MacRuby for Cocoa Developers

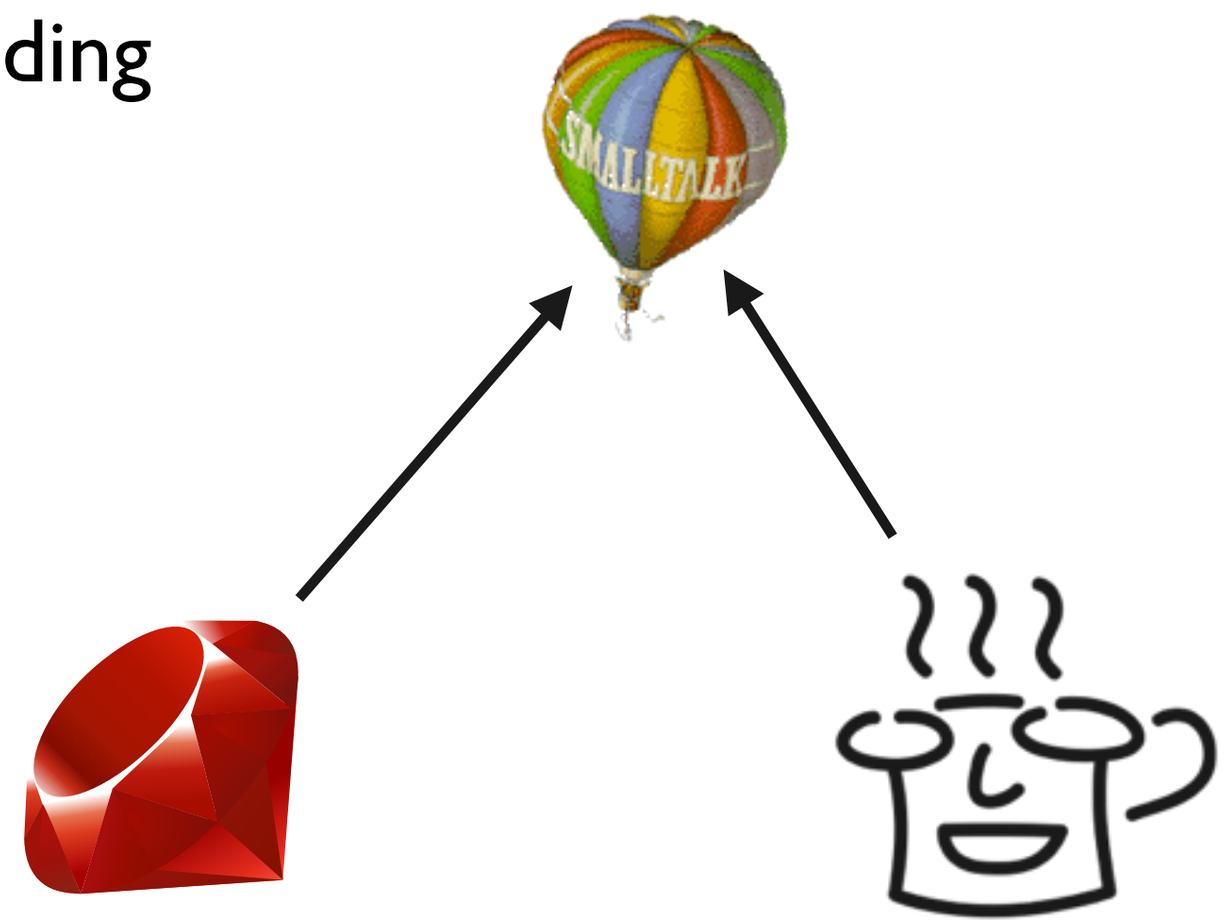
MacRuby for Cocoa Developers

Martijn Walraven
martijn@martijnwalraven.com

Ruby and Objective-C share a common ancestor in Smalltalk

Many similarities:

- Dynamic message sending
- Open classes
- Metaprogramming



In Ruby, everything is an object

- Can send messages to numbers
  - `10.times { print "Hello world" }`
- Can even define new methods on numbers
  - `7.lucky_number?`
- In most implementations, small integers (Fixnum) are immediate objects (tagged pointers)

Blocks in Ruby

- Every method invocation can have an associated block of code
  - `10.times {|i| print i }`
  -
    ```
    10.times do |i|
      print i
    end
    ```
- Blocks have access to the scope in which they are defined (e.g. they can keep accessing local variables)
- From within a method, access block either implicitly (`yield`) or through an explicit parameter

Blocks in Objective-C (≥10.6)

- Blocks are a C language extension introduced by Apple
  - `int multiplier = 7;`

```
int (^aBlock)(int) = ^(int number) {
    return number * multiplier;
};

printf("%d", aBlock(3));
```

- Every block is an Objective-C object (has `isa` pointer)

Blocks in Ruby and Objective-C compared

Objective-C:

```
[items enumerateObjectsUsingBlock:^(id item, NSUInteger index,
BOOL *stop) {
    NSLog(@"Item: %@", [item name]);
}];
```

Ruby:

```
items.each {|item| NSLog("Item: %@", item.name) }
```

Operators are ordinary methods

- Operators can be defined on any class
  - `x+y` ⇒ `x.+(y)`
- Even assignment is a method, making setters feel natural
  - `x.name = y` ⇒ `x.name=(y)`
- And so is indexing
  - `x[index]` ⇒ `x.[index]`
  - `x[index] = y` ⇒ `x.[index]=(y)`

And much more…

- Built-in support for regular expressions
  - `"Hello world".gsub(/[^aeiou]/) {|c| c.upcase }`
- Ranges and range literals
  - `(a..z).each {|letter| print letter }`

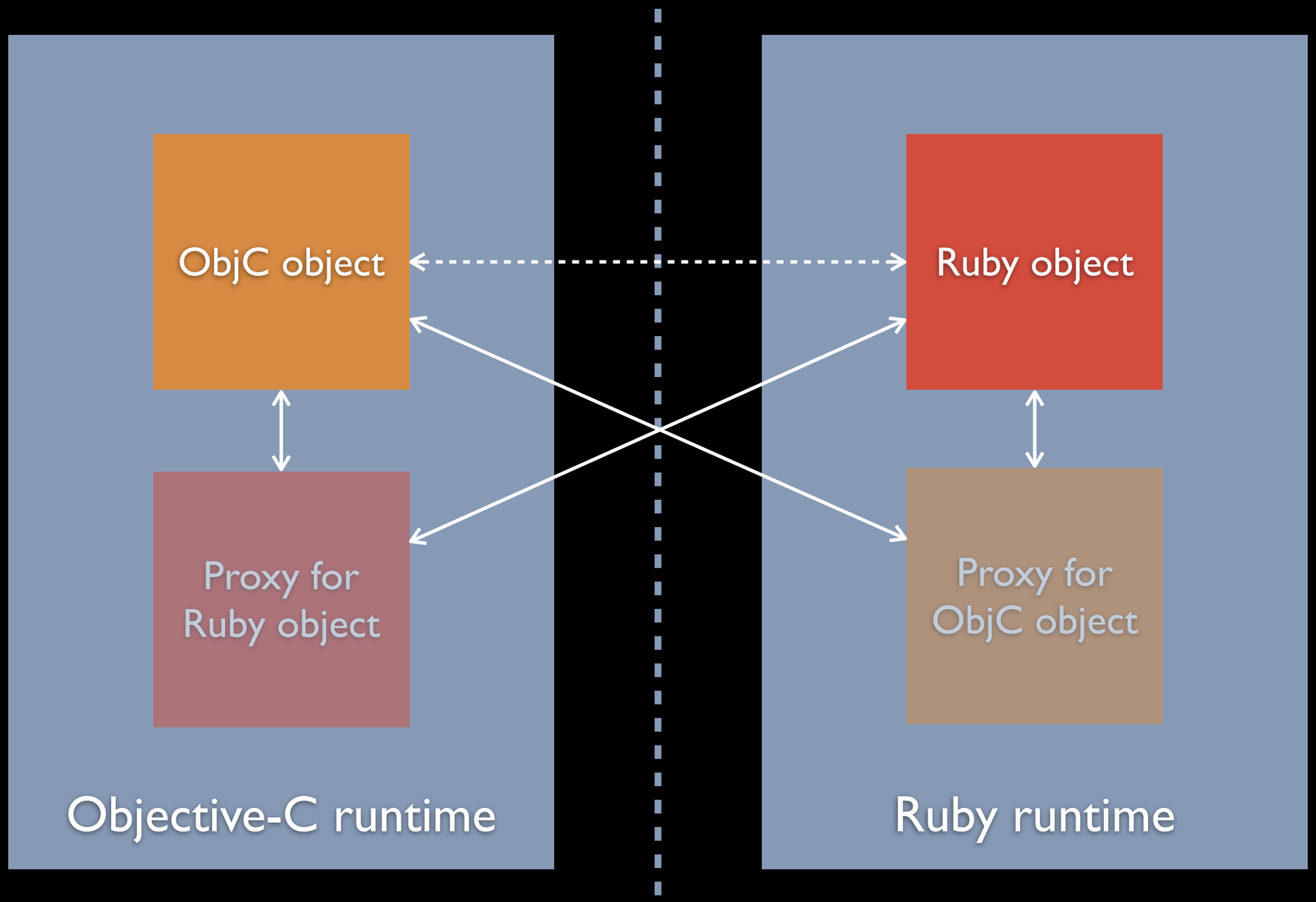
Many Ruby implementations

- Official Ruby 1.8 (MRI: Matz's Ruby Interpreter)
- Official Ruby 1.9 (YARV: Yet Another Ruby VM)
- JRuby (runs on Java VM)
- IronRuny (runs on .NET)
- Rubinius (custom VM, core classes written in Ruby)
- MagLev (based on Smalltalk VM with built-in persistence)

RubyCocoa

- Bridge between Ruby and Objective-C, manipulate Objective-C objects from Ruby, and vice-versa
- Based on Ruby 1.8
- Included in Mac OS X 10.5

Bridging the two environments



Bridges are expensive

- Two runtimes
  - Two garbage collectors
- Proxy objects
- Object conversions

MacRuby

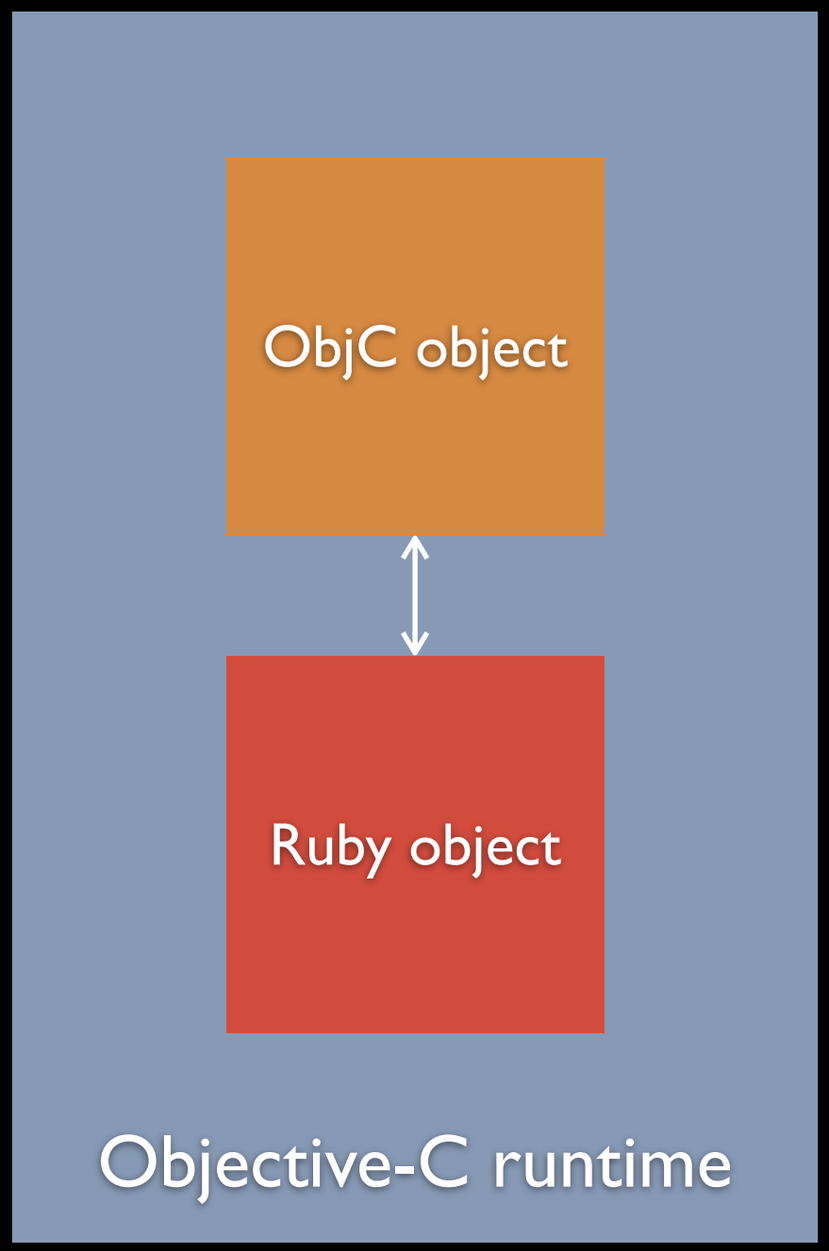
Open source project sponsored by Apple,
lead developer is Apple employee
Laurent Sansonetti

MacRuby

Posted by Laurent Sansonetti (Guest) on 28.02.2008 03:43

```
Hi,

I am honored to announce the beginning of the MacRuby project!

MacRuby is a version of Ruby that runs on top of Objective-C. More
precisely, MacRuby is currently a port of the Ruby 1.9 implementation
for the Objective-C runtime and garbage collector.

You can learn more about the project on its homepage:

http://trac.macosforge.org/projects/ruby/wiki/MacRuby

MacRuby is still extremely experimental, but a first release is
expected very soon.

Enjoy,
Laurent
```

MacRuby



- Fresh implementation of Ruby 1.9 based on the Objective-C runtime
- Every Ruby class is an Objective-C class
- Every Ruby object is an Objective-C object
- Every Ruby method is an Objective-C method

Objective-C runtime

- `objc_msgSend(receiver, selector, arg1, arg2, ...)`
- Method implementation is ordinary C function with two extra parameters (`self` and `_cmd`)
- Can add classes and methods at runtime
  - `objc_allocateClassPair()`, `objc_registerClassPair()`
  - `class_addMethod(class, selector, imp, types)`
- Dynamic method resolution
  - `+resolveClassMethod:`
  - `+resolveInstanceMethod:`

Core Ruby classes implemented on top of Cocoa classes

Object	NSObject
String	NSMutableString
Array	NSMutableArray
Hash	NSMutableDictionary
Numeric	NSNumber

Core Ruby classes implemented on top of Cocoa classes

- `"Hello world".class` ⇒ `NSMutableString`
- `["a", "b", "c"].class` ⇒ `NSMutableArray`
- `{"a"=>1, "b"=>2, "c"=>3}.class` ⇒ `NSMutableDictionary`
- `1.class.ancestors` ⇒ `[Fixnum, Precision, Integer, Precision, Numeric, Comparable, NSNumber, NSValue, NSObject, Kernel]`

Plays well with all of Cocoa

- Ruby Fixnum converted to C int, Float to double
- Boxing when necessary
  - e.g. Fixnum to NSNumber
- Uses BridgeSupport for C types, functions and constants
  - Treat some types as objects
    - `NSRect.new(NSPoint.new(0, 0), NSSize.new(100, 200))`
  - Convenience conversions built-in
    - `[100, 200]` ⇒ `NSPoint, NSSize`
    - `[0, 0, 100, 200]` ⇒ `NSRect`

No need for awkward syntax

- `[setObject:red forKey:color]` (Objective-C)
  `setObject_forKey_(red, color)` (RubyCocoa)
  `setObject(red, forKey:color)` (MacRuby)
- (Pet peeve: camel case vs. underscores)
- Heuristics for matching selectors
  - `fruit.color = red` (sends `setColor:red`)
  - `fruit.round?` (sends `isRound`)
- If the selector is not found, a Hash object is built and sent instead

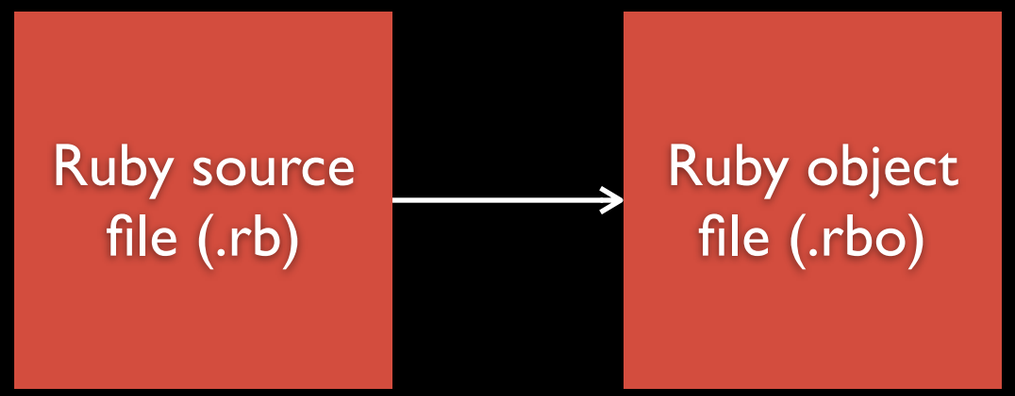
Out parameters

- Used by Cocoa for error reporting
  - `NSError *error = nil;`
    `BOOL success = [file doSomethingWith:anObject error:&error];`
- MacRuby has Pointer class
  - `errorPointer = Pointer.new_with_type(:object)`
    `success = file.doSomethingWith(anObject, error:errorPointer)`
    `error = errorPointer[0]`
- Multiple return values would be a more natural solution
  - `success, error = file.doSomethingWith(anObject)`

Fully concurrent threading

- No Global Interpreter Lock
  - Fully reentrant
- Every Ruby thread is a native thread
- Uses Objective-C garbage collector, which runs in a separate thread
- Recently added support for Grand Central Dispatch (will discuss in more detail later)

Compiler based on LLVM



- MacRuby supports both Just in Time (JIT) and Ahead of Time (AOT) compilation
- Even with AOT, the JIT is used for dynamic code
- No need to ship source

By no means finished, but pretty amazing already

- 0.5 just out
- Still many bugs and missing features
- Only runs on Intel, no PowerPC
- Mostly developed on 10.6
  - Need to compile yourself for 10.5
- Runs better in 64-bit than 32-bit

Does not run on iPhone OS (yet)

- No code interpretation allowed
- No garbage collection

So why would you want to program in Ruby?

Treating code as an essay

- Writing Ruby code feels like telling someone what the code is supposed to do
- With Objective-C, I constantly feel I'm repeating myself or am being overly wordy
- Ruby eliminates redundancies
  - Header files
  - Type information
  - Avoids unnecessary syntax

More descriptive is not always more readable

```
[aString stringByAppendingString:anotherString]
[aString concat:anotherString]
aString + anotherString
```

Arrays

Objective-C:

```
NSArray *fruits = [NSArray arrayWithObjects:apple, orange, banana, nil];
Fruit goodPotassiumSource = [fruits objectAtIndex:2];
NSArray *applesAndOranges = [fruits subarrayWithRange:NSMakeRange(0, 2)];
[fruits addObject:pear];
[fruits replaceObjectAtIndex:2 withObject:strawberry];
NSArray *fruitsAndNuts = [fruits arrayByAddingObjectsFromArray:nuts];
```

Ruby:

```
fruits = [apple, orange, banana]
goodPotassiumSource = fruits[2]
applesAndOranges = fruits[0..1]
fruits << pear
fruits[2] = strawberry
fruitsAndNuts = fruits + nuts
```

Dictionaries

Objective-C:

```
NSArray *keys = [NSArray arrayWithObjects:@"key1", @"key2", @"key3", nil];
NSArray *objects = [NSArray arrayWithObjects:@"value1", @"value2", @"value3", nil];
NSDictionary *dictionary = [NSDictionary dictionaryWithObjects:objects forKeys:keys];
```

Ruby:

```
dictionary = {"key1" => "value1", "key2" => "value2", "key3" => "value3"}
```

Ruby avoids unnecessary syntax

```
task :default => [:test]
task :test do
  ruby "test/unittest.rb"
end

task({:default => [:test]})
task(:test, &lambda(){
  ruby "test/unittest.rb"
})
```

Not a word too much

```
def decadeForYear(year)
  case year
    when 1970..1979 then "Seventies"
    when 1980..1989 then "Eighties"
    when 1990..1999 then "Nineties"
  end
end
```

Blocks again

Objective-C:

```
NSPredicate *redPredicate = [NSPredicate predicateWithBlock:^(id
evaluatedObject, NSDictionary *bindings) {
    return [[evaluatedObject color] isEqual:@"red";
}];

NSArray *redFruits = [fruits
filteredArrayUsingPredicate:redPredicate];
```

Ruby:

```
redFruits = fruits.select {|fruit| fruit.color == "red" }
```

Metaprogramming

Writing code that writes code,
or more generally does something at runtime
that is normally done through writing code

Metaprogramming

- Introspection
  - `-respondsToSelector:`
- Invoking methods dynamically at runtime
  - `[anObject performSelector:sel_getUid("doSomething")];`
  - `anObject.send("doSomething")`
- Responding to unknown messages
  - `-doesNotRecognizeSelector:`
  - `method_missing`
- Evaluating code at runtime
  - Not really possible in Objective-C
  - `define_method`, `eval` (in various forms)

```ruby
class Person
  attr_accessor :first_name, :last_name
end

def attr_accessor(*accessors)
  accessors.each do |attribute|

    define_method(attribute) do
      instance_variable_get("@#{attribute}")
    end

    define_method("#{attribute}=") do |value|
      instance_variable_set("@#{attribute}", value)
    end
  end
end
```

Metaprogramming in Objective-C

- Target-action mechanism
- Key-Value Coding and Key-Value Observing
  - `[fruit valueForKey:@"color"]` vs. `[fruit color]`
- Could be used much more

Domain Specific Languages

```
describe "The Array class" do
  it "is a direct subclass of NSMutableArray" do
    Array.class.should == Class
    Array.superclass.should == NSMutableArray
  end

  it "can be subclassed and later instantiated" do
    k = Class.new(Array)
    a = k.new
    a.class.should == k
    a << 42
    a[0].should == 42
  end
end
```

HotCocoa

```
application do |appl
  win = window :size => [100,50]
  b = button :title => 'Hello'
  b.on_action { puts 'World!' }
  win << b
end
```

Grand Central Dispatch

GCD — Objective-C

```
dispatch_queue_t waitingChairs = dispatch_queue_create("com.madebysofa.waitingChairs", 0);
dispatch_queue_t barber = dispatch_queue_create("com.madebysofa.barber", 0);
dispatch_semaphore_t semaphore = dispatch_semaphore_create((long)3);
NSInteger index = -1;

while (YES) {
    index++;
    long success = dispatch_semaphore_wait(semaphore, DISPATCH_TIME_NOW);
    if (success != 0) {
        NSLog(@"Customer turned away %i", index);
        continue;
    }
    dispatch_async(waitingChairs, ^{
        NSLog(@"Customer taking a seat %i", index);
        dispatch_async(barber, ^{
            dispatch_semaphore_signal(semaphore);
            NSLog(@"Shave and a haircut %i", index);
        });
    });
}
dispatch_release(waitingChairs);
dispatch_release(barber);
dispatch_release(semaphore);
```

GCD — MacRuby

```
# Create a new serial queue.
queue = Dispatch::Queue.new('org.macruby.examples.gcd')
# Synchronously dispatch some work to it.
queue.sync do
  puts 'Starting work!'
  sleep 1.0
  puts 'Done!'
end
# Asynchronously dispatch some work to it.
queue.async do
  puts 'Starting work!'
  sleep 1.0
  puts 'Done!'
end
```

GCD — MacRuby

```
class Future
  def initialize(&block)
    # Each thread gets its own FIFO queue upon which we will dispatch
    # the delayed computation passed in the &block variable.
    Thread.current[:futures] ||= Queue.new("futures-#{Thread.current.object_id}")
    @group = Group.new
    # Asynchronously dispatch the future to the thread-local queue.
    Thread.current[:futures].async(@group) { @value = block.call }
  end
  def value
    # Wait for the computation to finish. If it has already finished, then
    # just return the value in question.
    @group.wait
    @value
  end
end
```

Martijn Walraven
martijn@martijnwalraven.com