new("Salary Report Q4 2018", "ACME")
doc.print(Time.now)

# => Printing Salary Report Q4 2018
#    From company ACME at date 2017-05-25 12:12:45 +0200
```

Using Abstract Classes and Virtual Types

Ruby doesn't have native support for interfaces and abstract classes like Java or C#. In both Ruby and Crystal, the concept of an interface is implemented through modules, as you'll see in the next chapter. But Crystal also knows the concept of an abstract class, so if you're a Rubyist, the following will be new to you.

Not all classes are destined to produce objects, and `abstract` classes are a good example. These serve instead as a blueprint for subclasses to implement their

methods. Here you see a class, Rect (describing rectangles), forced to implement all abstract methods from class Shape:

classes_and_structs/inheritance.cr

```
abstract class Shape
  abstract def area
  abstract def perim
end

class Rect < Shape
  def initialize(@width : Int32, @height : Int32)
  end

  def area
    @width * @height
  end

  def perim
    2 * (@width + @height)
  end
end

s = Shape.new       # => can't instantiate abstract class Shape
Rect.new(3, 6).area # => 18
```

If one of the methods (say perim) isn't implemented, the compiler issues an error like the following:

```
error: "abstract `def Shape#perim()` must be implemented by Rect"
```

This lets you create class hierarchies where you can be confident that all necessary methods are implemented.

You can create more intricate structures as well. In the following example, class Document is called a *virtual type* because it combines different types from the same type hierarchy—in this case, different documents:

classes_and_structs/virtual.cr

```
class Document
end

class PDFDocument < Document
  def print
    puts "PDF header"
  end
end

class XMLDocument < Document
  def print
    puts "XML header"
  end
end
```

```
class Report
  getter doc

  def initialize(@name : String, @doc : Document)
  end
end

salq4 = Report.new "Salary Report Q4", PDFDocument.new
taxQ1 = Report.new "Tax Report Q1", XMLDocument.new
```

This virtual type is indicated by the compiler as type Document+, meaning that all types inherit from Document, including Document itself. It comes into play in situations like the one that follows where you'd expect d to be of a union type (PDFDocument | XMLDocument):

```
if 4 < 5
  d = PDFDocument.new
else
  d = XMLDocument.new
end
typeof(d) # => Document
```

Instead, d is of type Document. Internally the compiler uses this as a virtual type Document+ instead of the union type (PDFDocument | XMLDocument), because union types quickly become very complex in class-hierarchies.

If you call a method in a subclass of Document, you get an error:

```
salq4.doc.print # => Error: undefined method 'print' for Document
```

To remove this error, simply make the class Document abstract.

classes_and_structs/virtual.cr

```
abstract class Document
end

salq4.doc.print # => PDF header
```

Your Turn 2

➤ Shape: Subclass Shape with classes Square and Circle. (Hint: Use PI from the Math module with: include Math.)