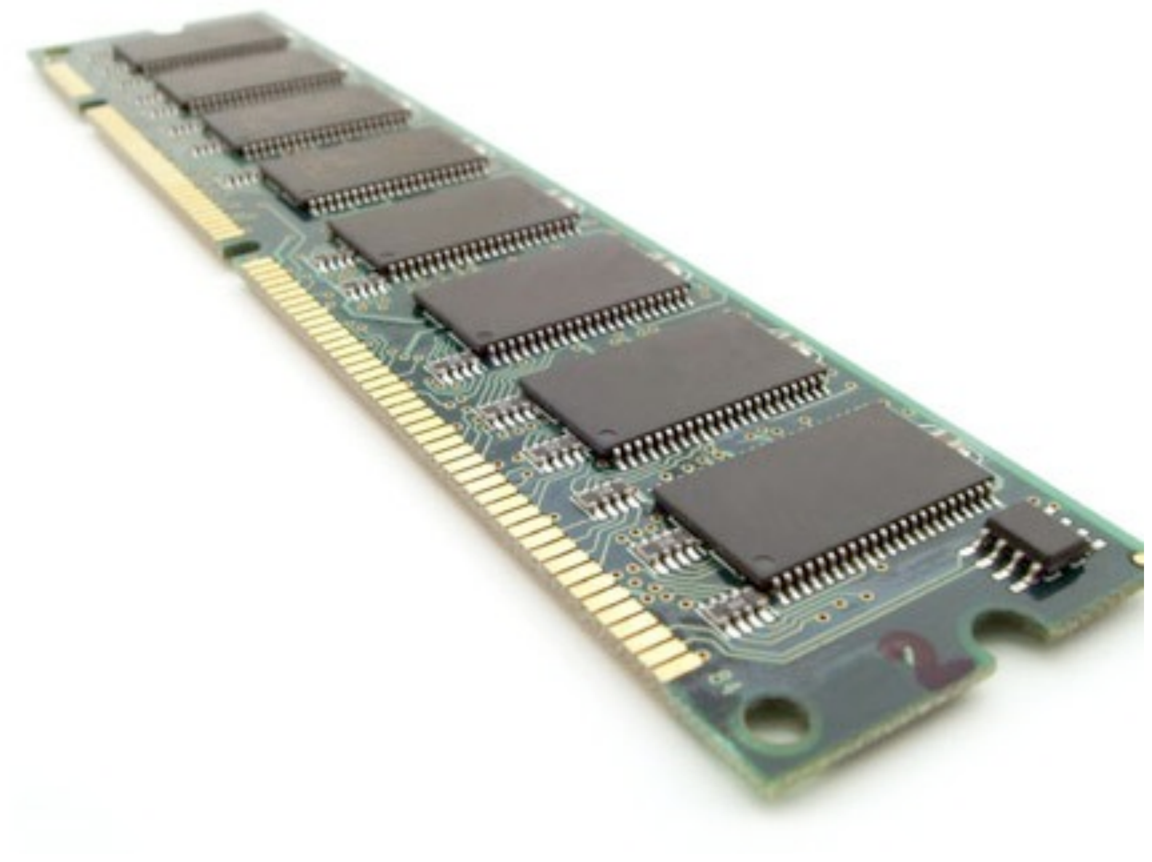
Objective-C Internals

André Pang
Realmac Software

Nice license plate, eh?

In this talk, we peek under the hood and have a look at Objective-C's engine: how objects are represented in memory, and how message sending works.

What is an object?



To understand what an object really is, we dive down to the lowest-level of the object: what it actually looks like in memory. And to understand Objective-C’s memory model, one must first understand C’s memory model...

What is an object?

int i;

i = 0xdeadbeef;

de	ad	be	ef

Simple example: here’s how an *int* is represented in C on a 32-bit machine. 32 bits = 4 bytes, so the int might look like this.

What is an object?

int i;

i = 0xdeadbeef;

ef	be	af	de

Actually, the int will look like this on Intel-Chip Based Macs (or ICBMs, as I like to call them), since ICBMs are little-endian.

What is an object?

```
int i;

i = 0xdeadbeef;
```

de	ad	be	ef

… but for simplicity, we'll assume memory layout is big-endian for this presentation, since it's easier to read.

What is an object?

```
typedef struct _IntContainer {
    int i;
} IntContainer;

IntContainer ic;

        ic.i = 0xdeadbeef;
```

de	ad	be	ef

This is what a C *struct* that contains a single int looks like. In terms of memory layout, a struct with a single int looks *exactly* the same as just an int[1]. This is very important, because it means that you can cast between an IntContainer and an int for a particular value with no loss of precision. This is also how Core Foundation and Cocoa's "toll-free bridging" works: a CFType has exactly the same memory layout as an NSObject.

1. Maybe there's something in the C language definition that may make this untrue on some stupid platform with a stupid compiler, but hopefully you're not coding for a DeathStation 9000.

What is an object?

```
typedef struct _NSPoint {
   CGFloat x;
   CGFloat y;
} NSPoint;
```

```
NSPoint p;
p.x = 1.0; p.y = 2.0;
```

3f	80	00	00	40	00	00	00

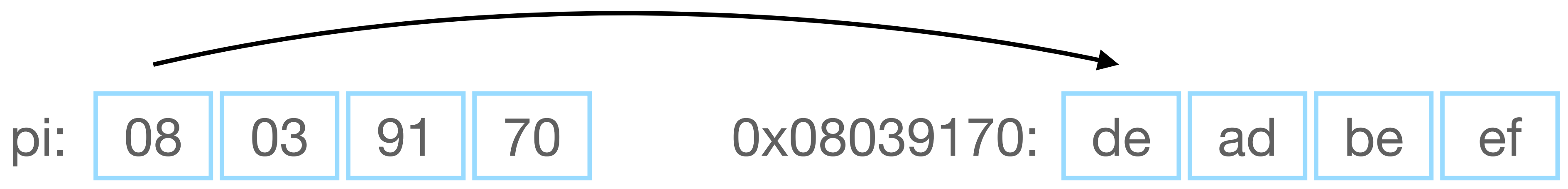
x: 1.0 y: 2.0

Here's a slightly more complex example that you'll have used a hundred times before: Cocoa's *NSPoint* struct, which consists of two *CGFloat*s. (A CGFloat is a typedef for a *float* on 32-bit platforms, or a *double* on 64-bit platforms.) So, what does a struct with more than one field look like? It's simply each value in memory one-after-another; i.e. contiguous (with some extra rules for value alignment and padding that are ultimately pretty boring, and not relevant for this talk).

(Note how the floats look completely different in memory to your more regular ints: if you're interested, two articles that are very worthwhile to read are "What Every Computer Scientist Should Know About Floating-Point Arithmetic", and Jim Blinn's much-harder-to-find-but-oh-so-worth-it "Floating-Point Tricks" article, which shows you how to do things such as square roots and powers on floats by using much faster bit arithmetic.)

What is an object?

```
int* pi;

*pi = 0xdeadbeef;
```



This is what a pointer looks like: it points to another location of a particular type in memory. (And who said pointers were hard?) In this case, *pi* here contains the hex value 0x08039170. The * operator *dereferences* the pointer and sets the value at the pointed-to memory address. So, in this case, the value at memory location 0x08039170 contains the actual int value that we want.

What is an object?

```
@interface NSObject <NSObject>     {
        Class                          isa;
}
```

Now we're equipped to look at an Objective-C class definition and talk about how objects are represented in memory. Type "open -h NSObject.h" into a Terminal, and it'll bring up the NSObject.h header file that contains the definition for the *NSObject* root object. This is what it looks like.

What is an object?

```
struct NSObject                          {
        Class                      isa;
}
```

The Objective-C *@interface* keyword is just a fancy way of declaring a struct with the same name, and also telling the compiler that the given name for it is an Objective-C class name. In other words, an NSObject is simply a struct with a single field, named *isa* (“is a”, as in “a car is a vehicle”) that points to some sort of *Class* type. But, what’s this Class thing?

(Note: this slide was not in the original Cocoaheads Sydney presentation, and has been added for clarification.)

What is an object?

```
struct NSObject                              {
        struct objc_class*     isa;
}
```

It turns out that Class is defined in <objc/objc.h> as a typedef for *struct objc_class**. In other words, an NSObject is simply a single pointer to an Objective-C class definition: that’s it.

What is an object?

```
struct NSObject                    {
        struct objc_class*     isa;
}
```

```
struct objc_class {
    Class isa;
    Class super_class;
    const char *name;
    long version;
    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
}
```

So, the next question is what an Objective-C class looks like. Search around in the */usr/include/objc/* directory on your Mac and you'll find that it looks like this[1]. This contains all the information that the Objecitve-C runtime needs to do absolutely anything with the object: find out what protocols it conforms to, what methods it has, the layout of the object's ivars (instance variables) in memory, what its superclass is, etc.

One interesting thing is that the first field of the *objc_class* struct is the same type as the single field in the NSObject struct. This means that an objc_class *is* an object, because its memory model is the same; therefore, all of the Objective-C operations—such as message sending—that work on instance objects work on class objects too. This increases *uniformity*, which means (much) less special-case code to distinguish between class objects and instance objects. But then what does this class object's *isa* field point to?

The class object's isa field points to something called a *metaclass* object, which, as the type of the field indicates, is just another objc_class struct. Every class definition therefore has a class object and a metaclass object. The rationale for this is that a class object's list of methods are for *instances of that class*; i.e. the class object's *methodLists* field contains information about *instance methods*. The metaclass object's methodLists field then contains information about *class methods*. Again, this increases uniformity, and reduces the need for special-case code. Of course, the next question is what the metaclass object's isa field points to: does it point to a metametaclass object? As it turns out, since there's no such thing as metaclass methods, there's no need for a metametaclass object, so the metaclass object's isa field simply points to itself, terminating the cycle.

1. In the actual header file, you'll see that this struct layout has been deprecated in Objective-C 2.0, to make the struct opaque. This enables the Objective-C engineers to modify the layout and add/remove fields to it; instead of accessing the values of the struct directly via *myClass->name*, you simply use functions such as *class_getName()* and *class_setName()* instead. At the lowest-level, though, even the Objective-C 2.0 definition of an objc_class struct will look very similar to this.

What is an object?

```
struct NSObject                                {
        struct objc_class*     isa;
}
```

```
struct objc_class {
    Class isa;
    Class super_class;
    const char *name;
    long version;
    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
}
```

```
(gdb) p NSApp
    $2 = (struct objc_object *) 0x8039170
(gdb) p *NSApp
    $3 = {
      isa = 0x15f8e0
    }
(gdb) p NSApp->isa
    $4 = (struct objc_class *) 0x15f8e0
(gdb) p *NSApp->isa
    $5 = {
      isa = 0x160de0,
      super_class = 0x22d3ea0,
      name = 0x1322de "RWApplication",
      version = 0,
      info = 12206145,
      instance_size = 100,
      ivars = 0x169720,
      methodLists = 0x80391e0,
      cache = 0x809d710,
      protocols = 0x15b064
    }
```

As proof, let's dive into *gdb* and peek at the *NSApp* global variable in a running application. First, you'll see that NSApp is, indeed, just a pointer to an objc_object struct. (Remember that in Objective-C, all object references are pointers.) Dereferencing NSApp shows that it does indeed have an isa field, and that isa points to a class object is at a memory address of 0x15f8e0. Dereference that, and you can start seeing details about the class, such as the size of one of its instances and what the name of the class is. Here, we presume that *NSApp->isa->isa is the RWApplication metaclass, and *NSApp->isa->superclass is the NSApplication class, which RWApplication subclasses.

What is an object?

```
@interface NSObject {
        Class                    isa;
}

@interface MySubclass : NSObject {
        int i;
}

@interface MySubsubclass : MySubclass {
        float f;
}
```

```
struct NSObject        {
    Class isa;
}

struct MySubclass      {
    Class isa;
    int i;
}

struct MySubsubclass  {
    Class isa;
    int i;
    float f;
}
```

For subclasses, each ivar is simply appended to the end of the subclass that it inherits from. In this example, MySubsubclass inherits from MySubclass which inherits from NSObject, so, in order, it contains all the ivars in NSObject first, then all the ivars in MySubclass, and then the ivars of its own class definition.

(Note: this slide was not in the original Cocoaheads Sydney presentation, and has been added for clarification.)

Messaging

Given what we know now about what objects really look like in memory, let’s talk about the fun stuff: message sending.

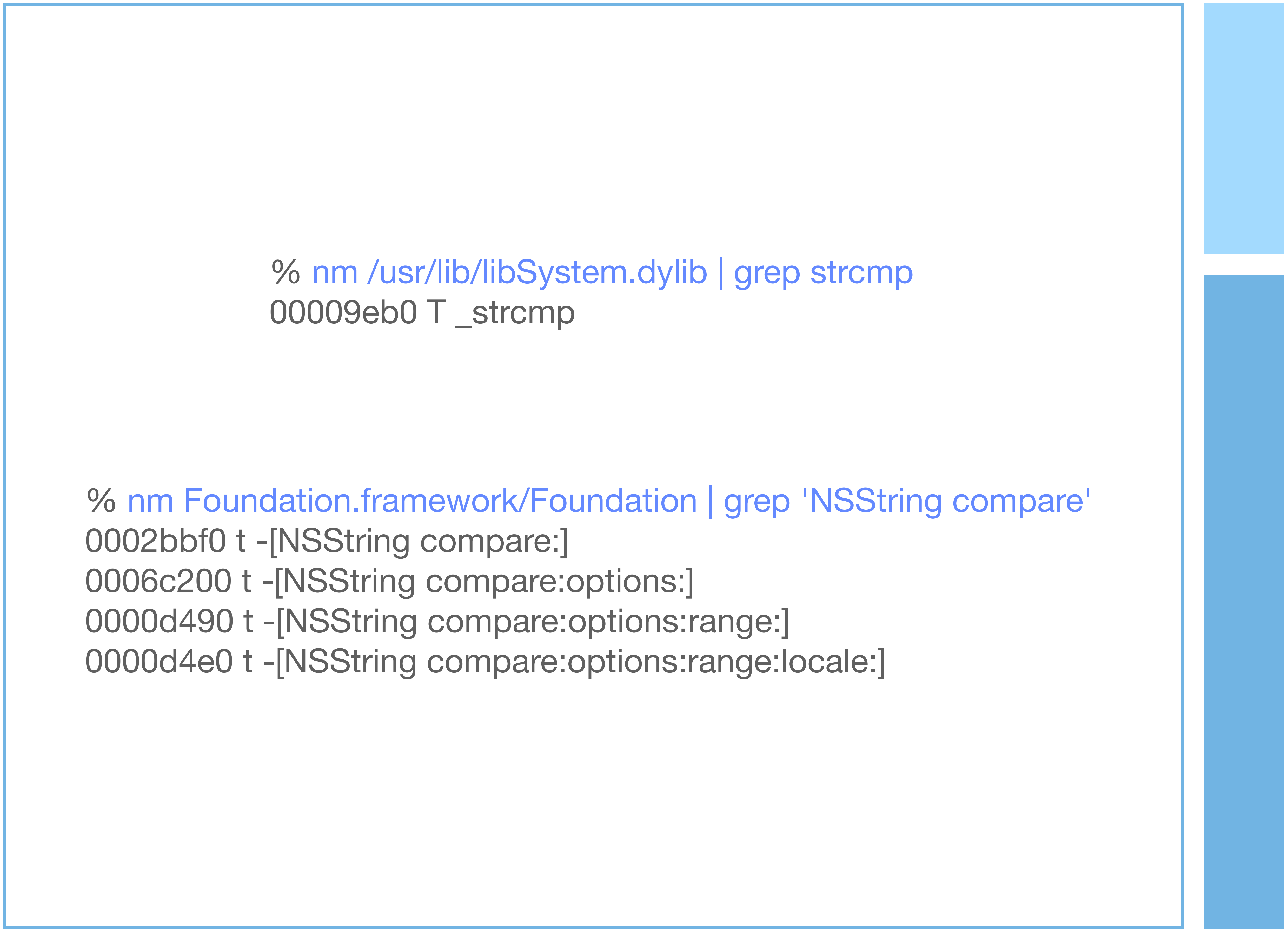
```
@implementation NSMutableString

- (void)appendString:(NSString*)aString
{
        ...;
}

@end
```

IMP

```
void -[NSMutableString appendString:](id self, SEL _cmd, NSString* aString)
{
        ...;
}
```

First, what is a method? Analogous to how we discussed Objective-C objects in terms of C, let's talk about Objective-C methods in terms of C. When you write a method definition between an *@implementation...@end* block in your code, the compiler actually *transforms* that into a standard C *function* (transforms, ha ha, get it?). The only two different things about this C function are that (1) it takes two extra arguments—*self* and *_cmd*—and (2) the function name has some characters that are normally disallowed in C functions (-, [and]). Other than (2), though, it really is a completely standard C function, and if you can somehow get a function pointer to that function, you can call it just like you would any other standard C function. The two extra arguments are how you can access the "hidden" self and _cmd variables inside the method. (Everyone uses self, but _cmd also exists, and you can use it to do some pretty funky things.)

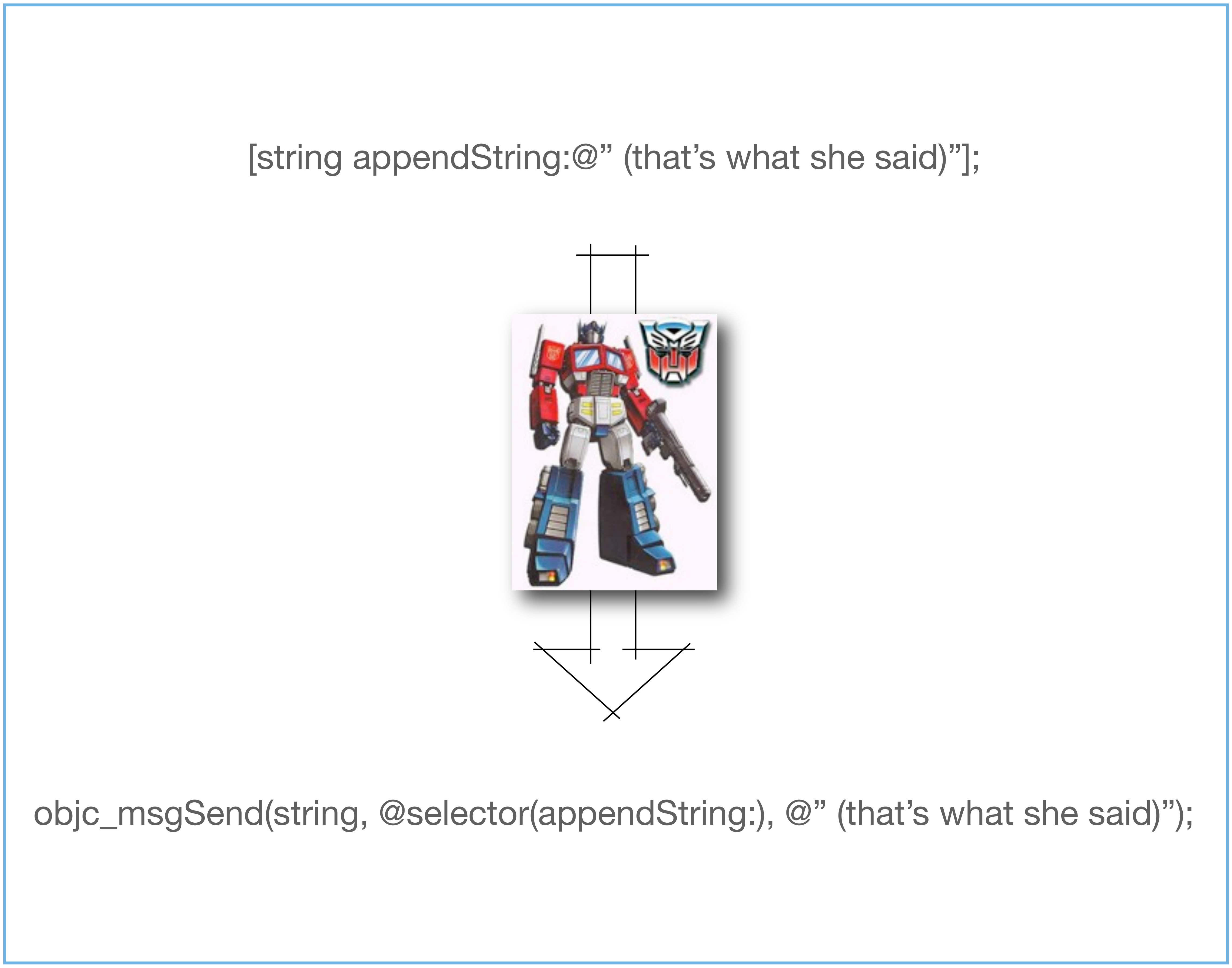
Note that in Objective-C lingo, the actual C function that is the method implementation is called an *IMP*. We'll be using this later on.



```
% nm /usr/lib/libSystem.dylib | grep strcmp
00009eb0 T _strcmp
```

```
% nm Foundation.framework/Foundation | grep 'NSString compare'
0002bbf0 t -[NSString compare:]
0006c200 t -[NSString compare:options:]
0000d490 t -[NSString compare:options:range:]
0000d4e0 t -[NSString compare:options:range:locale:]
```

And, just as proof, if you dump the symbols in a binary using the command-line *nm* tool, you can see that Objective-C methods really are standard C functions, just with special names. On Mac OS X, C functions have a _ prepended to their symbol name, but in comparison, the C names of the Objective-C methods do not.



```
[string appendString:@” (that’s what she said)”];
```

```
objc_msgSend(string, @selector(appendString:), @” (that’s what she said)”);
```

Now, what happens when you use the [...] syntax to send a message to an object? The compiler actually transforms that into a call to a function named *objc_msgSend()* that’s part of the Objective-C runtime[1]. objc_msgSend() takes at least two arguments: the object to send the message to (*receiver* in Objective-C lingo), and something called a *selector*, which is simply jargon for “a method name”.

Conceptually, you can think of a selector as simply a C string. In fact, a selector *is* a C string: it has the same memory model as a C string—NUL-terminated char* pointer—just as our IntContainer struct is the same memory model as a simple int. The only difference between a selector and a C string is that the Objective-C runtime ensures that there’s only one unique instance of each selector—i.e. one unique instance of each method name—in the entire memory address space. If you simply used char*s for method names, you could have two char*s that both had a value of “appendString:”, but residing at different memory addresses (e.g. 0xdeadbeef and 0xcafebabe). This means that testing whether one method name is equal to another method name requires a *strcmp()*, doing a character-by-character comparison that’s hilariously slow when you want to simply perform a function call. By ensuring that there is only one unique memory address for each selector, selector equality can simply be done by a pointer comparison, which is far quicker. As a result of this, selectors have a different type (*SEL*) to a char*, and you need to use the *sel_registerName()* function to “convert” a C string to a selector.

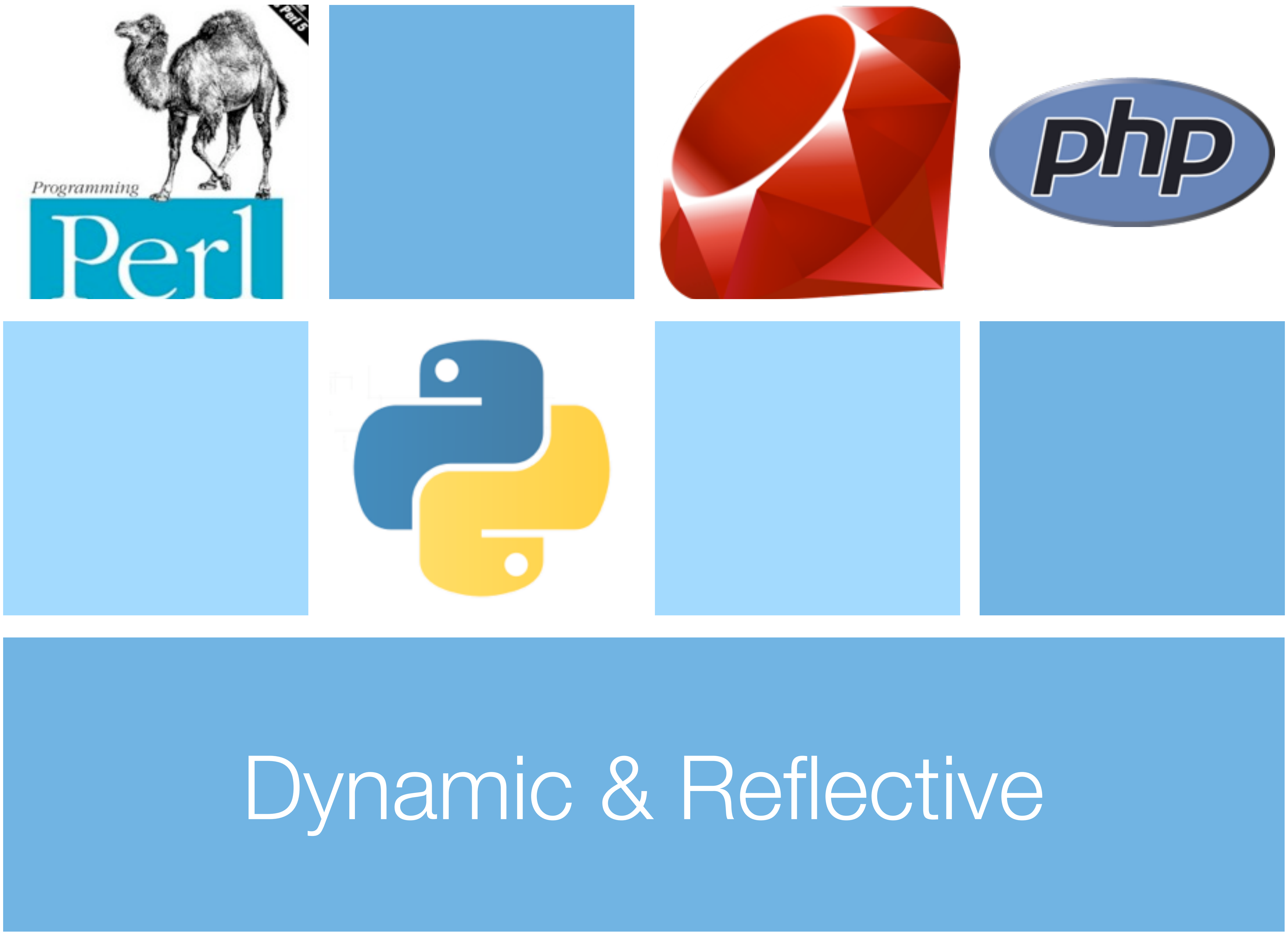
Note that objc_msgSend() is a varargs function, where the rest of the parameters after the first two are the message parameters.

1. The Objective-C runtime is simply a C library named *objc*; you can link to it as you would any other C library, and it resides at */usr/lib/libobjc.dylib*.

```
IMP class_getMethodImplementation(Class cls, SEL name);

id objc_msgSend(id receiver, SEL name, arguments…)
{
     IMP function = class_getMethodImplementation(receiver->isa, name);
     return function(arguments);
}
```

```
objc_msgSend(string, @selector(appendString:), @” (that’s what she said)”);
```

So, what would an implementation of objc_msgSend() look like? Conceptually, it may look similar to this, although in practice, it’s hand-rolled, highly optimised assembly because it’s a function that needs to very fast. The Objective-C runtime has a method named *class_getMethodImplementation()* that, given a class object and a selector, returns the IMP—the C function implementation—for that method. It does this by simply looking up the class’s method list and finding the selector that matches the one you’ve passed it, and returns the IMP that matches the selector. Now that you have an IMP, and an IMP is actually just a C function pointer, you can call it just like you would any other C function. So, all *objc_msgSend()* does is grab the receiver’s class object via the isa field, finds the IMP for the selector, and bang, we’ve got message sending. That’s really it: no more black magic.

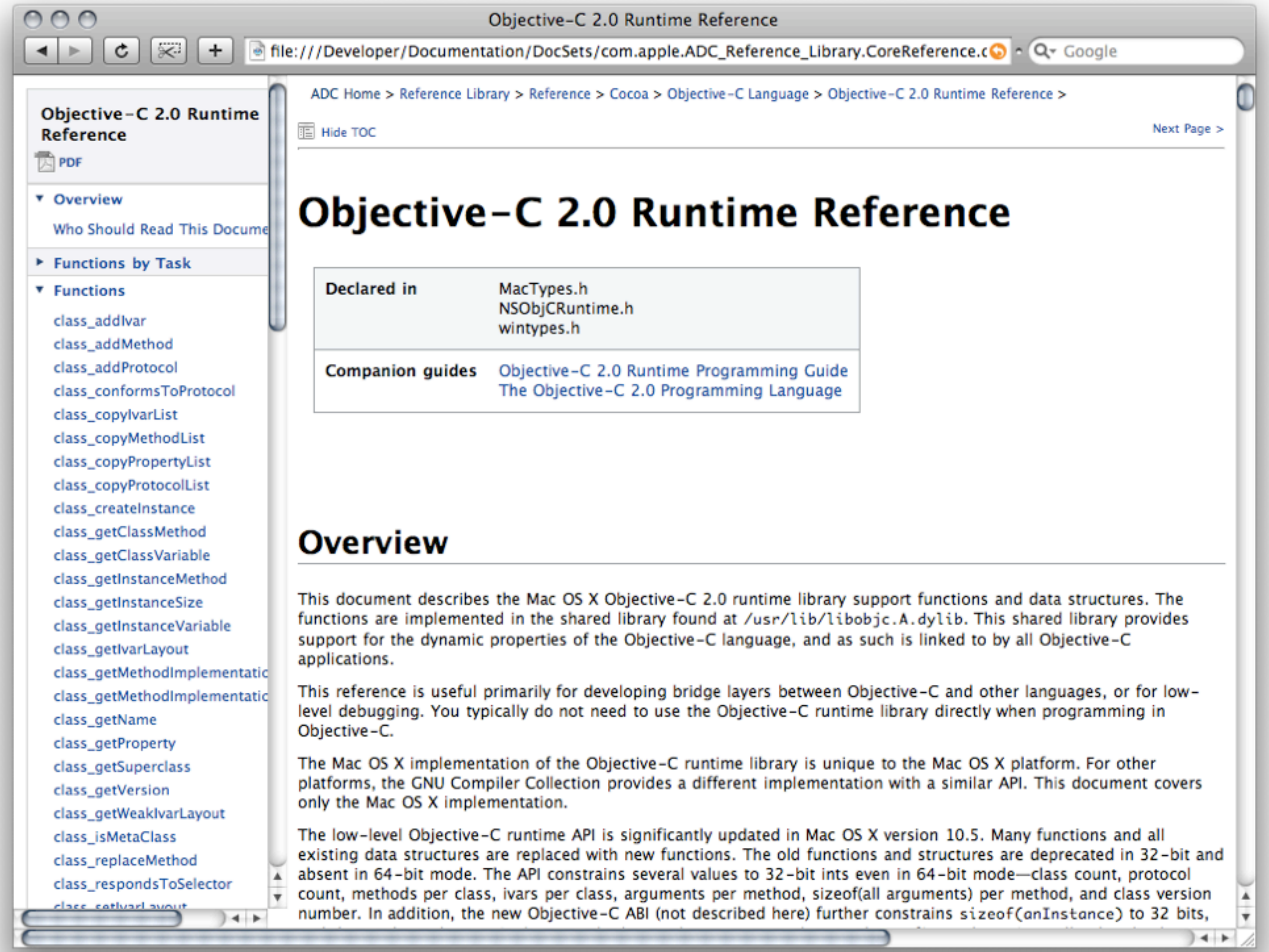


Dynamic & Reflective

The Objective-C memory model and message sending semantics has some very interesting consequences. First, since the class object contains all the information about what methods it implements, what the name of the class is, etc, and that information is accessible via the Objective-C runtime APIs, the language is *reflective* (a.k.a. *introspective*), meaning you can find out information about the class hierarchy. It's possible to get information about the entire class hierarchy and find out how many total methods there are, for example.

Second, since APIs are available to modify the class objects and since message-sending is tunnelled through a single C function, the language is highly *dynamic*. You can do things such as add classes at runtime and even swap pre-defined method implementations with your own ones. Objective-C messaging also allows objects to have a "second chance" to respond to messages: if you send an object a message that it doesn't understand, the runtime invokes a method named -*forwardInvocation:* (and also *+resolveInstanceMethod:* in Objective-C 2.0) that is passed information about the message you wanted to send, and the object can then do whatever it likes with the message, such as forward it on to another object.

These capabilities puts Objective-C in the same league as the more well-known dynamic languages and scripting languages, such as Perl, Python, Ruby, PHP, and JavaScript: the main difference is that Objective-C is compiled to native code. (No pedantry about JIT engines here, please.) It's pretty much just as dynamic as the rest of the scripting languages, though. For comparison, C++ has basically no introspection (only the RTTI) and no dynamism, while Java is similar to Objective-C (reflective & dynamic) except that it doesn't have an equivalent to *-forwardInvocation:*, and has far more verbose reflection APIs. COM & CORBA are more-or-less C++ with Objective-C-like features shovelled on top of it to have some dynamism & reflection, except that it's butt-ugly and it sucks.


Objective-C 2.0 Runtime Reference

file:///Developer/Documentation/DocSets/com.apple.ADC_Reference_Library.CoreReference.c

ADC Home > Reference Library > Reference > Cocoa > Objective-C Language > Objective-C 2.0 Runtime Reference >

Hide TOC | Next Page >

Objective-C 2.0 Runtime Reference
PDF

- Overview
  - Who Should Read This Docume
- Functions by Task
- Functions
  - class_addIvar
  - class_addMethod
  - class_addProtocol
  - class_conformsToProtocol
  - class_copyIvarList
  - class_copyMethodList
  - class_copyPropertyList
  - class_copyProtocolList
  - class_createInstance
  - class_getClassMethod
  - class_getClassVariable
  - class_getInstanceMethod
  - class_getInstanceSize
  - class_getInstanceVariable
  - class_getIvarLayout
  - class_getMethodImplementatic
  - class_getMethodImplementatic
  - class_getName
  - class_getProperty
  - class_getSuperclass
  - class_getVersion
  - class_getWeakIvarLayout
  - class_isMetaClass
  - class_replaceMethod
  - class_respondsToSelector

Objective-C 2.0 Runtime Reference

Declared in	MacTypes.h NSObjCRuntime.h wintypes.h
Companion guides	Objective-C 2.0 Runtime Programming Guide The Objective-C 2.0 Programming Language

Overview

This document describes the Mac OS X Objective-C 2.0 runtime library support functions and data structures. The functions are implemented in the shared library found at `/usr/lib/libobjc.A.dylib`. This shared library provides support for the dynamic properties of the Objective-C language, and as such is linked to by all Objective-C applications.

This reference is useful primarily for developing bridge layers between Objective-C and other languages, or for low-level debugging. You typically do not need to use the Objective-C runtime library directly when programming in Objective-C.

The Mac OS X implementation of the Objective-C runtime library is unique to the Mac OS X platform. For other platforms, the GNU Compiler Collection provides a different implementation with a similar API. This document covers only the Mac OS X implementation.

The low-level Objective-C runtime API is significantly updated in Mac OS X version 10.5. Many functions and all existing data structures are replaced with new functions. The old functions and structures are deprecated in 32-bit and absent in 64-bit mode. The API constrains several values to 32-bit ints even in 64-bit mode—class count, protocol count, methods per class, ivars per class, arguments per method, sizeof(all arguments) per method, and class version number. In addition, the new Objective-C ABI (not described here) further constrains `sizeof(anInstance)` to 32 bits,

You can get more information on the Objective-C runtime by looking at Apple's "Objective-C 2.0 Runtime Reference" documentation. As you can see on the left of the slide, there's an extensive amount of C functions you can call to peek & poke at the runtime.

RMModelObject

```
@interface MyBlogEntry : NSObject
{
    NSString* _title; NSCalendarDate* _postedDate; …
}

@property (copy) NSString* title;
@property (copy) NSCalendarDate* postedDate;
@property (copy) NSAttributedString* bodyText;
@property (copy) NSArray* tagNames;
@property (copy) NSArray* categoryNames;
@property BOOL isDraft;

@end
- (NSString*)title; (void)setTitle:(NSString*)value; …
- (BOOL)isEqual:(id)other;
- (id)copyWithZone:(NSZone*)zone;
- (id)initWithCoder:(NSCoder*)decoder;
- (void)encodeWithCoder:(NSCoder*)encoder;
- (void)dealloc;
```

Apart from just being cool, having a highly dynamic runtime can actually be useful. Ruby is famous for making extensive use of metaprogramming techniques to do "cool stuff" such as ActiveRecord. Here, we utilise all of Objective-C's reflective and dynamic features in a class named RMModelObject. It's designed to write simple model classes.

For example, if you're writing a website building program that may or may not rhyme with WapidReaver and want to model a blog entry, you might have a few properties such as a *title*, a *postedDate*, the *bodyText*, *tagNames* for the entry, etc. However, then you realise that you have to (1) declare ivars to serve as backing stores for the properties, (2) write accessors for all the properties, (3) implement -isEqual: and -hash, (4) write -copyWithZone:, (5) write -initWithCoder: and -encodeWithCoder: for NSCoding support, and (6) write dealloc to property release all the ivars. Not so much fun now, huh?

RMModelObject

```
@interface MyBlogEntry : RMModelObject
{

}

@property (copy) NSString* title;
@property (copy) NSCalendarDate* postedDate;
@property (copy) NSAttributedString* bodyText;
@property (copy) NSArray* tagNames;
@property (copy) NSArray* categoryNames;
@property BOOL isDraft;

@end

@implementation MyBlogEntry

@dynamic title, postedDate, bodyText, tagNames, categoryNames, isDraft;

@end
```

With RMModelObject, all you have to do is declare your properties as *@dynamic* in your @implementation context, and only then to silence some compiler warnings. That's it: RMModelObject does the rest. It dynamically creates a new class for you, adds ivars and accessor methods to the class, and implements the other required methods such as -initWithCoder:/-encodeWithCoder:. If you want a nice use-case and some example code for how to play with the Objective-C 2.0 runtime APIs, RMModelObject may be a good case study.

Other very cool projects that take advantage of the Objective-C 2.0 runtime are an Objective-C port of ActiveRecord, and SQLitePersistentObjects. Check 'em out.

Higher-Order Messaging

And now, I finally get to say my two favourite words: “higher order”…

```
@implementation NSArray (Mapping)

// Usage:
// NSArray* result = [myArray map:@selector(uppercaseString:)];

- (NSArray*)map:(SEL)selector
{
  NSMutableArray* mappedArray = [NSMutableArray array];

  for(id object in self)
  {
    id mappedObject = [object performSelector:selector];
    [mappedArray addObject:mappedObject];
  }

  return mappedArray;
}

@end
```

This is a new NSArray method named *map*. It’s simple: it takes in a selector, invokes that selector on each of the items in the array, and returns a new NSArray that contains the results of the invocations. The code’s not important here; what’s important is the general concept. If you’re a Smalltalk person, this is typically called *collect* in Smalltalk lingo. (“map” comes from the functional programming world.)

Higher-Order Messages

```
NSArray* result = [myArray map:@selector(uppercaseString:)];
```

vs

```
NSArray* result = [[myArray map] uppercaseString];

id tramponline = [myArrap map];
NSArray* result = [trampoline uppercaseString];
```

- map creates a proxy object (*trampoline*)
- trampoline receives uppercaseString message
- trampoline sends uppercaseString to each object in the original array and collects the results
- trampoline returns new array

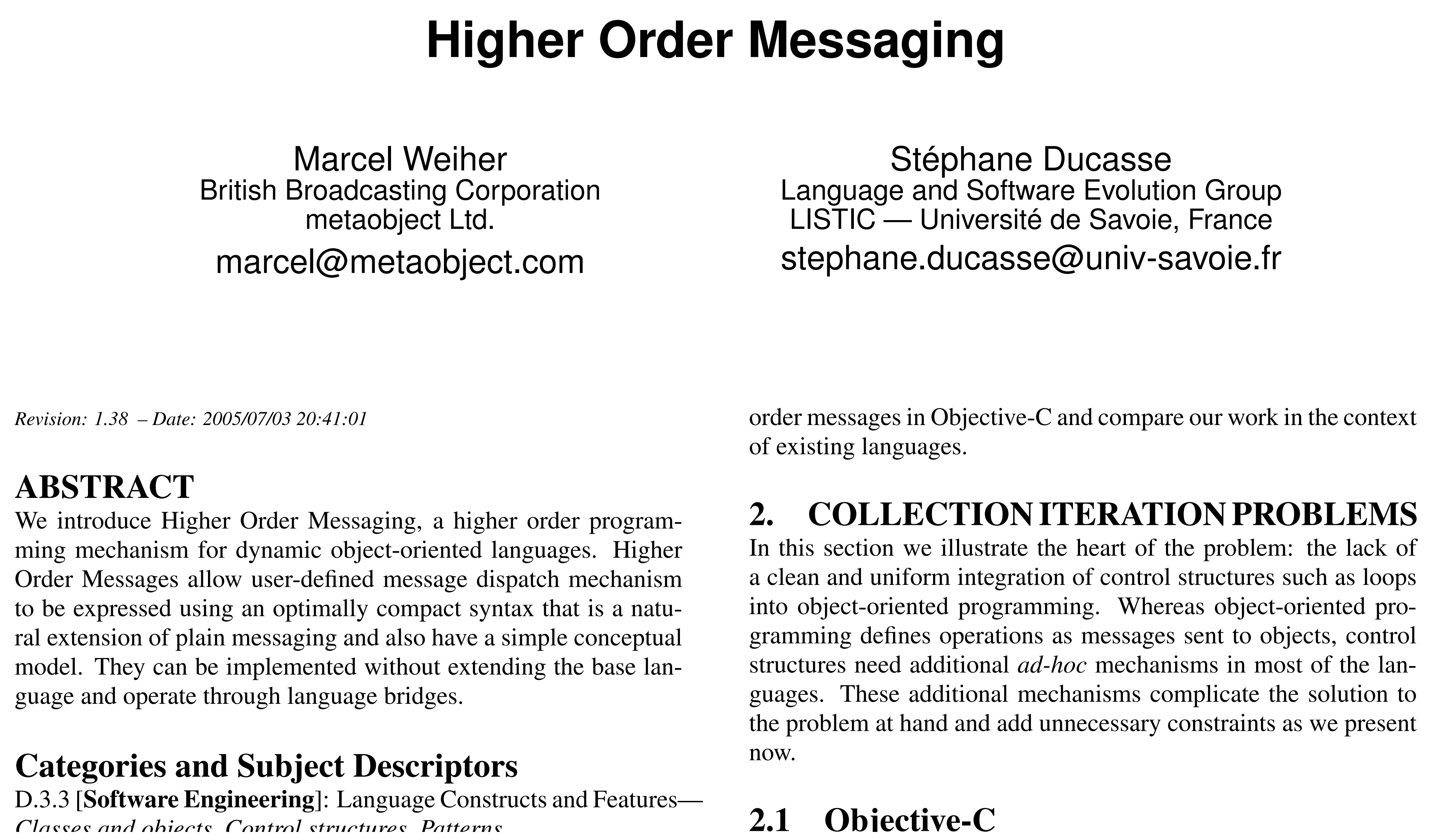
However, the syntax for the -map: method looks rather long-winded and unwieldy. Instead of writing “[myArray map:@selector(uppercaseString:)]”, what if we could write the shorter “[[myArray map] uppercaseString]”?

Marcel Weiher and an enterprising group of Objective-C developers at the cocoadev.com website calls this technique “Higher-Order Messaging”. Details about how it works are on the slide; hopefully it’s clear enough without further explanation.

Higher-Order Messages

- Threading:
  ```
  [[earth inBackground] computeAnswerToUniverse];
  [[window inMainThread] display];
  ```
- Threading:
  ```
  [myArray inPlaceMergeSort]; // synchronous
  [[myArray future] inPlaceMergeSort]; // asynchronous
  ```
- Parallel Map:
  ```
  [[myArray parallelMap] uppercaseString];
  ```
- Control Flow:
  ```
  [[myArray logAndIgnoreExceptions] stupidMethod];
  ```

There are many uses for higher-order messaging outside of the usual functional-programming-style map/fold (neé reduce)/filter collection functions. Some examples are shown here.



Higher Order Messaging

Marcel Weiher
British Broadcasting Corporation
metaobject Ltd.
marcel@metaobject.com

Stéphane Ducasse
Language and Software Evolution Group
LISTIC — Université de Savoie, France
stephane.ducasse@univ-savoie.fr

Revision: 1.38 – Date: 2005/07/03 20:41:01

ABSTRACT

We introduce Higher Order Messaging, a higher order programming mechanism for dynamic object-oriented languages. Higher Order Messages allow user-defined message dispatch mechanism to be expressed using an optimally compact syntax that is a natural extension of plain messaging and also have a simple conceptual model. They can be implemented without extending the base language and operate through language bridges.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Language Constructs and Features—*Classes and objects, Control structures, Patterns*

order messages in Objective-C and compare our work in the context of existing languages.

2. COLLECTION ITERATION PROBLEMS

In this section we illustrate the heart of the problem: the lack of a clean and uniform integration of control structures such as loops into object-oriented programming. Whereas object-oriented programming defines operations as messages sent to objects, control structures need additional *ad-hoc* mechanisms in most of the languages. These additional mechanisms complicate the solution to the problem at hand and add unnecessary constraints as we present now.

2.1 Objective-C

OOPSLA 2005

“Higher Order Messaging”
Marcel Weiher & Stéphane Ducasse

For more information on Higher-Order Messaging, simply Google for it: you can find the paper that Marcel Weiher and Stéphane Ducasse submitted to OOPSLA 2005 about it. It explains the subtle differences between HOM and more traditional higher-order functions better than I can.

Thanks!

All images © by their respective holders, and were found via Google Image Search.

Contact

http://algorithm.com.au/
ozone@algorithm.com.au

twitter

http://twitter.com/AndrePang

That’s it; thanks for listening folks! Feel free to contact me if you have any questions, of course.