# Per-object Behavior in Ruby*

## ** and why I love it*

David A. Black
Director
Ruby Power and Light, LLC

presented at
Ruby East 2007

# Why listen to me?

You seem to be obsessed with singleton methods.

*-- Ryan "ZenSpider" Davis, to me, c. 2006*

# Five questions about per-object behavior

1. What is its rationale?
2. How is it implemented?
3. How do you do it?
4. When should you do it?
5. What are its ramifications?

# 1.

What is its rationale?

# “It’s *object*-oriented, not class-oriented” -- Dave Thomas

- Classes launch objects into object-space
  - but the objects are what matters
- Classes don’t model the real world anyway
  - "I feel you're relying too much on inheritance hierarchy.  In Ruby, it's at best implementation sharing." -- Matz
- Objects reflect the character of Ruby
  - “Ruby is dynamic, like human nature” -- Matz

# 2.

How is it implemented?

# Singleton classes

- All methods live in a class or module
- Every object has a "birth class"
  - Every object (almost) can also have a "singleton class"
- Birth class can spawn other objects
  - but singleton class is for this object only
- Singleton class comes *first in the method look-up path*

# Singleton methods

- Singleton classes contain singleton methods
- Singleton methods can be called by one object only
  - with one exception (discussed later)

# The principle is what matters

- The implementation is secondary
- The main thing is: *Ruby objects are not constrained by their original classes*

# Matz has said as much

"I assume your question is why there's no method to obtain singleton class (per object hidden class), right? That's because I see no real 'need' for it.

"Besides seeing no real need, such method is highly connected to the current implementation."

*i.e., the current implementation of per-object behavior*

# In other words...

- Singleton classes are the *means, not the end*
- The end – the goal – is per-object behavior
- As long as this goal is met, and however it's implemented, Ruby is doing what it's supposed to.

# 3.

How do you do it?

# Leveraging per-object behavior

1. Add methods directly to the singleton class
2. Extend an object with one or more modules
   - ➔ equivalent to mixing the module into the object's singleton class

# Adding methods to singleton class

- `class` keyword lets you into singleton class context
- Inside the singleton class, you use `def`, just lik anywhere else

# Open the singleton class

The Person class

The singleton class of the david object

```
class Person
  def greet
    puts "Hello"
  end
end

david = Person.new

class << david
  def greet
    puts "Hi"
  end
end

david.greet
```

*The class keyword is happy with either a constant or an expre in the form: << objec*

# Alternative notation

```
class Person
  def greet
    puts "Hello"
  end
end

david = Person.new

def david.greet
  puts "Hi"
end

david.greet
```

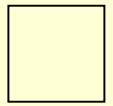
*Same as class << da*
*except for some issu*
*of constant-scoping*
*which*
*usually don't matter t*
*much.*

# Extend an object with a module

```
class Person
  def greet
    puts "Hello"
  end
end

module M
  def greet
    puts "Hi"
  end
end

david = Person.new

david.extend(M)

david.greet
```

# It's all about method lookup

```
class Person
  def greet
    puts "Hello"
  end
end

david = Person.new

david.greet
```

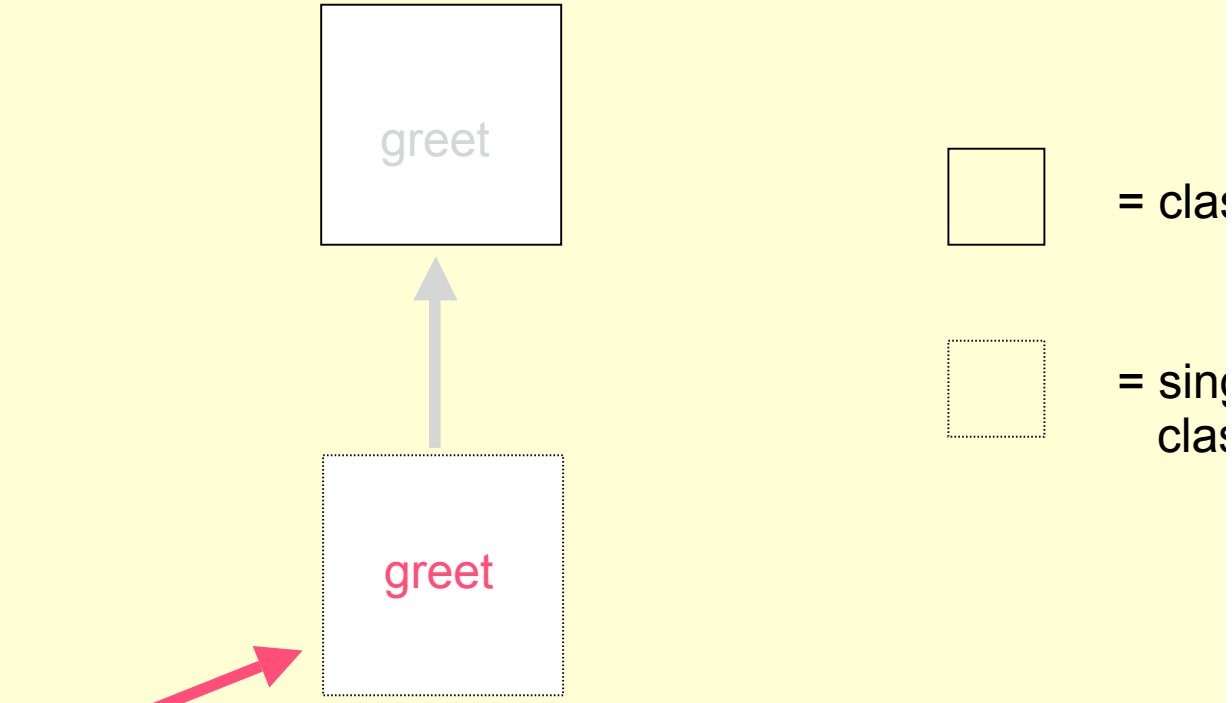



= cla

= sin
cla

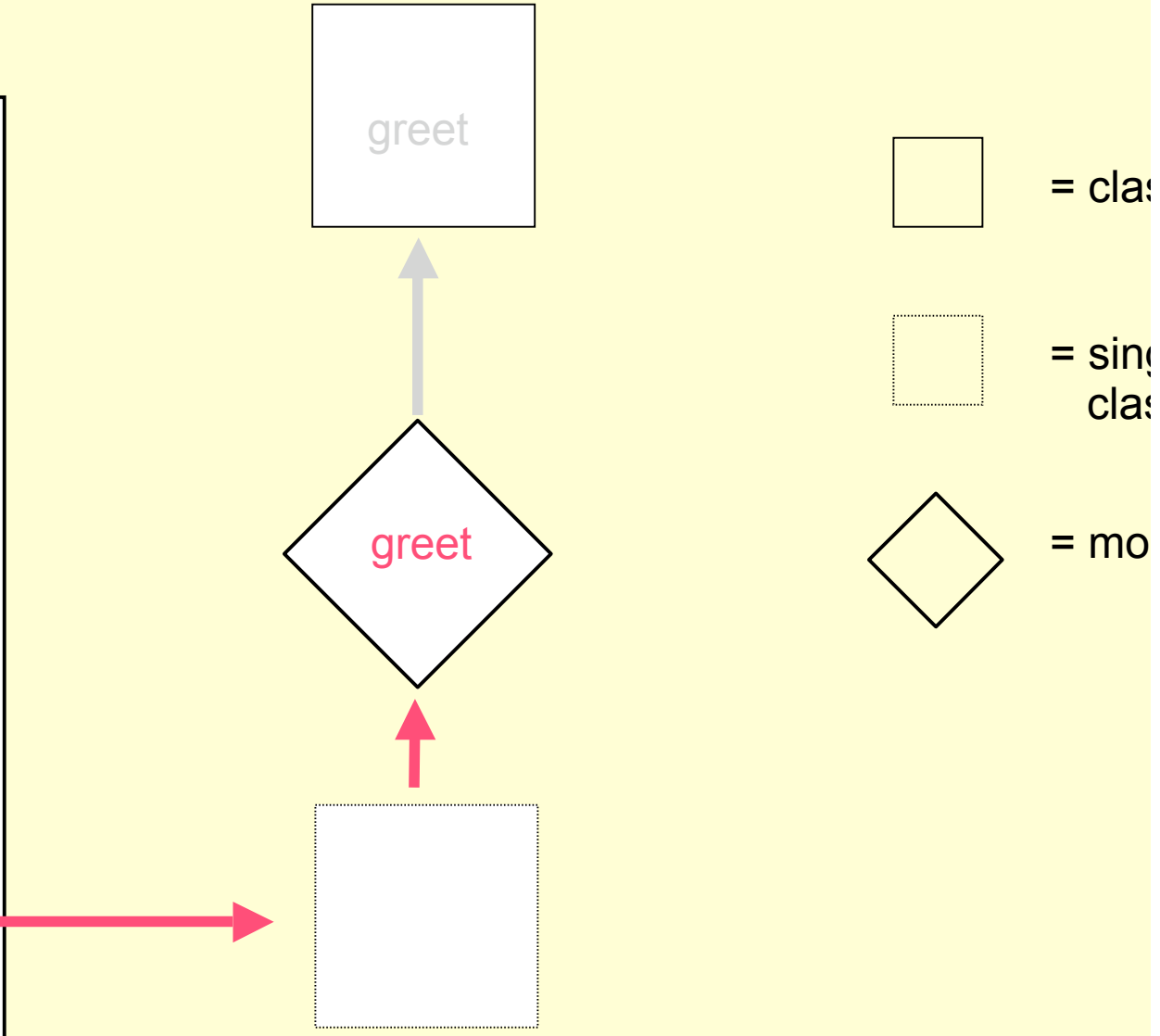
# It's all about method lookup

```
class Person
  def greet
    puts "Hello"
  end
end

david = Person.new

class << david
  def greet
    puts "Hi"
  end
end

david.greet
```



# It's all about method lookup

```
class Person
  def greet
    puts "Hello"
  end
end

module M
  def greet
    puts "Hi"
  end
end

david = Person.new

david.extend(M)

david.greet
```



# obj.extend(M)

```
class Person
  def greet
    puts "Hello"
  end
end

module M
  def greet
    puts "Hi"
  end
end

david = Person.new

david.extend(M)

david.greet
```

```
class Person
  def greet
    puts "Hello"
  end
end

module M
  def greet
    puts "Hi"
  end
end

david = Person.new

class << david
  include M
end

david.greet
```

# Singleton class/method limitations

Fixnums don't have singleton ("virtual") classes.

Floats don't have singleton methods.

Floats can have singleton classes, but each float is a different instance.

No singleton class for literals.

You can add methods to literals, but it's pointless since you can't call them!

```
>> class << 3; end
TypeError: no virtual class for Fixnum

>> class << 1.2; def x; end; end
TypeError: can't define singleton method "x" for Float

>> class << 1.2; A=1; end
=> 1
>> class << 1.2; p A; end
NameError: uninitialized constant Float::A

>> def "literal string".x; end
SyntaxError: compile error

>> class << "literal string"; def x; end; end
=> nil
```

# A word about naming...

- Most general term: singleton class
- Sometimes used: metaclass, eigenclass, virtua class....
  - "metaclass" sometimes just for singleton classes o class objects
    - not a good general term, since there's nothing particular "meta" about singleton classes
  - "virtual class" not used after Ruby 1.8.6
  - "eigenclass" and countless others have a following
- Much – *much! -- more can be found in ruby-tal archives*

# 4.

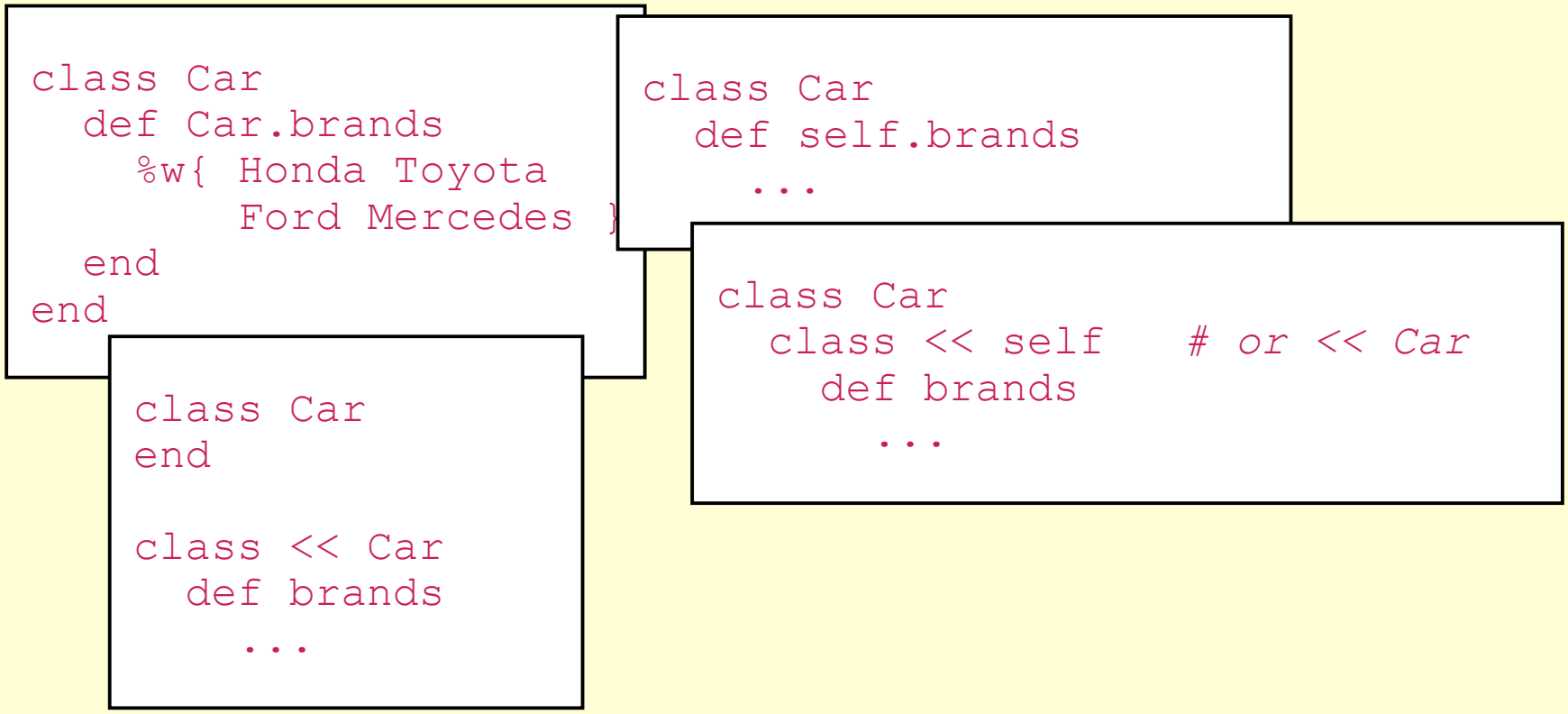
When should you do it?

# Major use cases include:

1. Class methods
2. To avoid changing core classes
3. Adding methods based on dynamically determined names

# Class methods

Class methods are singleton methods on class objects, and behave very much like singleton methods on other objects...



# Class methods

...with one slight anomaly.

```
class Sedan < Car
end

puts Sedan.brands
```

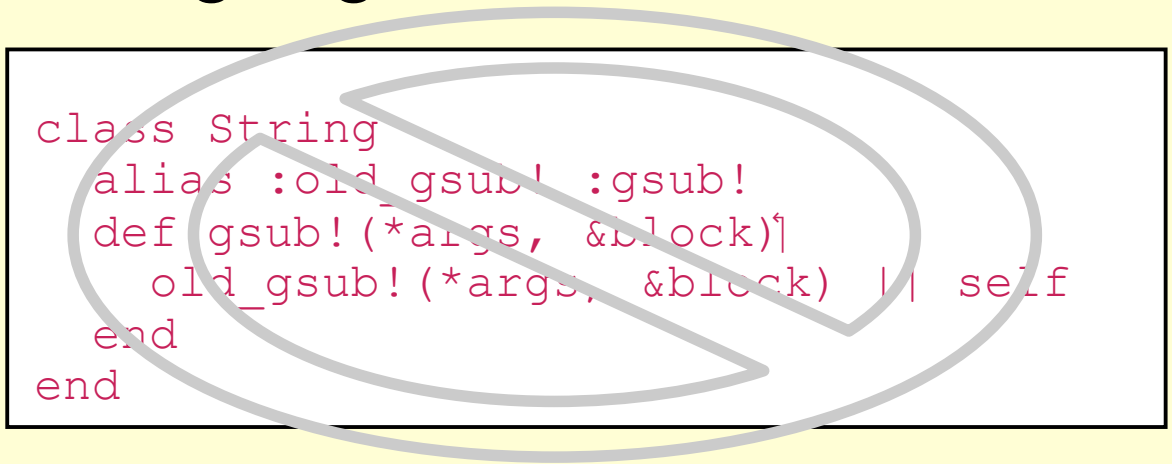
Subclasses can call the singleton methods (class methods) of their superclasses.

It's the only time that Object X can call the singleton methods of Object Y.

"Singleton" ceases to be, perhaps, the best term.

# Avoiding changes to core Ruby

- Changing core Ruby is almost always bad
- Looking grave and saying, "You shouldn't do this unless you understand the dangers" doesn't make it any better
- You affect things globally, including the rest of the core language and all libraries



# Enter per-object behavior

```
module GsubBangChanger
  def self.extended(obj)
    class << obj
      alias :old_gsub! :gsub!
      def gsub!(*args, &block)
        old_gsub!(*args, &block) || self
      end
    end
  end
end

string = "abc"
p string.gsub!(/z/,"y")

string.extend(GsubBangChanger)
p string.gsub!(/z/,"y")
```

nil

"abc"

# Dynamically adding methods

- Demo: emulating OpenStruct

# 5.

What are the ramifications?

# Class/Type == Nature/Nurture

- Type branches away from class
- Type is tautological
  - `obj` is of type: *The type that objects that can do what `obj` does are of*

*Which is why..*

- Class/module-checking (with `is_a?` or `kind_of?`) is not a robust way to determine object behavior
  - It isn't necessarily accurate
  - Even when it works, it sets unnecessary constraints
- Per-object behavior is part and parcel of duck typing
  - and "duck typing" is just a term for "letting Ruby be Ruby"

# Conclusion

- I love per-object behavior because it lies at the heart of Ruby
- Do not dismiss it as "wizardry", "running with scissors", etc.
  - Learn it and use it, like everything else, when it makes sense
- Do allow per-object behavior to trigger further learning about method lookup, singleton classes, and much of the rest of Ruby!

# Thank you!

David A. Black
Director
Ruby Power and Light, LLC

For Ruby & Rails training and consulting:
http://www.rubypal.com
dblack@rubypal.com

*(New Jersey-based! :-)*