Cocoaheads Paris 2016-03

SWIFT
PATTERNS
IN OBJECTIVE-C

That's maybe the last talk I'll ever make about objc, so I'll talk about patterns that we see in Swift, and how to do the same in ObjC.

OBJC
PATTERNS
IN OBJECTIVE-C

Well that was the initial intro I had when I prepared this talk, and I realized the patterns I was about to talk about (optionals, functional programming) were actually very "objective-C".
Or at least, my implementation was using concepts that were not strange at all to ObjC old-timers.

Objective-C

```
if([obj respondsToSelector:@selector(foo)])
{
    id foo = [self foo];
}
```

The starting point of this is: I've had enough with respondsToSelector:
Who has ever written something like that?

Swift

```
let foo = obj.foo?()
```

Swift has this very convenient "optional call". Could I do the same in ObjC?

Objective-C

```
id foo = [obj.please foo];
```

This is what I want.
I've been writing ObjC for more than ten years now, and I was sure it was possible, but hard, and never actually tried that until recently.
Turns out it's not that hard, but we'll have to cover a lot of topics to present the solution.
The solution is in fact very objc in spirit. I'm just a little disappointed to only make it now.
Also: we've been using it at Captain Train for a few months now, quite

- Objective-C Message Dispatch
- Forwarding
- NSInvocations
- Boxing

Actually, all this is an excuse to discuss the ObjC Runtime. Here's the agenda for today.

OBJECTIVE-C MESSAGE DISPATCH

Messages are a fundamental feature of Objective-C, far more than objects and class inheritance.
Classes are just the basic support for the message dispatch mechanism.
Actually, it should have been called Messaging-C.

https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtHowMessagingWorks.html
http://www.friday.com/bbum/2009/12/18/objc_msgsend-part-1-the-road-

---

ObjC Message Dispatch

- Clang emits objc_msgSend
- The Regular Method Lookup
- Try This Other Guy
- Wait I Just Found A Method!
- doesNotRespondToSelector:

Clang

```
[obj foo]
⬇
objc_msgSend(obj, "foo")
```

At compile time, clang actually emits function calls to objc_msgSend for all the messages.
There are a lot of scary details with how parameters are passed in registers and on the stack. One of the biggest problems is about methods that return structures.

https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtHowMessagingWorks.html

---

Clang

```
[view frame]
⬇
objc_msgSend_stret(view, "frame")
```

Which are actually emitted as the "_stret" variant, because well, C structures can't be returned in CPU registers.
(There's a similar, but different for double-floating point return values. But hey.)
Anyway, keep this in mind, it'll be important later.

---

objc_msgSend

- isa
- SEL
- IMP
  - *(id self, SEL _cmd, ...)
- Method Signature

The (regular) method dispatch uses the isa and the SEL to find the IMP.
isa: classes are just a list of methods that can be called to a target
SEL: selectors are (almost) C strings, uniqued at compile-time, so you can just compare the pointer
IMP: a C function pointer, that takes at least two arguments, self and _cmd
With the other parameters, and the return type, it forms the "method signature", that's actually used to setup the call.

Objective-C type encodings

```
- (void)foo:(int)bar;

"v@:i"
```

Method Signatures are actually expressed using the “objc type encoding”.
Any C or ObjC type can be expressed using a string.

Of course, the return type of an IMP has to match the emitted objc_msgSend variant, otherwise Bad Things Happen.

WHAT IF THERE’S NO IMP?

We’ve just explained the normal case. But that’s not the end of the story.

http://arigrant.com/blog/2013/12/13/a-selector-left-unhandled

---

```
- doesNotRespondToSelector:
```

Worst scenario: crash.
You can reimplement it if you like. You may even avoid crashing, by throwing (and catching) your own exception.
(If you return regularly from this method, you’re killed.)

```
- resolveInstanceMethod:
```

Before that, It’s possible to dynamically add methods to classes. CoreData does it routinely for accessors, among others.
In practice, respondsToSelector may return NO, while the actual call would work!

```
- forwardInvocation:
```

Before that, there’s the forwarding mechanism, which gives the opportunity to let another object handle the message, or to manipulate the message itself.

MESSAGE FORWARDING

That’s what we’re going to use to solve our problem.

INVOCATIONS

In the forwarding mechanism, messages are passed as objects. An NSInvocation is an object that represents an actual message (a selector) send to an object (its target, or receiver), with its arguments.
Invocations are strange and beautiful beasts.

INVOCATIONS ARE BLOCKS

Not really, but still. Closures, or similar concepts:

* Both are a “function object”.
* They capture variables in the scope.

INVOCATIONS EMIT CODE

But NSInvocations can also be worked with entirely at runtime.
A compiler emits the correct call to obj_sendMsg.
NSInvocation does the same thing, but at runtime.

NSInvocation

```
NSInvocation * i =
  [NSInvocation
   invocationWithMethodSignature:/*...*/];
i.target = (id) ;
i.selector = (SEL) ;
[i setArgument:(id) atIndex:(NSUInteger)];
...
[i invoke];
```

We’re basically setting up a call to objc_msgSend here.
(I’ve left out the method signature argument, but that’s basically the C string of the Objective-C type encoding.)

NSInvocation

```
NSInvocation * i =
  [NSInvocation
   invocationWithMethodSignature:/*...*/];
i.target = (id) ;
i.selector = (SEL) ;
[i setArgument:(id) atIndex:(NSUInteger)];
...
[i setReturnValue:(void*)];
```

But we can also set the return value ourselves!
In cases where we’re not emitting a call, but we’re receiving an invocation, that’s how we can set the return value for the caller.

BOXING

The next thing we’re going to need is called boxing.
That’s what we use when we needobject, but really only have a scalar types like integers or structs, or anything non-object. It’s called boxing.

```
CGRect r = obj.please.frame;
NSInteger i = obj.please.count;
```

We're going to need it because well, methods sometimes return scalars and structs.
And sometimes, those methods are optional too.

```
[obj pleaseOtherwiseReturn:42]
.count;
```

And I've left out an important detail since the beginning.
What do we return if the method isn't implemented?
Nil is the obvious answer when the method returns an object.
But for integers? for structs? we need a way to specify a default value.

BOXING

```
<scalar or struct>
```

```
NSValue
```

Boxing is putting stuff in an NSValue.
NSNumbers are NSValues, by the way.
It's used to put integers in an array, but also, all the time with key-value coding.
The KVC primitives return objects, but the underlying properties may return scalars, in which case the value is auto-boxed.

```
typedef struct { int x, y } MyStruct;
MyStruct myStruct = ...;
[NSValue valueWithBytes:&myStruct
               objCType:@encode(MyStruct)]
```

Unsurprisingly, NSValue works with objc type encoding.
Any C or Obj-C variable can be put in an NSValue.
It’s a raw memory copy, Bad Things Happen when you put an object in there, especially with ARC.

```
@( <thing> )
```

That’s also what the @() syntax does, at compile time. It just creates an NSValue, or directly an NSNumber with the contents.

```
-respondstoSelector:
```

Let’s go back to our initial problem. We don’t want to call that method anymore.

Putting the Pieces Together

```
[[obj performIfResponds] doSomething];
[[obj performOrReturn:@"foo"] foo];
[[obj performOrReturnValue:@YES] bar];

github.com/n-b/PerformIfResponds
```

performIfResponds returns a trampoline, that's going to forward all of its messages to the original object.

```
@interface NSObject (PerformProxy)
- (instancetype)performIfResponds;
- (instancetype)performOrReturn:(id)object;
- (instancetype)performOrReturnValue:(NSValue*)value;
@end
```

Our public methods are these three methods on NSObject.
instancetype, of course, is a lie, to make clang and the autocompletion happy
The actually returned object is a proxy, a trampoline that will try to forward each message to the original receiver.
We need two "return" variants to deal with the boxing issues I've explained before.
The underlying objc_msgSend variant is different, the NSInvocation has to

---

```
@interface PerformProxy : NSObject
{
    id _target;
    const char * _returnType;
    void(^_returnValueHandler)(NSInvocation*);
}
...
@end
```

Internally, we're going to have that Trampoline object, PerformProxy.
The Trampoline object has this single initializer, that's going to be called from each of the three public methods.

```
@implementation NSObject (PerformProxy)
- (instancetype)performIfResponds
{
    return [[PerformProxy alloc]
             initWithTarget:self
                 returnType:/*...*/
         returnValueHandler:/*...*/];
}
...
```

The public methods simply pass self as the target, and we'll see about the return types and handler in a minute.

```
@implementation PerformProxy
...
- (id)forwardingTargetForSelector:(SEL)sel_
{
    if ([_target respondsToSelector:sel_]) {
        return _target;
    } else {
        return self;
    }
}
...
```

In the PerformProxy, forwarding is done in two phases: the first is the "fast path", when you really just want another object to handle the method. That's what we want, if the target actually responds to the selector.

```
- (NSMethodSignature *)methodSignatureForSelector:(SEL)sel_
{
    return [NSMethodSignature signatureWithObjCTypes:
           @"%s%s%s",_returnType,@encode(id),@encode(SEL)];
}

- (void)forwardInvocation:(NSInvocation *)invocation_
{
    _returnValueHandler(invocation_);
}
```

(I cheated for the string formatting to make the slide readable.)
If the target does not respond to the message, we have to handle it ourselves. The runtime asks first for the signature, then does the actual invocation.
We see, here again, the two implicit methods of all invocations.
The return type and the handled were passed to the PerformProxy initializer a minute ago.

```
- (instancetype)performIfResponds
{
    return [[PerformProxy alloc]
            initWithTarget:self
                returnType:@encode(void)
        returnValueHandler:^(NSInvocation* i){}];
}
```

Void returning messages: that’s easy.

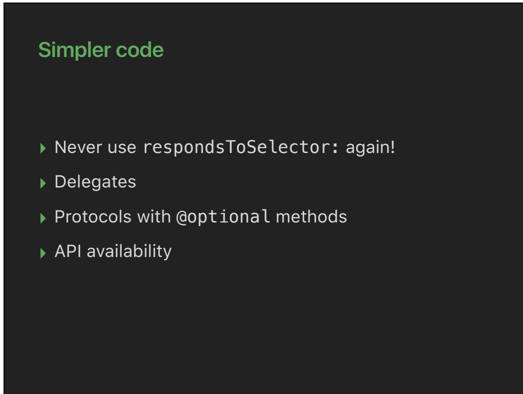
```
- (instancetype)performOrReturn:(id)object_
{
    return [[PerformProxy alloc]
            initWithTarget:self
                returnType:@encode(id)
        returnValueHandler:^(NSInvocation* i)
            {
               id obj = object_;
               [i setReturnValue:&obj];
            }];
}
```

Methods returning objects: just set the object as the return value.

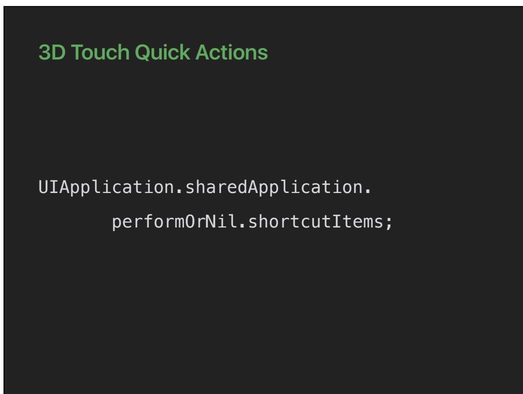
```
- (instancetype)performOrReturnValue:(NSValue*)value_
{
    return [[PerformProxy alloc]
            initWithTarget:self
                returnType:value_.objCType
        returnValueHandler:^(NSInvocation* i)
            {
               char buf[i.methodSignature.methodReturnLength];
               [value_ getValue:&buf];
               [invocation_ setReturnValue:&buf];
            }];
}
```

Method return scalars or structs: unbox the value in a local buffer, and return it

We're done! So what does it give us?



Existing patterns become easier to write, and easier to read.



Actual code snippet from CT.
Version compatibility (e.g. using a method that appeared in iOS 9 while maintaining compatibility with iOS 8.) becomes trivial.
We used to write compatibility abstraction layers for this, but now it's useless.

Optional All The Things!

New patterns appear as well!

It's ok not to have a default implementation!
It's ok not to check anything, just try to send the message and you'll see.

Contextual Help

```
obj.performIfResponds.helpTags
```

We've added Contextual Help in the latest update for CT: context is inferred from tags,
Tags come from all kinds of objects in the app, sometimes ViewControllers, because the context depends of where you are, sometimes model objects, because the help depends of what you see. But sometimes objects don't have specific tags.
We could have defined a protocol and made the objects actually conform to it. It was easier to just declare a selector, and loosely try it on each objects

---

Protocols Everywhere!

So instead of defining categories and superclasses, we just have abstract protocols.
PerformIfResponds allows to give a default value, and this is starting to look very similar to the "Protocol-Oriented-Programming" from that famous "Grumpy" talk at WWDC.

A few (minor) drawbacks

- Forwarding. is. very. slow.
- LOL Type Safety

```
NSForwarding: warning: method signature and compiler
disagree on struct-return-edness of 'someMethod'.
Signature thinks it does not return a struct, and
compiler thinks it does.
```

All is not so perfect.
Forwarding is 20x slower that regular dispatch, and that's if the real target responds to the selector.
The slow path with NSInvocation is 100x slower.
There is no type checking regarding the returned value.
What if we made a new language that made this kind of constructs easy, and safe?

SWIFT

Oh wait, that's just Swift.

Turns out, the thing that was the most Objective-C of Objective-C is one of the fundamental features of Swift. Mmm.

Merci !

```
source: github.com/n-b/PerformIfResponds
twitter: @_nb
jobs: captaintrain.com/jobs
betatest: ios@captaintrain.com
💚 Cocoaheads Paris 2016-03
```

Bonus

```
void(^block)() = ^{};

[(id)block invoke];
```

When I said Invocations were blocks:
NSBlock too has an `- invoke` method.

Bonus

- objc_msgSend does not appear in stack traces. Why?

Because it does “tail call optimization”. It directly jumps to the address of the IMP, instead of calling regularly.

Bonus #2

```
objc_boxable
```

Xcode 7.3 brings objc_boxable, which lets you declare boxing support for custom structures.

Bonus #3

NSProtocolChecker

I wanted to do this PerformProxy for a while, and I was sure this was possible, partly because of this strange class in Foundation.
It does something similar: it prevents from sending messages not specified in a given protocol to an object.
It was used a long time ago with distributed objects. Anyway.