Exploring Meta programming with Ruby & Lisp

Neil Jeetendra Handa - IMT2021037
Danesh Manoj Toshniwal - IMT2021094
Jashwanth Kadaru - IMT2021095

Table of contents

01

What is Meta Programming

Introduction

02

How Meta programming works

How Meta Programming is supported in languages like ruby

03

Impact of MP on Language & Types

Meta feature Syntax, Dynamic vs Static Binding, Typing, Introspec -tion Features & Lisp Macros

04

Object Model (Ruby MP)

Open Classes, instance_methods
Modules, method Lookup

05

Dynamic Methods & Ghost Methods (R)

Dynamic method, method_missing, send, Ghost methods, respond_to, Blank Slates

06

Blocks & Evals

Blocks, Closures, scope gates, instance_eval, Procs, class_eval, singleton methods & classes, aliases, wrappers, class macros

Table of contents

07

Practical Applications of MP

DSLs, Active Records, Sinatra

08

Code Demo & Conclusion

Conclusion of presentation.

01

What is
Meta Programming ?

What is Meta Programming ?

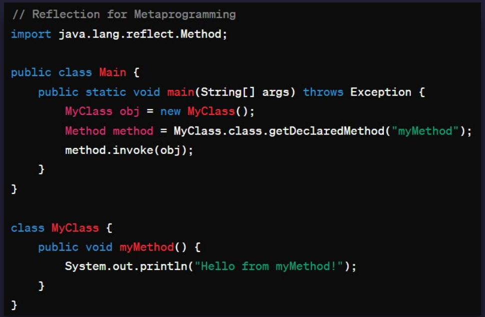
Introduction:

- *“Metaprogramming is writing code that writes code.”*

- *“It is writing code that manipulates language constructs at runtime.”*

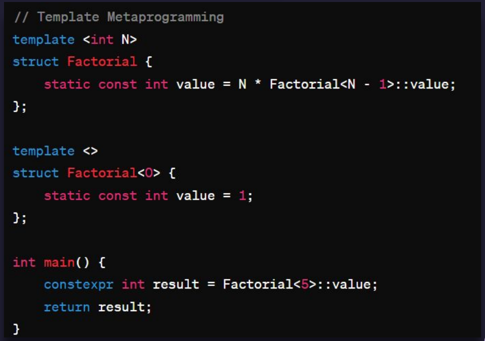
❖ Language constructs refer to programming language syntax such as class, method definitions & declarations, Modules, Instance variables, constants, etc.

Examples: 1) Java :



```java
// Reflection for Metaprogramming
import java.lang.reflect.Method;

public class Main {
    public static void main(String[] args) throws Exception {
        MyClass obj = new MyClass();
        Method method = MyClass.class.getDeclaredMethod("myMethod");
        method.invoke(obj);
    }
}

class MyClass {
    public void myMethod() {
        System.out.println("Hello from myMethod!");
    }
}
```

4) C++ :



```cpp
// Template Metaprogramming
template <int N>
struct Factorial {
    static const int value = N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0> {
    static const int value = 1;
};

int main() {
    constexpr int result = Factorial<5>::value;
    return result;
}
```

What is Meta Programming ?

- *“Metaprogramming is writing code that writes code.”*

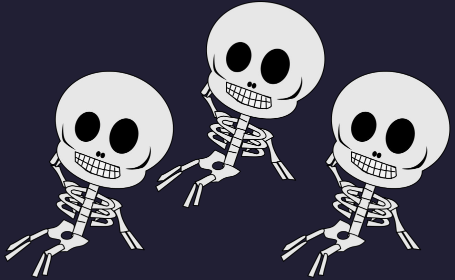
02

How Meta Programming Works?

How Meta Programming works?

- *“Metaprogramming is writing code that writes code.”*

Language Constructs



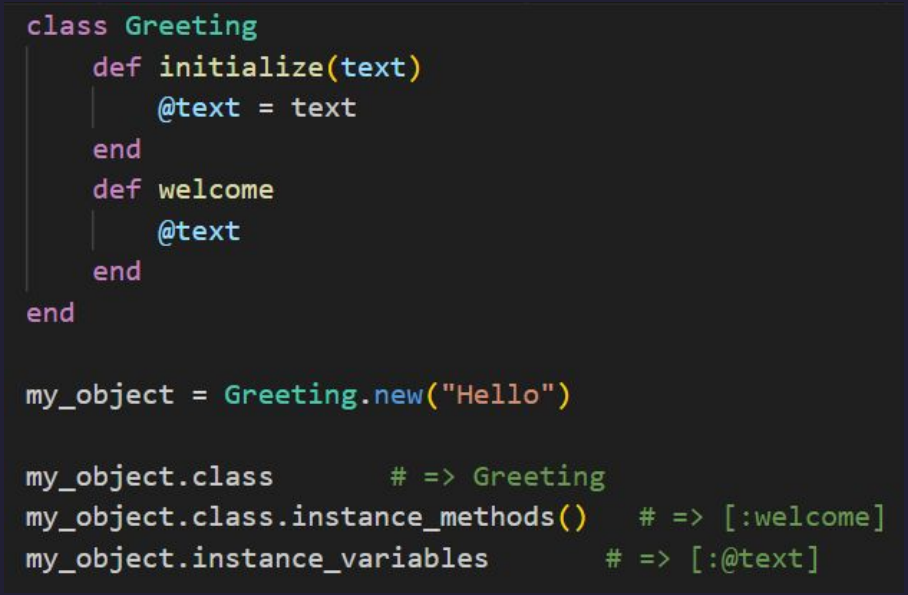
Source code ✅

Runtime ❌

Source code ✅

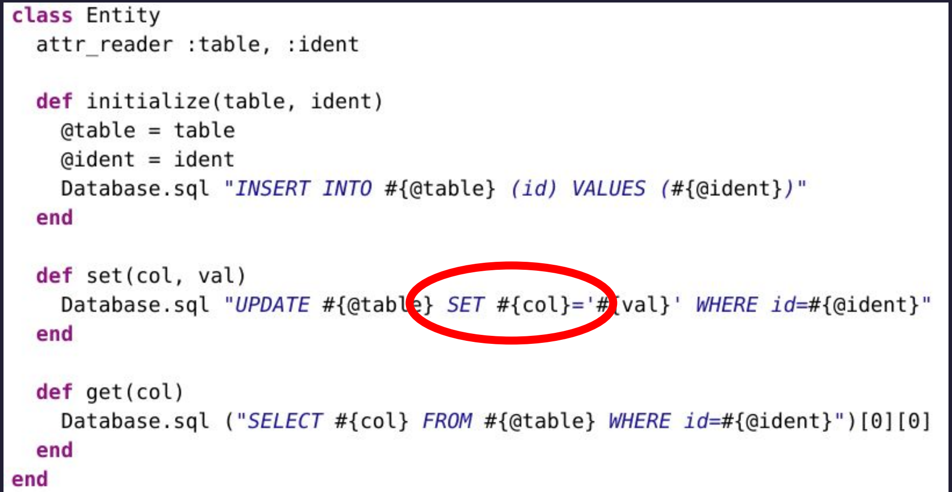
Runtime ✅

How Meta Programming works?

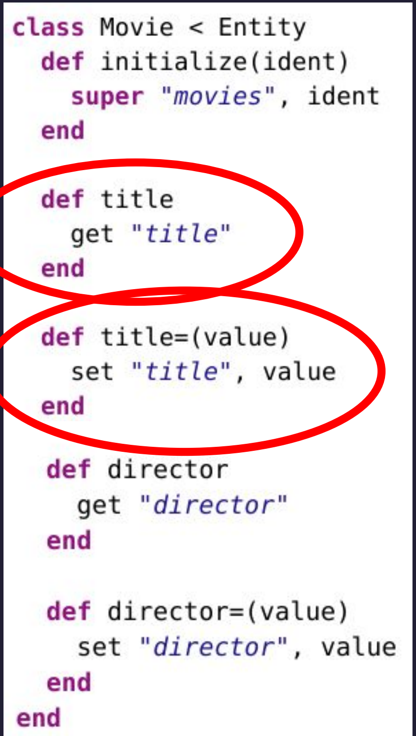


```
class Greeting
    def initialize(text)
        @text = text
    end
    def welcome
        @text
    end
end

my_object = Greeting.new("Hello")

my_object.class       # => Greeting
my_object.class.instance_methods()   # => [:welcome]
my_object.instance_variables       # => [:@text]
```

- *“It is writing code that manipulates language constructs at runtime.”*

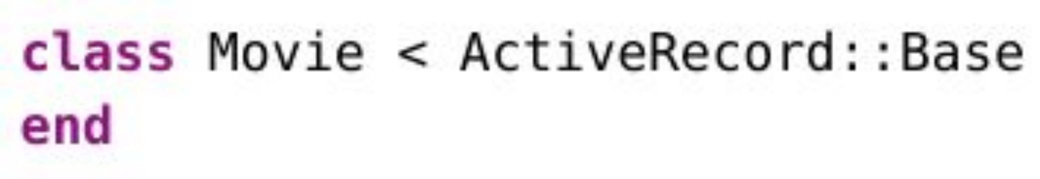


```ruby
class Entity
  attr_reader :table, :ident

  def initialize(table, ident)
    @table = table
    @ident = ident
    Database.sql "INSERT INTO #{@table} (id) VALUES (#{@ident})"
  end

  def set(col, val)
    Database.sql "UPDATE #{@table} SET #{col}='#{val}' WHERE id=#{@ident}"
  end

  def get(col)
    Database.sql ("SELECT #{col} FROM #{@table} WHERE id=#{@ident}")[0][0]
  end
end
```



```ruby
class Movie < Entity
  def initialize(ident)
    super "movies", ident
  end

  def title
    get "title"
  end

  def title=(value)
    set "title", value
  end

  def director
    get "director"
  end

  def director=(value)
    set "director", value
  end
end
```

movie = Movie.new(1)
movie.title = "Man of Steel"
movie.director = "Zack Snyder”



```ruby
class Movie < ActiveRecord::Base
end
```

```ruby
movie = Movie.create
movie.title = "Man of Steel"
movie.title # => "Zack Snyder"
```

ActiveRecord Library - Famous Ruby Library that maps objects to database tables.

1) Given “Movie” class, it automatically finds out its plural word “movies” as the table name in the database schema.
2) Automatically defines the getter and setter methods for all the attributes in the table.
3) With creation of every object, it adds a new entity to the same table.

03

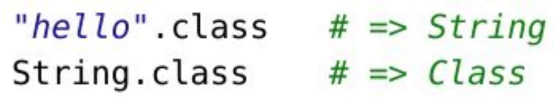
Impact of Meta programming on Languages and Types

Built-in Support of Metaprogramming features	**VS**	**Languages without dedicated syntax**
Dynamic Typing	**VS**	**Static Typing**

04

Object Model (Ruby)

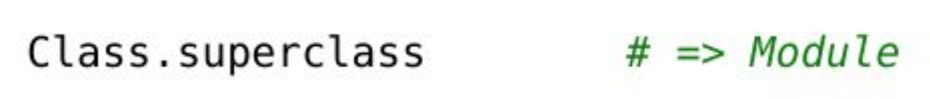
Classes themselves are nothing but objects.



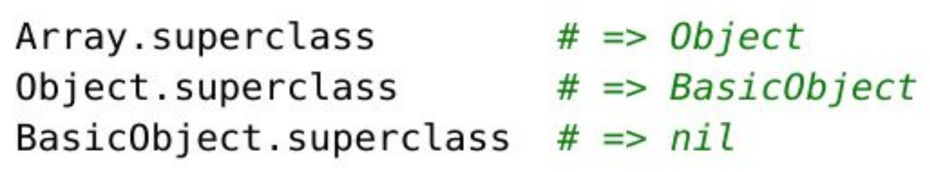
```
"hello".class    # => String
String.class     # => Class
```

Because a class is an object, everything that applies to objects also applies to classes. Classes, like any object, have their own class, called “Class”

A Ruby **class** inherits from its **superclass.**



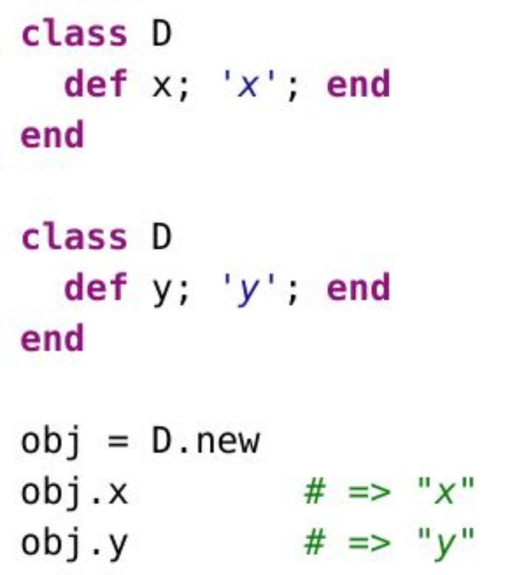
```
Class.superclass          # => Module
```



```
Array.superclass          # => Object
Object.superclass         # => BasicObject
BasicObject.superclass    # => nil
```

BasicObject is the root of the Ruby class hierarchy.

Open Classes



```
class D
  def x; 'x'; end
end

class D
  def y; 'y'; end
end

obj = D.new
obj.x          # => "x"
obj.y          # => "y"
```

The class keyword in Ruby is more like a scope operator than a class declaration.

We can always reopen existing classes, even standard library classes such as String or Array, and modify them on the fly.

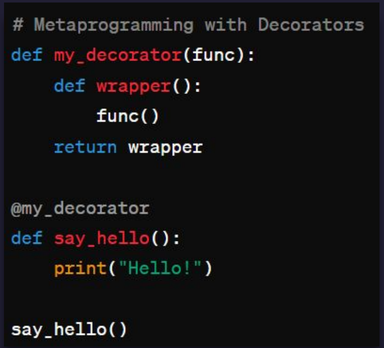
What is Meta Programming ?

Introduction:

- *The word “meta” means higher level of abstraction.*
- *Higher abstraction helps in writing better & more flexible code.*

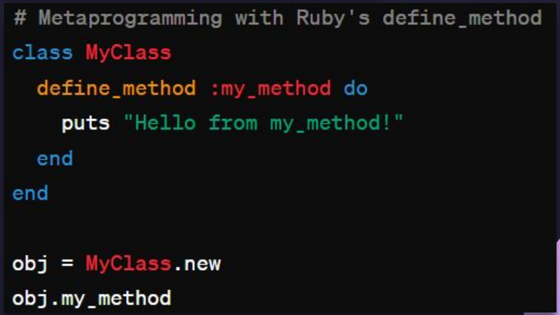
- Procedural programming abstracts blocks of code as functions. OOP abstracts bundle of data & methods as classes & objects.
- Meta programming abstracts the program itself and provides ***mechanisms to examine & modify, its structure & behaviour*** *at run-time.*

3) Python :



```
# Metaprogramming with Decorators
def my_decorator(func):
    def wrapper():
        func()
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

4) Ruby :



```
# Metaprogramming with Ruby's define_method
class MyClass
  define_method :my_method do
    puts "Hello from my_method!"
  end
end


obj = MyClass.new
obj.my_method
```

What is Meta Programming ?

Introduction:

- *Introspection: The ability of a program to inspect its own internal state at runtime (e.g., examining methods, properties of objects & classes)*

- *Modification: The ability of a program to dynamically alter its structure and behavior at runtime (e.g., adding methods, changing ancestor chain, etc).*

Let's Dive Into Mechanics & Implications Of Metaprogramming in a language !!!

02

How Meta Programming Works?

Concept of Live Runtime Environment, Meta-Classes & Dynamic binding. An overview of how MP is supported in Ruby and why not in C.

How Meta Programming Works?

Overview:

- Think of source code as a world of programming language constructs.

- In languages like C, these programming constructs do not exist at runtime in their original form, as defined in the source code.

❖ In metaprogramming languages like Ruby, Lisp, python, most of these language constructs are still alive and stored somewhere in memory, preserving the meaning as it was in the source code.

How Meta Programming Works?

Overview:

- In C, nothing survives the compilation phase, and by runtime, everything is converted to machine code and loses its meaning & definition as it was in source code.

- In languages like C++, Java, some of these programming constructs do survive the compilation and persist during runtime. Allowing users to leverage some meta-features.

❖ In metaprogramming languages like Ruby, there is no compilation phase and most of the constructs exist during runtime, allowing Ruby to provide more robust meta features.

How Meta Programming Works?

Things to Note:

- The constructs are stored by creating meta-classes and meta-objects that store the definitions and other semantics of the constructs, in a place in memory.

- Ruby for example, by default creates meta-classes for every class defined.

- ❖ The persistence & existence of constructs, and meta-classes, allow us to examine the source code by querying the meta-class and also modify it, during run-time. The lack of this kind of design, obstructs a language from providing meta-programming to users.

03

Impact of MP on Language & Types

This section explores impact of metaprogramming on language design (syntax, semantics, features) and type systems (safety, inference).

Impact of MP on Language types

Impact on Language syntax, semantics, & features:

- Think of source code as a world of programming language constructs.

- In languages like C, these programming constructs do not exist at runtime in their original form, as defined in the source code.

- In metaprogramming languages like Ruby, Lisp, python, most of these language constructs are still alive and stored somewhere in memory, preserving the meaning as it was in the source code.

04

Object Model (Ruby MP)

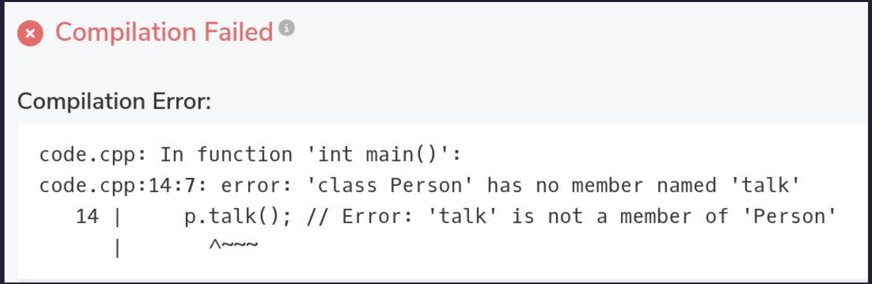
This section explores the object model of Ruby, and various methods & syntax in-place for introspection and code modification during runtime.

05

Dynamic Methods & Ghost Methods

This section explores various techniques that can be used to avoid duplication of code using meta-programming.

```
class Computer
  def initialize(computer_id, data_source)
  @id = computer_id
  @data_source = data_source
  end

  def mouse
  info = @data_source.get_mouse_info(@id)
  price = @data_source.get_mouse_price(@id)
  result = "Mouse: #{info} ($#{price})"
  return "* #{result}" if price >= 100
  result
  end

  def cpu
  info = @data_source.get_cpu_info(@id)
  price = @data_source.get_cpu_price(@id)
  result = "Cpu: #{info} ($#{price})"
  return "* #{result}" if price >= 100
  result
  end

SO ON............
```

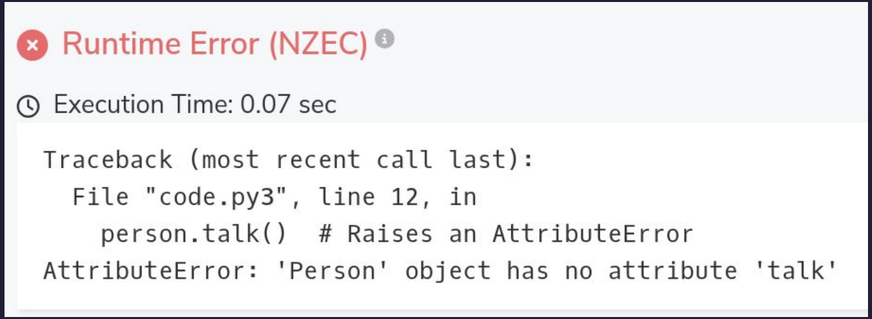


Compilation Failed

Compilation Error:

```
code.cpp: In function 'int main()':
code.cpp:14:7: error: 'class Person' has no member named 'talk'
   14 |     p.talk(); // Error: 'talk' is not a member of 'Person'
      |       ^~~~
```

Static Languages



Runtime Error (NZEC)

Execution Time: 0.07 sec

```
Traceback (most recent call last):
  File "code.py3", line 12, in
    person.talk()  # Raises an AttributeError
AttributeError: 'Person' object has no attribute 'talk'
```

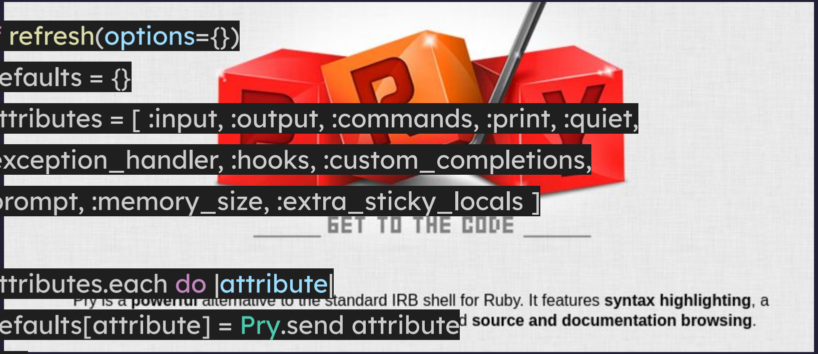
Dynamic Languages

Dynamic Methods

```ruby
class MyClass
  def my_method(my_arg)
  my_arg * 2
  end
end
obj = MyClass.new
obj.my_method(3) # => 6
```

Dynamic
Dispatch

```ruby
obj.send(:my_method, 3) # => 6
```



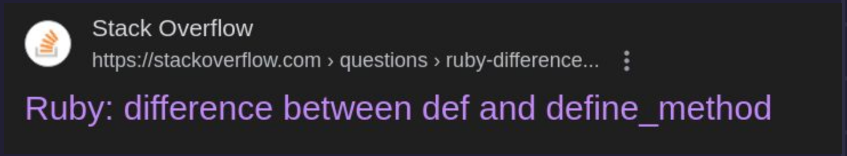
Pry is a powerful alternative to the standard IRB shell for Ruby. It features **syntax highlighting**, a … **source and documentation browsing**.

```ruby
def refresh(options={})
  defaults = {}
  attributes = [ :input, :output, :commands, :print, :quiet,
  :exception_handler, :hooks, :custom_completions,
  :prompt, :memory_size, :extra_sticky_locals ]

  attributes.each do |attribute|
  defaults[attribute] = Pry.send attribute
  end
# ...
  defaults.merge!(options).each do |key, value|
  send("#{key}=", value) if respond_to?("#{key}=")
  end
  true
end
```

```ruby
def refresh(options={})
ptions[:memory_size]
  defaults[:quiet] = Pry.quiet
  self.quiet = options[:quiet] if options[:quiet]
  # same for all the other attributes...
end
```

Calling methods ✅

Defining methods ❓



Stack Overflow
https://stackoverflow.com › questions › ruby-difference...

Ruby: difference between def and define_method

```
class MyClass
  define_method :my_method do |my_arg|
    my_arg * 3
  end
end

obj = MyClass.new
obj.my_method(2) # => 6
```

Little bit of Introspection too

Step 1   Step 2

```
class Computer
  def initialize(computer_id, data_source)
    @id = computer_id
    @data_source = data_source
  end
  def mouse
    component :mouse
  end
  def keyboard
    component :keyboard
  end
end
```

```
class Computer
  def initialize(computer_id, data_source)
    @id = computer_id
    @data_source = data_source
    data_source.methods.grep(/^get_(.*)_info$/) { Computer.define_component $1 }
  end

  def self.define_component(name)
    define_method(name) do
      # ...
    end
  end
end
```

Continued for readability

```
class Computer
  def initialize(computer_id, data_source)
    @id = computer_id
    @data_source = data_source
  end

  def self.define_component(name)
    define_method(name) do
      info = @data_source.send "get_#{name}_info", @id
      price = @data_source.send "get_#{name}_price", @id
      result = "#{name.capitalize}: #{info} ($#{price})"
      return "* #{result}" if price >= 100
      result
    end
  end

  define_component :mouse
  define_component :cpu
  define_component :keyboard
end
```

METHOD_MISSING

```ruby
class Computer
  def initialize(computer_id, data_source)
    @id = computer_id
    @data_source = data_source
  end
  def method_missing(name)
    super if !@data_source.respond_to?("get_#{name}_info")
    info = @data_source.send("get_#{name}_info", @id)
    price = @data_source.send("get_#{name}_price", @id)
    result = "#{name.capitalize}: #{info} ($#{price})"
    return "* #{result}" if price >= 100
    result
  end
end
```

Handle Dynamically

PROBLEM

By overriding

By Extracting info

```ruby
Object.instance_methods.grep /^d/ # =>
[:dup, :display, :define_singleton_method]
```

Implement logic

SOLN: Blank States

06

Blocks & Evals

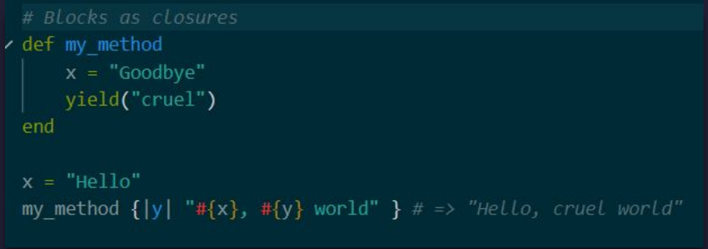
This section explores blocks, scoping, and eval methods, that add more meta programming features to Ruby.

Blocks

A block can only be defined and passed to a method.

Method can call block, using yield. It's a call back.

Blocks preserve their original bindings, from the scope they were defined ⇨ Closures.

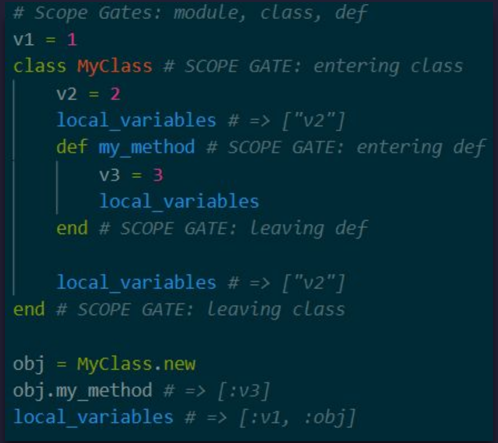


```ruby
# Blocks as closures
def my_method
    x = "Goodbye"
    yield("cruel")
end

x = "Hello"
my_method {|y| "#{x}, #{y} world" } # => "Hello, cruel world"
```

Scope Gates

A scope is defined by 3 keywords: module, class, def (method).

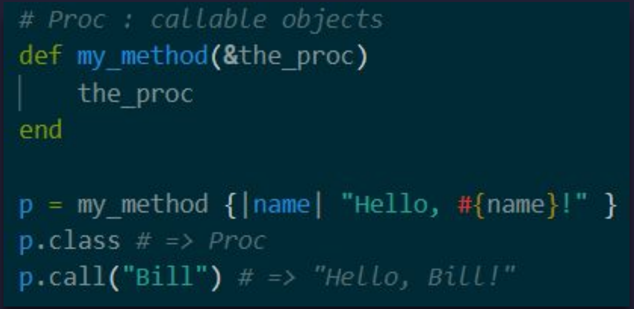
When moving from 1 scope to another, previous scope bindings are lost



```ruby
# Scope Gates: module, class, def
v1 = 1
class MyClass # SCOPE GATE: entering class
    v2 = 2
    local_variables # => ["v2"]
    def my_method # SCOPE GATE: entering def
        v3 = 3
        local_variables
    end # SCOPE GATE: leaving def

    local_variables # => ["v2"]
end # SCOPE GATE: leaving class

obj = MyClass.new
obj.my_method # => [:v3]
local_variables # => [:v1, :obj]
```

Procs

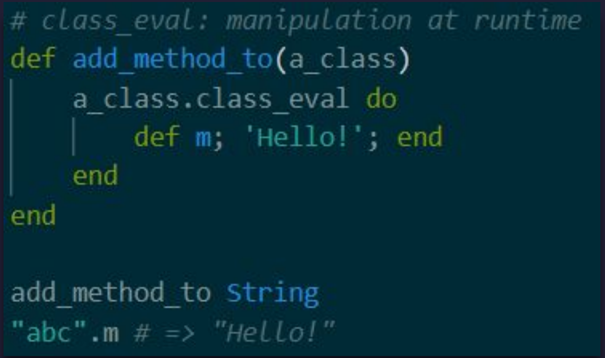
Procs allow us to run blocks (code) dynamically instead of statically.



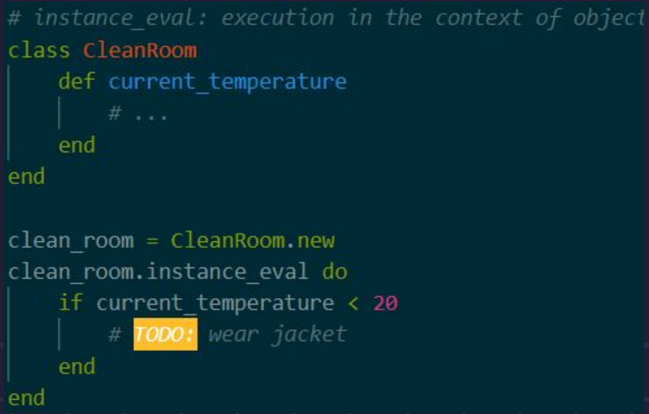
```ruby
# Proc : callable objects
def my_method(&the_proc)
    the_proc
end

p = my_method {|name| "Hello, #{name}!" }
p.class # => Proc
p.call("Bill") # => "Hello, Bill!"
```

Evals

- The instance eval method enables flattening of scope.
- Class_eval & instance_eval are used to execute code in the context of class & instance respectively.



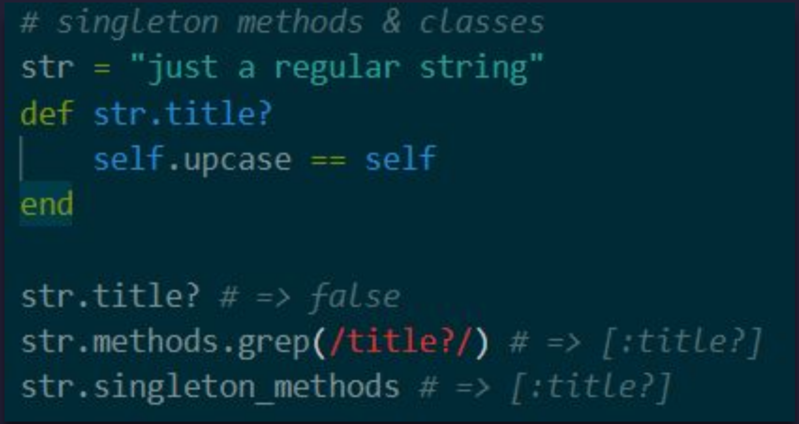
```ruby
# class_eval: manipulation at runtime
def add_method_to(a_class)
    a_class.class_eval do
        def m; 'Hello!'; end
    end
end

add_method_to String
"abc".m # => "Hello!"
```



```ruby
# instance_eval: execution in the context of object
class CleanRoom
    def current_temperature
        # ...
    end
end

clean_room = CleanRoom.new
clean_room.instance_eval do
    if current_temperature < 20
        # TODO: wear jacket
    end
end
```

Singleton methods & Classes

- A method specific to a single object instance is singleton method.
- A class, thus created, which is specific to a single object is called singleton class or eigen class

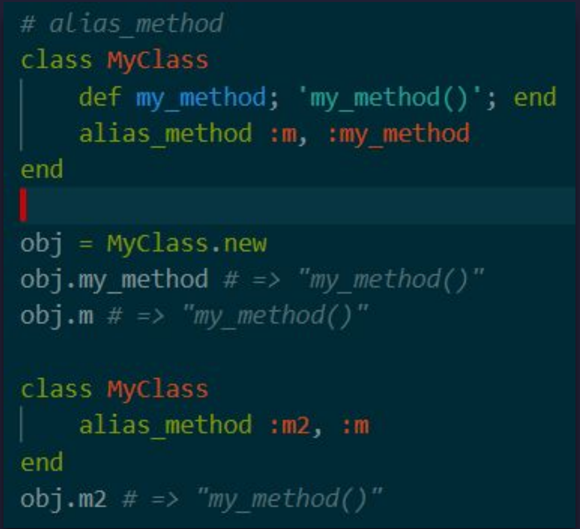


```
# singleton methods & classes
str = "just a regular string"
def str.title?
    self.upcase == self
end

str.title? # => false
str.methods.grep(/title?/) # => [:title?]
str.singleton_methods # => [:title?]
```

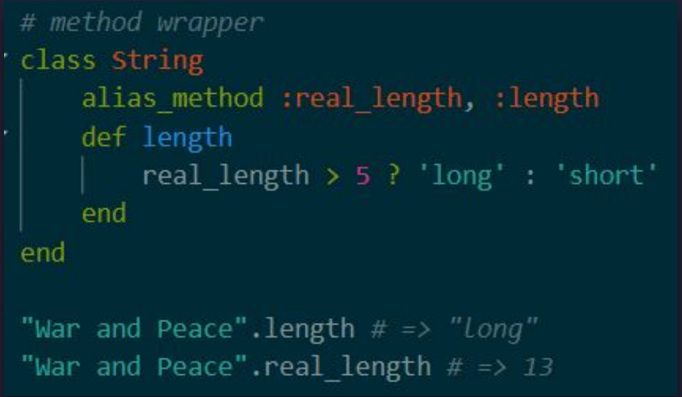
Aliases & Method Wrapper

Module#*alias_method* : gives alternate name to a Ruby method. *alias_method new_nm old_nm*

Wrapper: Wrapping additionally functionality around existing methods, without breaking dependent calls in code.



```ruby
# alias_method
class MyClass
    def my_method; 'my_method()'; end
    alias_method :m, :my_method
end

obj = MyClass.new
obj.my_method # => "my_method()"
obj.m # => "my_method()"

class MyClass
    alias_method :m2, :m
end
obj.m2 # => "my_method()"
```



```ruby
# method wrapper
class String
    alias_method :real_length, :length
    def length
        real_length > 5 ? 'long' : 'short'
    end
end

"War and Peace".length # => "long"
"War and Peace".real_length # => 13
```

Class Macros

Class macros are class methods that allow us to avoid code duplication & handle runtime events by using macros.

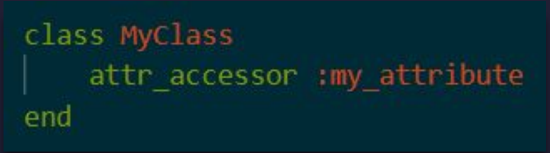
Class Macros

attr_accessor => defines getters + setters

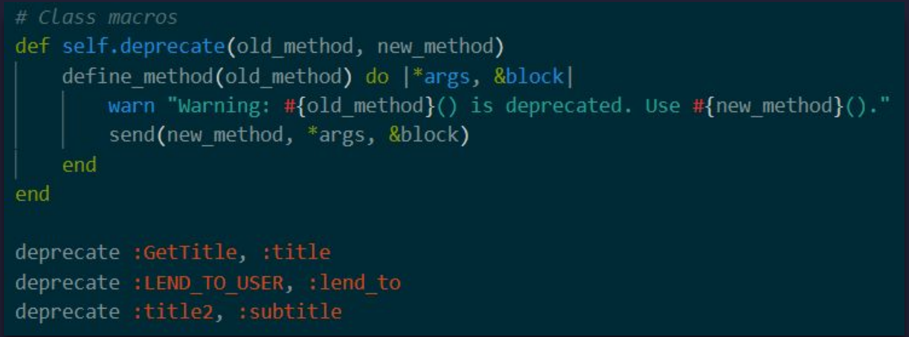
deprecate => catch calls to old_deprecated & send to new_method

attr_reader => generates getters only

attr_writer => generates setters only



```ruby
class MyClass
    attr_accessor :my_attribute
end
```



```ruby
# Class macros
def self.deprecate(old_method, new_method)
    define_method(old_method) do |*args, &block|
        warn "Warning: #{old_method}() is deprecated. Use #{new_method}()."
        send(new_method, *args, &block)
    end
end

deprecate :GetTitle, :title
deprecate :LEND_TO_USER, :lend_to
deprecate :title2, :subtitle
```

07

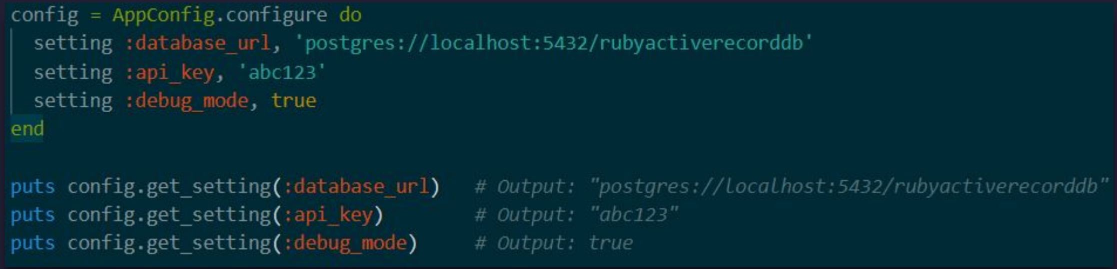
Practical Applications Of Meta Programming

Here, we discuss applications such as DSLs, Active Records, Sinatra, which internally rely on metaprogramming features discussed before.

DSLs

Domain-Specific Languages, are specialized languages designed to solve problems within a specific domain or context. They provide a higher level of abstraction, making it easier to express solutions.

In Ruby, DSLs are often implemented using metaprogramming features such as blocks and yield.

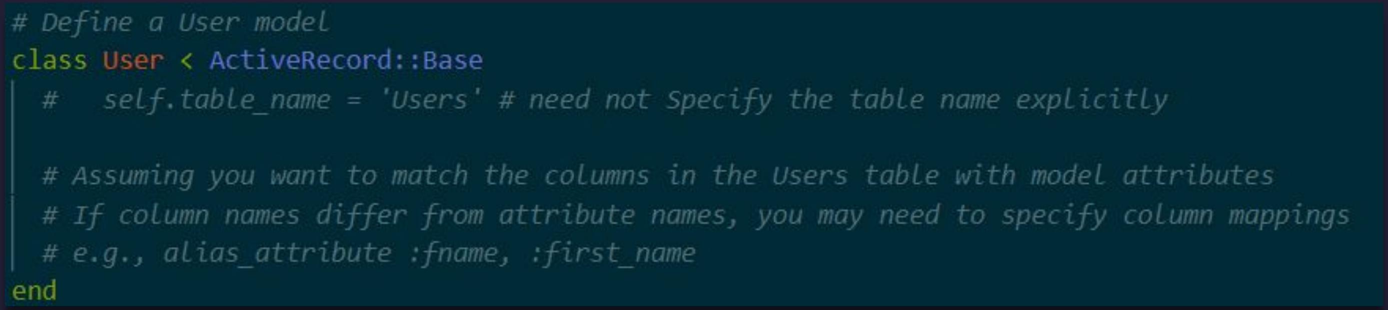


```ruby
config = AppConfig.configure do
  setting :database_url, 'postgres://localhost:5432/rubyactiverecorddb'
  setting :api_key, 'abc123'
  setting :debug_mode, true
end

puts config.get_setting(:database_url)   # Output: "postgres://localhost:5432/rubyactiverecorddb"
puts config.get_setting(:api_key)        # Output: "abc123"
puts config.get_setting(:debug_mode)     # Output: true
```

Active Records

ActiveRecord is an Object-Relational Mapping (ORM) library in Ruby on Rails

ActiveRecord leverages metaprogramming features in Ruby to dynamically generate code, reducing boilerplate code duplication



```ruby
# Define a User model
class User < ActiveRecord::Base
  #   self.table_name = 'Users' # need not Specify the table name explicitly

  # Assuming you want to match the columns in the Users table with model attributes
  # If column names differ from attribute names, you may need to specify column mappings
  # e.g., alias_attribute :fname, :first_name
end
```

Active Records

Here's how ActiveRecord leverages metaprogramming features:

1. **Dynamic Attribute Methods:** ActiveRecord dynamically generates attribute methods for each column in a database table. This is done using metaprogramming techniques like define_method, which generates getter and setter methods for attributes at runtime.

2. **Dynamic Query Methods**: ActiveRecord dynamically generates query methods for common database operations like find_by, where, order, etc. These methods are generated based on the names of columns in the database table, reducing the need to write custom SQL queries.

Sinatra: Web App Framework

Sinatra uses metaprogramming techniques to provide a concise and expressive Domain-Specific Language (DSL) for defining routes and application logic.

- **Route Definition:** Methods like get, post, put, and delete are not predefined methods. They are defined dynamically within the Sinatra::Application class using define_method syntax, unlike traditional frameworks.
- **Blocks as Handlers**: When you define a route with a block, Sinatra uses metaprogramming to convert that block into a callable method that handles the request. (Uses Procs)

Conclusion: Code Demo

Demo in laptop

Thank You !!!