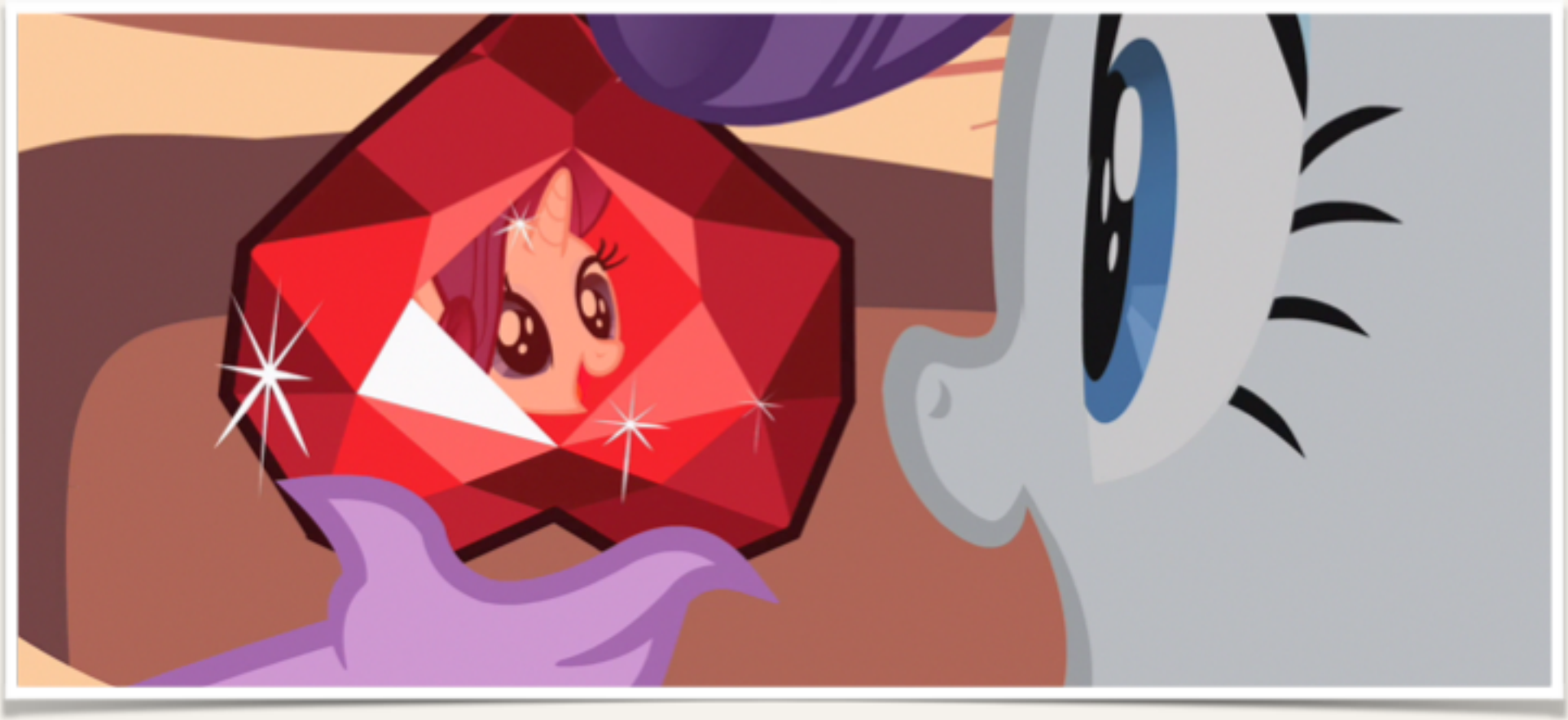


Jason Voegele / @jvoegele

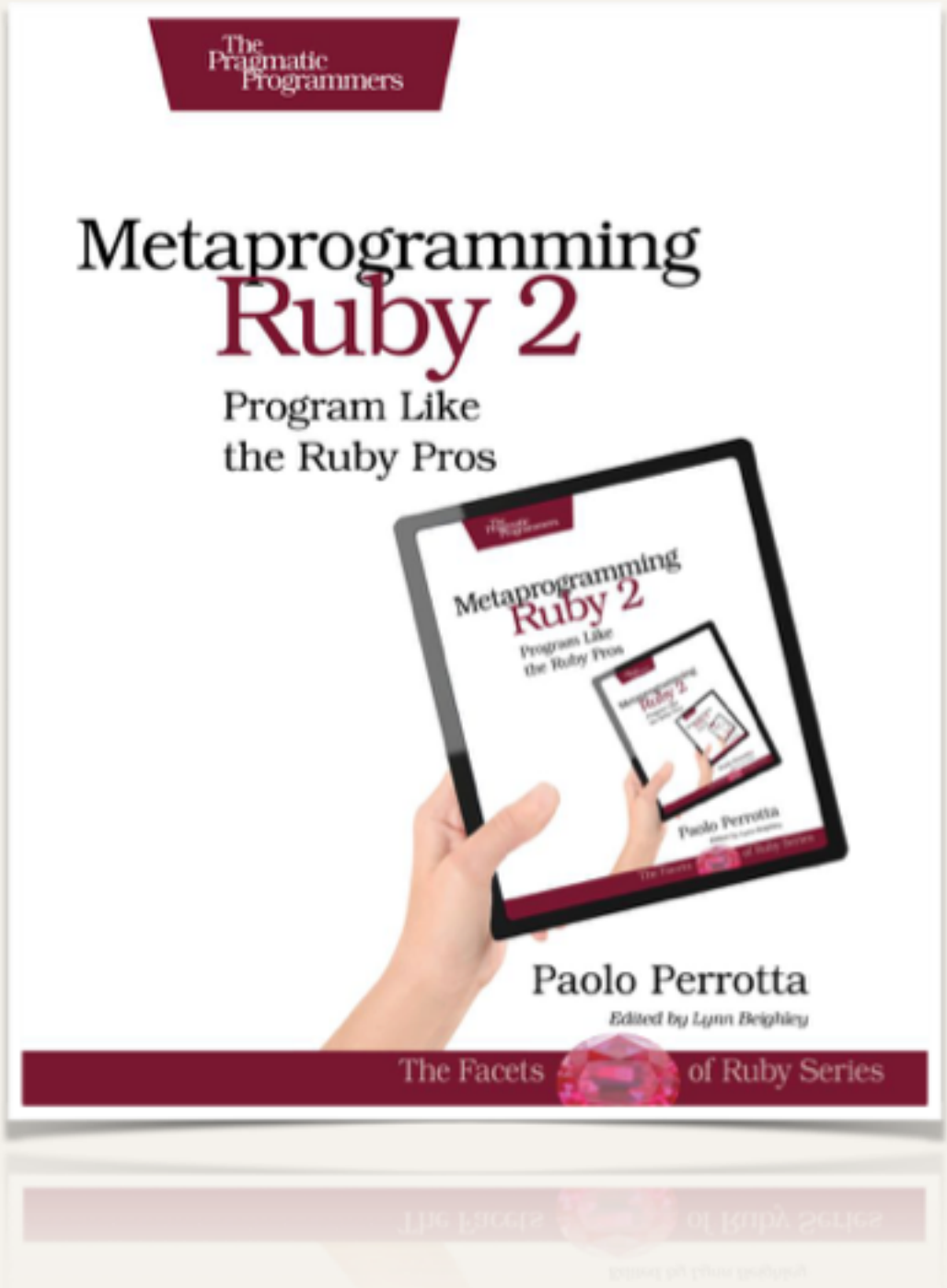
Reflecting on Ruby

- The Ruby Object Model
- Reflection
- Metaprogramming

Metaprogramming Ruby

Based on a true story

Fantastic book by Paolo Perrotta,
published by Pragmatic Programmers.



What is metaprogramming?

- Writing code that writes code…not exactly.
- Metaprogramming is writing code that manipulates language constructs at *runtime*:
  - define classes on the fly,
  - add methods to objects,
  - wrap methods with custom before and/or after behavior,
  - create domain specific languages, etc.
- Contrast with *static* code generators and compilers.

The first rule of metaprogramming:

DON'T DO IT!

— Geoff Lane

The Ruby Object Model

- Every value in Ruby is an object
  - (but some are more objecty than others)
- Ruby objects are structures that contain:
  - pointer to a class
  - some flags (tainted, frozen, etc.)
  - array of instance variables
- What's missing?

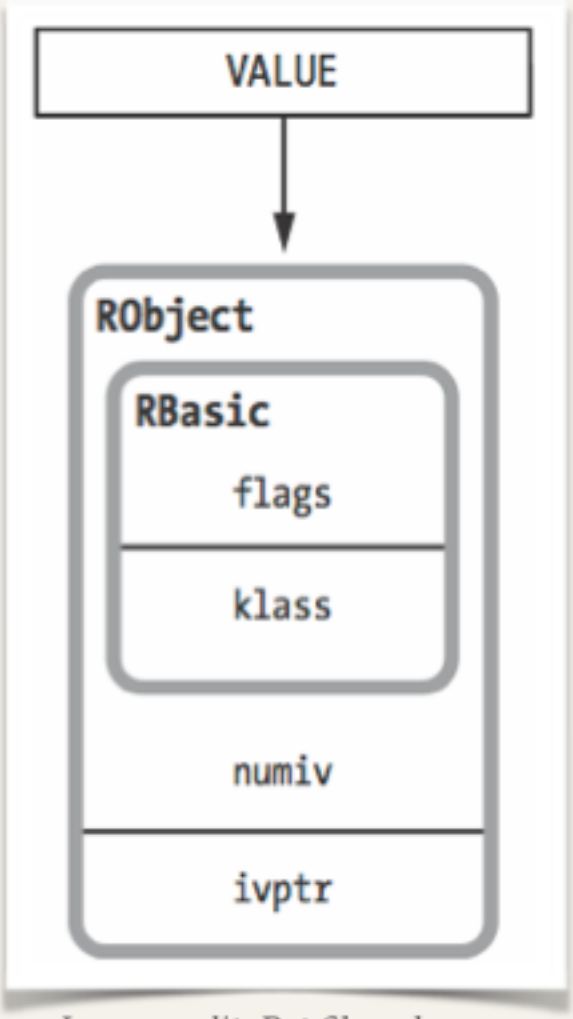


Image credit: Pat Shaughnessy

Where are the methods?

- Methods exist at runtime only in *classes*, not objects.
  - (Even when appearances are to the contrary.)
- The methods of an object are the instance methods of its class.
- Ruby has a slew of methods for discovering methods, including `Object#methods`, `Module#instance_methods`, etc.
- Methods can be turned into objects using `Object#method`

Blocks, Procs, Lambdas, and Closures

- Blocks are just chunks of code between {…} or do…end.
- Methods can take a single block as an implicit parameter, and yield control to the block.
- Blocks can be turned into objects using:
  - Proc.new
  - lambda or ->
  - The & operator
- Blocks are closures, meaning they capture their surrounding scope.
- Lambdas are procs that act more like methods.

Classes

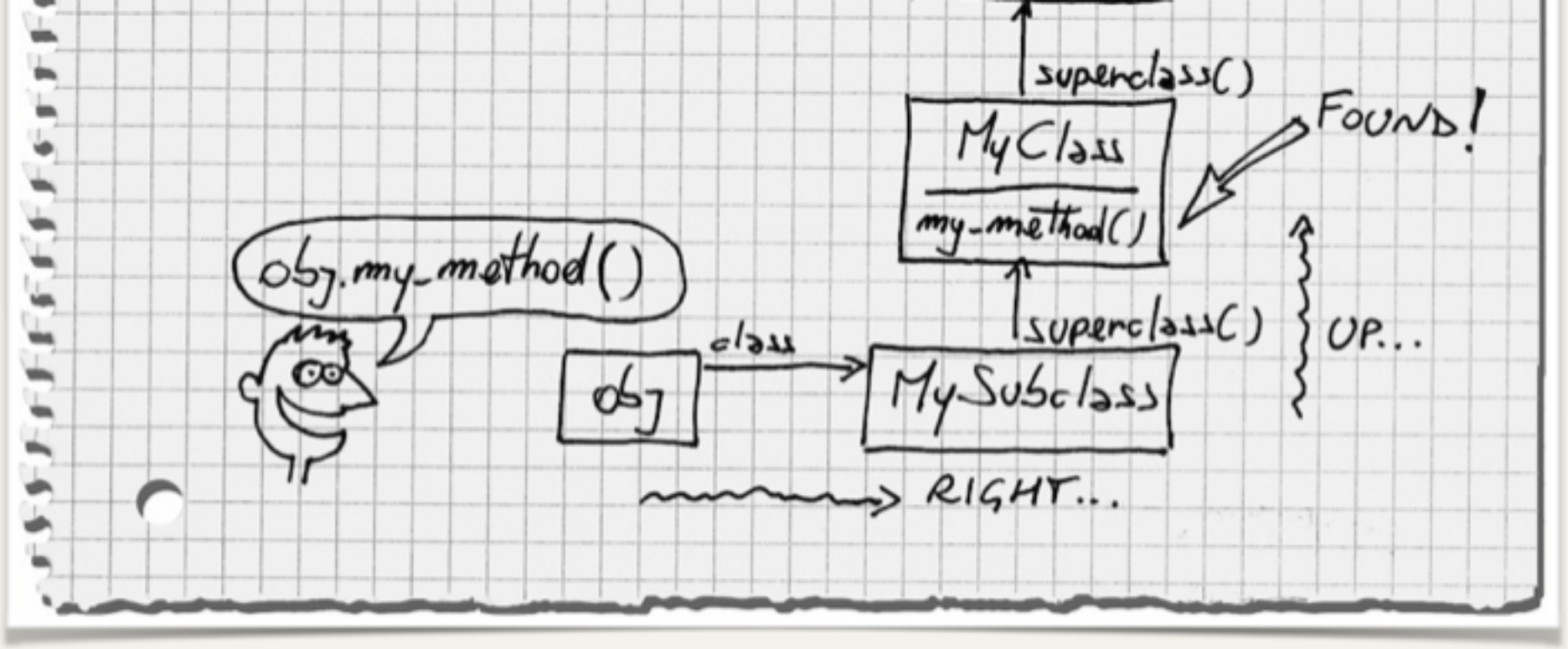
Classes

- Every object is an instance of some class.
- Classes are objects too. A class is an instance of **`Class`**.
- The name of the class (i.e. **`String`**) is a constant that refers to the **`Class`** object.
- The **`class`** keyword is a scope operator where:
  - Inside is normal, executable Ruby code.
  - **`self`** refers to the **`Class`** object.
- Classes can be created dynamically with `Class.new`

Class Inheritance

Class Inheritance

- All classes (except one) have exactly one superclass.
- If not otherwise specified, the superclass is **`Object`**.
- **`Object`** in turn inherits from **`BasicObject`**, which is the only Ruby class that does not have a superclass.
- A class's superclass can be obtained with the `superclass` method.
- The entire chain of superclasses can be obtained with the `ancestors` method.



Method Lookup

Ruby finds methods by looking at the receiver's class, then climbing the ancestors chain.

Modules

- Modules are like classes, except totally different.
- Provide a scope for defining methods.
- Modules are objects of class `Module`.
- Cannot create new objects from modules.
- Modules do not participate in inheritance, but can be included in the ancestors chain in interesting ways.

`include` and `prepend`

- Both `include` and `prepend` insert the module into the ancestors chain of the calling class (or module).
- `include` inserts directly after the caller.
- `prepend` inserts directly before the caller.
- This affects what **`super`** means when used in methods.

Ruby Trivia

- Where is the mysterious method defined?
- How can we call it?

```
class C
  def m1
    def mysterious; end
  end
end

class D < C; end
```

Singleton Methods

- Ruby allows for defining methods on individual objects.
- They are defined only on the object, not other objects of the same class.
- Class methods (and module methods) are actually singleton methods where the object happens to be a class.

Singleton Classes

- Recall that methods live only in classes, not objects.
- Singleton methods cannot live in the object's class, since that is shared by other objects.
- Ruby creates a special singleton class for each object. Also known as a metaclass or eigenclass.
- You can access it with `singleton_class` method or this curious syntax:

```
class << object
  # In here, self is the singleton class
end
```

Object#extend

- "Every single day, somewhere in the world, a Ruby

- "Every single day, somewhere in the world, a Ruby programmer tries to define a class method by including a module."
- Including a module gets the module's instance methods, not the singleton methods (i.e. class methods).
- So, include the module in the singleton class.
- Or, equivalently, use `extend`.
- `extend self`: make `self`'s instance methods also be singleton methods

Singleton Classes Inception

- Singleton classes are classes, and classes are objects, and objects have singleton classes. So, singleton classes themselves have singleton classes.
- You will never use this knowledge.

Ruby Trivia

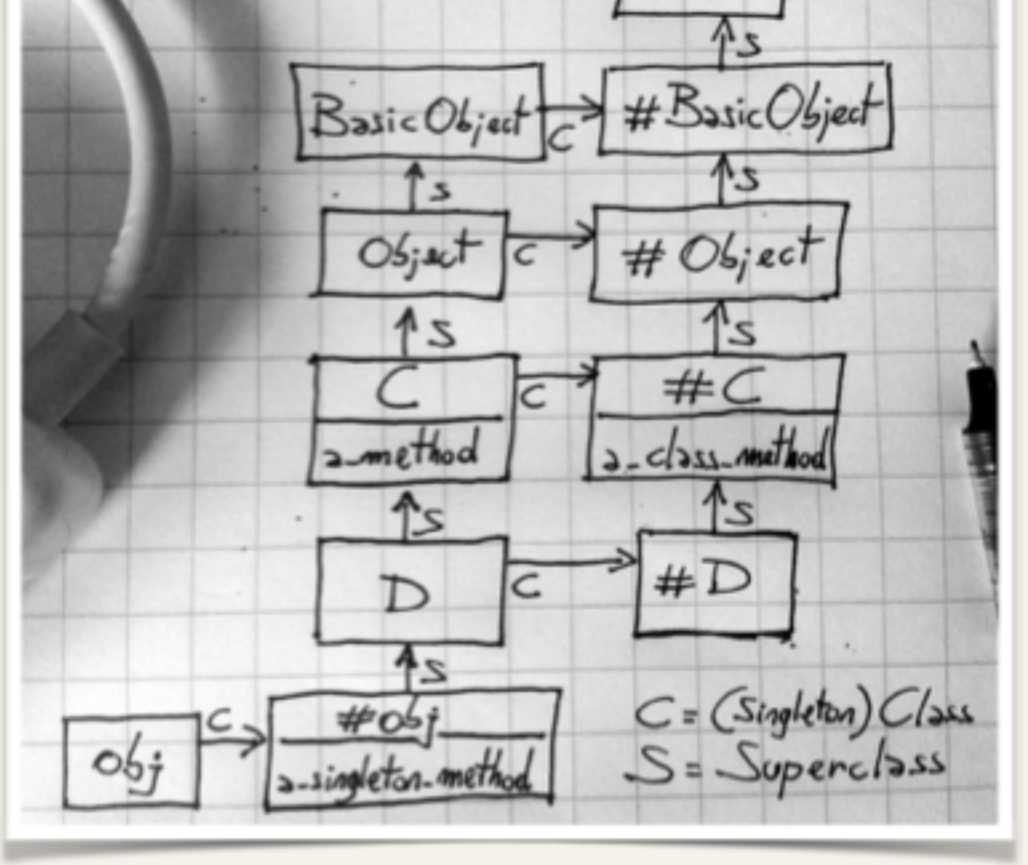
- If O is an object, what is the superclass of O's singleton

class?

- The superclass of the singleton class of an object is the object's class.
- If C is a class, what is the superclass of C's singleton class?
- The superclass of the singleton class of a class is the singleton class of the class's superclass.

Method Lookup Revisited

- "If an object has a singleton class, Ruby starts looking for methods in the singleton class rather than the conventional class." — The Book
- This is why you can call a class method on a subclass of the class in which it is defined.



method_missing

- If Ruby cannot find a method in the ancestors chain, it invokes `method_missing`.

- Classes can redefine method_missing to respond to messages ("ghost methods").
- Fraught with peril:
  - `respond_to?` lies (unless you also redefine `respond_to_missing?`)
  - Ghost methods cannot be discovered with reflection
- However, can be used for great good: https://github.com/jimweirich/builder

---

`define_method`

- `define_method` is a singleton method defined in the

- `define_method` is a singleton method defined in the **`Module`** class.
- Given a block, create a named instance method on the calling class.
- Useful for defining families of related methods.

Better Monkey Patching With `prepend`

- `prepend` inserts a module into the ancestors chain *before* the class or module that calls it.

- Methods defined in the prepended module can call super to access the original.
- Advantages:
  - No need to alias the original method to call it.
  - Documentation: the monkey patched method must be defined in a module that is inserted into the ancestors chain.

The `eval` Family of Methods

- `Kernel#eval`: blunt instrument, stay away
- `Module#class_eval`: evaluates a block in the context of an existing class.

an existing class.

- `BasicObject#instance_eval`: evaluates a block in the context of a specific object. Redefines `self` to be the receiver. This totally breaks encapsulation, but can be very useful.
- Often used for defining Domain Specific Languages (DSLs) in Ruby.

The `exec` Family of Methods

- Blocks passed to `eval` methods are still closures, but since `self` is redefined they don't have access to instance variables in their surrounding scope.

- The exec methods provide a workaround: any arguments passed to the method are made available to the block.
- `Module#class_exec(*args)`
- `BasicObject#instance_exec(*args)`
- Not used frequently, but RSpec makes heavy use internally.