CFHipsterRef

Low-Level Programming on iOS & OS X

First Edition

Matt Thompson

Mattt Thompson

CFHipsterRef: Low-Level Programming on iOS & Mac OS X

Illustrated by Conor Heelan

ISBN 978-0-9912182-4-0

NSHipster
Portland, Oregon
http://nshipster.com

Contents

Chapter 1

Kernel

It’s often remarked that the architecture of software reflects the structure of the organization building it. [1]

Therefore, if you’re at all familiar with the story of how Apple acquired NeXT, you already have an intuitive sense about the architecture of the OS X kernel—even if you don’t really know what a kernel *is*.

So, what is a kernel?

A kernel is a program that coordinates input and output between software and the underlying hardware. It’s the fundamental part of an operating system, responsible for reading and writing information to and from the CPU, memory, and any other mounted devices. How a kernel should go about coordinating access to these resources across all of the contending processes has been an ongoing quandary for industry and academia alike.

As it so happens, this question of computer architecture played an important role in a pivotal moment of Apple’s history.

Apple, NeXT, & Be

The mid 1990’s were a bad time for Apple. After years of uninspired, over-priced computers, the company had become all but irrelevant.

1 As codified by Melvin Conway in his eponymous law: "Organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations."

PCs were more performant, more ubiquitous, and more cost-efficient than anything coming out of Cupertino at the time. Between '94 and '97, the company posted losses in the hundreds of millions, and may have been forced to declare bankruptcy had it not been for a cash infusion from Microsoft, which was fighting an antitrust lawsuit at the time.

It wasn't just hardware innovation that had stalled. By the release of System 7, the technical debt built up since the original Macintosh had become insurmountable: in order to remain competitive, it needed to be addressed. In 1994, development started on Copland, a project to rewrite the underlying Macintosh OS for its 8.0 release. However, as deadlines slipped, and the completion date continued to be pushed back, it was decided that the project was dead-on-arrival.

The only way to get a new operating system would be through an acquisition.

On the table were two companies with strong connections to Apple: Be Inc., founded by former Apple exec Jean-Louis Gassée [2], and NeXT, the company Steve Jobs started after he was forced out of Apple.

2 Gassée was appointed to Jobs' former position as head of Macintosh development by Apple CEO John Sculley. Notable achievements during his tenure include the Macintosh Portable and Newton MessagePad

Both companies had built modern, eponymous operating systems—Be Inc.'s BeOS, and NeXT's NeXTSTEP—from the ground-up, but were struggling to gain a foothold into the mainstream computer market. [3] BeOS, with an impressively forward-looking architecture and focus on multimedia, was in many ways the technological forerunner, but it was ultimately NeXTSTEP that would be given the nod. [4]

NeXTSTEP [5] would go on to be ported to PowerPC and eventually become the OS X everyone knows today.

Darwin

Darwin is the UNIX core of OS X. The term "Darwin" specifically refers to the operating system, composed of the XNU kernel, Objective-C runtime, and I/O Kit driver framework, though it is often used interchangeably to refer to any individual component. [6]

3 Around the time of acquisition, NeXT had nearly given up on their operating system, and had considered shifting focus onto WebObjects instead.

4 Regarding the acquisition, Gil Amelio famously quipped, *"We choose Plan A instead of Plan Be."*

5 Or, to be pedantic, OPENSTEP, which was a specific implementation of the OpenStep API (note the capitalization), which was jointly developed by NeXT and Sun Microsystems (hence the `NS` prefix for "NeXT/Sun")

6 Each major release of Darwin is coordinated with a minor release of OS X. For a given release of OS X, `10.x.y`, the corresponding Darwin release is `(x + 4).y.0`.

The source code for Darwin and other underlying technologies can be downloaded from Apple's Open Source website. [a]

a http://opensource.apple.com

XNU, the Darwin kernel, is an acronym for "X is Not UNIX". [7] It is a hybrid kernel, incorporating both the microkernel architecture of Mach and features of the monolithic BSD kernel. [8]

```
$ uname -a

Darwin NSHipster.local 13.3.0 Darwin Kernel Version 13.3.0: Tue Jun  3 21:27:35 ↩
      PDT 2014; root:xnu-2422.110.17~1/RELEASE_X86_64 x86_64
```

It's endearing to view this marriage of Mach and BSD in the XNU kernel as an extension of Apple's philosophy of pragmatic eclecticism. By combining competing technologies in ways that compliment one another, one gets the best of both worlds. Like how the ideas of Smalltalk were bolted onto a C implementation to form Objective-C—under the right circumstances, the whole can be much greater than the sum of its individual parts.

7 Though, as of OS X 10.5, Darwin has been certified under the Single UNIX Specification version 3 (SUSv3)

8 As such, OS X is often described as having a "Mach/BSD" kernel.

Actually, the parallels between Mach/BSD and Smalltalk/C run deeper than that.

Both Mach and Smalltalk are designed around the concept of message passing. For Smalltalk, it's objects sending messages to one another in order to invoke methods. For Mach, it's untyped interprocess communication (IPC) and remote procedure calls (RPC) as a means of sending information between processes. Message passing is a safe and elegant abstraction, but necessarily incurs a performance penalty because of this indirection.

High-minded idealism is wasted without some basis in reality. That's where C & BSD come in.

For Objective-C, transforming message sends into highly-optimized C function calls makes the cost of objects negligible. For Mach, serious performance gains are made by linking kernel components into a single address space.

As a result, XNU gets the extensibility of Mach with the performance of the monolithic BSD kernel.

Mach

At the core of XNU is Mach, which provides a small set of abstractions for interacting with the system:

- Task: The unit of resource allocation, similar to a process, consisting of a virtual address space and port rights.
- Thread: The unit of CPU utilization for a task.
- Port: A simplex, or one-way, communication channel with send and receive capabilities made accessible via port rights.
- Message: A typed collection of data objects.
- Memory Object: An internal unit of memory management, representing data that can be mapped into address spaces.

Multitasking

The original Mac OS kernel was extremely simple. The allocation and usage of system resources was left up to the individual processes themselves, in a scheme known as cooperative multi-tasking. Applications would run until they would either exit or yield back to the OS. The idea was that by switching between tasks in rapid succession, it would give the illusion of running them all simultaneously. However, this behavior was entirely elective, resulting in a "tragedy of the commons", where a single bad actor could bring the entire system to a halt.

OS X, on the other hand, is a preemptive multitasking environment, which means that the kernel enforces cooperation by scheduling resources for processes to share. Under this scheme, an application can still "beachball", but it’s usually possible to recover by force-quitting any unresponsive processes.

As the saying goes, "Good fences make good neighbors".

Mach-O

Darwin also adopts the Mach-O file format for executables, object code, and shared libraries. [9] [10]

Mach-O has been the exclusive file format for NeXTSTEP, OS X, and iOS. One feature of particular importance for these platforms is Mach’s support of multi-architecture, or "fat", binaries.

For example, a single iOS binary can have six instruction set architectures:

9 Commonly known by their file extensions: `.o`, `.dylib`, and `.bundle`.

10 Nearly every other Unix-based system adopts the ELF format for executables and shared libraries. Since Darwin is already POSIX compliant, this would be one of the last steps to compatibility with Solaris, BSD, and Unix.

Architectures & Supported Devices

`ARMv6`	iPhone, 3G; 1st & 2nd generation iPod touch
`ARMv7`	iPhone 3GS, 4, 4S; iPad, 2, 3rd generation; 3rd, 4th, & 5th generation iPod touch
`ARMv7s`	iPhone 5; 4th generation iPad
`ARMv8`	iPhone 5S
`x86`	iPhone Simulator (32-bit)
`x86_64`	iPhone Simulator (64-bit)

BSD

The BSD layer provides the operating system personality of Darwin, as well as a POSIX-compliant interface to everything one can expect on a Unix system. It's implemented on top of Mach in such a way that allows developers to interact with high-level abstractions, rather than the underlying primitives.

Among the most visible features of BSD are its ubiquitous sockets API for TCP & UDP networking, along with Kqueue, an efficient event pipeline between kernel and user space. Everything else, from memory protection to multiuser access, are more notable for how they're extended by Darwin.

Darwin Additions

iOS vs. OS X

From a kernel and operating systems perspective, iOS is OS X with some slight modifications. For one, the architecture of the kernel and binaries is ARM-based, rather than the Intel i386 or x86_64 used on desktops. [11]

Perhaps the most significant difference between the OS X & iOS kernels, however, are the additional protections built to keep iOS on lockdown. A cynical take on these measures would be that they serve to protect the interests of a business driven by vendor lock-in. In fairness, many of these decisions follow sound technical reasoning that acknowledges the fundamental differences in use cases and requirements between iOS and OS X.

11 With this new platform, Apple decided not to open source the ARM-based kernel.

Basic Security Module

Darwin incorporates Sun's Basic Security Module, or BSM, to provide an auditing log of actions taken by users and processes. It's enabled on OS X by default, but disabled on iOS, since it makes less sense on the single-user platform.

Mandatory Access Control

Mandatory Access Control (MAC), limits access to protected resources like sockets and shared memory segments to specific processes. It was originally developed as part of TrustedBSD, and incorporated into OS X 10.5.

MAC is the substrate of both OS X Sandboxing and iOS Entitlements.

Jetsam

OS X & iOS implement a memory status mechanism called Jetsam, or alternatively, Memorystatus. Similar to the "Out-Of-Memory" handler `oom` on Linux, Jetsam is used to monitor the memory usage of processes and kill any that are consuming more than their fair share.

On a regular interval, `launchd` takes a snapshot of how many pages of memory are being used by each process. Jetsam then takes this information and builds an ordered list of candidates that should be killed if memory pressure exceeds a certain threshold. The current list of snapshots and candidates can be queried using the `sysctl` system call, or the APIs exposed in `<sys/kern_memorystatus.h>`

Under a constrained memory environment like iOS, Jetsam is critical for ensuring that no running application in the foreground or background over-utilizes system resources.

This important to keep in mind when investigating the source of a crash on iOS. Consider the following crash log:

```
Incident Identifier: <GUID>
CrashReporter Key:   <Checksum>
Hardware Model:      <Apple Model Identifier>
OS Version:          <Version>
Kernel Version:      <Version (output of `$ uname -a`)>
Date:                <Date>
Time since snapshot: <Timestamp>

Free pages:          <Number of Remaining Pages of Memory>
Wired pages:         <Number of Resident Pages of Memory>
```

```
Purgeable pages:     <Number of Pages to be Freed>
Largest process:     <Process Causing Crash>

 Processes
     Name                 UUID                     Count resident pages
    <Name>               <UUID>                    <Count> (jettisoned) (active)
```

If a process is marked as `(jettisoned)`, it means that Memorystatus / Jetsam killed it for consuming too much memory. [12] The easiest way to mitigate the chance of an application being jettisoned is to respond to the following low memory warnings:

- `UIApplicationDelegate -applicationDidReceiveMemoryWarning:` delegate method
- `UIViewController -didReceiveMemoryWarning` delegate method
- `UIApplicationDidReceiveMemoryWarningNotification` notification

Process Hibernation

As of iOS 5 and OS X 10.7 Lion, high-memory processes identified by Jetsam can be frozen instead of immediately killed. When a process is frozen, its state is captured in such a way that it can be thawed and

12 An `(active)` designation indicates that the process was in the foreground at the time of the log.

resumed when memory pressure subsides. This is known as hibernation.

Hibernate is only enabled on iOS, again, due to the prevalence of low-memory conditions on mobile devices relative to the desktop.

From a user's perspective, hibernation occurs as a consequence of switching between applications. By default, a screenshot of an application's current window is taken before hibernation, and used in the multitasking UI. By listening for the `UIApplicationWillResignActiveNotification` notification, an application has the opportunity to customize what is displayed in the application switcher.

Kernel Address Space Layout Randomization

For the vast majority of applications, Address Space Layout Randomization (ASLR) is a completely irrelevant implementation detail. However for hackers, it has profound implications.

One of the primary attack vectors for software is injecting behavior at particular memory addresses. Know the position of the stack, heap, and libraries in a process's address space, and that process can easily be exploited. By randomizing memory offsets within that address space, however, a process becomes much less susceptible to attack.

I/O Kit

The last piece of the XNU kernel is I/O Kit, a framework for developing device drivers for OS X & iOS.

> The source code for I/O Kit and its companion library `libkern` can be downloaded from Apple's Open Source website. [a]
>
> a http://opensource.apple.com

I/O Kit was created to replace NeXT's Driver Kit, which critically lacked the capabilities of hot-swappable hardware and automatically configuration. Unlike Driver Kit, which is written in Objective-C, I/O Kit is implemented in Embedded C++, a subset that omits languages features deemed problematic within a multithreaded kernel environment. [13]

I/O Kit is organized into several different families, each responsible for a particular type of interface:

13 Specifically: no exceptions, no multiple inheritance, and no templates. (Sounds pretty nice, right?)

- ADB (Apple Desktop Bus)
- AGP (Accelerated Graphics Port)
- ATA & ATAPI (ATA Packet Interface)
- Audio
- FireWire
- Graphics
- HID (Human Interface Devices)
- Network
- PCI (Peripheral Component Interconnect)
- SBP-2 (Serial Bus Protocol 2)
- SCSI (Small Computer System Interface [14])
- Serial
- Storage
- USB (Universal Serial Bus)

Families are, themselves, divided into logical layers that are represented in a class hierarchy. Consistent with best practices in application level development, most drivers inherit from the most specific class available. For example, a keyboard driver would inherit from `IOHIKeyboard` rather than `IOHIDevice`, or even `IOService`.

Although consumer software development doesn't often require much in the way of kernel programming, it's still fun to poke around to get a

14 Pronounced `/ˈskʌzi/`

better sense of how everything fits together. To that end, the following command-line utilities offer a safe way to explore the hidden world of drivers (`io` prefix) and kernel extensions (`kext` prefix):

Driver Utilities

`ioreg`	Prints the contents of the I/O Registry (a command-line version of the I/O Registry Explorer application).
`iostat`	Displays kernel I/O statistics on terminal, disk, and CPU operations.
`ioclasscount`	Displays instance count of a specified class.
`ioalloccount`	Displays some accounting of memory allocated by I/O Kit objects in the kernel.

Kernel Extension Utilities

`kextstat`	Prints statistics about currently loaded drivers and other kernel extensions.
`kextload`	Loads a kernel extension (such as device driver) or generates a statically linked symbol file for remote debugging.
`kextunload`	Unloads a kernel extension (if possible).

Chapter 2

Objective-C Runtime

Objective-C, by itself, is just talk. All of its high-falutin' ideas about message passing and dynamism is nothing but a bunch of hot air without a runtime to do the hard work.

It's almost unfair how the Objective-C *language* gets all of the credit, when it's really the Objective-C *runtime* things happen.

"'What's that? Your object-oriented paradigm was inspired by the interactions of microorganisms, you say? [15] *Get a job, hippy!"*

The Objective-C runtime acts as a kind of meta operating system, facilitating the data structures and function calls that that implement the dynamic features of Objective-C.

Every class declaration, method invocation, and expression evaluation is compiled into equivalent C functions to interact with the runtime. Indeed, most interactions a developer has with the Objective-C runtime is through the Objective-C language. But as a strict superset of C, these runtime functions can be invoked directly as well.

15 Alan Kay took inspiration from the autonomous, message-passing interactions of cells within an organism when he created Smalltalk, the language credited with bringing object-oriented programming to the mainstream.

The Objective-C 2.0 Runtime is open source, and available for download from Apple's Open Source website. [a]

[a] http://opensource.apple.com/source/objc4

libobjc

`libobjc` is the shared library for the Objective-C 2.0 runtime. It can be used directly from an Objective-C application to introspect and change its own behavior.

```
#import <objc/runtime.h>
```

Objective-C developers are conditioned to be wary of whatever follows this ominous incantation. And for good reason: messing with the Objective-C runtime changes the very fabric of reality for all of the code that runs on it.

In the right hands, the functions of <objc/runtime.h> have the potential to add powerful new behavior to an application or framework, in ways that would otherwise be impossible.

Message Sending

At the heart of Objective-C's object-oriented paradigm is the concept of message passing. It's enshrined in the syntax of square brackets, which delimit the act of sending a message to an object.

Consider the following Objective-C code:

```
[object message];
```

The compiler will translate this into an equivalent `objc_msgSend` call: [16]

```
objc_msgSend(object, @selector(message));
```

A class (`Class`) maintains a dispatch table to resolve messages sent at runtime; each entry in the table is a method (`Method`), which keys a particular name, the selector (`SEL`), to an implementation (`IMP`), which is a pointer to an underlying C function.

16 Any parameters would be passed as additional arguments to `objc_msgSend`.

When a message is sent to an object, it consults its class's dispatch table to find an implementation associated with the message's selector. If a match is found, the associated function is invoked. If not, the dispatch table of the superclass is consulted, and so on, until either a match is found, or the selector is determined to be unrecognized.

With dynamic dispatch, the behavior of message-based code is not deterministic at compile time. Every aspect of execution is deferred until the last possible moment. The class of `object` could be changed, the method corresponding to the `message` selector could have its implementation replaced, or the ivar layout of `object` could just be sabotaged in such a way that it crashes the process. Anything is possible.

Such are the risks and rewards of hacking the Objective-C runtime.

You have nothing to gain and everything to lose by invoking `objc_msgSend` directly. Objective-C code has the benefit of an incredibly sophisticated compiler and analyzer, which can detect and safeguard against incorrect behavior. Cutting out the middle man means taking all of that responsibility on yourself, with little to no direct benefit.

But, if you like to live dangerously… and consider asinine development practices to qualify, here's what's what:

- `objc_msgSend`: Sends a message with a simple return value to a class.
- `objc_msgSend_stret`: Sends a message with a data-structure return value to a class.
- `objc_msgSendSuper`: Sends a message with a simple return value to the superclass of a class.
- `objc_msgSendSuper_stret`: Sends a message with a data-structure return value to the superclass of a class.

Metaprogramming with Properties

Properties define the public interface for an object's state. Using `libobjc`, a clever developer can work with classes on an entirely new level.

For instance, with access to a list of an object's properties, the drudgery of manually implementing `NSCoding` can be avoided with a dose of metaprogramming:

```
#pragma mark - NSCoding

- (id)initWithCoder:(NSCoder *)decoder {
    self = [super init];
    if (!self) {
        return self;
    }
    unsigned int count;
    objc_property_t *properties = class_copyPropertyList([self class], &count);
    for (NSUInteger i = 0; i < count; i++) {
        objc_property_t property = properties[i];
        NSString *key = [NSString stringWithUTF8String:property_getName( ↩
     property)];
        [self setValue:[decoder decodeObjectForKey:key]
                forKey:key];
    }
    free(properties);

    return self;
}

- (void)encodeWithCoder:(NSCoder *)coder {
    unsigned int count;
    objc_property_t *properties = class_copyPropertyList([self class], &count);
    for (NSUInteger i = 0; i < count; i++) {
        objc_property_t property = properties[i];
        NSString *key = [NSString stringWithUTF8String:property_getName( ↩
     property)];
        [coder encodeObject:[self valueForKey:key] forKey:key];
    }
    free(properties);
}
```

Associated Objects

Associated objects are a feature of the Objective-C runtime without a counterpart in the language itself.

This feature allows objects to associate arbitrary values for keys at runtime. It can be used to work around a language constraint that prevents categories from declaring new storage properties.

```
@interface NSObject (AssociatedObject)
@property (nonatomic, strong) id associatedObject;
@end

@implementation NSObject (AssociatedObject)
@dynamic associatedObject;

- (void)setAssociatedObject:(id)object {
     objc_setAssociatedObject(self, @selector(associatedObject), object,  ↩
     OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

- (id)associatedObject {
    return objc_getAssociatedObject(self, @selector(associatedObject));
}
```

Associated objects are fairly straightforward: use `objc_setAssociatedObject` to store an associated value, and `objc_getAssociatedObject` to retrieve it. [17]

Each associated object is referenced by a key, which can be any constant value. The easiest solution is to just pass the selector of the getter method. [18]

17 Anyone from a time before `@synthesize` will no doubt recognize this getter / setter pattern.

18 Selectors are guaranteed to be unique and constant in the runtime.

Dynamically Adding a Method

Just as properties describe the state of an object, methods comprise their behavior. In Objective-C, methods are declared with a leading + or –, to denote whether a method is associated with a class or instances of that class.

Normally, class declarations are relatively stable. Setting aside the confounding effect of extensions and categories, the methods of a class usually remain stable over an application's execution. However, using the Objective-C runtime, methods can be added and removed as desired.

Consider the following category:

```
@interface NSObject ()
- (NSString *)greetingWithName:(NSString *)name;
@end

@implementation NSObject ()
- (NSString *)greetingWithName:(NSString *)name {
  return [NSString stringWithFormat:@"Hello, %@!", name];
}
@end
```

By including this code in a source file at compile time, `greetingWithName:` will be available to all instances of `NSObject`.

However, the equivalent behavior could instead be achieved at runtime:

```
Class c = [NSObject class];
IMP greetingIMP = imp_implementationWithBlock((NSString *)^(id self, NSString * ↩
     name){
    return [NSString stringWithFormat:@"Hello, %@!", name];
});
const char *greetingTypes = [[NSString stringWithFormat:@"%s%s%s", @encode(id), ↩
      @encode(id), @encode(SEL)] UTF8String];
class_addMethod(c, @selector(greetingWithName:), greetingIMP, greetingTypes);
```

The ability to add new methods and properties makes it possible to define complex behavior in terms of macro-based DSLs, or work around compatibility bugs within an SDK.

It's a lot of power at a *very* reasonable price.

Method Swizzling

Method swizzling is the process of changing the implementation of an existing selector. It's a technique made possible by the fact that method invocations in Objective-C can be modified at runtime, by changing how selectors are mapped to underlying functions in a class's dispatch table.

Consider the task of tracking how many times each view controller in an application is presented during its lifetime:

Each view controller could add tracking code to an overridden implementation of `viewDidAppear:`, but that would make for a ton of duplicated boilerplate code. Subclassing is another possibility, but it would require subclassing `UIViewController`, `UITableViewController`, `UINavigationController`, and every other view controller class—an approach that also suffers from excessive code duplication.

With method swizzling, the solution is rather elegant:

```
#import <objc/runtime.h>

@implementation UIViewController (Tracking)

+ (void)load {
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        Class class = [self class];

        SEL originalSelector = @selector(viewWillAppear:);
        SEL swizzledSelector = @selector(xxx_viewWillAppear:);

        Method originalMethod = class_getInstanceMethod(class, originalSelector ↩
      );
        Method swizzledMethod = class_getInstanceMethod(class, swizzledSelector ↩
      );
```

```
        BOOL didAddMethod =
            class_addMethod(class,
                originalSelector,
                method_getImplementation(swizzledMethod),
                method_getTypeEncoding(swizzledMethod));
        if (didAddMethod) {
            class_replaceMethod(class,
                swizzledSelector,
                method_getImplementation(originalMethod),
                method_getTypeEncoding(originalMethod));
        } else {
            method_exchangeImplementations(originalMethod, swizzledMethod);
        }
    });
}

#pragma mark - Method Swizzling

- (void)swizzled_viewWillAppear:(BOOL)animated {
    [self swizzled_viewWillAppear:animated];
    NSLog(@"viewWillAppear: %@", self);
}

@end
```

Now, when any instance of `UIViewController` or one of its subclasses invokes `viewWillAppear:`, a log statement will be printed. [19]

Because method swizzling affects global state, it is important to

19 It may appear that `swizzled_viewWillAppear:` would cause an infinite loop by calling itself in its own implementation, but that's not the case. In the process of swizzling, `swizzled_viewWillAppear:` has been reassigned to the original implementation of `UIViewController -viewWillAppear:`.

minimize the possibility of race conditions. `+load` is guaranteed to be executed during class initialization, thereby affording a modicum of consistency for changing system-wide behavior. By contrast, `+initialize` provides no such guarantee of when it will be executed. [20]

Injecting behavior into the view controller lifecycle, responder events, view drawing, or the Foundation networking stack are all reasonable examples of how method swizzling can be used to great effect. There are a number of other occasions when swizzling would be an appropriate technique, and they become increasingly apparent the more seasoned an Objective-C developer becomes.

Dynamically Creating a Class

Combining the previous concepts of dynamic property and method definition, `libobjc`'s ultimate feat is creating a class at runtime.

Consider the following interface and implementation for a simple `Product` class, with `name` and `price` properties:

20 In fact, unless a class is messaged directly by the app, its `+initialize` method won't be called.

```
@interface Product : NSObject
@property (readonly) NSString *name;
@property (readonly) double price;

- (instancetype)initWithName:(NSString *)name
                       price:(double)price;
@end

@implementation Product

- (instancetype)initWithName:(NSString *)name
                       price:(double)price
{
    self = [super init];
    if(!self) {
        return nil;
    }

    self.name = name;
    self.price = price;

    return self;
}
@end
```

If ever there was a sympathetic case to be made for Objective-C's syntax, it's how elegant that code looks in comparison to its runtime equivalent:

```
Class c = objc_allocateClassPair([NSObject class], "Product", 0);
class_addIvar(c, "name", sizeof(id), log2(sizeof(id)), @encode(id));
class_addIvar(c, "price", sizeof(double), sizeof(double), @encode(double));

Ivar nameIvar = class_getInstanceVariable(c, "name");
ptrdiff_t priceIvarOffset =
    ivar_getOffset(class_getInstanceVariable(c, "price"));

IMP initIMP = imp_implementationWithBlock(
    ^(id self, NSString *name, double price)
{
    object_setIvar(self, nameIvar, name);

    char *ptr = ((char *)(__bridge void *)self) + priceIvarOffset;
    memcpy(ptr, &price, sizeof(price));

    return self;
});
const char *initTypes = [[NSString stringWithFormat:@"%s%s%s%s%s%s",
    @encode(id), @encode(id), @encode(SEL), @encode(id), @encode(id),
    @encode(NSUInteger)] UTF8String];
class_addMethod(c,
                @selector(initWithFirstName:lastName:age:),
                initIMP,
                initTypes);

IMP nameIMP = imp_implementationWithBlock(^(id self) {
    return object_getIvar(self, nameIvar);
});
const char *nameTypes =
    [[NSString stringWithFormat:@"%s%s%s",
          @encode(id), @encode(id), @encode(SEL)] UTF8String];
class_addMethod(c, @selector(name), nameIMP, nameTypes);

IMP priceIMP = imp_implementationWithBlock(^(id self) {
    char *ptr = ((char *)(__bridge void *)self) + priceIvarOffset;
    double price;
    memcpy(&price, ptr, sizeof(price));

    return price;
});
const char *priceTypes = [[NSString stringWithFormat:@"%s%s%s", @encode(double) ↩
     , @encode(id), @encode(SEL)] UTF8String];
class_addMethod(c, @selector(age), priceIMP, priceTypes);

objc_registerClassPair(c);
```

There's a lot going on here, so let's take things one step at a time.

First, a class is allocated with `objc_allocateClassPair`, which specifies the class's superclass and name. [21]

After that, instance variables are added to the class using `class_addIvar`. That fourth argument is used to determine the variable's minimum alignment, which depends on the ivar's type and the target platform architecture. [22]

The next step is to define the implementation for the initializer, with `imp_implementationWithBlock:`. To set `name`, the call is simply `object_setIvar`. `price` is set by performing a `memcpy` at the previously-calculated offset.

21 Why a class *pair*? Every class has a meta-class, which manages the dispatch table for messages sent to the class itself. Classes are objects in Objective-C; class methods are *actually* instance methods for class objects.

22 For variables of any pointer type, the correct alignment is `log2(sizeof(type))`.

In order to add the initializer, the type encodings of each argument needs to be calculated. It's an awkward mess of `@encode` and string interpolation, but it does the job.

Adding methods for the ivar getters follows much the same process. [23]

Finally, once all of the methods are added, the class is registered with the runtime. And from that point on, `Product` can be interacted with in Objective-C like any other class:

```
Product *widget = [[Product alloc] initWithName:@"Widget"
                                          price:50.00];

NSLog(@"%@: %g", widget.name, widget.price);
```

23 Although this example declares `readonly` properties, setter methods would be implemented in similar manner to the initializer.

Chapter 3

Clang

Apple's adoption of LLVM in the mid-00's was the most important technical decision made since the acquisition of NeXT. In many ways, it serves as the demarcation point for when the company started to hit its technological stride once again.

Originally created as a research project at the University of Illinois by Vikram Adve and Chris Lattner, LLVM has grown to encompass a range of low-level toolchain technologies, including assemblers, compilers, and debuggers. Its modern, modular architecture and non-Copyleft [24] [25] license made it a particularly appealing alternative to GCC. As such, Apple took an early interest in the project, and has been its primary steward ever since—going so far as to hire Lattner to head a new developer technologies team within the organization in 2005.

Clang is architected as a 3-phase compiler:

```
- Source Code
1. Front-end
2. Optimizer
```

24 GCC is released under the terms of the GNU General Public License, which has a Copyleft provision that mandates redistributed software to also be released under the same license. Companies like Apple have often taken a defensive stance against such terms, preferring non-GPL-licensed projects and even going as far as to create their own alternatives.

25 LLVM is released under the University of Illinois/NCSA Open Source License, which is based on the MIT and BSD 3-Clause Licenses, and lacks any such Copyleft provisions that might constrain the distribution of proprietary software.

```
3. Back-end
- Machine Code
```

The genius of this approach is that both the front-end and back-end can be swapped out to support any source language input (C, Haskell, Ruby, etc.) or instruction set output (x86, PowerPC, ARM), without falling victim to the `M x N` combinatorics of supporting each combination individually. Instead, each LLVM front-end compiles source code into the LLVM Intermediate Representation (IR), which is analyzed and transformed by the common optimizer, and then passed into the desired back-end. [26]

Clang is the front-end to the LLVM compiler for the C language family. It has a deep understanding of the syntax and semantics of Objective-C, and is largely responsible for how Objective-C came to be the capable language it is today.

libclang

`libclang` is the C interface to the Clang LLVM front-end. It's a powerful way for C & Objective-C programs to introspect their own internal structure and composition.

26 LLVM was developed for maximum compatibility with GCC, and as such, can use GCC font-ends as a fallback for situations where an LLVM front-end is not provided.

There’s a lot of functionality baked into Clang, which `libclang` organizes it into several different components:

Clang Components

`libsupport`	Basic support library, from LLVM.
`libsystem`	System abstraction library, from LLVM.
`libbasic`	Diagnostics, SourceLocations, SourceBuffer abstraction, file system caching for input source files.
`libast`	Provides classes to represent the C AST, the C type system, builtin functions, and various helpers for analyzing and manipulating the AST (visitors, pretty printers, etc).
`liblex`	Lexing and preprocessing, identifier hash table, pragma handling, tokens, and macro expansion.
`libparse`	Parsing. This library invokes coarse-grained *Actions* provided by the client (e.g. `libsema` builds ASTs) but knows nothing about ASTs or other client-specific data structures.
`libsema`	Semantic Analysis. This provides a set of parser actions to build a standardized AST for programs.
`libcodegen`	Lower the AST to LLVM IR for optimization & code generation.
`librewrite`	Editing of text buffers (important for code rewriting transformation, like refactoring).

(continued)

`libanalysis`	Static analysis support.

Translation Units

The first step to working with source code is to load it into memory. Clang operates on *translation units*, which are the ultimate form of C or Objective-C source code, after all of the `#include`, `#import`, and any other preprocessor directives have been evaluated.

A translation unit is created by passing a path to the source code file, and any command-line compilation arguments:

```
CXIndex idx = clang_createIndex(0, 0);
const char *filename = "path/to/Source.m";
int argc;
const char *argv[];

CXTranslationUnit tu =
    clang_parseTranslationUnit(idx,
                               filename,
                               0,
                               argv,
                                 argc,
                                 0,
                                 CXTranslationUnit_None);
{ ... }
clang_disposeTranslationUnit(tu);
clang_disposeIndex(idx);
```

AST

In order to understand the structure of a translation unit, Clang constructs an Abstract Syntax Tree (AST). ASTs are the platonic ideal of what code, derived by distilling constructs from their representation.

Each node in an AST is represented by a cursor to its respective declaration, definition, statement, or reference. A cursor provides the name, location, range, and type of the node, as well as a pointer to any children it may have.

To iterate over the nodes of an AST, use `clang_visitChildren`, passing a pointer to a function that evaluates each node, and decides whether to recurse into children, continue onto the next sibling node, or terminate:

```
static unsigned Visitor(CXCursor cursor,
                        CXCursor parent,
                        CXClientData data)
{
    switch (clang_getCursorKind(cursor)) {
        case CXCursor_FunctionDecl:
            // Function
            break;
        case CXCursor_VarDecl:
            // Variable
            break;
        case CXCursor_ObjCInstanceMethodDecl:
            // Objective-C Instance Method
            break;
        // ...
        default:
            break;
    }

    return CXChildVisit_Recurse;
}


CXCursor cursor = clang_getTranslationUnitCursor(tu);
clang_visitChildren(cursor, Visitor, 0);
```

Because a translation unit may `#include` code included from another source file, it's often useful to first check that the cursor is contained within that specific source file:

```
CXSourceLocation location = clang_getCursorLocation(cursor);
```

```
CXFile file;
clang_getFileLocation(location, &file, 0, 0, 0);
const char *filename = "/path/to/Source.m";

if (filename != clang_getCString(clang_getFileName(file))) {
    return CXChildVisit_Continue;
}
```

For reference types, Clang can find all of the other local references within the AST. Xcode uses this functionality to jump between a reference and its declaration, as well as refactor by renaming all occurrences. This information could also be used for semantic highlighting, or using colors to differentiate between individual references.

Tokens

It may be useful to abstract away syntax, but for syntax highlighting or code formatting, the representation is what actually matters.

A token represents a keyword, identifier, punctuation, literal, or comment, which exists at a particular location and range. `clang_tokenize` tokenizes the content of a translation unit within a particular range:

```
CXCursor cursor = clang_getTranslationUnitCursor(tu);
CXSourceRange range = clang_getCursorExtent(cursor);

CXToken* tokens;
unsigned count;
clang_tokenize(tu, range, &tokens, &count);

for (unsigned i = 0; i < count; i++) {
    CXToken token = tokens[i];
    unsigned line, column, offset;
    clang_getFileLocation(clang_getTokenLocation(tu, token),
                          &file, &line, &column, &offset);
    switch (clang_getTokenKind(token)) {
        case CXToken_Punctuation:
        case CXToken_Keyword:
        case CXToken_Identifier:
        case CXToken_Literal:
        case CXToken_Comment:
            // ...
            break;
        default:
            break;
    }
}
```

Xcode uses this to provide syntax highlighting for source code. In order to avoid recoloring the entire document on every keypress, the highlighter can constrain its tokenization to just the immediate lexical scope of the change.

Diagnostics

Where Clang really starts to show off its smarts are through diagnostics.

Clang diagnostics are ranked into different levels of severity, much like log statements:

`CXDiagnostic_Ignored`	A diagnostic that has been suppressed, e.g., by a command-line option.
`CXDiagnostic_Note`	This diagnostic is a note that should be attached to the previous (non-note) diagnostic.
`CXDiagnostic_Warning`	This diagnostic indicates suspicious code that may not be wrong.
`CXDiagnostic_Error`	This diagnostic indicates that the code is ill-formed.
`CXDiagnostic_Fatal`	This diagnostic indicates that the code is ill-formed such that future parser recovery is unlikely to produce useful results.

Diagnostics are evaluated in the scope of an entire translation unit, since the correctness of any one declaration is dependent on its preceding context. `clang_getNumDiagnostics` gets the total number of diagnostics for the translation unit, which can be enumerated in a `for` statement:

```
for (unsigned i = 0; i < clang_getNumDiagnostics(tu); i++) {
    CXDiagnostic diagnostic = clang_getDiagnostic(tu, i);
    CXString string =
        clang_formatDiagnostic(diagnostic,
                               clang_defaultDiagnosticDisplayOptions());

    switch (clang_getDiagnosticSeverity(diagnostic)) {
        case CXDiagnostic_Note:
        case CXDiagnostic_Warning:
        case CXDiagnostic_Error:
        case CXDiagnostic_Fatal:
            // ...
            break;
        case CXDiagnostic_Ignored:
        default:
            break;
    }

    // ...

    clang_disposeString(string);
}
```

Xcode annotates source code with Clang diagnostics with visual indicators in the gutter, which can be disclosed to highlight offending code in-line. [27]

Fix-Its

It's one thing to be able to point out problems, but it's another thing entirely to fix them as well. Clang fix-its take diagnostics to a whole new level.

For each diagnostic, there are any number of potential changes that can be made to address the issue. These options would typically be presented to the end-user in order to determine the best course of action:

```
for (unsigned j = 0; j < clang_getDiagnosticNumFixIts(diagnostic); j++) {
    CXSourceRange range;
    CXString fixIt =
        clang_getDiagnosticFixIt(diagnostic, j, &range);

    CXSourceLocation start = clang_getRangeStart(range);
    unsigned startLine, startColumn;
    clang_getSpellingLocation(start, 0, &startLine , &startColumn, 0);

    // ...
```

27 Yellow caution icons are displayed for `CXDiagnostic_Note` and `CXDiagnostic_Warning`. Red error icons are displayed for `CXDiagnostic_Error` and `CXDiagnostic_Fatal`.

```
    clang_disposeString(fixIt);
}
```

Xcode denotes fix-its with a dot over the yellow or red diagnostic gutter icon. When activated, all of the available options are presented, which when selected, automatically make the necessary changes to (hopefully) fix the code.

Clang CLI

Clang's insight into code can also be accessed via the command line.

Consider the following piece of source code:

```
@import Foundation;

/**
  This is documentation.
*/
@interface Calculator : NSObject

+ (int)add:(int)a
        to:(int)b;

@end

@implementation Calculator
```

```
+ (int)add:(int)a
        to:(int)b
{
    int sum = a + b;
    return sum;
}

@end
```

If we pass this into `xcrun clang` and include the `-ast-dump` flag, the output is an abstract syntax tree of the source file:

```
$ xcrun -sdk iphoneos clang -x objective-c -Xclang -ast-dump -fsyntax-only ↩
     Calculator.m
```

```
|-ObjCInterfaceDecl 0x1036f8040 <Calculator.m:6:1, line:11:2> Calculator
| |-super ObjCInterface 0x10501c3b0 'NSObject'
| |-ObjCImplementation 0x1036f82d0 'Calculator'
| |-FullComment 0x1036f8820 <line:4:1, col:24>
| | `-ParagraphComment 0x1036f87f0 <col:1, col:24>
| |   `-TextComment 0x1036f87c0 <col:1, col:24> Text="  This is documentation."
| `-ObjCMethodDecl 0x1036f8170 <line:8:1, line:9:18> + add:to: 'int'
|   |-ParmVarDecl 0x1036f8200 <line:8:13, col:17> a 'int'
|   `-ParmVarDecl 0x1036f8260 <line:9:13, col:17> b 'int'
`-ObjCImplementationDecl 0x1036f82d0 <line:13:1, line:22:1> Calculator
  |-ObjCInterface 0x1036f8040 'Calculator'
  `-ObjCMethodDecl 0x1036f8390 <line:15:1, line:20:1> + add:to: 'int'
    |-ImplicitParamDecl 0x1036f84f0 <<invalid sloc>> self 'Class':'Class'
    |-ImplicitParamDecl 0x1036f8550 <<invalid sloc>> _cmd 'SEL':'SEL *'
    |-ParmVarDecl 0x1036f8420 <line:15:13, col:17> a 'int'
    |-ParmVarDecl 0x1036f8480 <line:16:13, col:17> b 'int'
    |-VarDecl 0x1036f85c0 <line:18:5, col:19> sum 'int'
    | `-BinaryOperator 0x1036f8698 <col:15, col:19> 'int' '+'
    |   |-ImplicitCastExpr 0x1036f8668 <col:15> 'int' <LValueToRValue>
    |   | `-DeclRefExpr 0x1036f8618 <col:15> 'int' lvalue ParmVar 0x1036f8420 ' ↩
     a' 'int'
    |   `-ImplicitCastExpr 0x1036f8680 <col:19> 'int' <LValueToRValue>
    |     `-DeclRefExpr 0x1036f8640 <col:19> 'int' lvalue ParmVar 0x1036f8480 ' ↩
     b' 'int'
    `-CompoundStmt 0x1036f8738 <line:17:1, line:20:1>
      |-DeclStmt 0x1036f86c0 <line:18:5, col:20>
      | `-VarDecl 0x1036f85c0 <col:5, col:19> sum 'int'
      |   `-BinaryOperator 0x1036f8698 <col:15, col:19> 'int' '+'
      |     |-ImplicitCastExpr 0x1036f8668 <col:15> 'int' <LValueToRValue>
      |     | `-DeclRefExpr 0x1036f8618 <col:15> 'int' lvalue ParmVar 0 ↩
     x1036f8420 'a' 'int'
      |     `-ImplicitCastExpr 0x1036f8680 <col:19> 'int' <LValueToRValue>
      |       `-DeclRefExpr 0x1036f8640 <col:19> 'int' lvalue ParmVar 0 ↩
     x1036f8480 'b' 'int'
      `-ReturnStmt 0x1036f8718 <line:19:5, col:12>
        `-ImplicitCastExpr 0x1036f8700 <col:12> 'int' <LValueToRValue>
          `-DeclRefExpr 0x1036f86d8 <col:12> 'int' lvalue Var 0x1036f85c0 'sum' ↩
      'int'
```

Look past the noisy memory pointers and angled brackets, and the structure of the source code emerges from the AST branches. At the root is the top-level class declaration for `Calculator`, with its immediate children pointing to its superclass implementation, `NSObject` and the class implementation that follows, as well as the comment and class method declaration. In the implementation, there is a reference back to the interface, as well as the class method implementation, with parameters, returns type, variable declarations, and return statement.

Static Analyzer

Within the context of programming, static analysis refers to algorithms and techniques used to analyze source code in order to automatically find bugs. The idea is similar in spirit to compiler warnings, which can be useful for finding coding errors, but taken a step further to find bugs that would otherwise be encountered at runtime.

Bug-finding tools have evolved over several decades, from basic syntactic checkers to those that find deep bugs by reasoning about the semantics of code. The goal of the Clang Static Analyzer is to provide a industrial-quality static analysis framework for analyzing C, C++, and Objective-C programs that is freely available, extensible, and has a high quality of implementation. Just a cursory look at all of the detectable warnings, it's clear that Clang has lived up to its own expectations:

Core Analyzer Warnings

`core.CallAndMessage`	Check for logical errors for function calls and Objective-C message expressions (e.g., uninitialized arguments, null function pointers).
`core.DivideZero`	Check for division by zero.
`core.NonNullParamChecker`	Check for null pointers passed as arguments to a function whose arguments are marked with the nonnull attribute.
`core.NullDereference`	Check for dereferences of null pointers.
`core.StackAddressEscape`	Check that addresses of stack memory do not escape the function.
`core.UndefinedBinaryOperatorResult`	Check for undefined results of binary operators.
`core.VLASize`	Check for declarations of VLA of undefined or zero size.
`core.uninitialized.ArraySubscript`	Check for uninitialized values used as array subscripts.
`core.uninitialized.Assign`	Check for assigning uninitialized values.

(continued)

`core.uninitialized.Branch`	Check for uninitialized values used as branch conditions.
`core.uninitialized.CapturedBlockVariable`	Check for blocks that capture uninitialized values.
`core.uninitialized.UndefReturn`	Check for uninitialized values being returned to the caller.

Dead Code Analyzer Warnings

`deadcode.DeadStores`	Check for values stored to variables that are never read afterwards.

OS X Analyzer Warnings

`osx.API`	Check for proper uses of various Apple APIs (`dispatch_once`)
`osx.SecKeychainAPI`	Check for improper uses of the Security framework's Keychain APIs

(continued)

`osx.cocoa.AtSync`	Check for nil pointers used as mutexes for @synchronized.
`osx.cocoa.ClassRelease`	Check for sending retain, release, or autorelease directly to a class.
`osx.cocoa.IncompatibleMethodTypes`	Check for an incompatible type signature when overriding an Objective-C method.
`alpha.osx.cocoa.MissingSuperCall`	Warn about Objective-C methods that lack a necessary call to super.
`osx.cocoa.NSAutoreleasePool`	Warn for suboptimal uses of NSAutoreleasePool in Objective-C GC mode (-fobjc-gc compiler option).
`osx.cocoa.NSError`	Check usage of NSError** parameters.
`osx.cocoa.NilArg`	Check for prohibited nil arguments in specific Objective-C method calls (`compare:`, et. al.)
`osx.cocoa.RetainCount`	Check for leaks and violations of the Cocoa Memory Management rules.
`osx.cocoa.SelfInit`	Check that self is properly initialized inside an initializer method.
`osx.cocoa.UnusedIvars`	Warn about private ivars that are never used.

(continued)

`osx.cocoa.VariadicMethodTypes`	Check for passing non-Objective-C types to variadic collection initialization methods that expect only Objective-C types.
`osx.coreFoundation.CFError`	Check usage of CFErrorRef* parameters.
`osx.coreFoundation.CFNumber`	Check for improper uses of CFNumberCreate.
`osx.coreFoundation.CFRetainRelease`	Check for null arguments to CFRetain, CFRelease, CFMakeCollectable.
`osx.coreFoundation.containers.OutOfBounds`	Checks for index out-of-bounds when using CFArray API.
`osx.coreFoundation.containers.PointerSizedValues`	Warns if CFArray, CFDictionary, CFSet are created with non-pointer-size values.

`security.FloatLoopCounter`	Warn on using a floating point value as a loop counter (CERT: FLP30-C, FLP30-CPP).
`security.insecureAPI.UncheckedReturn`	Warn on uses of functions whose return values must be always checked: `setuid`, `setgid`, `seteuid`, `setegid`, `setreuid`, `setregid`
`security.insecureAPI.getpw`	Warn on uses of the getpw function.
`security.insecureAPI.gets`	Warn on uses of the gets function.
`security.insecureAPI.mkstemp`	Warn when mktemp, mkstemp, mkstemps or mkdtemp is passed fewer than 6 X's in the format string.
`security.insecureAPI.mktemp`	Warn on uses of the mktemp function.
`security.insecureAPI.rand`	Warn on uses of inferior random number generating functions (only if `arc4random` function is available): `drand48`, `erand48`, `jrand48`, `lcong48`, `lrand48`, `mrand48`, `nrand48`, `random`, `rand_r`

(continued)

`security.insecureAPI.strcpy`	Warn on uses of the strcpy and strcat functions.
`security.insecureAPI.vfork`	Warn on uses of the vfork function.

Unix Analyzer Warnings

`unix.API`	Check calls to various UNIX/POSIX functions: `open`, `pthread_once`, `calloc`, `malloc`, `realloc`, `alloca`
`unix.Malloc`	Check for memory leaks, double free, and use-after-free and offset problems involving malloc.
`unix.MallocSizeof`	Check for dubious malloc, calloc or realloc arguments involving sizeof.
`unix.MismatchedDeallocator`	Check for mismatched deallocators (e.g. passing a pointer allocating with new to free()).
`unix.cstring.BadSizeArg`	Check the size argument passed to strncat for common erroneous patterns. Use -Wno-strncat-size compiler option to mute other strncat-related compiler warnings.

(continued)

`unix.cstring.NullArg`	Check for null pointers being passed as arguments to C string functions: `strlen`, `strnlen`, `strcpy`, `strncpy`, `strcat`, `strncat`, `strcmp`, `strncmp`, `strcasecmp`, `strncasecmp`

Xcode integrates Clang's static analysis into the Build & Analyze command, which does a pretty remarkable job at tracing problems back to their root cause. However, the same functionality is available via the command line as well.

Clang Analyzer is not included as a standalone utility in the Xcode Developer Tools, and must be compiled from source, which is available for download on the project website. [a].

[a] http://clang-analyzer.llvm.org

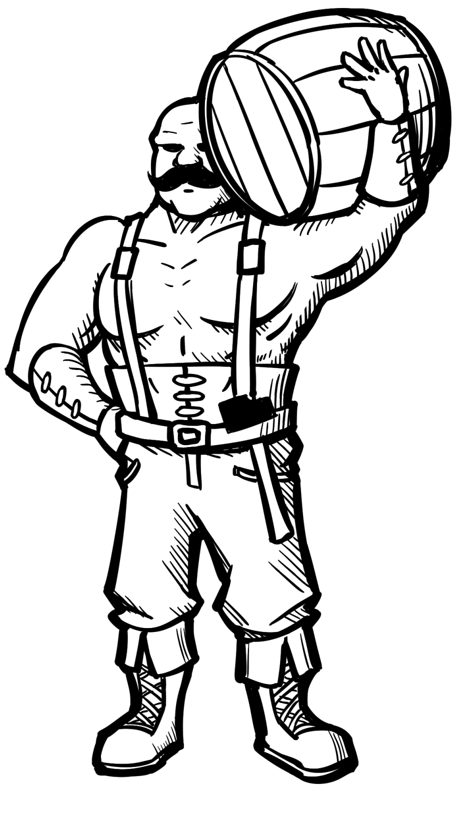
After using `xcodebuild` to build an Xcode project, execute `scan-build` with the same arguments to get an HTML-formatted readout of all of the analyzer warnings. [28]

28 Always build using the debug configuration when running the static analyzer.

```
$ xcodebuild -project /path/to/Project.xcodeproj -scheme YOU_APP_SCHEME -sdk ↩
     iphonesimulator7.1 analyze
$ scan-build !!
```

Chapter 4

OSAtomic



It is telling that the word used to describe structure and organization in a system is "order".

Human logic and understanding are entirely contingent on a universe operating in linear, chronological fashion, with every action traceable back to its mover, through an unbroken chain of causality.

This is what makes concurrent programming so difficult. Allowing state to be mutated by multiple threads at the same time creates chaos. There can be no guarantees that anything will be as it was from moment to moment, making bugs difficult to predict, understand, and reproduce. Worse still is that looks are often deceiving. Many operations that seem atomic in nature are, in reality, decomposable into several sub-operations, that, without safeguards, are vulnerable to race conditions.

`<libkern/OSAtomic.h>` are the kernel-level concurrency API headers for iOS and OS X. Although many, if not most, of these functions are dispreferred to higher-level APIs, namely those provided by `libdispatch`, they are still informative as core constructs in concurrent programming.

OSAtomic functionality can be divided up into four distinct categories:

- Integer Operations
- Compare & Swap Operations
- Spinlocks
- Queues

Integer Operations

One of the most basic operations in programming is incrementing an integer. It's so common a task, that it can be accomplished with several equivalent operators:

```
int x = 3;

// Equivalent
x = x + 1;
x += 1;
x++;
```

However, what looks like one operation—incrementing an integer variable—is actually comprised of three distinct steps.

```
* Get x (3)
* Perform Addition (3 + 1)
* Set x (4)
```

For a single-threaded application, there is a reasonable guarantee `x` will only ever be accessed in that order: "get-add-set". But throw another thread into the mix, and any such guarantee is up in smoke.

As an example, imagine that two threads, T_1 & T_2, both attempt to perform `x++` in short succession of one another:

```
- T~1~: Get X (3)
- T~2~: Get X (3)
- T~1~: Perform Addition (3+1)
- T~1~: Set X (4)
- T~2~: Perform Addition (3+1)
- T~2~: Set X (4)
```

In this scenario, the result would be a missed increment, such that `x ==4` despite it being incremented twice from `x ==3`. [29]

OSAtomic provides thread-safe operators that ensure that a memory address is only read by a single thread at any given moment.

The following code demonstrates a thread-safe OSAtomic equivalent to the previous code:

29 This is known as the ABA problem, wherein a the same value is read by two different threads at the same time, resulting in missed work.

```
int64_t x = 3;

// Equivalent
OSAtomicAdd64(1, &x);
OSAtomicIncrement64(&x);
```

As a matter of simplicity, OSAtomic operations work with a constrained set of fixed-length types—specifically integers of either 32-bit or 64-bit length. Operations include atomic increment, decrement, and add (subtraction can be accomplished by negating the addend), as well as bitwise `AND`, `OR`, & `XOR`.

Increment / Decrement / Addition

```
int64_t x = 3, y = 5;
```

`x++`	`OSAtomicIncrement64(&x)`
`x--`	`OSAtomicDecrement64(&x)`
`x +=y`	`OSAtomicAdd64(y, &x)`
`x -=y`	`OSAtomicAdd64(-y, &x)`

Bitwise AND / OR / XOR

```
uint32_t p = 0xf00, q = 0xf0f;
```

`p &=q`	`OSAtomicAnd32(q, &p)`
`p |=q`	`OSAtomicOr32(q, &p)`
`p ˆ=q`	`OSAtomicXor32(q, &p)`

Compare & Swap Operations

Compare & Swap is the fundamental atomic operation, from which all others can be derived. After all, the challenge of multi-threaded programming is to ensure that values don't get read when they're about to be written to by another process. Controlling access to a memory address is the key to synchronizing concurrent processes.

Still not convinced? Here's how one might implement the previous atomic integer addition function using `OSAtomicCompareAndSwap32`:

```
int32_t Add32(int32_t amount, volatile int32_t *value) {
    BOOL success;
    int32_t new;
    do {
        int32_t old = *value;
        new = old + amount;
        success = OSAtomicCompareAndSwap32(old, new, value);
    } while(!success);

    return new;
}
```

Compare & Swap is actually a single operation in OSAtomic. A naive, non-thread-safe C implementation might look like this:

```
BOOL CompareAndSwap(int old, int new, int *value) {
    if(*value == old) {
        *value = new;
        return YES;
    } else {
        return NO;
    }
}
```

Like the integer operations, Compare & Swap functions comes in both 32- and 64-bit, as well as barrier and non-barrier varieties. In addition, there are versions that take `int` and `long` arguments, as well as `OSAtomicCompareAndSwapPtr`, which can be used to safely heap-allocated object pointers.

Memory Barriers

Many OSAtomic functions provide both barrier and non-barrier variants. A barrier enforces ordering of memory access across threads. When a barrier is created, operations before the barrier are guarantee to finish before those created after the barrier can begin. The barrier and non-barrier varieties of the aforementioned functions differ by whether or not they incorporate a memory barrier.

30

Test & Set / Clear

Similar to Compare & Swap, `OSAtomicTestAndSet` atomically sets a bit in the specified variable to 1, and returns that value. Its negation, `OSAtomicTestAndClear` does the same, but setting a 0 instead.

These functions can be used to create a semaphore to direct concurrent execution flow. Again, like most things in OSAtomic, `libdispatch` offers a safer & more convenient API, so there's really no reason to use anything but `dispatch_semaphore` at the application level.

Spin Locks

Spinlocks are perhaps the easiest type of lock to understand. They instruct the thread to hang out and wait until the lock is released. It's a

30 As a general rule, semaphores, counters, and other standalone values that fit within a single 32- or 64-bit memory address don't need barriers. Anything where values aren't self-contained, or involve data outside of the value should use a barrier.

busy form of waiting, akin to directing an airplane into a holding pattern until other traffic clears the runway. This might sound inefficient, but in situations where the lock is not held onto for very long, a spinlock has superior performance characteristics over its alternatives.

OSAtomic spin locks sit between pthread spinlocks and `NSLock` in terms of abstraction. In most situations where a spinlock might be called for, a serial dispatch queue or `NSLock` would be likely be preferable.

Before libdispatch, though, `OSAtomicLock/Unlock` in combination with `OSAtomicTestAndSet` offered one of the most effective thead-safe singleton patterns available at the time. Again, this is strictly worse to `dispatch_once` in a modern application, but in the interest of curiosity, here's what that looks like:

```
static id _sharedInstance = nil;
static int32_t onceToken = 0;
if (!OSAtomicTestAndSet(1, &onceToken)) {
    static OSSpinLock lock = OS_SPINLOCK_INIT;
    OSSpinLockLock(&lock);
    _sharedInstance = [[[self class] alloc] init];
    OSSpinLockUnlock(&lock);
}
```

Queues

`OSAtomicEnqueue` & `OSAtomicDequeue` are a lock-free, thread-safe implementation of a LIFO queue. There are also FIFO queue variants `OSAtomicFifoEnqueue` & `OSAtomicFifoDequeue`.

```
typedef struct node {
    int value;
    volatile struct node *link;
} node_t;

node_t a = {1, NULL};
node_t b = {2, NULL};

OSQueueHead queue = OS_ATOMIC_QUEUE_INIT;
OSAtomicEnqueue(&queue, &a, offsetof(node_t, link));
OSAtomicEnqueue(&queue, &b, offsetof(node_t, link));

node_t *n;
n = OSAtomicDequeue(&queue, offsetof(node_t, link));
// n == &b

n = OSAtomicDequeue(&queue, offsetof(node_t, link));
// n == &a
```

Before GCD queues, `OSAtomicEnqueue/Dequeue` provided a respectable mechanism for safely scheduling work in a multi-threaded environment.

But since GCD *does* exist, you're almost certainly better off with `dispatch_queue`, which has a number of additional benefits, which are discussed in depth in the next chapter.

Chapter 5

Grand Central Dispatch

One of the most striking things about process control is how close reality matches the domain models of computers. You won't happen upon a `String` as you walk down the street, or decide to hold a lunch conversation over a `socket`. But you will stand in a `queue`, and wait for a `signal` before crossing a street. Semaphores were flags and locks were made of metal long before they were a cause of application deadlock, after all.

Determining how best to schedule resources in order to perform work is as directly applicable to everyday life as programming gets.

Maybe it's for this reason that concurrency is such a mainstay of programmer humor. *Jokes in which a thread order of the the changes punchline, or locks sudd—*. There's definitely an element of gallows humor to it, because—let's be honest—threading is really hard to get right. But perhaps moreso, these jokes evoke absurdity about the way things are supposed to operate.

Setting aside the philosophy of humor, one thing is clear: for concurrent programming, Grand Central Dispatch is seriously awesome.

Grand Central Dispatch (GCD) is a technology for optimizing performance on multiprocessor systems. Introduced with the

C-language blocks extension in iOS 4 and OS X 10.6, GCD is used throughout Cocoa APIs to make applications faster and more efficient.

Apple's implementation of GCD, `libdispatch`, is open source, and available for download from Mac OS Forge. [a]

[a] https://libdispatch.macosforge.org

Queues

In GCD, work is divided up into discrete blocks or functions, which are scheduled on dispatch queues. Queues abstract the concept of threads from programmers. The system provides a main queue, which executes work on the main thread, in addition to several global queues that execute on background threads at different priority levels. In addition to these, users can create their own queues.

Custom queues can either be serial (executing one task at a time) or concurrent (executing multiple tasks at once).

```
dispatch_queue_t mainQueue = dispatch_get_main_queue();

dispatch_queue_t globalQueue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

```
dispatch_queue_t customSerialQueue =
    dispatch_queue_create("com.example.serial", DISPATCH_QUEUE_SERIAL);

dispatch_queue_t customConcurrentQueue =
    dispatch_queue_create("com.example.concurrent", DISPATCH_QUEUE_CONCURRENT);
```

Work is scheduled on a queue to execute either synchronously or asynchronously. Specifying synchronous execution will have the queue wait until the block or function terminates, whereas asynchronous means that execution will proceed to the next statement immediately.

```
dispatch_queue_t mainQueue = dispatch_get_main_queue();
dispatch_sync(mainQueue, ^{
    sleep(3);
    NSLog(@"Finished");
});
```

A common pattern with GCD is to dispatch work to a background queue, and then return results on the main queue. This is especially important for things like updating the UI, which needs to be done on the main thread.

```
dispatch_queue_t mainQueue = dispatch_get_main_queue();
dispatch_queue_t globalQueue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(globalQueue, ^{
    sleep(3);

    dispatch_async(mainQueue, ^{
        NSLog(@"Finished");
    });
});
```

Tasks can also be scheduled to be run on a queue after a specified delay:

```
dispatch_queue_t mainQueue = dispatch_get_main_queue();
int64_t delay = 1 * NSEC_PER_SEC;
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, delay), mainQueue, ^{
    NSLog(@"Finished");
});
```

A lesser-known, yet useful feature of GCD queues is applying a block over a range of integers. When run on a concurrent queue, `dispatch_apply` offers a highly concurrent alternative to garden variety `for` loops:

```
dispatch_queue_attr_t attributes =  DISPATCH_QUEUE_CONCURRENT;
dispatch_queue_t queue = dispatch_queue_create(NULL, attributes);
dispatch_apply(1000, queue, ^(size_t n) {
    size_t square = n * n;
    printf("%zu: %zu\n", n, square);
    sleep(1);
});
```

As mentioned in the previous chapter, GCD can be used to implement robust, thread-safe implementations of common atomic operations. For example, `dispatch_once` can guarantee that a statement is executed exactly once—making it perfectly suited for creating singletons:

```
+ (instancetype)sharedInstance {
    static id _sharedInstance = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        _sharedInstance = [[self alloc] init];
    });

    return _sharedInstance;
}
```

Groups

Tasks can also be scheduled in groups, providing a callback for when all tasks within that group are finished executing:

```
dispatch_queue_t queue =
    dispatch_queue_create(NULL, DISPATCH_QUEUE_CONCURRENT);
dispatch_group_t group =
dispatch_group_create();

dispatch_apply(10, queue, ^(size_t n) {
    dispatch_group_enter(group);
    dispatch_group_async(group, queue, ^{
        sleep((int)n);
        dispatch_group_leave(group);
    });
});

dispatch_group_notify(group, queue, ^{
    // ...
});
```

Semaphores

Semaphores play a crucial role in GCD, by allowing asynchronous code to wait and block execution, thereby becoming synchronous. For many applications, asynchronous execution is strictly preferable to the alternative. However, in some cases, an API must be run synchronously. And it is in those cases, where dispatch semaphore shines:

```
dispatch_queue_t queue =
  dispatch_queue_create(NULL, DISPATCH_QUEUE_CONCURRENT);

dispatch_semaphore_t semaphore =
  dispatch_semaphore_create(0);

dispatch_async(queue, ^{
    sleep(3);
    dispatch_semaphore_signal(semaphore);
});

dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
```

Barriers

Barriers are another concept explored in the previous chapter. In the case of GCD, barriers are used to synchronize access to shared state.

Without Barrier

```
dispatch_queue_t queue =
    dispatch_queue_create(NULL, DISPATCH_QUEUE_CONCURRENT);

dispatch_async(queue,^{
   sleep(3);

   dispatch_async(queue,^{
      sleep(5);

      dispatch_async(dispatch_get_main_queue(), ^{
          NSLog(@"Finished");
      });
   });
});
```

With Barrier

```
dispatch_queue_t queue =
    dispatch_queue_create(NULL, DISPATCH_QUEUE_CONCURRENT);

dispatch_async(queue,^{
    sleep(5);
});

dispatch_async(queue,^{
    sleep(3);
});

dispatch_barrier_async(queue,^{
    dispatch_async(dispatch_get_main_queue(),^{
        // ...
    });
});
```

This is especially useful for methods that mutate a collection, such as a backing array or dictionary:

```
@property NSMutableDictionary *mutableDictionary;
@property dispatch_queue_t queue;

- (void)setObject:(id)object
           forKey:(id)key
{
    dispatch_barrier_async(self.queue, ^{
        self.mutableDictionary[key] = object;
    });
}
```

Sources

GCD can be used to handle events from sources like timers, processes, mach ports, and file descriptors. Dispatch sources start suspended, and must be explicitly resumed in order to start.

A timer dispatch event source can be thought of as a more flexible alternative to `dispatch_after`, with the ability to be cancelled, and offer leeway to minimize the performance impact of temporally misaligned instructions:

```
dispatch_queue_t queue =
    dispatch_queue_create(NULL, DISPATCH_QUEUE_CONCURRENT);

dispatch_source_t timer =
    dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER, 0, 0, queue);

int64_t delay = 30 * NSEC_PER_SEC;
int64_t leeway = 5 * NSEC_PER_SEC;
dispatch_source_set_timer(timer, DISPATCH_TIME_NOW, delay , leeway);

dispatch_source_set_event_handler(timer, ^{
    NSLog(@"Ding Dong!");
});

dispatch_resume(timer);
```

To monitor a file or directory for changes, create a dispatch event for a file descriptor. Whenever one of the watched events is triggered, the event handler will be scheduled on the specified queue:

```
dispatch_queue_t queue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

NSURL *fileURL = [[[NSFileManager defaultManager]
                    URLsForDirectory:NSDocumentDirectory
                          inDomains:NSUserDomainMask] firstObject];

int fileDescriptor =
  open([fileURL fileSystemRepresentation], O_EVTONLY);

unsigned long mask = DISPATCH_VNODE_EXTEND |
                     DISPATCH_VNODE_WRITE  |
                     DISPATCH_VNODE_DELETE;
__block dispatch_source_t source =
    dispatch_source_create(DISPATCH_SOURCE_TYPE_VNODE,
                           fileDescriptor,
                           mask,
                           queue);

dispatch_source_set_event_handler(source, ^{
    dispatch_source_vnode_flags_t flags =
        dispatch_source_get_data(source);

    if (flags) {
        dispatch_source_cancel(source);
        dispatch_async(dispatch_get_main_queue(), ^{
            // ...
        });
    }
});

dispatch_source_set_cancel_handler(source, ^{
    close(fileDescriptor);
});

dispatch_resume(source);
```

A similar approach can be used to read from STDIN:

```
dispatch_queue_t globalQueue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_source_t stdinReadSource =
    dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
                           STDIN_FILENO,
                           0,
                           globalQueue);

dispatch_source_set_event_handler(stdinReadSource, ^{
    uint8_t buffer[1024];
    int length = read(STDIN_FILENO, buffer, sizeof(buffer));
    if (length > 0) {
        NSString *string =
            [[NSString alloc] initWithBytes:buffer
                                     length:length
                                   encoding:NSUTF8StringEncoding];
        NSLog(@"%@", string);
    }
});

dispatch_resume(stdinReadSource);
```

Finally, a dispatch source can listen for process signals, such as a `SIGTERM`:

```
pid_t ppid = getppid();

dispatch_queue_t globalQueue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_source_t source =
    dispatch_source_create(DISPATCH_SOURCE_TYPE_PROC,
                           ppid,
                           DISPATCH_PROC_EXIT,
                           globalQueue);
if (source) {
    dispatch_source_set_event_handler(source, ^{
        NSLog(@"pid: %d Exited", ppid);
        dispatch_source_cancel(source);
    });

    dispatch_resume(source);
}
```

I/O

Although dispatch sources provide a convenient way to interact with a input and output, it requires quite a bit of responsibility on the part of the API consumer. GCD's I/O APIs allow the developer to hand over most of that responsibility, which not only makes for less code to write, but greatly improves the overall capacity for concurrent I/O operations by reducing resource contention.

The dispatch I/O APIs operate on channels. Each channel manages a file descriptor, reading data either as a stream or allowing for random access of content. When a channel is created, it takes control of the file descriptor until one of the following occurs: the channel is closed, all references to the channel are released, or an error occurs.

Here's how to create a dispatch channel to STDIN:

```
dispatch_queue_t queue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_io_t stdinChannel =
  dispatch_io_create(DISPATCH_IO_STREAM,
                     STDIN_FILENO,
                     queue,
  ^(int error) {
    if (error) {
      NSLog(@"stdin: (%d) %s", error, strerror(error));
    }
  });
```

Before reading from a channel, it's important to tune it specifically for the desired use case. The critical factor of working with I/O is determining how often to process data. You can do this one of two ways: wait for a certain amount of data to accumulate, or wait for a certain amount of time to pass.

Specifying what constitutes a meaningful amount of data can be accomplished by setting a low and high water mark—the minimum and maximum amount of data to gather before invoking the handler. In the case of reading STDIN, input is usually interactive, so it makes sense to set a low water mark of a single byte:

```
dispatch_io_set_low_water(stdinChannel, 1);
```

While it doesn't make sense to do so in this example, the corresponding `dispatch_io_set_high_water` function can lower the upper bounds from its default `SIZE_MAX` value, to something more reasonable for situations like parsing large amounts of data streamed over a socket.

Specifying the time interval to wait between processing data can be accomplished with the `dispatch_io_set_interval` function, which accepts time intervals at nanosecond resolution. Again, this isn't a great fit for processing from STDIN, but it would be a great approach for things like capturing audio or video data from a peripheral for sampling.

Once a channel is configured, it can start reading data:

```
off_t offset = 0;         // Ignored for stream
UInt length = SIZE_MAX;   // Read until EOF
dispatch_io_read(stdinChannel, offset, length, queue,
  ^(bool done, dispatch_data_t data, int error) {
    // ...
});
```

Data can also be written to a channel in a similar manner. Creating a new channel to a file path makes for a rudimentary logging tool:

```
dispatch_io_t fileChannel =
  dispatch_io_create_with_path(DISPATCH_IO_STREAM,
                               "/path/to/file",
                               O_RDONLY,
                               0,
                               queue,
  ^(int error) {
    if (error) {
      NSLog(@"file: (%d) %s", error, strerror(error));
    }
  });

dispatch_io_read(stdinChannel, offset, length, queue,
  ^(bool done, dispatch_data_t data, int error) {
    if (data) {
      dispatch_write(fileChannel, 0, data, queue, nil);
    }
});
```

Alternatively, GCD also provides `dispatch_read` and `dispatch_write`, which are convenience methods built on top of `dispatch_io_read` and `dispatch_io_write` for simple, one-off I/O operations.

Data

When first introduced, dispatch data objects were unique within Apple's SDKs for being a container for both contiguous and noncontiguous data. The major implication being that two data objects could be concatenated in constant time, without having to copy over into a single continuous segment. As of iOS 7 and OS X 10.9, `NSData` added support for noncontiguous access, as well as a one-way cast from `dispatch_data_t` objects. [31] [32]

Perhaps the best way to understand dispatch data is to say that it has all of the convenience of `NSData` in a low-level C interface.

`dispatch_data_create` constructs a dispatch data object from a buffer:

31 According to the Foundation release notes: "In 64-bit apps using either manual retain/release or ARC, dispatch_data_t can now be freely cast to NSData *, though not vice versa."

32 Although no specific details are made explicit in the documentation, one could reasonably speculate that `NSData` and `NSFileHandle` both had their underlying implementations replaced to use their GCD equivalents, dispatch data and dispatch I/O, in this release.

```
size_t length;
void *buffer = malloc(length);
dispatch_data_t data =
  dispatch_data_create(buffer,
                       length,
                       NULL,
                       DISPATCH_DATA_DESTRUCTOR_DEFAULT);
free(buffer);
```

It can even do the `free` call automatically, by passing the destructor `DISPATCH_DATA_DESTRUCTOR_FREE`.

`dispatch_data_create_concat` can create a new dispatch data object by concatenating two existing objects:

```
dispatch_data_t first, second;
dispatch_data_t combined = dispatch_data_create_concat(first, second);
```

Using this, and taking advantage of the new `enumerateByteRangesUsingBlock:` method in `NSData`, a function can be created to construct a dispatch data object from `NSData` (whereas only `NSData` to `dispatch_data_t` is provided by the framework):

```
dispatch_data_t dispatch_data_create_with_nsdata(NSData *data) {
    dispatch_queue_t queue =
        dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    __block dispatch_data_t container;
    [data enumerateByteRangesUsingBlock:
     ^(const void *bytes, NSRange byteRange, BOOL *stop) {
        if (container) {
            dispatch_data_t region =
                dispatch_data_create(bytes,
                                     byteRange.length,
                                     queue,
                                     DISPATCH_DATA_DESTRUCTOR_DEFAULT);
            container = dispatch_data_create_concat(container, region);
        }
    }];

    return container;
}
```

GCD's equivalent to `NSData -enumerateByteRangesUsingBlock:` is `dispatch_data_apply`, which executes a block for each memory region contained within the container:

```
dispatch_data_apply(data, ^(dispatch_data_t region, size_t offset, const void * ↩
     buffer, size_t size) {
    // ...
    return true;
});
```

Or, when interacting with calls that operate on a single, contiguous buffer, use `dispatch_data_create_map`:

```
void *buffer;
size_t length;
dispatch_data_create_map(data, &buffer, &length);
```

Debugging

As of OS X 10.8 and iOS 6.0, GCD types are full-fledged NSObject subclasses, which respond to `-debugDescription`. This means that doing `po` in lldb will return useful diagnostic output, e.g.:

```
<OS_dispatch_queue_root:
    com.apple.root.default-priority[0x2024100] = {
        xrefcnt = 0x80000000,
        refcnt = 0x80000000,
        suspend_cnt = 0x0,
        locked = 1,
        target = [0x0],
        width = 0x7fffffff,
        running = 0x1,
        barrier = 0
    }
>
```

Benchmarking

`dispatch_benchmark` is part of libdispatch, but is not publicly available. To use it, it must be re-declared:

```
uint64_t dispatch_benchmark(size_t count, void (^block)(void));;
```

`dispatch_benchmark` executes a block the specified number of times, and then returns the average execution runtime, in nanoseconds:

```
size_t const objectCount = 1000;
uint64_t n = dispatch_benchmark(10000, ^{
    @autoreleasepool {
        id obj = @42;
        NSMutableArray *array = [NSMutableArray array];
        for (size_t i = 0; i < objectCount; ++i) {
            [array addObject:obj];
        }
    }
});
NSLog(@"-[NSMutableArray addObject:] : %llu ns", n);
```

Chapter 6

Inter-Process Communication

Up until this point in the book, the guiding narrative has been technologies fusing together through happy accidents of history to create something better than before. And, while this is true for many aspects of Apple's technology stack, inter-process communication is a flagrant counter-example.

Rather than taking the best of what was available at each juncture, solutions just kinda piled up. As a result, a handful of overlapping, mutually-incompatible IPC technologies are scattered across various abstraction layers. [33]

- Mach Ports
- Distributed Notifications
- Distributed Objects
- AppleEvents & AppleScript
- Pasteboard
- XPC

Ranging from low-level kernel abstractions to high-level, object-oriented APIs, they each have particular performance and

33 Whereas all of these are available on OS X, only Grand Central Dispatch and Pasteboard (albeit to a lesser extent) can be used on iOS.

security characteristics. But fundamentally, they're all mechanisms for transmitting and receiving data from beyond a context boundary.

Mach Ports

All inter-process communication ultimately relies on functionality provided by Mach kernel APIs.

Mach ports are light-weight and powerful, but poorly documented [34] and inconvenient to use directly. [35]

Sending a message over a given Mach port comes down to a single `mach_msg_send` call, but it takes a bit of configuration in order to build the message to be sent:

```
natural_t data;
mach_port_t port;

struct {
    mach_msg_header_t header;
    mach_msg_body_t body;
    mach_msg_type_descriptor_t type;
} message;
```

34 How poorly documented? The most up-to-date authoritative resource was a Mach 3.0 PostScript file circa 1990 tucked away on a Carnegie Mellon University FTP server.

35 How inconvenient? Well, just look at the code samples.

```
message.header = (mach_msg_header_t) {
    .msgh_remote_port = port,
    .msgh_local_port = MACH_PORT_NULL,
    .msgh_bits = MACH_MSGH_BITS(MACH_MSG_TYPE_COPY_SEND, 0),
    .msgh_size = sizeof(message)
};

message.body = (mach_msg_body_t) {
    .msgh_descriptor_count = 1
};

message.type = (mach_msg_type_descriptor_t) {
    .pad1 = data,
    .pad2 = sizeof(data)
};

mach_msg_return_t error = mach_msg_send(&message.header);

if (error == MACH_MSG_SUCCESS) {
    // ...
}
```

Things are a little easier on the receiving end, since the message only needs to be declared, not initialized:

```
mach_port_t port;

struct {
    mach_msg_header_t header;
    mach_msg_body_t body;
    mach_msg_type_descriptor_t type;
    mach_msg_trailer_t trailer;
} message;

mach_msg_return_t error = mach_msg_receive(&message.header);

if (error == MACH_MSG_SUCCESS) {
    natural_t data = message.type.pad1;
    // ...
}
```

Fortunately, higher-level APIs for Mach ports are provided by Core Foundation and Foundation. `CFMachPort` / `NSMachPort` are wrappers on top of the kernel APIs that can be used as a runloop source, while `CFMessagePort` / `NSMessagePort` facilitate synchronous communication between two ports.

`CFMessagePort` is actually quite nice for simple one-to-one communication. In just a few lines of code, a local named port can be attached as a runloop source to have a callback run each time a message is received:

```
static CFDataRef Callback(CFMessagePortRef port,
                          SInt32 messageID,
                          CFDataRef data,
                          void *info)
{
    // ...
}
```

```
CFMessagePortRef localPort =
    CFMessagePortCreateLocal(nil,
                             CFSTR("com.example.app.port.server"),
                             Callback,
                             nil,
                             nil);

CFRunLoopSourceRef runLoopSource =
    CFMessagePortCreateRunLoopSource(nil, localPort, 0);

CFRunLoopAddSource(CFRunLoopGetCurrent(),
                   runLoopSource,
                   kCFRunLoopCommonModes);
```

Sending data is straightforward as well. Just specify the remote port, the message payload, and timeouts for sending and receiving. `CFMessagePortSendRequest` takes care of the rest:

```
CFDataRef data;
SInt32 messageID = 0x1111; // Arbitrary
CFTimeInterval timeout = 10.0;

CFMessagePortRef remotePort =
    CFMessagePortCreateRemote(nil,
                              CFSTR("com.example.app.port.client"));

SInt32 status =
    CFMessagePortSendRequest(remotePort,
                             messageID,
                             data,
                             timeout,
                             timeout,
                              NULL,
                              NULL);
if (status == kCFMessagePortSuccess) {
    // ...
}
```

Distributed Notifications

There are many ways for objects to communicate with one another in Cocoa:

There is, of course, sending a message directly. There are also the target-action, delegate, and callbacks, which are all loosely-coupled, one-to-one design patterns. KVO allows for multiple objects to subscribe to events, but it strongly couples those objects together. Notifications, on the other hand, allow messages to be broadcast globally, and intercepted by any object that knows what to listen for. [36]

Each application manages its own `NSNotificationCenter` instance for infra-application pub-sub. But there is also a lesser-known Core Foundation API, `CFNotificationCenterGetDistributedCenter` that allows notifications to be communicated system-wide as well.

36 It's pretty astonishing just how many notifications are fired off during the lifecycle of an application. Try adding `NSNotificationCenter -addObserverForName:object:queue:usingBlock` with `nil` values for `name` and `object` just as an application launches, and see just how many times that block fires.

To listen for notifications, add an observer to the distributed notification center by specifying the notification name to listen for, and a function pointer to execute each time a notification is received:

```
static void Callback(CFNotificationCenterRef center,
                     void *observer,
                     CFStringRef name,
                     const void *object,
                     CFDictionaryRef userInfo)
{
    // ...
}

CFNotificationCenterRef distributedCenter =
    CFNotificationCenterGetDistributedCenter();

CFNotificationSuspensionBehavior behavior =
        CFNotificationSuspensionBehaviorDeliverImmediately;

CFNotificationCenterAddObserver(distributedCenter,
                                NULL,
                                Callback,
                                CFSTR("notification.identifier"),
                                NULL,
                                behavior);
```

Sending a distributed notification is even simpler; just post the identifier, object, and user info:

```
void *object;
```

```
CFDictionaryRef userInfo;

CFNotificationCenterRef distributedCenter =
    CFNotificationCenterGetDistributedCenter();

CFNotificationCenterPostNotification(distributedCenter,
                                     CFSTR("notification.identifier"),
                                     object,
                                     userInfo,
                                     true);
```

Of all of the ways to link up two applications, distributed notifications are by far the easiest. It wouldn't be a great idea to use them to send large payloads, but for simple tasks like synchronizing preferences or triggering a data fetch, distributed notifications are perfect.

Distributed Objects

Distributed Objects (DO) is a remote messaging feature of Cocoa that had its heyday back in the mid-90's with NeXT. And though its not widely used any more, the dream of totally frictionless IPC is still unrealized in our modern technology stack.

Vending an object with DO is just a matter of setting up an `NSConnection` and registering it with a particular name:

```
@protocol Protocol;

id <Protocol> vendedObject;

NSConnection *connection = [[NSConnection alloc] init];
[connection setRootObject:vendedObject];
[connection registerName:@"server"];
```

Another application would then create a connection registered for that same registered name, and immediately get an atomic proxy that functioned as if it were that original object:

```
id proxy = [NSConnection rootProxyForConnectionWithRegisteredName:@"server"  ↩
     host:nil];
[proxy setProtocolForProxy:@protocol(Protocol)];
```

Any time a distributed object proxy is messaged, a Remote Procedure Call (RPC) would be made over the `NSConnection` to evaluate the message against the vended object and return the result back to the proxy. [37]

Distributed Objects are simple, transparent, and robust. And they would have been a flagpole feature of Cocoa had any of it worked as advertised.

37 Behind the scenes, a shared `NSPortNameServer` instance managed by the operating system was responsible for hooking up named connections.

In reality, Distributed Objects can't be used like local objects, if only because any message sent to a proxy could result in an exception being thrown. Unlike other languages, Objective-C doesn't use exceptions for control flow. As a result, wrapping everything in a `@try/@catch` is a poor fit to the conventions of Cocoa.

DO is awkward for other reasons, too. The divide between objects and primitives is especially pronounced when attempting to marshal values across a connection. Also, connections are totally unencrypted, and the lack of extensibility for the underlying communication channels makes it a deal-breaker for most serious usage.

All that's really left are traces of the annotations used by Distributed Objects to specify the proxying behavior of properties and method parameters:

- `in`: Argument is used as input, but not referenced later
- `out`: Argument is used to return a value by reference
- `inout`: Argument is used as input and returned by reference
- `const`: Argument is constant
- `oneway`: Return without blocking for result
- `bycopy`: Return a copy of the object
- `byref`: Return a proxy of the object

AppleEvents & AppleScript

AppleEvents are the most enduring legacies of the classic Macintosh operating system. Introduced in System 7, AppleEvents allowed apps to be controlled locally using AppleScript, or remotely using a feature called Program Linking. To this day, AppleScript, using the Cocoa Scripting Bridge, remains the most direct way to programmatically interact with OS X applications. [38]

That said, it's easily one of the weirdest technologies to work with.

AppleScript uses a natural language syntax, intended to be more accessible to non-programmers. And while it does succeed in communicating intent in a human-understandable way, it's a nightmare to write.

To get a better sense of the nature of the beast, here's how to tell Safari to open a URL in the active tab in the frontmost window:

38 Mac OS's Apple Event Manager provided the initial low-level transport mechanism for AppleEvents, but was reimplemented on top of Mach ports for OS X.

```
tell application "Safari"
  set the URL of the front document to "http://nshipster.com"
end tell
```

In many ways, AppleScript's natural language syntax is more of a liability than an asset. English, much like any other spoken language, has a great deal of redundancy and ambiguity built into normal constructions. While this is perfectly acceptable for humans, computers have a tough time resolving all of this.

Even for a seasoned Objective-C developer, it's nearly impossible to write AppleScript without constantly referencing documentation or sample code.

Fortunately, the Scripting Bridge provides a proper programming interface for Cocoa applications.

Cocoa Scripting Bridge

In order to interact with an application through the Scripting Bridge, a programming interface must first be generated:

```
$ sdef /Applications/Safari.app | sdp -fh --basename Safari
```

`sdef` generates scripting definition files for an application. These files can then be piped into `sdp` to be converted into another format—in this case, a C header. The resulting `.h` file can then be added and `#import`-ed into a project to get a first-class object interface to that application.

Here's the same example as before, expressed using the Cocoa Scripting Bridge:

```
#import "Safari.h"

SafariApplication *safari = [SBApplication applicationWithBundleIdentifier:@" ↩
     com.apple.Safari"];

for (SafariWindow *window in safari.windows) {
    if (window.visible) {
        window.currentTab.URL = [NSURL URLWithString:@"http://nshipster.com"];
        break;
    }
}
```

It's a bit more verbose than AppleScript, but this is much easier to integrate into an existing codebase. It's also a lot clearer to understand how this same code could be adapted to slightly different behavior (though that could just be the effect of being more familiar with Objective-C).

Alas, AppleScript's star appears to be falling, as recent releases of OS X and iWork applications have greatly curtailed their scriptability. At this point, it's unlikely that adding support in your own applications will be worth it.

Pasteboard

Pasteboard is the most visible inter-process communication mechanism on OS X and iOS. Whenever a user copies or pastes a piece of text, an image, or a document between applications, an exchange of data from one process to another over mach ports is being mediated by the `com.apple.pboard` service.

On OS X there's `NSPasteboard`, and on iOS there's `UIPasteboard`. They're pretty much the same, although like most counterparts, iOS provides a cleaner, more modern set of APIs that are slightly less capable than what's found on OS X.

Programmatically writing to the Pasteboard is nearly as simple as invoking `Edit > Copy` in a GUI application:

```
NSImage *image;

NSPasteboard *pasteboard = [NSPasteboard generalPasteboard];
[pasteboard clearContents];
[pasteboard writeObjects:@[image]];
```

The reciprocal paste action is a bit more involved, requiring an iteration over the Pasteboard contents:

```
NSPasteboard *pasteboard = [NSPasteboard generalPasteboard];

if ([pasteboard canReadObjectForClasses:@[[NSImage class]] options:nil]) {
    NSArray *contents = [pasteboard readObjectsForClasses:@[[NSImage class]]  ↩
      options: nil];
    NSImage *image = [contents firstObject];
}
```

What makes Pasteboard especially compelling as a mechanism for transferring data is the notion of simultaneously providing multiple representations of content copied onto a pasteboard. For example, a selection of text may be copied as both rich text (RTF) and plain text (TXT), allowing, for example, a WYSIWYG editor to preserve style information by grabbing the rich text representation, and a code editor to use just the plain text representation.

These representations can even be provided on an on-demand basis by conforming to the `NSPasteboardItemDataProvider` protocol. This allows derivative representations, such as plain text from rich text, to be generated only as necessary.

Each representation is identified by a Unique Type Identifier (UTI), a concept discussed in greater detail in the next chapter.

XPC

XPC is the state-of-the-art for inter-process communication in the SDKs. Its architectural goals are to avoid long-running process, to adapt to the available resources, and to lazily initialize wherever possible. The motivation to incorporate XPC into an application is not to do things that are otherwise impossible, but to provide better privilege separation and fault isolation for inter-process communication.

It's a replacement for `NSTask`, and a whole lot more.

Introduced in 2011, XPC has provided the infrastructure for the App Sandbox on OS X, Remote View Controllers on iOS, and App Extensions on both. It is also widely used by system frameworks and first-party applications:

```
$ find /Applications -name \*.xpc
```

By surveying the inventory of XPC services in the wild, one can get a much better understanding of opportunities to use XPC in their own application. Common themes in applications emerge, like services for image and video conversion, system calls, webservice integration, and 3rd party authentication.

XPC takes responsibility for both inter-process communication and service lifecycle management. Everything from registering a service, getting it running, and communicating with other services is handled by `launchd`. An XPC service can be launched on demand, or restarted if they crash, or terminated if they idle. As such, services should be designed to be completely stateless, so as to allow for sudden termination at any point of execution.

As part of the new security model adopted by iOS and backported in OS X, XPC services are run with the most restricted environment possible by default: no file system access, no network access, and no root privilege escalation. Any capabilities must be whitelisted by a set of entitlements.

XPC can be accessed through either the `libxpc` C API, or the `NSXPCConnection` Objective-C API. [39]

XPC services either reside within an application bundle or are advertised to run in the background using launchd.

39 Though one should always try to use the highest-level API available to accomplish a particular task, this book *does* have the words "Low-Level" in the title, so the examples in this section will use `libxpc`.

Services call `xpc_main` with an event handler to receive new XPC connections:

```
static void connection_handler(xpc_connection_t peer) {
    xpc_connection_set_event_handler(peer, ^(xpc_object_t event) {
        peer_event_handler(peer, event);
    });

    xpc_connection_resume(peer);
}

int main(int argc, const char *argv[]) {
   xpc_main(connection_handler);
   exit(EXIT_FAILURE);
}
```

Each XPC connection is one-to-one, meaning that the service operates on distinct connections, with each call to `xpc_connection_create` creating a new peer. [40] :

```
xpc_connection_t c = xpc_connection_create("com.example.service", NULL);
xpc_connection_set_event_handler(c, ^(xpc_object_t event) {
    // ...
});
xpc_connection_resume(c);
```

When a message is sent over an XPC connection, it is automatically

40 This is similar to `accept` in the BSD sockets API, in that the server listens on a single file descriptor that creates additional descriptors for each inbound connection.

dispatched onto a queue managed by the runtime. As soon as the connection is opened on the remote end, messages are dequeued and sent.

Each message is a dictionary, with string keys and strongly-typed values:

```
xpc_dictionary_t message = xpc_dictionary_create(NULL, NULL, 0);
xpc_dictionary_set_uint64(message, "foo", 1);
xpc_connection_send_message(c, message);
xpc_release(message)
```

XPC objects operate on the following primitive types:

- Data
- Boolean
- Double
- String
- Signed Integer
- Unsigned Integer
- Date
- UUID
- Array
- Dictionary
- Null

XPC offers a convenient way to convert from the `dispatch_data_t` data type, which simplifies the workflow from GCD to XPC:

```
void *buffer;
size_t length;
dispatch_data_t ddata =
    dispatch_data_create(buffer,
                         length,
                         DISPATCH_TARGET_QUEUE_DEFAULT,
                         DISPATCH_DATA_DESTRUCTOR_MUNMAP);

xpc_object_t xdata = xpc_data_create_with_dispatch_data(ddata);
```

```
dispatch_queue_t queue;
xpc_connection_send_message_with_reply(c, message, queue,
    ^(xpc_object_t reply)
{
      if (xpc_get_type(event) == XPC_TYPE_DICTIONARY) {
         // ...
      }
});
```

Registering Services

XPC can also be registered as launchd jobs, configured to automatically start on matching IOKit events, BSD notifications or

CFDistributedNotifications. These criteria are specified in a service's `launchd.plist` file:

launchd.plist

```
<key>LaunchEvents</key>
<dict>
  <key>com.apple.iokit.matching</key>
  <dict>
      <key>com.example.device-attach</key>
      <dict>
          <key>idProduct</key>
          <integer>2794</integer>
          <key>idVendor</key>
          <integer>725</integer>
          <key>IOProviderClass</key>
          <string>IOUSBDevice</string>
          <key>IOMatchLaunchStream</key>
          <true/>
          <key>ProcessType</key>
          <string>Adaptive</string>
      </dict>
  </dict>
</dict>
```

A recent addition to `launchd` property lists is the `ProcessType` key, which describe at a high level the intended purpose of the launch agent. Based on the prescribed contention behavior, the operating system will automatically throttle CPU and I/O bandwidth accordingly.

Process Types and Contention Behavior

Standard	Default value
Adaptive	Contend with apps when doing work on their behalf
Background	Never contend with apps
Interactive	Always contend with apps

To register a service to run approximately every 5 minutes (allowing a grace period for system resources to become more available before scheduling at a more aggressive priority), a set of criteria is passed into `xpc_activity_register`:

```
xpc_object_t criteria = xpc_dictionary_create(NULL, NULL, 0);
xpc_dictionary_set_int64(criteria, XPC_ACTIVITY_INTERVAL, 5 * 60);
xpc_dictionary_set_int64(criteria, XPC_ACTIVITY_GRACE_PERIOD, 10 * 60);

xpc_activity_register("com.example.app.activity",
                      criteria,
                      ^(xpc_activity_t activity)
{
    // Process Data

    xpc_activity_set_state(activity, XPC_ACTIVITY_STATE_CONTINUE);

    dispatch_async(dispatch_get_main_queue(), ^{
        // Update UI
        xpc_activity_set_state(activity, XPC_ACTIVITY_STATE_DONE);
    });
});
```

Chapter 7

Core Services

Let's take a moment to really think about what a file is.

Fundamentally, a file is a resource with information. It's persisted in such a way that it remains durable—available beyond the scope of the program that originally created it. A file itself is usually encoded as a one-dimensional byte array, ideally in sequential portions of the storage medium.

By themselves, files are meaningless blobs of 0's and 1's. It is only through the programs that read and write them that any sense can be made of it all.

A common practice for interchange formats—especially image encodings like PNG, GIF, and JPEG—is to use a unique value at the start of the file, also known as a file signature or "magic number". Information about a file, such as its name, size, and other attributes are stored as metadata by the filesystem. Programs reading from a file use this to determine how to parse and interpret the file's data.

How this information is structured and encoded is one of the main differentiating factors of a filesystem.

Data & Resource Forks

The old school Apple Macintosh Filesystem (MFS) associated a data fork, a resource fork, and multiple named forks for each file entry. For applications, the data fork would hold the binary executable, while the resource fork would contain things like icon bitmaps and localized strings. Not all files had resource forks, but they were useful for separating content from presentation, like in the case of a word processing documentation.

On Mac OS, file types and creator codes were represented by an `OSType`, a four-byte identifier, most often encoded as four ASCII / Macintosh Roman characters. Common examples of file types are listed in the figure below:

OSType of Common File Types

`CODE`	Executable Binary
`Text`	Plain Text
`PICT`	QuickDraw Image
`SND`	Sound

`OSType` was also used to denote the type of pasteboard contents, error codes, and AppleEvents, the inter-process communication mechanism introduced in System 7.

Actually, that last point about `OSType` being repurposed for tasks like handling copy-paste between applications raises an interesting point: files are but one of many ways that data gets passed around.

One such example is information downloaded over the internet.

The vast majority of networking done in applications today is over HTTP. Although the type of resource may sometime be gleaned from the extension of the URI, the canonical identifier is found in the `Content-Type` HTTP header field. Values for this header use MIME types. [41]. MIME types are defined by a central authority, The Internet Assigned Numbers Authority (IANA), which also manages root name servers and IP address blocks.

Therefore, any system that would replace `OSType` would have to be able to accommodate internet media, as well as existing file types.

That system was Universal Type Identifiers (UTI), and it was introduced with OS X.

UTI

UTIs provide an extensible, hierarchical classification system that affords the developer great flexibility in handling even the most exotic

41 MIME types were originally used in the development of mail applications using SMTP

file types. For example, a Ruby source file (`.rb`) is categorized as "Ruby Source Code > Source Code > Text > Content > Data"; a QuickTime Movie file (`.mov`) is categorized as "Video > Movie > Audiovisual Content > Content > Data".

Not limited to files, UTIs can be used to identify a number of different entities:

- Files
- Directories
- Pasteboard Data
- Bundles
- Frameworks
- Internet Media
- Streaming Data
- Aliases and Symbolic Links

Type Identifiers

The public domain is reserved for common or standard types that are of general use to most applications, such as:

- `public.text`

- `public.plain-text`
- `public.jpeg`
- `public.html`

Dynamic Type Identifiers

Sometimes a data type does not have a UTI declared for it. UTIs handle this case transparently by creating a dynamic identifier. Dynamic identifiers have the domain `dyn`, with the rest of the string that follows being opaque. They can be thought of as a UTI-compatible wrapper around an otherwise unknown filename extension, MIME type, OSType, and so on.

Custom Type Identifiers

When creating a custom type identifier, the aim is to have the UTI conform to both a physical and functional hierarchy:

- A physical hierarchy involves the nature of the item, such as whether it's a file or directory. This should inherit from `public.item`.

- A functional hierarchy relates to how the item is used. This should not inherit from `public.item`, but instead something like `public.content` or `public.executable`.

Working with UTIs

The Core Services framework on OS X and Mobile Core Services framework on iOS provide functions that identify and categorize data types by file extension and MIME type, according to Universal Type Identifiers.

Comparing

There are two functions for comparing UTIs. `UTTypeEqual` does an equality check, which is the equivalent of a case insensitive string comparison. `UTTypeConformsTo` is more compelling, as it consults the functional and physical hierarchies to find a match. It's the same difference between `isMemberOfClass:` and `isKindOfClass:`. [42]

42 Like any other function that takes string identifiers, use constants rather than literal value when possible (e.g. `kUTTypeApplication`, instead of `"com.apple.application"`). The examples here use literal strings for clarity.

```
UTTypeConformsTo(CFSTR("public.jpeg"),
                 CFSTR("public.item")); // YES

UTTypeConformsTo(CFSTR("public.jpeg"),
                 CFSTR("public.image")); // YES

UTTypeEqual(CFSTR("public.jpeg"),
            CFSTR("public.image")); // NO

UTTypeConformsTo(CFSTR("public.jpeg"),
                 CFSTR("public.png")); // NO
```

Copying Declarations

In addition to its unique identifier, each UTI is registered with a property list of attributes. Those attributes can be retrieved with `UTTypeCopyDeclaration`:

```
UTTypeCopyDeclaration(CFSTR("public.png"));
```

```
{
    UTTypeConformsTo = "public.image";
    UTTypeDescription = "Portable Network Graphics image";
    UTTypeIdentifier = "public.png";
    UTTypeTagSpecification =     {
        "com.apple.nspboard-type" = "Apple PNG pasteboard type";
        "com.apple.ostype" = PNGf;
        "public.filename-extension" = png;
        "public.mime-type" = "image/png";
    };
}
```

Converting

A common task, when working with UTIs, is to get their equivalent MIME type or filename extension. This can be accomplished using the function `UTTypeCopyPreferredTagWithClass`:

```
NSString *contentType =
    (__bridge_transfer NSString *)
        UTTypeCopyPreferredTagWithClass(CFSTR("public.text"),
                                        kUTTagClassMIMEType);
```

The converse, determining the UTI for a MIME type, can be accomplished with `UTTypeCreatePreferredIdentifierForTag`:

```
NSString *UTI =
    (__bridge_transfer NSString *)
        UTTypeCreatePreferredIdentifierForTag(kUTTagClassFilenameExtension,
                                              CFSTR("jpg"),
                                              NULL);
```

The tag class arguments for `UTTypeCopyPreferredTagWithClass` & `UTTypeCreatePreferredIdentifierForTag` can be any of the following string constants: [43]

```
const CFStringRef kUTTagClassFilenameExtension;
const CFStringRef kUTTagClassMIMEType;
const CFStringRef kUTTagClassNSPboardType; // OS X
const CFStringRef kUTTagClassOSType; // OS X
```

As a means of reconciling the growing and evolving landscape of data and file transfer, UTIs have performed remarkably well. The shift from filesystem-specific mechanisms like "magic number" signatures and resource forks to a formalized type hierarchy has afforded applications great flexibility in how data is handled.

Making the most of UTIs will ensure that applications interact with files responsibly, and play well with others.

43 For OSType values containing only printable 7-bit ASCII characters, you can still use the CFSTR macro with a four-character string literal (for example, `CFSTR("TEXT")` to create a valid OSType tag.

Chapter 8

ImageIO

Image I/O is a powerful, albeit lesser-known framework for working with images. Independent of Core Graphics, it can read and write between between many different formats, access photo metadata, and perform common image processing operations. The framework offers the fastest image encoders and decoders on the platform, with advanced caching mechanisms and even the ability to load images incrementally.

Supported Image Types

According to the official docs, Image I/O supports "most image formats". Rather than take the docs at their word and guess what exactly that entails, this information can be retrieved programmatically.

`CGImageSourceCopyTypeIdentifiers` returns list of UTIs for image types supported:

Image I/O UTIs

UTI	iOS	OS X
`com.adobe.photoshop-image`		x
`com.adobe.raw-image`	x	x
`com.apple.icns`		x

(continued)

UTI	iOS	OS X
com.apple.macpaint-image		x
com.apple.pict		x
com.apple.quicktime-image		x
com.canon.cr2-raw-image	x	x
com.canon.crw-raw-image	x	x
com.canon.tif-raw-image	x	x
com.compuserve.gif	x	x
com.epson.raw-image	x	x
com.fuji.raw-image	x	x
com.hasselblad.3fr-raw-image	x	x
com.hasselblad.fff-raw-image	x	x
com.ilm.openexr-image		x
com.kodak.flashpix-image		x
com.kodak.raw-image	x	x
com.konicaminolta.raw-image	x	x
com.leafamerica.raw-image	x	x
com.leica.raw-image	x	x
com.leica.rwl-raw-image	x	x
com.microsoft.bmp	x	x
com.microsoft.cur	x	x
com.microsoft.ico	x	x
com.nikon.nrw-raw-image	x	x

(continued)

UTI	iOS	OS X
com.nikon.raw-image	x	x
com.olympus.or-raw-image	x	x
com.olympus.raw-image	x	x
com.olympus.sr-raw-image	x	x
com.panasonic.raw-image	x	x
com.panasonic.rw2-raw-image	x	x
com.pentax.raw-image	x	x
com.samsung.raw-image	x	x
com.sgi.sgi-image		x
com.sony.arw-raw-image	x	x
com.sony.raw-image	x	x
com.sony.sr2-raw-image	x	x
com.truevision.tga-image	x	x
public.jpeg	x	x
public.jpeg-2000	x	x
public.mpo-image		x
public.png	x	x
public.radiance		x
public.tiff	x	x
public.xbitmap-image	x	x

As it turns out, that does seem like *most* formats. At least the ones that matter for applications today. There is universal support for common formats: TIFF, JPEG, GIF, PNG, RAW, and Windows Bitmap, Icon, and Cursor. Additionally, several vendor-specific RAW camera formats are supported on iOS, but OS X supports a few more of them.

Writing to a File

Image I/O offers advanced output configuration without much overhead.

Specify the UTI of the desired output format, as well as any options, like compression quality, orientation, or whether to ignore alpha channels. A `CGImageDestinationRef` is created for the destination, has the `CGImage` added to it, and is then finalized:

```
UIImage *image = ...;

NSURL *fileURL = [NSURL fileURLWithPath:@"/path/to/output.jpg"];
NSString *UTI = @"public.jpeg";
NSDictionary *options = @{
                          (__bridge id) ↩
    kCGImageDestinationLossyCompressionQuality: @(0.75),
                          (__bridge id)kCGImagePropertyOrientation: @(4),
                          (__bridge id)kCGImagePropertyHasAlpha: @(NO)
                          };
```

```
CGImageDestinationRef imageDestinationRef =
    CGImageDestinationCreateWithURL((__bridge CFURLRef)fileURL,
                                    (__bridge CFStringRef)UTI,
                                    1,
                                    NULL);

CGImageDestinationAddImage(imageDestinationRef, [image CGImage], (__bridge  ↩
     CFDictionaryRef)options);
CGImageDestinationFinalize(imageDestinationRef);
CFRelease(imageDestinationRef);
```

Reading from a File

Reading an image from a file is a very similar process to writing.

Create a file URL to the desired input file, and set any desired flags for caching or image type hinting. A `CGImageSourceRef` is created with that URL, which then reads the data and creates a `CGimage` with a call to `CGImageSourceCreateImageAtIndex`.

```
NSURL *fileURL = [NSURL fileURLWithPath:@"/path/to/input.jpg"];
NSDictionary *options = @{
                          (__bridge id)kCGImageSourceTypeIdentifierHint: @" ↩
     public.jpeg",
                          (__bridge id)kCGImageSourceShouldCache: @(YES),
                          (__bridge id)kCGImageSourceShouldAllowFloat: @(YES),
                          };
```

```
CGImageSourceRef imageSourceRef =
    CGImageSourceCreateWithURL((__bridge CFURLRef)fileURL, NULL);
CGImageRef imageRef =
    CGImageSourceCreateImageAtIndex(imageSourceRef,
                                    0,
                                    (__bridge CFDictionaryRef)options);

UIImage *image = [UIImage imageWithCGImage:imageRef];

CFRelease(imageRef);
CFRelease(imageSourceRef);
```

Incrementally Reading an Image

The previous example can be extended to load the image incrementally, which may contribute a better user experience for especially large or remote images.

Since many applications load images over HTTP, the session task delegate method `URLSession:dataTask:didReceiveData:` is a great opportunity for performance gains:

```
- (void)URLSession:(NSURLSession *)session
          dataTask:(NSURLSessionDataTask *)dataTask
    didReceiveData:(NSData *)data
{
    [self.mutableResponseData appendData:data];
    CGImageSourceUpdateData(self.imageSourceRef,
                            (__bridge CFDataRef)self.mutableResponseData,
                            [self.mutableResponseData length]
                                [dataTask countOfBytesExpectedToReceive]);

    if (CGSizeEqualToSize(self.imageSize, CGSizeZero)) {
        NSDictionary *properties =
            (__bridge_transfer NSDictionary *)
                CGImageSourceCopyPropertiesAtIndex(self.imageSourceRef,
                                                   0,
                                                   NULL);
        if (properties) {
            NSNumber *width = properties[(__bridge id) ↩
     kCGImagePropertyPixelWidth];
            NSNumber *height = properties[(__bridge id) ↩
     kCGImagePropertyPixelHeight];

            if (width && height) {
                self.imageSize = CGSizeMake([width floatValue],
                                            [height floatValue]);
            }
        }
    }

    CGImageRef imageRef = CGImageSourceCreateImageAtIndex(self.imageSourceRef,
                                                          0,
                                                          NULL);
    UIImage *image = [UIImage imageWithCGImage:imageRef];
    CFRelease(imageRef);

    dispatch_async(dispatch_get_main_queue(), ^{
        // delete or block callback to update with image
    });
}
```

Given a `CGImageSourceRef`, which would have been initialized when the request began loading, this delegate method calls `CGImageSourceUpdateData` to update with the response data buffer.

If enough data has been loaded to determine the final dimensions of the image, `CGImageSourceCopyPropertiesAtIndex` can retrieve that information and cache it. From that point on, a delegate or block callback would be able to send the partially-loaded image to update the UI on the main thread.

Image Metadata

In the incremental image loading example, the image's width and height were retrieved from its metadata so that it could be properly sized—even before all of the data was loaded.

That same approach can be used to retrieve image metadata, such as GPS data (location), camera EXIF (lens, exposure, shutter speed, etc.), or IPTC (information suitable for publication, like creator and copyright).

Metadata is divided into several different dictionaries, which can be specified with any of the following keys:

- `kCGImagePropertyTIFFDictionary`
- `kCGImagePropertyGIFDictionary`
- `kCGImagePropertyJFIFDictionary`
- `kCGImagePropertyExifDictionary`
- `kCGImagePropertyPNGDictionary`
- `kCGImagePropertyIPTCDictionary`
- `kCGImagePropertyGPSDictionary`
- `kCGImagePropertyRawDictionary`
- `kCGImagePropertyCIFFDictionary`
- `kCGImageProperty8BIMDictionary`
- `kCGImagePropertyDNGDictionary`
- `kCGImagePropertyExifAuxDictionary`

Retrieving image metadata properties with Image I/O is pretty self-explanatory: one call to `CGImageSourceCopyPropertiesAtIndex`, and it's all standard `NSDictionary` access on `CGImageProperty` keys from there on out:

```
NSDictionary *properties =
    (__bridge_transfer NSDictionary *)
        CGImageSourceCopyPropertiesAtIndex(self.imageSourceRef,
                                           0,
                                           NULL);

NSDictionary *EXIF = properties[(__bridge id)kCGImagePropertyExifDictionary];
if (EXIF) {
    NSString *Fnumber = EXIF[(__bridge id)kCGImagePropertyExifFNumber];
    NSString *exposure = EXIF[(__bridge id)kCGImagePropertyExifExposureTime];
    NSString *ISO = EXIF[(__bridge id)kCGImagePropertyExifISOSpeedRatings];

    NSLog(@"Shot Information: %@ %@ %@", Fnumber, exposure, ISO);
}

NSDictionary *GPS = properties[(__bridge id)kCGImagePropertyGPSDictionary];
if (GPS) {
    NSString *latitude = GPS[(__bridge id)kCGImagePropertyGPSLatitude];
    NSString *latitudeRef = GPS[(__bridge id)kCGImagePropertyGPSLatitudeRef];

    NSString *longitude = GPS[(__bridge id)kCGImagePropertyGPSLongitude];
    NSString *longitudeRef = GPS[(__bridge id)kCGImagePropertyGPSLongitudeRef];

    NSLog(@"GPS: %@ %@ / %@ %@", latitude, latitudeRef, longitude, longitudeRef ↩
     );
}
```

Chapter 9

Accelerate

Over the last decade, there has been a focus on doing more with less, when it comes to hardware.

As each generation of microprocessor pushes the physical limits of silicon, a greater emphasis has been placed on doing more with less. Moore's Law, the observation that the number of transistors that can be fit onto a chip doubles every 18 months or so, has to end sometime, and that time is fast approaching.

At the same time, the rise of mobile computing has turned the performance paradigm on its head, emphasizing battery life over power.

In response to these two emerging realities, hardware manufacturers have built muti-core CPUs, which software developers have learned to exploit with high degrees of parallelism and concurrency.

On iOS and OS X, the best way to harness these advanced capabilities is the Accelerate Framework.

With over 2,000 APIs, Accelerate is easily the single largest framework in the iOS and OS X SDKs. But far from being monolithic, it's really more of an umbrella framework, with several inter-related component parts.

At the top level, Accelerate can be split up between vecLib & vImage. vecLib contains data types and C functions for digital signal processing (vDSP) as well as vector and matrix math, including those covered by the Linear Algebra Package (LAPACK) and the Basic Linear Algebra Subprograms (BLAS) standard. vImage contains a wide range of image manipulation functionality, including alpha compositing, conversion, convolution, morphology, transformation, and histogram generation.

Learning to use Accelerate can be overwhelming just in terms of the volume of APIs. And for anyone not coming from a math or high performance computing background, the conceptual overhead alone is enough to send most people running.

SIMD

If there is a single, unifying concept for Accelerate, it's SIMD, or "single instruction, multiple data". SIMD means that a computer can perform the same operation on several data points simultaneously using a single command.

For example, given an array of unsigned integers, the maximum value could be found using a single, optimized hardware instruction (e.g., `PMAXUB` for SSE).

The hardware found in iPhones, iPads, and Macs boast impressive capabilities. On x86 architectures (Mac), the key technologies are SSE, AVX, and AVX2; for AMD (iPhone & iPad), it's NEON.

Accelerate provides a single, unified set of APIs that adapt to provide the same behavior across all of these different architectures and hardware environments to ensure maximum performance and stability without any compilation flags or platform hacks.

Benchmarking Performance

How much of a difference do these advanced routines make in practice? Consider the following benchmarks for common arithmetic operations:

Populating an Array

```
NSUInteger count = 10000000;
float *array = malloc(count * sizeof(float));
```

Baseline

```
for (NSUInteger i = 0; i < count; i++) {
    array[i] = i;
}
```

Accelerate

```
float initial = 0;
float increment = 1;
vDSP_vramp(&initial, &increment, array, 1, count);
```

Baseline	Accelerate	Δ
20.664600 msec	2.495000 msec	10x

Multiplying an Array

Baseline

```
for (NSUInteger i = 0; i < count; i++) {
    array[i] *= 2.0f;
}
```

Accelerate

```
cblas_sscal(count, 2.0f, array, 1);
```

Baseline	Accelerate	Δ
19.969440 msec	2.541220 msec	10x

Summing an Array

Baseline

```
float sum = 0;
for (NSUInteger i = 0; i < count; i++) {
  sum += array[i];
}
```

Accelerate

```
float sum = cblas_sasum(count, array, 1);
```

Baseline	Accelerate	Δ
41.704483 msec	2.165160 msec	20x

Searching

Create random array

```
for (NSUInteger i = 0; i < count; i++) {
    array[i] = (float)arc4random();
}
```

Baseline

```
NSUInteger maxLocation = 0;
for (NSUInteger i = 0; i < count; i++) {
  if (array[i] > array[maxLocation]) {
    maxLocation = i;
  }
}
```

Accelerate

```
NSUInteger maxLocation = cblas_isamax(count, array, 1);
```

Baseline	Accelerate	Δ
22.339838 msec	5.110880 msec	4x

From these benchmarks, it's clear that for Accelerate can have huge performance benefits for operations on large data sets. Of course, like any optimization, not all situations will benefit equally. The best approach is always to use Instruments to find bottlenecks in your code, and measure alternative implementations.

In order to understand and identify situations that might benefit from Accelerate, though, we need to get a sense of everything it can do.

vecLib

vecLib is comprised of the following 9 headers:

`cblas.h` / `vBLAS.h`	Interface for BLAS functions
`clapack.h`	Interface for LAPACK functions
`vectorOps.h`	Vector implementations of the BLAS routines.
`vBasicOps.h`	Basic algebraic operations. 8-, 16-, 32-, 64-, 128-bit division, saturated addition / subtraction, shift / rotate, etc.
`vfp.h`	Transcendental operations (sin, cos, log, etc.) on single vector floating point quantities.
`vForce.h`	Transcendental operations on arrays of floating point quantities.
`vBigNum.h`	Operations on large numbers (128-, 256-, 512-, 1024-bit quantities)
`vDSP.h`	Digital signal processing algorithms including FFTs, signal clipping, filters, and type conversions.

vDSP.h

Fast-Fourier Transform (FFT) is the fundamental digital signal processing algorithm. It decomposes a sequence of values into components with different frequencies.

Although they have wide-ranging applications across mathematics and engineering, most application developers encounter FFTs for audio or video processing, as a way of determining the critical values in a noisy signal.

Fast-Fourier Transform

```
int x = 8;
int y = 8;

int dimensions = x * y;

int log2_x = (int)log2((double)x);
int log2_y = (int)log2((double)y);

DSPComplex *data = (DSPComplex *)malloc(sizeof(DSPComplex) * dimensions);
for (NSUInteger i = 0; i < dimensions; i++) {
    data[i].real = (float)i;
    data[i].imag = (float)(dimensions - i) - 1.0f;
}

DSPSplitComplex input = {
    .realp = (float *)malloc(sizeof(float) * dimensions),
    .imagp = (float *)malloc(sizeof(float) * dimensions),
};

vDSP_ctoz(data, 2, &input, 1, dimensions);

FFTSetup weights = vDSP_create_fftsetup(fmax(log2_x, log2_y), kFFTRadix2);
vDSP_fft2d_zip(weights, &input, 1, 0, log2_x, log2_y, FFT_FORWARD);
vDSP_destroy_fftsetup(fft_weights);

vDSP_ztoc(&input, 1, data, 2, dimensions);

for (NSUInteger i = 0; i < dimensions; i++) {
    NSLog(@"%g %g", data[i].real, data[i].imag);
}

free(input.realp);
free(input.imagp);
free(data);
```

vImage

vImage is comprised of 6 headers:

`Alpha.h`	Alpha compositing functions.
`Conversion.h`	Converting between image format (e.g. `Planar8` to `PlanarF`, `ARGB8888` to `Planar8`).
`Convoluton.h`	Image convolution routines (e.g. blurring and edge detection).

`Geometry.h`	Geometric transformations (e.g. rotate, scale, shear, affine warp).
`Histogram.h`	Functions for calculating image histograms and image normalization.
`Morphology.h`	Image morphology procedures (e.g. feature detection, dilation, erosion).
`Tranform.h`	Image transformation operations (e.g. gamma correction, colorspace conversion).

`Planar8`	The image is a single channel (one color or alpha value). Each pixel is an 8-bit unsigned integer value. The data type for this image format is Pixel_8.
`PlanarF`	The image is a single channel (one color). Each pixel is a 32-bit floating-point value. The data type for this image format is Pixel_F.
`ARGB8888`	The image has four interleaved channels, for alpha, red, green, and blue, in that order. Each pixel is 32 bits, an array of four 8-bit unsigned integers. The data type for this image format is Pixel_8888.
`ARGBFFFF`	The image has four interleaved channels, for alpha, red, green, and blue, in that order. Each pixel is an array of four floating-point numbers. The data type for this image format is Pixel_FFFF.

`RGBA8888`	The image has four interleaved channels, for red, green, blue, and alpha, in that order. Each pixel is 32 bits, an array of four 8-bit unsigned integers. The data type for this image format is Pixel_8888.
`RGBAFFFF`	The image has four interleaved channels, for red, green, blue, and alpha, in that order. Each pixel is an array of four floating-point numbers. The pixel data type for this image format is Pixel_FFFF.

`Alpha.h`

Alpha compositing is a process of combining multiple images according to their alpha components. For each pixel in an image, the alpha, or transparency, value is used to determine how much of the image underneath it will be shown.

vImage functions are available for blending or clipping. The most common operation is compositing a top image onto a bottom image:

Compositing Two Images

```
UIImage *topImage, *bottomImage = ...;
CGImageRef topImageRef = [topImage CGImage];
CGImageRef bottomImageRef = [bottomImage CGImage];
```

```
CGDataProviderRef topProvider = CGImageGetDataProvider(topImageRef);
CFDataRef topBitmapData = CGDataProviderCopyData(topProvider);

size_t width = CGImageGetWidth(topImageRef);
size_t height = CGImageGetHeight(topImageRef);
size_t bytesPerRow = CGImageGetBytesPerRow(topImageRef);

vImage_Buffer topBuffer = {
    .data = (void *)CFDataGetBytePtr(topBitmapData),
    .width = width,
    .height = height,
    .rowBytes = bytesPerRow,
};

CGDataProviderRef bottomProvider = CGImageGetDataProvider(bottomImageRef);
CFDataRef bottomBitmapData = CGDataProviderCopyData(bottomProvider);

vImage_Buffer bottomBuffer = {
    .data = (void *)CFDataGetBytePtr(bottomBitmapData),
    .width = width,
    .height = height,
    .rowBytes = bytesPerRow,
};

void *outBytes = malloc(height * bytesPerRow);
vImage_Buffer outBuffer = {
    .data = outBytes,
    .width = width,
    .height = height,
    .rowBytes = bytesPerRow,
};

vImage_Error error = vImagePremultipliedAlphaBlend_ARGB8888(&topBuffer, & ↩
     bottomBuffer, &outBuffer, kvImageDoNotTile);
```

```
if (error) {
    NSLog(@"Error: %ld", error);
}
```

Conversion.h

Images are comprised of pixels, each of which have a color represented by a combination of discrete values for red, green, and blue intensities. Altogether, these intensity values comprise a channel for each color, as well an alpha channel, representing transparency.

There are two ways that images encode this information: *interleaved*, such that each pixel has its red, green, blue, and alpha values represented together, or *planar*, where all of the values in a channel are set, followed by the values in the next channel, and so on.

vImage provides functions for exchanging the intensities from one channel to another, for a given image format:

Permuting Color Channels

```
UIImage *image = ...;
CGImageRef imageRef = [image CGImage];
```

```
size_t width = CGImageGetWidth(imageRef);
size_t height = CGImageGetHeight(imageRef);
size_t bitsPerComponent = CGImageGetBitsPerComponent(imageRef);
size_t bytesPerRow = CGImageGetBytesPerRow(imageRef);

CGDataProviderRef sourceImageDataProvider = CGImageGetDataProvider(imageRef);
CFDataRef sourceImageData = CGDataProviderCopyData(sourceImageDataProvider);
vImage_Buffer sourceImageBuffer = {
    .data = (void *)CFDataGetBytePtr(sourceImageData),
    .width = width,
    .height = height,
    .rowBytes = bytesPerRow,
};

uint8_t *destinationBuffer = malloc(CFDataGetLength(sourceImageData));
vImage_Buffer destinationImageBuffer = {
    .data = destinationBuffer,
    .width = width,
    .height = height,
    .rowBytes = bytesPerRow,
};



const uint8_t channels[4] = {0, 3, 2, 1}; // ARGB -> ABGR
vImagePermuteChannels_ARGB8888(&sourceImageBuffer, &destinationImageBuffer,  ↩
     channels, kvImageNoFlags);

CGColorSpaceRef colorSpaceRef = CGColorSpaceCreateDeviceRGB();
CGContextRef destinationContext =
    CGBitmapContextCreateWithData(destinationBuffer,
                                  width,
                                  height,
                                  bitsPerComponent,
                                  bytesPerRow,
                                  colorSpaceRef,
                                                kCGBitmapByteOrderDefault |
                                                kCGImageAlphaPremultipliedFirst,
                                                NULL,
                                                NULL);

CGImageRef permutedImageRef = CGBitmapContextCreateImage(destinationContext);
UIImage *permutedImage = [UIImage imageWithCGImage:permutedImageRef];

CGImageRelease(permutedImageRef);
CGContextRelease(destinationContext);
CGColorSpaceRelease(colorSpaceRef);
```

`Convolution.h`

Image convolution is the process of multiplying each pixel and its adjacent pixels by a *kernel*, or square matrix with a sum of 1. Depending on the kernel, a convolution operation can either blur, sharpen, emboss, or detect edges.

Except for specific situations where a custom kernel is required, convolution operations would be better-served by the Core Image framework, which utilizes the GPU. However, for a straightforward CPU-based solution, vImage delivers:

Blurring an Image

```
UIImage *inImage = ...;
CGImageRef inImageRef = [inImage CGImage];

CGDataProviderRef inProvider = CGImageGetDataProvider(inImageRef);
CFDataRef inBitmapData = CGDataProviderCopyData(inProvider);

vImage_Buffer inBuffer = {
    .data = (void *)CFDataGetBytePtr(inBitmapData),
    .width = CGImageGetWidth(inImageRef),
    .height = CGImageGetHeight(inImageRef),
    .rowBytes = CGImageGetBytesPerRow(inImageRef),
};

void *outBytes = malloc(CGImageGetBytesPerRow(inImageRef) * CGImageGetHeight( ↩
     inImageRef));
vImage_Buffer outBuffer = {
    .data = outBytes,
    .width = inBuffer.width,
    .height = inBuffer.height,
    .rowBytes = inBuffer.rowBytes,
};

uint32_t length = 5; // Size of convolution
vImage_Error error =
  vImageBoxConvolve_ARGB8888(&inBuffer,
                             &outBuffer,
                             NULL,
                             0,
                             0,
                             length,
                             length,
                             NULL,
                             kvImageEdgeExtend);
if (error) {
    NSLog(@"Error: %ld", error);
}

CGColorSpaceRef colorSpaceRef = CGColorSpaceCreateDeviceRGB();
CGContextRef c =
  CGBitmapContextCreate(outBuffer.data,
                        outBuffer.width,
                        outBuffer.height,
                        8,
                        outBuffer.rowBytes,
                        colorSpaceRef,
                        kCGImageAlphaNoneSkipLast);
CGImageRef outImageRef = CGBitmapContextCreateImage(c);
UIImage *outImage = [UIImage imageWithCGImage:outImageRef];

CGImageRelease(outImageRef);
CGContextRelease(c);
CGColorSpaceRelease(colorSpaceRef);
CFRelease(inBitmapData);
```

`Geometry.h`

Resizing an image is another operation that is perhaps more suited for another, GPU-based framework, like Image I/O. For a given vImage buffer, it might be more performant to scale with Accelerate using `vImageScale_*` rather than convert back and forth between a `CGImageRef`:

Resizing an Image

```
double scaleFactor = 1.0 / 5.0;
void *outBytes = malloc(trunc(inBuffer.height * scaleFactor) * inBuffer. ↩
     rowBytes);
vImage_Buffer outBuffer = {
    .data = outBytes,
    .width = trunc(inBuffer.width * scaleFactor),
    .height = trunc(inBuffer.height * scaleFactor),
    .rowBytes = inBuffer.rowBytes,
};

vImage_Error error =
  vImageScale_ARGB8888(&inBuffer,
                       &outBuffer,
                       NULL,
                       kvImageHighQualityResampling);
if (error) {
    NSLog(@"Error: %ld", error);
}
```

Histogram.h

Detecting if an Image Has Transparency

```
UIImage *image;
CGImageRef imageRef = [image CGImage];
CGDataProviderRef dataProvider = CGImageGetDataProvider(imageRef);
CFDataRef bitmapData = CGDataProviderCopyData(dataProvider);

vImagePixelCount a[256], r[256], g[256], b[256];
vImagePixelCount *histogram[4] = {a, r, g, b};
vImage_Buffer buffer = {
    .data = (void *)CFDataGetBytePtr(bitmapData),
    .width = CGImageGetWidth(imageRef),
    .height = CGImageGetHeight(imageRef),
    .rowBytes = CGImageGetBytesPerRow(imageRef),
};

vImage_Error error =
  vImageHistogramCalculation_ARGB8888(&buffer,
                                      histogram,
                                      kvImageNoFlags);
if (error) {
    NSLog(@"Error: %ld", error);
}

BOOL hasTransparency = NO;
for (NSUInteger i = 0; !hasTransparency && i < 255; i++) {
    hasTransparency = histogram[3][i] == 0;
}

CGDataProviderRelease(dataProvider);
CFRelease(bitmapData);
```

Morphology.h

Dilating an Image

```
size_t width = image.size.width;
size_t height = image.size.height;
size_t bitsPerComponent = 8;
size_t bytesPerRow = CGImageGetBytesPerRow([image CGImage]);

CGColorSpaceRef colorSpaceRef = CGColorSpaceCreateDeviceRGB();
CGContextRef sourceContext = CGBitmapContextCreate(NULL,
                                             width,
                                             height,
                                             bitsPerComponent,
                                             bytesPerRow,
                                             colorSpaceRef,
                                             kCGBitmapByteOrderDefault |
                                             kCGImageAlphaPremultipliedFirst);


CGContextDrawImage(sourceContext,
      CGRectMake(0.0f, 0.0f, width, height), [image CGImage]);



void *sourceData = CGBitmapContextGetData(sourceContext);
vImage_Buffer sourceBuffer = {
    .data = sourceData,
    .width = width,
    .height = height,
    .rowBytes = bytesPerRow,
};

size_t length = height * bytesPerRow;
void *destinationData = malloc(length);
vImage_Buffer destinationBuffer = {
    .data = destinationData,
    .width = width,
    .height = height,
    .rowBytes = bytesPerRow,
};

static unsigned char kernel[9] = {
    1, 1, 1,
    1, 1, 1,
    1, 1, 1,
};

vImageDilate_ARGB8888(&sourceBuffer,
                      &destinationBuffer,
                      0,
                      0,
                      kernel,
                      9,
                      9,
                      kvImageCopyInPlace);

CGContextRef destinationContext =
    CGBitmapContextCreateWithData(destinationData,
                                  width,
                                  height,
                                  bitsPerComponent,
                                  bytesPerRow,
                                  colorSpaceRef,
                                  kCGBitmapByteOrderDefault |
                                  kCGImageAlphaPremultipliedFirst,
                                  NULL,
                                  NULL);

CGImageRef dilatedImageRef = CGBitmapContextCreateImage(destinationContext);
UIImage *dilatedImage = [UIImage imageWithCGImage:dilatedImageRef];

CGImageRelease(dilatedImageRef);
CGContextRelease(destinationContext);
CGContextRelease(sourceContext);
CGColorSpaceRelease(colorSpaceRef);
```

Chapter 10

Security

In this era of widespread surveillance, diminishing privacy, and ubiquitous connectivity, security is no longer the pet subject of paranoids—it's something everyone would do well to understand.

As if the concepts in cryptography weren't difficult enough to navigate, the alphabet soup of technologies one must understand to do it correctly makes things nearly incomprehensible. SHA, HMAC, AES, PBKDF, NIST, RSA, NSA, FIPS, DES, …a career in infosec would be just as serviced by a degree in computer science as it would be political science.

Seeing things in code does a lot to help clarify these concepts, though. For anyone prone to squeamishness when it comes to cryptographic acronyms, just take a deep breath and read carefully. All of the really hard stuff is taken care of by the Security framework.

The Security framework can be divided up into Keychain Services, Cryptographic Message Syntax, Security Transform Services, and CommonCrypto.

Keychain Services

Keychain is the password management system on iOS & OS X. It stores certificates and private keys, as well as passwords for websites, servers, wireless networks, and other encrypted volumes.

Using the Security framework, applications can access the keychain programmatically, allowing protected access to user data, without constantly being prompted for authentication. [44] [45]

Unfortunately, convenience for the end user comes at the expense of the developer, for there are few APIs as cumbersome in Cocoa as those for interacting with the Keychain.

Interactions with the Keychain are mediated through queries, rather than direct manipulation. The queries themselves can be quite complicated, and cumbersome with a C API.

A query is a dictionary consisting of the following components:

- The class of item to search for, either "Generic Password", "Internet Password", "Certificate", "Key", or "Identity".
- The return type for the query, either "Data", "Attributes", "Reference", or "Persistent Reference".
- One or more attribute key-value pairs to match on.
- One or more search key-value pairs to further refine the results, such as whether to match strings with case sensitivity, only match trusted certificates, or limit to just one result or return all.

44 Made even more convenient with iCloud Keychain, which syncs credentials to all connected devices.

45 …which used to be available with MobileMe, but was disabled in 2011 when iCloud was introduced.

String constants are used for nearly all of the keys, and many of the values, which makes for a lot of `__bridge id` casts and documentation lookups.

Getting Keychain Items

To get a keychain item, construct a query, and pass it to `SecItemCopyMatching`:

```
NSString *service = @"com.example.app";
NSString *account = @"username";

NSDictionary *query = @{
    (__bridge id)kSecClass: (__bridge id)kSecClassGenericPassword,
    (__bridge id)kSecAttrService: service,
    (__bridge id)kSecAttrAccount: key,
    (__bridge id)kSecMatchLimit: kSecMatchLimitOne,
};

CFTypeRef result;
OSStatus status =
    SecItemCopyMatching((__bridge CFDictionaryRef)query, &result);
```

In this example, the query tells the keychain to find all generic password items for the service `com.example.app` with the matching username. `kSecAttrService` defines the scope of credentials, while kSecAttrAccount acts as a unique identifier. The search option kSecMatchLimitOne is passed to ensure that only the first match is returned, if any.

If status is equal to errSecSuccess (0), then result should be populated with the matching credential.

Adding and Updating Keychain Items

Perhaps the main sticking point of the Keychain Services APIs, however, is that in order to write to the keychain, one must read from it first. There are two write functions: SecItemAdd and SecItemUpdate. Calling SecItemAdd with attributes that already match an existing item returns the status code errSecDuplicateItem. Calling SecItemUpdate with attributes that do not match an existing item returns the status code errSecItemNotFound. For lack of an UPSERT-type command, one is resigned to respond conditionally each time:

```
NSData *data = ...;
if (status == errSecSuccess) {
    NSDictionary *updatedAttributes =
        @{(__bridge id)kSecValueData: data};
    SecItemUpdate((__bridge CFDictionaryRef)query,
                  (__bridge CFDictionaryRef)updatedAttributes);
} else {
    NSMutableDictionary *attributes = [query mutableCopy];
    attributes[(__bridge id)kSecValueData] = data;
    attributes[(__bridge id)kSecAttrAccessible] =
        (__bridge id)kSecAttrAccessibleAfterFirstUnlock;

    SecItemAdd((__bridge CFDictionaryRef)attributes, NULL);
}
```

Following from the previous example, arbitrary data is being set on the item using the `kSecValueData` attribute key. The original query is copied and merged into the additional attributes for `SecItemAdd`, whereas on the updated attributes are passed for `SecItemUpdate`.

Cryptographic Message Syntax

Cryptographic Message Syntax is the IETF's standard for public key encryption and digital signatures for S/MIME messages. Apple's Cryptographic Message Syntax Services in the Security framework provide APIs that implement these industry standard algorithms.

As described in the Core Services chapter, MIME is the internet standard that makes email, well, useful. Without it, email wouldn't support non-ASCII characters or attachments. the S in S/MIME refers to how these messages are sent and received securely.

Messages can either be signed, encrypted, or both, by any number of signers or recipients. To sign a message is to allow the recipient to verify its sender. To encrypt a message is to ensure that it kept secret from everyone but the recipients, who alone are able to decrypt the message's content. These two operations are orthogonal, but cryptographically related.

Encoding a Message

```
NSData *data;
SecCertificateRef certificateRef;

CMSEncoderRef encoder;
CMSEncoderCreate(&encoder);

// Encrypt
CMSEncoderUpdateContent(encoder, [data bytes], [data length]);
CMSEncoderAddRecipients(encoder, certificateRef);

// Sign
SecIdentityRef identityRef = nil;
SecIdentityCreateWithCertificate(nil, certificateRef, &identityRef);
CMSEncoderUpdateContent(encoder, [data bytes], [data length]);
CMSEncoderAddSigners(encoder, identityRef);
CFRelease(identityRef);

CMSEncoderUpdateContent(encoder, [data bytes], [data length]);
```

```
CMSEncoderAddSignedAttributes(encoder, kCMSAttrSmimeCapabilities);

CFDataRef encryptedDataRef;
CMSEncoderCopyEncodedContent(encoder, &encryptedDataRef);
NSData *encryptedData = [NSData dataWithData:(__bridge NSData *) ↩
     encryptedDataRef];

CFRelease(encoder);
```

Decoding a Message

```
CMSDecoderRef decoder;
CMSDecoderCreate(&decoder);

CMSDecoderUpdateMessage(decoder, [encryptedData bytes], [encryptedData length]) ↩
     ;
CMSDecoderFinalizeMessage(decoder);

CFDataRef decryptedDataRef;
CMSDecoderCopyContent(decoder, &decryptedDataRef);
NSData *decryptedData = [NSData dataWithData:(__bridge NSData *) ↩
     decryptedDataRef];

CFRelease(decryptedDataRef);
CFRelease(decoder);
```

Certificate, Key, and Trust Services

A digital certificate is used to verify the identity of its holder or sender.

The best way to understand certificates is to open one up and see what's inside:

```
$ openssl x509 -in certificate.pem -noout -text

Certificate:
   Data:
       Version: 1 (0x0)
       Serial Number: 4919 (0x1337)
       Signature Algorithm: md5WithRSAEncryption
       Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc,
               OU=Certification Services Division,
               CN=Thawte Server CA/emailAddress=server-certs@thawte.com
       Validity
           Not Before: Jun 2 18:00:00 2014 GMT
           Not After : Jun 2 18:00:00 2015 GMT
       Subject: C=US, ST=Oregon, L=Portland, O=Mattt Thompson,
                OU=NSHipster, CN=nshipster.com/emailAddress=mattt@nshipster.com
       Subject Public Key Info:
           Public Key Algorithm: rsaEncryption
           RSA Public Key: (1024 bit)
               Modulus (1024 bit):
                   cb:1c:00:aa:bb:89:a0:4c:26:cd:8c:4b:0b:13:88:...
               Exponent: 65537 (0x10001)
   Signature Algorithm: md5WithRSAEncryption
       f5:5c:d6:a0:bf:39:95:fb:fa:ba:f5:f5:5a:d5:d9:f8:42:6b:...
```

Contrary to their unwieldy reputation, certificates are remarkably easy to parse and understand—even for someone who doesn't have a strong grasp on cryptography in general.

Scanning through the plain text output, several pieces of information come to the surface:

- Certificate issuer
- Validity period
- Certificate holder
- Public key of the owner
- Digital signature from the certification authority

Each certificate is verified by its issuing certificate, thus establishing trust along a chain of certificates all the way up to a root certificate issued by a certificate authority.

Certificates are the basis for the cryptographic infrastructure used to secure the internet. One of the most common interactions an iOS or OS X developer has certificates is an authentication challenge from a URL request:

```
NSURLAuthenticationChallenge *challenge = ...;
SecTrustRef trust = challenge.protectionSpace.serverTrust;
SecPolicyRef X509Policy = SecPolicyCreateBasicX509();
SecTrustSetPolicies(serverTrust, (__bridge CFArrayRef)@[(__bridge id)X509Policy ↩
     ]);

SecTrustResultType result;
assert(SecTrustEvaluate(trust, &result) == errSecSuccess);
```

`SecTrustEvaluate` validates a certificate by verifying its signature, along with the signatures of the certificates in its certificate chain—all the way up to the anchor certificate. A certificate chain is created for each specified policy, starting with the leaf certificate, and checking each certificate in the chain until an invalid certificate is encountered, no more certificates remain, or a certificate with a non-default trust setting is found. [46]

Validating the identity of the certificate holder during an authentication challenge is critical, as it ensures that the server is who it claims to be, and can be trusted with sensitive information.

Security Transform Services

The fundamental concern of cryptography is to preserve the meaning of a message from the time it's encrypted by the sender until it's decrypted by the receiver. When that message is raw binary data, it's sometimes necessary to take an additional step of encoding that data into a text representation. [47] [48]

46 Usually, this an anchor certificate, or one explicitly trusted or distrusted by the user. After evaluation, the result is stored in the trust management object.

47 A lot can go wrong when sending raw binary: characters could be mis-interpreted as control commands, line endings could be misinterpreted, bytes could be sent in the wrong order, or any number of other bizarre possibilities.

48 This is known colloquially as "ASCII Armor".

Base64 encoding maps binary data into 8bit chunks, which are then represented by 64 printable ASCII characters. It's a popular choice because of how it hits a sweet spot between efficiency and practicality. It encodes data with just a 33% overhead, doesn't rely on esoteric characters, and has a relatively straightforward implementation. It's sort of the UTF-8 of binary-to-text encoding.

Base64 is used for everything from HTTP basic authorization to embedded `data-uri` assets in CSS documents, to visually comparing fixed-size byte sequences, like MD5 or SHA-1 checksums.

The security framework provides built in support for Base64 (as well as Base32) encoding and decoding, using the `SecTransformExecute` function.

To encode data, create an instance of `SecTransformRef` with the `kSecBase64Encoding` option and do `SecTransformExecute`:

Base64 Encoding

```
SecTransformRef transform =
    SecEncodeTransformCreate(kSecBase64Encoding, NULL);

SecTransformSetAttribute(transform,
                         kSecTransformInputAttributeName,
                         (__bridge CFDataRef)data,
                         NULL);

NSData *encodedData =
    (__bridge_transfer NSData *)SecTransformExecute(transform, NULL);

CFRelease(transform);
```

The reverse is nearly identical, except passing `kSecBase64Decoding` to `SecEncodeTransformCreate`:

Base64 Decoding

```
SecTransformRef transform =
    SecEncodeTransformCreate(kSecBase64Decoding, NULL);
NSData *decodedData =
    (__bridge_transfer NSData *)SecTransformExecute(transform, NULL);

CFRelease(transform);
```

Randomization Services

Cryptography is predicated on unpredictable, random values. Without such a guarantee, it's all just security theater.

`SecRandomCopyBytes` reads from `/dev/random`, which generates cryptographically-secure random bytes. `/dev/random` is a special file on Unix systems that streams entropy based on the environmental noise of the device.

```
NSUInteger length = 1024;

NSMutableData *mutableData =
    [NSMutableData dataWithLength:length];

OSStatus success =
    SecRandomCopyBytes(kSecRandomDefault,
                       length,
                       mutableData.mutableBytes);

__Require_noErr(success, exit);
```

CommonCrypto

CommonCrypto offers convenient APIs to common cryptographic operations, and is available on OS X 10.5+, and iOS 5.0+.

Digests

Cryptographic hash functions play an important role in information security. Referred to as checksums, fingerprints, or digests, the output of a cryptographic hash function cannot practically be reversed to find the input. [49]

For example, the SHA-1 checksum of "NSHipster" is `7c33b28cb6fe3515548ee58812131de07afeef1b`, whereas performing the same hash function on "CFHipsterRef" generates "342924012ebde06234135698b372e10c5b86c5b2".

To calculate a checksum in code, use the `CC_SHA1` function:

```
NSData *data = ...;

uint8_t output[CC_SHA1_DIGEST_LENGTH];
CC_SHA1(data.bytes, data.length, output);

NSData *digest = [NSData dataWithBytes:output
                                length:CC_SHA1_DIGEST_LENGTH];
```

There are many cryptographic hashing functions out there, each with different security characteristics and use cases. It is the developer's responsibility to evaluate the requirements of their own product to

49 Small changes in the source often cause chaotic differences in generated hash values.

determine the most appropriate security technologies to incorporate.

HMAC

A keyed-hash message authentication code (HMAC) uses cryptographic hash function and secret key to generate a code that can be used to simultaneously verify both the integrity and the authenticity of a message. The strength of an HMAC is contingent on the strength of the cryptographic hash function as well as the size of the secret key. HMACs are often used by web services to ensure that protected calls are only accessible to verified users.

Common Crypto provides the `CCHmac` for generating HMACs:

```
NSData *data, *key;

unsigned int length = CC_SHA1_DIGEST_LENGTH;
unsigned char output[length];

CCHmac(kCCHmacAlgSHA1, key.bytes, key.length, data.bytes, data.length, output);
```

Symmetric Encryption

At the time of writing, AES-128 & PBKDF2 is a reasonable approach for secure symmetric encryption—that is, encrypting and decrypting messages.

The Advanced Encryption Standard (AES) is an encryption specification established by the U.S. National Institute of Standards and Technology (NIST). PBKDF2 is a way of using a hash function that is commonly used to generate the key for a block or stream cypher, like AES.

The Security framework provides the building blocks to do symmetric encryption, but requires developers to roll their own specific implementation.

The first step is to create a function that generates a PBKDF2 key from a salted password. A salt is random data used as an additional input to a one-way function performed on a password.

```
static NSData * AES128PBKDF2KeyWithPassword(NSString *password,
                                            NSData *salt,
                                            NSError * __autoreleasing *error)
{
    NSCParameterAssert(password);
    NSCParameterAssert(salt);

    NSMutableData *mutableDerivedKey = [NSMutableData dataWithLength: ↩
     kCCKeySizeAES128];
    CCCryptorStatus status = CCKeyDerivationPBKDF(kCCPBKDF2, [password  ↩
     UTF8String], [password lengthOfBytesUsingEncoding:NSUTF8StringEncoding],  ↩
     [salt bytes], [salt length], kCCPRFHmacAlgSHA256, 1024, [ ↩
     mutableDerivedKey mutableBytes], kCCKeySizeAES128);

    NSData *derivedKey = nil;
    if (status != kCCSuccess) {
        if (error) {
            *error = [[NSError alloc] initWithDomain:nil code:status userInfo: ↩
     nil];
        }
    } else {
        derivedKey = [NSData dataWithData:mutableDerivedKey];
    }

    return derivedKey;
}
```

Next, a function to encrypt the data can be created, which takes the data to encrypt and password, and returns the generated salt and initialization, as well as any error encountered in performing the operation, as out arguments:

```
static NSData * AES128EncryptedDataWithData(NSData *data,
                                            NSString *password,
                                            NSData * __autoreleasing *salt,
                                            NSData * __autoreleasing * ↩
     initializationVector,
                                            NSError * __autoreleasing *error)
{
    NSCParameterAssert(initializationVector);
    NSCParameterAssert(salt);

    uint8_t *saltBuffer = malloc(8);
    SecRandomCopyBytes(kSecRandomDefault, 8, saltBuffer);
    *salt = [NSData dataWithBytes:saltBuffer length:8];

    NSData *key = AES128PBKDF2KeyWithPassword(password, *salt, error);

    uint8_t *initializationVectorBuffer = malloc(kCCBlockSizeAES128);
    SecRandomCopyBytes(kSecRandomDefault, kCCBlockSizeAES128, ↩
     initializationVectorBuffer);
    *initializationVector = [NSData dataWithBytes:initializationVector length: ↩
     kCCBlockSizeAES128];

    size_t size = [data length] + kCCBlockSizeAES128;
    void *buffer = malloc(size);

    size_t numberOfBytesEncrypted = 0;
    CCCryptorStatus status = CCCrypt(kCCEncrypt, kCCAlgorithmAES128, ↩
     kCCOptionPKCS7Padding, [key bytes], [key length], [*initializationVector ↩
     bytes], [data bytes], [data length], buffer, size, & ↩
     numberOfBytesEncrypted);

    NSData *encryptedData = nil;
    if (status != kCCSuccess) {
        if (error) {
            *error = [[NSError alloc] initWithDomain:nil code:status userInfo: ↩
     nil];
        }
    } else {
        encryptedData = [[NSData alloc] initWithBytes:buffer length: ↩
     numberOfBytesEncrypted];
    }

    return encryptedData;
```

```
}
```

Finally, to decrypt the data, do the same process in reverse, this time passing the data and password along with the salt and initialization vector generated from the encryption function:

```
static NSData * AES128DecryptedDataWithData(NSData *data, NSString *password, ↩
     NSData *salt, NSData *initializationVector, NSError * __autoreleasing * ↩
     error) {
    NSData *key = AES128PBKDF2KeyWithPassword(password, salt, error);

    size_t size = [data length] + kCCBlockSizeAES128;
    void *buffer = malloc(size);

    size_t numberOfBytesDecrypted = 0;
    CCCryptorStatus status = CCCrypt(kCCDecrypt, kCCAlgorithmAES128, ↩
     kCCOptionPKCS7Padding, [key bytes], [key length], [initializationVector ↩
     bytes], [data bytes], [data length], buffer, size, & ↩
     numberOfBytesDecrypted);


    NSData *encryptedData = nil;
    if (status != kCCSuccess) {
        if (error) {
            *error = [[NSError alloc] initWithDomain:nil code:status userInfo: ↩
     nil];
        }
    } else {
        encryptedData = [[NSData alloc] initWithBytes:buffer length: ↩
     numberOfBytesDecrypted];
    }
    return encryptedData;
}
```

Chapter 11

System Configuration

System Configuration contains C APIs for determining hardware configuration and network status.

Most developers' exposure to the framework comes by way of a piece of Apple sample code, known simply as "Reachability". What was intended to demonstrate the use of System Configuration calls to determine internet connectivity, has instead become a part of thousands of shipping applications. In this way, one could argue that Reachability is a victim of its own success. Rather than documenting System Configuration API, it has inadvertently made them obsolete.

The unfortunate consequence is that most developers have a poor understanding of what reachability is, and how it should be used. As a result, many of those applications with `Reachability.m` in their source tree may actually be making the user experience worse than if nothing had been done at all.

Reachability

"Am I connected to the internet?" is a deceptively hard question to answer.

From the user’s perspective, it should be pretty easy, right? Just type `http://apple.com` into Safari, and see if anything loads. Nope.

Networking is so impossibly ad-hoc and idiosyncratic, that it's honestly a surprise that any of this works at all.

Determining Network Reachability Synchronously

Like any networking, establishing reachability should *not* be done synchronously. However, for the purposes of building up to the more complicated asynchronous usage, here's what that would look like:

```
@import SystemConfiguration;

SCNetworkReachabilityRef networkReachability =
    SCNetworkReachabilityCreateWithName(kCFAllocatorDefault,
                                        [@"www.apple.com" UTF8String]);
SCNetworkReachabilityFlags flags =
    SCNetworkReachabilityGetFlags(networkReachability, &flags);

// Use flags to determine reachability

CFRelease(networkReachability);
```

`SCNetworkReachabilityRef` is the data type responsible for determining network reachability. It can be created by either passing host name, like in the previous example, or a `sockaddr` address:

```
BOOL ignoresAdHocWiFi = NO;

struct sockaddr_in ipAddress;
bzero(&ipAddress, sizeof(ipAddress));
ipAddress.sin_len = sizeof(ipAddress);
ipAddress.sin_family = AF_INET;
ipAddress.sin_addr.s_addr =
  htonl(ignoresAdHocWiFi ? INADDR_ANY : IN_LINKLOCALNETNUM);

SCNetworkReachabilityRef networkReachability =
    SCNetworkReachabilityCreateWithAddress(kCFAllocatorDefault,
                        (struct sockaddr *)ipAddress);
```

`SCNetworkReachabilityFlags` is a synchronous call that determines the reachability of the available network interfaces. Because there are so many different factors that affect reachability, the returned value of this function is not a simple `YES` / `NO`, but a bitmask of characteristics:

Reachability Flag Values

Reachable	The specified node name or address can be reached using the current network configuration.
Transient Connection	The specified node name or address can be reached via a transient connection, such as PPP.
Connection Required	The specified node name or address can be reached using the current network configuration, but a connection must first be established.

(continued)

Connection On Traffic	The specified node name or address can be reached using the current network configuration, but a connection must first be established. Any traffic directed to the specified name or address will initiate the connection.
Connection On Demand	The specified node name or address can be reached using the current network configuration, but a connection must first be established.
Intervention Required	The specified node name or address can be reached using the current network configuration, but a connection must first be established. In addition, some form of user intervention will be required to establish this connection, such as providing a password, an authentication token, etc.
Is Local Address	The specified node name or address is one that is associated with a network interface on the current system.
Is Direct	Network traffic to the specified node name or address will not go through a gateway, but is routed directly to one of the interfaces in the system.

While the intricacies of network interfaces are interesting, they are little more than an academic exercise for app developers, who, like their users, would honestly prefer a `YES` or `NO` answer.

Here’s how to boil down a complex truth:

```
BOOL isReachable =
  ((flags & kSCNetworkReachabilityFlagsReachable) != 0);

BOOL needsConnection =
  ((flags & kSCNetworkReachabilityFlagsConnectionRequired) != 0);

BOOL canConnectionAutomatically =
  (((flags & kSCNetworkReachabilityFlagsConnectionOnDemand ) != 0) ||
   ((flags & kSCNetworkReachabilityFlagsConnectionOnTraffic) != 0));

BOOL canConnectWithoutUserInteraction =
  (canConnectionAutomatically &&
   (flags & kSCNetworkReachabilityFlagsInterventionRequired) == 0);

BOOL isNetworkReachable =
  (isReachable &&
   (!needsConnection || canConnectWithoutUserInteraction));
```

Taking it a step further, reachability flags can also be used to determine which network interface is being used:

```
if (isNetworkReachable == NO) {
    // Not Reachable
}
```

```
#if TARGET_OS_IPHONE
else if ((flags & kSCNetworkReachabilityFlagsIsWWAN) != 0) {
    // Reachable via WWAN
}
#endif
else {
    // Reachable via WiFi
}
```

Calling `SCNetworkReachabilityFlags` on the main thread invokes a DNS lookup with a 30 second timeout. [50] This is bad. Don't make blocking, synchronous calls to `SCNetworkReachabilityFlags`.

Determining Network Reachability Asynchronously

Thankfully, the System Configuration framework provides a set of APIs for monitoring reachability changes asynchronously.

First, define a static callback function, which takes the network reachability reference, the flags, and any additional context to be passed.

50 To put this into a perspective: an app can expect to be killed by the system watchdog process after 20 seconds.

```
static void ReachabilityCallback(
  SCNetworkReachabilityRef target,
  SCNetworkConnectionFlags flags,
  void *context)
{
   // ...
}
```

Once the callback function is declared, it can be set on an `SCNetworkReachabilityRef`, which is then scheduled on a runloop:

```
SCNetworkReachabilityContext context = {0, NULL, NULL, NULL, NULL};

SCNetworkReachabilitySetCallback(networkReachability,
                                 ReachabilityCallback,
                                 &context));

SCNetworkReachabilityScheduleWithRunLoop(reachability,
                                         CFRunLoopGetMain(),
                                         kCFRunLoopCommonModes));
```

Now, whenever the network reachability of the device changes, the `ReachabilityCallback` function will be called. This information can be then communicated to the application by posting to `NSNotificationQueue`, or by invoking a block passed into the

`SCNetworkReachabilityContext` argument.

Network Configuration Settings

One would be forgiven for thinking that System Configuration was a one-trick pony. Almost every mention to the framework is made in reference to network reachability—which itself, is a topic dominated by that darned Reachability sample.

The more obscure swathe of functionality has to do with querying the dynamic store of a system: `SCDynamicStore`. These APIs are, alas, only available on OS X.

Querying

`SCDynamicStoreRef` provides a key-value interface to the current system configuration, which is managed by the `configd` daemon.

A list of available keys can be divined using `SCDynamicStoreCopyKeyList`:

```
SCDynamicStoreContext context = { 0, NULL, NULL, NULL, NULL };
SCDynamicStoreRef store = SCDynamicStoreCreate(NULL, NULL, nil, &context);
NSArray *keys =
    (__bridge_transfer  NSArray *)
          SCDynamicStoreCopyKeyList(store, CFSTR(".+"));
```

- `Setup:/Network/Service/.../IPv4,`
- `Setup:/Network/Service/.../IPv6,`
- … 100+ Items …
- `State:/Network/Interface/p2p0/Link,`
- `State:/Network/Interface/lo0/IPv6,`
- `State:/IOKit/LowBatteryWarning,`
- `State:/Network/MulticastDNS,`
- `State:/Network/Global/Proxies,`
- `State:/Network/Interface/bridge0/Link`

For a complete list of keys, see Apple's System Configuration Programming Guide. [a]

[a] https://developer.apple.com/

With a list of keys in hand, the purpose of `SCDynamicStore` becomes a little clearer. An application can be notified of changes in IP address, network interfaces, or even when the device's battery is running low.

The semantics here are the same as what's used for monitoring network reachability.

As an example, a list of available network interfaces can be found with the `State:/Network/Interface` key:

```
SCDynamicStoreContext context = { 0, NULL, NULL, NULL, NULL };
SCDynamicStoreRef store = SCDynamicStoreCreate(NULL, NULL, nil, &context);
CFPropertyListRef propertyList =
    SCDynamicStoreCopyValue(store, CFSTR("State:/Network/Interface"));
NSArray *interfaces =
    (__bridge NSArray *)
          CFDictionaryGetValue(propertyList, CFSTR("Interfaces"));
```

- `lo0`
- `gif0`
- `stf0`
- `en0`
- `en1`
- `en2`
- `bridge0`
- `p2p0`
- `utun0`

Monitoring

The real utility of the dynamic store is being able to monitor changes, so that an application can immediately respond to things like Airport being turned on or the local IP address changing.

In a way, this is just a more generalized formulation of monitoring network reachability. Create a dynamic store, listen on a particular network service entity key, and set a callback function pointer:

```
static void Callback(SCDynamicStoreRef store,
                     CFArrayRef changedKeys,
                     void *info)
{
  if (info) {
    ((void (^)())info)(changedKeys);
  }
}

id callback = ^(NSArray *changedKeys){
  // ...
}

SCDynamicStoreContext context =
  {0, (__bridge void *)callback, NULL, NULL, NULL};

SCDynamicStoreRef store =
  SCDynamicStoreCreate(NULL,
                       CFSTR("IPv4AddressMonitor"),
                       Callback,
                        &context);

NSString *ipv4 =
  (NSString *)SCDynamicStoreKeyCreateNetworkServiceEntity(NULL,
                  kSCDynamicStoreDomainState,
                  kSCCompAnyRegex,
                  kSCEntNetIPv4);

SCDynamicStoreSetNotificationKeys(store, NULL, @[ipv4])
SCDynamicStoreSetDispatchQueue(store, dispatch_get_main_queue());
```

Chapter 12

International Components for Unicode

Language is the essence of our very consciousness. It's how we reason. It's why we can reason at all.

We understand the world in terms of nouns, and navigate through it with verbs. Adjectives focus discrete observations into feelings and judgements. We progress as a society through the written records of our fore-bearers. And it will be our own words that outlast us, in the end.

That we can communicate at all is a miracle. That ideas can travel from one mind to another is beyond belief—and yet is so well-understood that it barely registers a second thought. By communicating, we create understanding among one another. We evoke empathy, and expand the boundaries of our moral consideration to others. Our world gets bigger.

However, when linguistic distance is too much to overcome and communication breaks down, it becomes difficult to empathize. A sense of otherness forms. Whether interpersonally or internationally, failure to communicate remains the primary cause of conflict.

As a technology that has done so much to eliminate linguistic and cultural distance, there is a strong case to be made that Unicode is one of the most important technologies ever created for our species.

At least in terms of Apple's SDKs, the role Unicode plays is difficult to over-state. This chapter will look at how one aspect of Unicode in particular, the ICU, or International Components for Unicode, are used in Foundation and Core Foundation.

Unicode

Unicode is the bedrock of international computing. It all started in 1987 with three individuals: Joe Becker from Xerox, and Lee Collins & Mark Davis from Apple.

Their goals for Unicode were simple, yet ambitious:

- Be universal, addressing the needs of world languages.
- Be uniform, with fixed-width codes for efficient access.
- Be unique, such that each bit sequence has only one interpretation.

Since its inception, Unicode has succeeded in creating a universally-adopted standard, with over 100,000 characters represented for languages used by billions of people.

ICU

ICU, or International Components for Unicode, is the industry standard for providing Unicode and globalization support in software.

It was was created by IBM in the 90's, and has been continuously maintained since then.

ICU4C, its C/C++ libraries, form the backbone of Apple's operating systems, in the form of libicucore, a private-ish framework used extensively by Core Foundation and Foundation, but unavailable for public consumption. While it is possible to vendor libicucore, there is little practical advantage over simply using the public SDK APIs built on top of it, such as `NSLocale`, `NSCalendar`, and `CFStringTransform`.

As such, this chapter will investigate ICU as a way to better understanding how those higher-level APIs work, and how this information can be used to exploit undocumented API features.

CLDR

The CLDR, or Common Locale Data Repository, is what makes ICU so compelling as a technology. Weighing in at over 400MB of JSON data, the CLDR contains the authoritative encoding for all of humanity's cultural conventions.

The CLDR is available for download in both XML and JSON formats from the Unicode website. [a]

[a] http://cldr.unicode.org/index/downloads

The `main` directory of the CLDR contains a multitude of subdirectories—one for each available locale. Within each locale directory are a collection of files describing a particular aspect of that locale:

Calendars

There are 18 different calendars represented in the CLDR, from the standard Gregorian, to all manners of ancient, religious, and obscure systems.

- Calendar
- Buddhist
- Chinese
- Coptic (a.k.a Alexandrian)
- Dangi
- Ethiopic
- Ethiopic (Amete Alem)
- Hebrew
- Indian
- Islamic
- Islamic (Civil)

- Islamic (Saudi Arabia)
- Islamic (Tabular)
- Islamic (Um al-Qura)
- Minguo (Republic of China)
- Japanese
- Persian

Each calendar is represented in a separate file for each locale. Each file is several hundred lines long, and contains the months of the year and days of the week in various levels of abbreviation, as well as the formatting rules for dates and time intervals.

`NSCalendar` and `NSDateFormatter` use this information to parse and format dates into locale-appropriate formats: [51]

Characters

For each language spoken within a locale, an inventory of characters is provided, along with collation indexes and formatting rules for ellipsis.

51 Due to their differing release schedules, changes in the CLDR are leading indicators of the next version of iOS & OS X. For example, the CLDR v25 added information about the Coptic and Ethiopic calendars, which were then added to `NSCalendar` in iOS 8 & OS X 10.10.

Character inventories are likely used by `NSLinguisticTagger` as a low pass for evaluating the `NSLinguisticTagSchemeLanguage` of a string. A string with characters beyond the orthographic inventory of a language is unlikely to match. Conversely, the relative frequencies of exemplar characters may be informative in deciding between two likely candidates.

Exemplar characters for an `NSLocale` can be retrieved with the `NSLocaleExemplarCharacterSet` key.

Collation indexes are used by `UILocalizedIndexedCollation` to segment linguistic records appropriately for the current locale. American English uses the Latin alphabet to collate information like names:

```
[A B C D E F G H I J K L M N O P Q R S T U V W X Y Z]
```

However, Swedish extends the Latin alphabet, with a few extra characters of its own: [52]

```
[A Á B C Č D Đ E É F G H I J K L M N Ŋ O P Q R S Š T Ŧ U V W X Y Z Ž Ø Æ Å Ä Ö]
```

52 Order indeed matters! Putting Å entries immediately after A in Swedish would be as jarring and nonsensical as clustering b, d, p, and q together in English because they happen to look the same.

Ellipsis rules specify how truncated text should be formatted, which, depending on whether the truncation, is to be done in the initial, medial, or final position, and whether there's a word boundary at that point:

Example Ellipsis Rules

Initial	
Medial	
Final	
Word Initial	
Word Medial	
Word Final	

Currencies

Forget music or Esperanto; *money* is the real universal language. Each currency is listed according to its ISO 4217 code (USD, EUR, GBP, etc.), and includes its locale-specific symbol ($, €, £, etc.). Most of this information is consistent across various locales, but there is built-in redundancy to accommodate things like count-specific display names.

`NSLocale` uses this information to lookup the appropriate currency code and symbol for a specified locale. This information is, in turn, passed into `NSNumberFormatter` when presenting numbers with `NSNumberFormatterCurrencyStyle`.

Date Fields

In addition to calendar details, each locale has rules for how to do relative date formatting, such as "now", "yesterday" or "last week". Idiomatic deictics vary between different languages. For example, German has the word "vorgestern" to describe "the day before yesterday". Along with idiomatic phrases, there are formats for conventional / formulaic past and future deictics, such as "1 week ago" and "in 4 seconds".

This information is used by `NSDateFormatter` when `doesRelativeDateFormatting` is set to `YES`.

Delimiters

Each language has its own take on how to delimit quotations:

Quotation Delimiters

English	“I can eat glass, it doesn’t harm me.”
German	„Ich kann Glas essen, das tut mir nicht weh.“

The CLDR specifies primary (“ ”) and optional alternate (‘ ’) quotation start and end delimiters for each language used by a locale.

Quotation delimiters for an `NSLocale` can be retrieved with the `NSLocaleQuotationBeginDelimiterKey` / `NSLocaleAlternateQuotationBeginDelimiterKey` and `NSLocaleQuotationEndDelimiterKey` / `NSLocaleAlternateQuotationEndDelimiterKey` keys.

Languages

This one is kinda meta. The `languages` file specifies its respective locale’s way to refer to that language. [53]

As an example, `NSLocale` uses this information for `displayNameForKey` when passed the `NSLocaleLanguageCode` key:

53 Throughout the CLDR, the canonical identifier for a language is its ISO 639 code.

```
NSLocale *frLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"fr_FR"];
NSLog(@"fr: %@",
  [frLocale displayNameForKey:NSLocaleLanguageCode
                        value:@"fr"]);
NSLog(@"en: %@",
  [frLocale displayNameForKey:NSLocaleLanguageCode
                        value:@"en"]);
```

NSLocaleLanguageCode

`fr`	français
`en`	anglais

Layout

Layout details for a locale are pretty simple: they specify the character order (left-to-right or right-to-left) and line order (top-to-bottom or bottom-to-top) for each language.

List Patterns

Lists are the heart, body, and soul of the accusative case. Rules for how items in a list are delimited vary between locales and languages, and all of those differences are enumerated in the CLDR.

From an implementation perspective, it's interesting to see how these rules are codified. Instead of specifying, for example, delimiting character and conjunction, which would make sense at least for English, the CLDR specifies formats for start, middle, and end patterns. Languages may, in turn, have unique sets of patterns to accommodate different contexts, like units.

In practice, the main differences between locales are whether to use the standard or full-width comma, whether to use inter-item spacing, and whether to use a conjunction at the end. [54]

Foundation does not currently make use of this information, as it does not provide anything along the lines of an `NSArrayFormatter`.

54 Conjunctive list patterns also offer an alternative to accommodate the terminal delimiter, a.k.a. Oxford Comma.

Locale Display Names

Like `languages`, the `localeDisplayNames` file defines locale-specific display names for different types of information. Sort order, number systems, writing systems, calendars, and collation schemes are all represented here.

This information is used throughout iOS and OS X any time locale preferences are provided. However, not all of the information provided by the CLDR can be accessed through system APIs.

Measurement System Names

The world is *so* close to making this information irrelevant. If only we could get the USA on the SI train to metric town… what a world it would be.

Anyway, the `measurementSystemNames` file specifies the localized name of each measurement system (Metric, US, & UK).

To determine if a given `NSLocale` uses the metric system, use the `NSLocaleUsesMetricSystem` key in `-objectForKey:`.

Numbers

This file codifies all of the rules about number formatting for a locale: number system, formats for decimal, scientific, percent, and currency styles; preferred symbols for decimal (.), group (,), list (;), plus sign (+), minus sign (-), percent sign (%), per mille sign (‰), exponential (E), infinity (∞), and not a number (NaN); and patterns for number ranges. [55]

As you might expect, this is where `NSNumberFormatter` derives its formatting rules.

POSIX

This one is actually quite interesting—not something most English-speaking computer users probably have considered.

When Unix commands prompt for confirmation, they are expecting a yes or no answer. In English, it's pretty clear: `yes` / `y` or `no` / `no`.

But what about other languages?

55 Unicode nerds will be delighted by the fact that the generic currency symbol, "¤", is put to good use here.

In Italian, the options are `sì` / `si` / `s` or `no` / `n`. In Russian, `да` / `д` or `нет` / `н` are the acceptable answers.

The CLDR has rules for each language, making it straightforward for software developers using ICU to make agreeable software in all locales.

Since most iOS and OS X apps prefer GUIs to CLIs, this is not a prescient matter, and as such, is not supported by the SDKs.

Scripts

There is a many-to-many relationship between scripts and languages. Some languages have multiple scripts. Most scripts are used by more than one language.

ISO 15924 is the standard for identifying scripts. Each script is assigned both a 4 character and a numeric identifier.

For example, `Latn` is Latin script, `Hira` is Japanese Hiragana, and `Brai` is Braille.

For each locale in the CLDR, there are localized names for each script. `NSLocale` can tap into this by fetching the `NSLocaleScriptCode` key.

Territories

Here's where things start to become a little tense. A locale's `territories` file includes names for countries (according to ISO 3166) and world region, according to their United Nations geoscheme ID.

The difficulty here is how politically contentious names and geography can be. Some countries may not be recognized by other countries, or may have territory annexed as part of an armed conflict.

Since programmers don't have a dog in most geopolitical fights, adopting ICU standards is a smart choice, which minimizes the possibility of unintentionally sparking an international crisis.

Only country codes are exposed by `NSLocale`, with the `NSLocaleCountryCode` key. However, AddressBook and other frameworks harness the CLDR database to localize country names throughout the operating system.

Time Zone Names

Any programmer who knows enough about time zones knows that they want nothing to do with coding any of that themselves.

Timezones range from UTC−12 to UTC+14—spanning a total of 26 hours, which is weird, considering that a day only has 24. Some time zones observe daylight savings time, while others don't. And of the ones that do, some of them use partial offsets, ±30 or 45 minutes in some cases. Certain countries that sweep across a wide arc of longitude, like the United States and Canada, are split up across a number of different time zones. Other countries, like China, are standardized to only a single timezone for an equivalent span, meaning that by the time the sun rises in the western city of Kashgar at 8AM, it's nearly mid-day in Beijing.

With so many edge cases, it's as if every time zone is an exception to the rule. As such, each `timezones` file is several thousand lines long, and includes listings for hundreds of regions, countries, and cities around the world.

Thankfully, `NSTimeZone` makes its calculations based on this information so you don't have to.

Transform Names

A transform is the process by which text in one script or writing standard is converted into another. For major scripts, there are standardized conventions for transforms. BGN is used to transform Russian Cyrillic into Latin, Jamo for Korean Hangul to Latin, and Pinyin for Chinese to Latin. There are transforms for CJK (Chinese, Japanese, Korean) characters to go between half- and full-width representations. There is also the UNGEGN (United Nations Group of Experts on Geographical Names) transform, which standardizes the transliteration of toponyms, or place names.

Each of these standard transforms has a name associated with them, which vary across locales. The CLDR has correspondences for each language.

Units

There are many different types of units, each representing a particular physical quantity, like acceleration, angle, area, duration, length, mass, power, pressure, speed, temperature, or volume. Since a locale can have slightly different standards for how to format and represent these units, the CLDR provides patterns for each.

With the introduction of HealthKit, Foundation added formatters for energy, mass, and length. MapKit also provides a formatter for distance in miles and kilometers. Each of these take advantage of the unit formatting rules in the CLDR.

Variants

The `variants` records of a locale are a grab bag of localized names for BCP 47 subtags, which include dialects, orthographies, and transliteration schemes. These are significant alternatives to the accepted standards of a particular language, like, for example, the Wade-Giles and Hepburn romanization strategies for Mandarin Chinese and Japanese, which were made obsolete by Pinyin and Rōmaji, respectively.

Even for Unicode, many of these are pretty obscure. A tag for specifying the Late Middle French dialect based on Jean Nicot's 1606 foundational lexicographic text *Thresor de la langue francoyse*? Yeah, probably not going to be relevant for the next big social networking app.

Supplemental

Finally, in a totally separate top-level directory exists a supplemental directory of records. There's almost as much going on in here as in the individual locale records, but since not as much of it is currently available through Objective-C APIs, we'll just skim through:

- Calendar Data: Epochs of calendar system eras, and whether the calendar was based on the lunar or solar cycles.
- Calendar Preference Data: Ordered list of calendars supported in each locale, sorted by preference.
- Character Fallbacks: Simpler alternatives for less well-supported characters, such as `(C)` for "©" or `1/2` for "½", as well as currency symbols, ligatures, and compound characters in Korean and Hebrew.
- Code Mappings: Top-level domain code mappings.
- Currency Data: Histories of currencies used by different countries, including start and end date of usage.
- Day Periods: Various schemes for dividing up the hours of a day, from simple: "a.m. / p.m.", to excessively precise: "wee hours / early morning / morning / late morning / noon / mid day / afternoon / evening / late evening / night".
- Gender Rules for Plurals: Rules for how to gender plurals. [56]
- Language Data: A list of languages and their respective scripts and territories.
- Language Matching: Rules for how similar languages can be interchanged, such as Kazakh and Russian.
- Likely Subtags: Given a BCP 47 language tag, the most likely subtags to be associated.

56 In some languages, like Arabic, the addition of a single male-gendered word "taints" a collection of female-gendered words, making the entire collection male-gendered. Other languages do not have such a rule—or better yet, lack gendered nouns completely.

- Measurement Data: Which countries use metric vs. imperial units, or A4 vs. US-Letter for paper sizes. [57]
- Metadata: There be dragons.
- Metazones: Records establishing a regional hierarchy for localities.
- Numbering Systems: Inventory and rules for alternative numbering systems, like Arabic, Roman, full-width CJK, and spelled out English numerals.
- Ordinals: Rules or ordinal numbers for each language (i.e. 1st, 2nd, 3rd, etc.).
- Parent Locale: Establishes a directed graph relationship from regions to parent locale.
- Plural Rules: Each language uses any of the 6 distinct Unicode counting rules: zero, one, two, few, many, and other.
- Postal Code Data: Regular expressions describing the postal code rules for each country.
- Primary Zones: The primary time zones.
- References: A bibliography of sources used to determine all of these different rules. [58]
- Telephone Code Data: International dialing codes for each country.
- Territory Containment: Establishes the spatial relationships for the geographic areas of territories.

57 The rule "001", the UN geographic region code for the world, is used as a catch-all, such that only the exceptions need be defined.

58 A significant percentage of citations are for Wikipedia or the CIA World Factbook.

- Territory Information: A breakdown of regional statistics, including population, GDP, literacy rate, and language populations.
- Time Data: tl;dr - `{"_allowed":"H h", "_preferred":"h"}`.
- Week Data: For each locale, the minimum number of days in a week, and which day is the start of the week.
- Windows Zones: Legacy mapping of timezone information to however Microsoft did things in the past.

Transforms

Originally designed to convert text in one script to another, ICU transforms have evolved into a powerful tool for working with Unicode text, with case and width conversion, composite character sequence normalization, and removal of accents and diacritics.

ICU transforms are exposed through the `CFStringTransform` function in Core Foundation. About a dozen or so string constants are defined for common operations, like `kCFStringTransformToLatin`, which conveniently transliterates text into its corresponding Latin alphabet representation. Unfortunately, these constants have opaque values, which ends up obscuring the fact that `CFStringTransform` will accept any valid ICU transform.

An ICU transform consists of 1 or more semicolon-delimited mappings. Each mapping is either unidirectional or bidirectional between the left-hand side and right-hand side values.

For example, a bidirectional transform between a copyright symbol and its ASCII representation could be expressed as:

```
(c) <> ©;
```

The `}` operand constrains a rule to a particular context, like in this unidirectional mapping that only removes hyphens after lowercase letters:

```
[:lowercase letter:] } '-' > '';
```

Each mapping is evaluated in order, and therefore should be listed starting with the most specific rules and ending with the most general.

A complete reference for ICU transform rule syntax can be found on the ICU project website. [a]

[a] http://userguide.icu-project.org/transforms/general/rules

ICU provides a number of built-in transliterations for common and useful operations, which can be combined with other rules to accomplish virtually any automated text transformation task.

Text Processing

ICU has transliterations for basic text processing tasks like changing case or normalization:

Text Processing Transforms

`Any-Null`	Has no effect; leaves input text unchanged.
`Any-Remove`	Deletes input characters. This is useful when combined with a filter that restricts the characters to be deleted.
`Any-Lower`, `Any-Upper`, `Any-Title`	Converts to the specified case. See Case Mappings for more information.
`Any-NFD`, `Any-NFC`, `Any-NFKD`, `Any-NFKC`, `Any-FCD`, `Any-FCC`	Converts to the specified normalized form.
`Any-Publishing`	Converts between real punctuation and typewriter punctuation.

Accent and Diacritic Stripping

One of the most common normalization tasks is the removal of accents and diacritics. ICU transforms provide a flexible solution to this problem.

Normalization Transforms

"NFD; [:Mn:] Remove; NFC"	Removes all accents and diacritics.

Using this transform, "Énġlišh långuãge läcks iñterêsţing diaçrïtičş" becomes "English language lacks interesting diacritics"

The `kCFStringTransformStripCombiningMarks` constant can also be used to the same effect.

Unicode Symbol Naming

Every code point in the Unicode standard has an official name, which can be retrieved using an ICU transform:

Unicode Symbol Naming Transforms

`Any-Name`	Replaces each character with its Unicode name.

Applying this transform to "å" yields "{LATIN SMALL LETTER A WITH RING ABOVE}". [59] [60]

Script Transliteration

Script transliterations are a killer feature of ICU, to put it mildly. For the billions of people in the world who can only read or write in the scripts of their native languages, the ability to transform any text into something pronounceable is itself transformative to humanity.

The ICU includes the following transliterations:

- Latin <-> Arabic, Armenian, Bopomofo, Cyrillic, Georgian, Greek, Han, Hangul, Hebrew, Hiragana, Indic (Devanagari, Gujarati, Gurmukhi, Kannada, Malayalam, Oriya, Tamil, & Telegu), Jamo, Katakana, Syriac, Thaana, & Thai.

59 This transformation is especially useful (or at least entertaining) for Emoji.

60 `Any-Name` is equivalent to using the `kCFStringTransformToUnicodeName` constant defined in Core Foundation.

- Indic <-> Indic
- Hiragana <-> Katakana
- Simplified Chinese (`Hans`) <-> Traditional Chinese (`Hant`)

Source and target specifiers can be be script identifiers ("Latin" / "Latn"), Unicode language identifiers (`fr`, `en_US`, `zh_Hant`), or special tags (`Any`, `Hex`).

Here are some examples of transliterations chained together in useful ways:

Script Transliteration Transforms

`Any-Latin`	Transliterate text into Latin script, perhaps in order to make it pronounceable by English speakers.
`Any-Latin;Latin-Hangul`	Transliterate text into Hangul, using Latin as an intermediate representation
`Any-Latin;Latin-ASCII;[:^ASCII:] Remove`	Transliterate text to ASCII from intermediate Latin representation, removing any non-ASCII characters in the process.

(continued)

`[:Latin:];NFKD;Lower;Latin-Katakana;Fullwidth-Halfwidth`	For all Latin characters, normalize according to Normalization Form Compatibility Decomposition, change to lowercase, transliterate into Katakana, and then convert into the halfwidth representation.
`Any-Latin;Latin-NumericPinyin`	Transliterate text into Latin, changing Pinyin accents into their numeric equivalents .

Chapter 13

Dictionary Services

Though widely usurped of their "go-to reference" status by the Internet, dictionaries and word lists serve an important role behind the scenes. A vast array of functionality relies on this information, ranging from spell check, grammar check, and auto-correct to auto-summarization and semantic analysis.

So, as a reference, here's a look at the ways and means by which computers give meaning to the world through words, in Unix, OS X, and iOS.

Unix

Nearly all Unix distributions include a small collection of newline-delimited lists of words. On OS X, these can be found at `/usr/share/dict`:

```
$ ls /usr/share/dict
    README
    connectives
    propernames
    web2
    web2a
    words@ -> web2
```

Symlinked to `words` is the `web2` word list, which, though not exhaustive, is still a sizable corpus:

```
$ wc /usr/share/dict/words
    235886  235886 2493109
```

Skimming with `head` shows what fun lies herein. Such excitement is rarely so palpable as it is among words beginning with "a":

```
$ head /usr/share/dict/words
    A
    a
    aa
    aal
    aalii
    aam
    Aani
    aardvark
    aardwolf
    Aaron
```

These giant, system-provided text files make it easy to `grep` crossword puzzle clues, generate mnemonic pass phrases, and seed databases, but from a user perspective, `/usr/share/dict` 's monolingualism and lack of associated meaning render it pretty useless.

OS X builds upon this with its own system dictionaries. Never one to disappoint, the operating system's penchant for extending Unix functionality through strategically placed bundles and plist files is in full force with how dictionaries are distributed.

OS X

The OS X analog to `/usr/share/dict` can be found in `/Library/Dictionaries`. A quick peek into the directory demonstrates one immediate improvement over Unix, by acknowledging the existence of languages other than English:

```
$ ls /Library/Dictionaries/

  Apple Dictionary.dictionary/
  Diccionario General de la Lengua Espan˜ola Vox.dictionary/
  Duden Dictionary Data Set I.dictionary/
  Dutch.dictionary/
  Italian.dictionary/
  Korean - English.dictionary/
  Korean.dictionary/
  Multidictionnaire de la langue francaise.dictionary/
  New Oxford American Dictionary.dictionary/
  Oxford American Writer's Thesaurus.dictionary/
  Oxford Dictionary of English.dictionary/
  Oxford Thesaurus of English.dictionary/
  Sanseido Super Daijirin.dictionary/
  Sanseido The WISDOM English-Japanese Japanese-English Dictionary.dictionary/
  Simplified Chinese - English.dictionary/
  The Standard Dictionary of Contemporary Chinese.dictionary/
```

OS X ships with dictionaries in Chinese, English, French, Dutch, Italian, Japanese, and Korean, as well as an English thesaurus and a special dictionary for Apple-specific terminology.

Diving deeper into the rabbit hole, we peruse the `.dictionary` bundles to see them for what they really are:

```
$ ls "/Library/Dictionaries/New Oxford American Dictionary.dictionary/Contents"

    Body.data
    DefaultStyle.css
    EntryID.data
    EntryID.index
    Images/
    Info.plist
    KeyText.data
    KeyText.index
    Resources/
    _CodeSignature/
    version.plist
```

A filesystem autopsy reveals some interesting implementation details. In the case of the New Oxford American Dictionary in particular, contents include:

- Binary-encoded `KeyText.data`, `KeyText.index`, & `Content.data`
- CSS for styling entries
- 1207 images, from A-Frame to Zither

- Preference to switch between US English Diacritical Pronunciation and International Phonetic Alphabet (IPA)
- Manifest & signature for dictionary contents

Normally, proprietary binary encoding would mean the end of what one could reasonably do with data, but luckily, Core Services provides APIs to read this information.

Getting Definition of Word

To get the definition of a word on OS X, one can use the `DCSCopyTextDefinition` function, found in the Core Services framework:

```
#import <CoreServices/CoreServices.h>

NSString *word = @"apple";
NSString *definition = (__bridge_transfer NSString *)DCSCopyTextDefinition(NULL ↩
     , (__bridge CFStringRef)word, CFRangeMake(0, [word length]));
NSLog(@"%@", definition);
```

Wait, where did all of those great dictionaries go?

Well, they all disappeared into that first `NULL` argument. One might expect to provide a `DCSCopyTextDefinition` type here, as prescribed by the function definition. However, there are no public functions to construct or copy such a type, making `NULL` the only available option. The documentation is as clear as it is stern:

"This parameter is reserved for future use, so pass `NULL`. Dictionary Services searches in all active dictionaries."

"Dictionary Services searches in all active dictionaries", you say? Sounds like a loophole!

Setting Active Dictionaries

Now, there's nothing programmers love to hate to love more than the practice of exploiting loopholes to side-step Apple platform restrictions. Behold: an entirely error-prone approach to getting, say, thesaurus results instead of the first definition available in the standard dictionary:

```
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
NSMutableDictionary *dictionaryPreferences = [[userDefaults  ↩
     persistentDomainForName:@"com.apple.DictionaryServices"] mutableCopy];
NSArray *activeDictionaries = [dictionaryPreferences objectForKey:@" ↩
     DCSActiveDictionaries"];
dictionaryPreferences[@"DCSActiveDictionaries"] = @[@"/Library/Dictionaries/ ↩
     Oxford American Writer's Thesaurus.dictionary"];
```

```
[userDefaults setPersistentDomain:dictionaryPreferences forName:@"com.apple. ↩
     DictionaryServices"];
{
    NSString *word = @"apple";
    NSString *definition = (__bridge_transfer NSString *)DCSCopyTextDefinition( ↩
     NULL, (__bridge CFStringRef)word, CFRangeMake(0, [word length]));
    NSLog(@"%@", definition);
}
dictionaryPreferences[@"DCSActiveDictionaries"] = activeDictionaries;
[userDefaults setPersistentDomain:dictionaryPreferences forName:@"com.apple. ↩
     DictionaryServices"];
```

"But this is OS X, a platform whose manifest destiny cannot be contained by meager sandboxing attempts from Cupertino!", you cry. "Isn't there a more civilized approach? Like, say, private APIs?"

Why yes, yes there are.

Private APIs

Not publicly exposed, but still available through Core Services are a number of functions that cut closer to the dictionary services that we crave:

```
extern CFArrayRef DCSCopyAvailableDictionaries();
extern CFStringRef DCSDictionaryGetName(DCSDictionaryRef dictionary);
extern CFStringRef DCSDictionaryGetShortName(DCSDictionaryRef dictionary);
extern DCSDictionaryRef DCSDictionaryCreate(CFURLRef url);
extern CFStringRef DCSDictionaryGetName(DCSDictionaryRef dictionary);
extern CFArrayRef DCSCopyRecordsForSearchString(DCSDictionaryRef dictionary,  ↩
     CFStringRef string, void *, void *);

extern CFDictionaryRef DCSCopyDefinitionMarkup(DCSDictionaryRef dictionary,  ↩
     CFStringRef record);
extern CFStringRef DCSRecordCopyData(CFTypeRef record);
extern CFStringRef DCSRecordCopyDataURL(CFTypeRef record);
extern CFStringRef DCSRecordGetAnchor(CFTypeRef record);
extern CFStringRef DCSRecordGetAssociatedObj(CFTypeRef record);
extern CFStringRef DCSRecordGetHeadword(CFTypeRef record);
extern CFStringRef DCSRecordGetRawHeadword(CFTypeRef record);
extern CFStringRef DCSRecordGetString(CFTypeRef record);
extern CFStringRef DCSRecordGetTitle(CFTypeRef record);
extern DCSDictionaryRef DCSRecordGetSubDictionary(CFTypeRef record);
```

Private as they are, these functions aren't about to start documenting themselves, so let's take a look at how they're used:

Getting Available Dictionaries

```
NSMapTable *availableDictionariesKeyedByName =
    [NSMapTable mapTableWithKeyOptions:NSPointerFunctionsCopyIn
        valueOptions:NSPointerFunctionsObjectPointerPersonality];

for (id dictionary in (__bridge_transfer NSArray *)DCSCopyAvailableDictionaries ↩
     ()) {
```

```
    NSString *name = (__bridge NSString *)DCSDictionaryGetName((__bridge  ↩
     DCSDictionaryRef)dictionary);
    [availableDictionariesKeyedByName setObject:dictionary forKey:name];
}
```

Getting Definition for Word

With instances of the elusive `DCSDictionaryRef` type available at our disposal, we can now see what all of the fuss is about with that first argument in `DCSCopyTextDefinition`:

```
NSString *word = @"apple";

for (NSString *name in availableDictionariesKeyedByName) {
    id dictionary = [availableDictionariesKeyedByName objectForKey:name];

    CFRange termRange = DCSGetTermRangeInString((__bridge DCSDictionaryRef) ↩
     dictionary, (__bridge CFStringRef)word, 0);
    if (termRange.location == kCFNotFound) {
        continue;
    }

    NSString *term = [word substringWithRange:NSMakeRange(termRange.location,  ↩
     termRange.length)];

    NSArray *records = (__bridge_transfer NSArray *) ↩
     DCSCopyRecordsForSearchString((__bridge DCSDictionaryRef)dictionary, ( ↩
     __bridge CFStringRef)term, NULL, NULL);
    if (records) {
        for (id record in records) {
            NSString *headword = (__bridge NSString *)DCSRecordGetHeadword(( ↩
     __bridge CFTypeRef)record);
            if (headword) {
                NSString *definition = (__bridge_transfer NSString*) ↩
     DCSCopyTextDefinition((__bridge DCSDictionaryRef)dictionary, (__bridge  ↩
     CFStringRef)headword, CFRangeMake(0, [headword length]));
                NSLog(@"%@: %@", name, definition);

                NSString *HTML = (__bridge_transfer NSString*)DCSRecordCopyData ↩
     ((__bridge DCSDictionaryRef)dictionary, (__bridge CFStringRef)headword,  ↩
     CFRangeMake(0, [headword length]));
                NSLog(@"%@: %@", name, definition);
            }
        }
    }
}
```

Most surprising from this experimentation is the ability to access the raw HTML for entries, which combined with a dictionary's bundled CSS, produces what is shown in Dictionary.app.

iOS

iOS development is a decidedly more by-the-books affair, so attempting to reverse-engineer the platform would be little more than an academic exercise. Fortunately, a good chunk of functionality is available (as of iOS 5) through the obscure UIKit class `UIReferenceLibraryViewController`.

`UIReferenceLibraryViewController` is similar to an `MFMessageComposeViewController`, in that provides a minimally-configurable view controller around system functionality, intended to be presented modally.

Simply initialize with the desired term and present modally:

```
UIReferenceLibraryViewController *referenceLibraryViewController =
    [[UIReferenceLibraryViewController alloc] initWithTerm:@"apple"];
[viewController presentViewController:referenceLibraryViewController
                             animated:YES
                           completion:nil];
```

This is the same behavior that one might encounter by tapping the "Define" `UIMenuItem` on a highlighted word in a `UITextView`.

`UIReferenceLibraryViewController` also provides the class method `dictionaryHasDefinitionForTerm:`. A developer would do well to call this before presenting a dictionary view controller that will inevitably have nothing to display. [61]

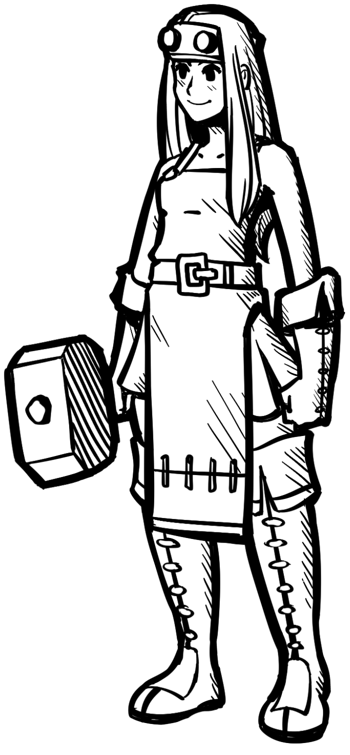
61 In both cases, it appears that `UIReferenceLibraryViewController` will do its best to normalize the search term, so stripping whitespace or changing to lowercase should not be necessary.

```
[UIReferenceLibraryViewController dictionaryHasDefinitionForTerm:@"apple"];
```

From Unix word lists to their evolved `.dictionary` bundles on OS X (and presumably iOS), words are as essential to application programming as mathematical constants and the "Sosumi" alert noise. Consider how the aforementioned APIs can be integrated into your own app, or used to create a kind of app you hadn't previously considered. There are a wealth of linguistics technologies baked into Apple's platforms, so take advantage of them.

Chapter 14

Xcode Toolchain



Though we all come from different backgrounds, with different perspectives that shape our experiences; though we do what we do with various motivations, with beliefs and biases and opinions that sets us apart from one another, there is one thing that brings us together:

We all have to use Xcode. One way or another. For better or worse.

As far as common causes go, honestly, things could be far more desperate. With a few notable exceptions, Xcode seems to get better with each release.

But of course, Xcode is not really just one application. Underneath the GUI lies a confederation of applications and command line tools, which are just as central to a developer's workflow as the editor itself.

Xcode Tools

xcode-select

As if to prove, definitively, that irony is not dead in Cupertino, everyone's journey with Xcode begins with a choice. `xcode-select` offers that choice, albeit one along the lines of the eternal question: "Cake or Death?"

As of Mavericks, getting started as a developer on the Mac takes a single command:

```
xcode-select --install
```

This will install the Command Line Tools, which are necessary for compiling Objective-C code.

xcrun

`xcrun` is the fundamental Xcode command line tool. With it, all other tools are invoked.

```
$ xcrun xcodebuild
```

In addition to running commands, `xcrun` can find binaries and show the path to an SDK:

```
$ xcrun --find clang
$ xcrun --sdk iphoneos --find pngcrush
$ xcrun --sdk macosx --show-sdk-path
```

Because `xcrun` executes in the context of the active Xcode version (as set by `xcode-select`), it is easy to have multiple versions of the Xcode toolchain co-exist on a single system. [62]

Using `xcrun` in scripts and other external tools has the advantage of ensuring consistency across different environments. For example, Xcode ships with a custom distribution of Git. By invoking `$ xcrun git` rather than just `$ git`, a build system can guarantee that the correct distribution is run.

xcodebuild

The second most important Xcode tool is `xcodebuild`, which, as the name implies, builds Xcode projects and workspaces.

Without passing any build settings, `xcodebuild` defaults to the scheme and configuration most recently used by Xcode.app:

```
$ xcodebuild
```

However, everything from scheme, targets, configuration, destination, SDK, and derived data location can be configured:

62 Such as when working with a Developer Preview.

```
$ xcodebuild -workspace NSHipster.xcworkspace \
             -scheme "NSHipster"
```

There are six build actions that can be invoked in sequence:

`build`	Build the target in the build root (`SYMROOT`). This is the default build action.
`analyze`	Build and analyze a target or scheme from the build root (`SYMROOT`). This requires specifying a scheme.
`archive`	Archive a scheme from the build root (`SYMROOT`). This requires specifying a scheme.
`test`	Test a scheme from the build root (`SYMROOT`). This requires specifying a scheme and optionally a destination.
`installsrc`	Copy the source of the project to the source root (`SRCROOT`).
`install`	Build the target and install it into the target's installation directory in the distribution root (`DSTROOT`).
`clean`	Remove build products and intermediate files from the build root (`SYMROOT`).

genstrings

The `genstrings` utility generates a `.strings` file from the specified C or Objective-C source files. A `.strings` file is used for localizing an application in different locales, as described under "Internationalization" in Apple's Cocoa Core Competencies. [63]

```
$ genstrings -a \
             /path/to/source/files/*.m
```

For each use of the `NSLocalizedString` macro in a source file, `genstrings` will append the key and comment into the target file. It's up to the developer to then create a copy of that file for each targeted locale and have that file translated.

fr.lproj/Localizable.strings

```
/* No comment provided by engineer. */
"Username"="nom d'utilisateur";
/* {User First Name}'s Profile */
"%@'s Profile"="profil de %1$@";
```

63 https://developer.apple.com/library/mac/documentation/general/conceptual/devpedia-cocoacore/-Internationalization.html

ibtool

`genstrings` is to source code as `ibtool` is to XIB files.

```
$ ibtool --generate-strings-file \
         Localizable.strings \
         en.lpoj/Interface.xib
```

Although localization is its primary use case, `ibtool` boasts several other features for working with Interface Builder documents. It can change all references to a class name with the `--convert` flag. It can upgrade a document to the latest version, with the `--upgrade` flag. It can even enable Auto Layout, and update frames & constraints with the `--enable-auto-layout`, `--update-frames`, and `--update-constraints` flags, respectively. [64]

iprofiler

`iprofiler` measure an app's performance without launching Instruments.app:

64 One might imagine integrating these commands into a Git pre-commit hook in a large project, to ensure consistency across developers and versions.

```
$ iprofiler -allocations \
            -leaks \
            -T 15s \
            -o perf \
            -a NSHipster
```

The preceding command will attach to the NSHipster app, run it for 15 seconds, instrument allocations and leaks, and then write the results to `perf`. This output can then be read and visualized by Instruments.app later.

xed

This command simply opens Xcode.

```
$ xed NSHipster.xcworkspace
```

By passing the `-w` flag, `xed` will wait until all opened windows are closed. This is useful for scripting user interactions, such as prompting a user to edit a file and continuing once finished.

agvtool

`agvtool` can be used to version Xcode projects, by reading and writing the appropriate values in the `Info.plist` file.

```
$ agvtool what-version
```

…returns the current version of the project.

```
$ agvtool next-version
```

…increments `CURRENT_PROJECT_VERSION` and `DYLIB_CURRENT_VERSION`. Passing the `-all` option will also update the `CFBundleVersion` key in `Info.plist`. [65]

Other Tools

In addition to the aforementioned Xcode tools, there are a score of other executables that can be invoked with `xcrun`:

65 Integrate `agvtool` into a build system to automatically track consecutive builds.

Compilation & Assembly

- `clang`: Compiles C, C`, Objective-C, and Objective-C` source files.
- `lldb`: Debugs C, C`, Objective-C, and Objective-C` programs
- `nasm`: Assembles files.
- `ndisasm`: Disassembles files.
- `symbols`: Displays symbol information about a file or process.
- `strip`: Removes or modifies the symbol table attached to the output of the assembler and link editor.
- `atos`: Converts numeric addresses to symbols of binary images or processes.

Processors

- `unifdef`: Removes conditional `#ifdef` macros from code.
- `ifnames`: Finds conditionals in C++ files.

Libraries

- `ld`: Combines object files and libraries into a single file.

- `otool`: Displays specified parts of object files or libraries.
- `ar`: Creates and maintains library archives.
- `libtool`: Creates a library for use with the link editor, `ld`.
- `ranlib`: Updates the table of contents of archive libraries.
- `mksdk`: Makes and updates SDKs.
- `lorder`: Lists dependencies for object files.

Scripting

- `sdef`: Scripting definition extractor.
- `sdp`: Scripting definition processor.
- `desdp`: Scripting definition generator.
- `amlint`: Checks Automator actions for problems.

Packages

- `installer`: Installs OS X packages.
- `pkgutil`: Reads and manipulates OS X packages.
- `lsbom`: List contents of a bom (Bill of Materials).

Documentation

- `headerdoc`: Processes header documentation.
- `gatherheaderdoc`: Compiles and links `headerdoc` output.
- `headerdoc2html`: Generates HTML from `headerdoc` output.
- `hdxml2manxml`: Translates from `headerdoc` XML output to a file for use with `xml2man`
- `xml2man`: Converts Man Page Generation Language (MPGL) XML files into manual pages.

Core Data

- `momc`: Compiles Managed Object Model (`.mom`) files
- `mapc`: Compiles Core Data Mapping Model (`.xcmappingmodel`) files

Chapter 15

Third-Party Tools

appledoc

There's an adage among Cocoa developers that Objective-C's verbosity lends itself to self-documenting code. Between `longMethodNamesWithNamedParameters:` and the explicit typing of those parameters, Objective-C methods don't leave much to the imagination.

But even self-documenting code can be improved with documentation, and just a small amount of effort can yield significant benefit to others.

In Objective-C, the documentation tool of choice is `appledoc` [66]. Using a Javadoc-like syntax, `appledoc` is able to generate HTML and Xcode-compatible `.docset` docs from `.h` files that look nearly identical Apple's official documentation.

Objective-C documentation is designated by a `/** */` comment block (note the extra initial star) preceding any `@interface` or `@protocol`, as well as any method or `@property` declarations. Documentation may also contain labels for systematic fields, like parameters or return value:

66 http://gentlebytes.com/appledoc/

- `@param [param] [Description]`: Describes what value should be passed or this parameter
- `@return [Description]`: Describes the return value of a method
- `@see [selector]`: Provide *"see also"* reference to related item
- `@discussion [Discussion]`: Provide additional background
- `@warning [description]`: Call out exceptional or potentially dangerous behavior

`appledoc` can be installed by following the latest installation instructions provided on the project page, or with Homebrew [67]:

```
$ brew install appledoc
```

To generate documentation, execute the `appledoc` command within the root directory of an Xcode project, passing metadata such as project and company name:

```
$ appledoc --project-name CFHipsterRef \
           --project-company "NSHipster" \
           --company-id com.nshipster \
           --output ~/Documents \
           .
```

67 http://brew.sh

This will generate and install an Xcode `.docset` file from the documentation in the headers found within the target directory.

Additional configuration options, including HTML output, can be found by passing `--help`:

```
$ appledoc --help
```

xctool

`xctool` is a drop-in replacement for `xcodebuild`, the utility underlying Xcode.app itself.

Every step of the build process is neatly organized and reported in a way that is understandable and visually appealing, with ANSI colorization and a splash of Unicode ornamentation, but `xctool`'s beauty is not just skin-deep: build progress can also be reported in formats that can be read by other tools:

```
$ xctool -reporter plain:output.txt build
```

- `pretty`: (*default*) a text-based reporter that uses ANSI colors and unicode symbols for pretty output.

- `plain`: like `pretty`, but with with no colors or Unicode.
- `phabricator`: outputs a JSON array of build/test results which can be fed into the Phabricator code-review tool.
- `junit`: produces a JUnit / xUnit compatible XML -ile with test results.
- `json-stream`: a stream of build/test events as JSON dictionaries, one -er line (example output).
- `json-compilation-database`: outputs a JSON Compilation Database of build events which can be used by Clang Tooling based tools, e.g. OCLint.

Another improvement over `xcodebuild` is that `xctool` will run application tests in your project in the same way Xcode.app does. [68]

For this reason alone, xctool has great implications for the emerging discipline of continuous integration testing within the Objective-C community.

To install `xctool`, run the following command:

```
$ brew install xctool
```

68 `xcodebuild` can't discern which targets in your scheme are test targets, let alone run them in the simulator

OCLint

OCLint is a static code analyzer that inspects C code for common sources of problems, like empty `if/else/try/catch/finally` statements,unused local variables and parameters, complicated code with high NCSS (Non Commenting Source Statements) or cyclomatic / NPath complexity, redundant code, code smells, and other bad practices.

The best way to install OCLint is by using Homebrew Cask: [69]

```
$ brew cask install oclint
```

Remember xctool's `json-compilation-database` reporter option? Use that with the `oclint-json-compilation-database` helper executable to kick off OCLint:

```
$ xctool -workspace NSHipster.xcworkspace \
         -scheme "NSHipster" \
         -reporter json-compilation-database \
         build > compile_commands.json

$ oclint-json-compilation-database
```

69 http://caskroom.io

xcpretty

`xcpretty` is similar to `xctool` in that it improves on `xcodebuild` build output, but instead of attempting to replace `xcodebuild`, `xcpretty` augments and improves it.

In fact, `xcpretty` exemplifies the Unix philosophy of composability by taking the piped output of `xcodebuild`, rather than instead of being invoked directly:

```
$ xcodebuild [flags] | xcpretty -c
```

One major benefit to this approach is that it's really fast—indeed, in some cases, `xcpretty` is actually a bit faster than invoking `xcodebuild` directly, as it saves on the amount of time spent printing to the console.

Another commonality with `xctool` is the reporters feature, which features formatted output into JUnit-style XML, HTML, or the aforementioned OCTool-compatible json-compilation-database format.

`xcpretty` can be installed using RubyGems [70], which is installed by default on OS X:

```
$ gem install xcpretty
```

Nomad

Nomad is a collection of world-class command-line utilities for iOS and OS X development. It automates the common administrative tasks, so that developers can focus on building and shipping software.

Each tool can be installed individually, or all together with a single command:

```
$ gem install nomad-cli
```

Cupertino [71]

The process of application provisioning is universally loathed by all Apple developers. [72]

70 http://rubygems.org

71 Named after Cupertino, CA: home to Apple, Inc.'s world headquarters.

72 The only thing even *close* to evoking that much vitriol is being prompted by Xcode to enable Snapshots.

Aside from the entire process being a nightmare from start to finish, many of the operations require interacting through a web interface. This not only requires a lot of extra clicking, but makes it very difficult to automate.

Cupertino provides a CLI for managing devices, provisioning profiles, app IDs, and certificates. [73]

```
$ ios devices:list

+------------------------------+---------------------------------------+
|      Listing 2 devices. You can register 98 additional devices.     |
+---------------------------+------------------------------------------+
| Device Name               | Device Identifier                        |
+---------------------------+------------------------------------------+
| Johnny Appleseed iPad     | 0123456789012345678901234567890123abcdef |
| Johnny Appleseed iPhone   | abcdef0123456789012345678901234567890123 |
+---------------------------+------------------------------------------+

$ ios devices:add "iPad 1"=abc123
$ ios devices:add "iPad 2"=def456 "iPad 3"=ghi789 ...
```

Shenzhen [74]

73 In lieu of an actual API for the Apple Provisioning Profile, Cupertino accomplishes this by mechanizing the browser actions and scraping the results. It's a messy business, but it gets the job done.

74 Named for the Chinese city famous for its role as the center of manufacturing for a majority of consumer electronics, including iPhones and iPads.

One thing web developers have on their iOS counterparts is the ability to continuously deploy code within seconds, as opposed to waiting a few days for Cupertino to approve (and sometimes reject!) an update.

Fortunately, a cottage industry has sprung up around development and enterprise distribution. Third party like HockeyApp, DeployGate, and TestFlight [75] offer developers an easier way to sign up test users and send out the latest builds for QA.

Shenzhen is a tool for further automating this process, by building an `.ipa` file and then distributing to an FTP/SFTP server, S3 bucket, or any of the aforementioned third party services.

```
$ cd /path/to/iOS Project/
$ ipa build
$ ipa distribute:sftp --host HOST -u USER -p PASSWORD -P FTP_PATH
```

Houston [76]

Houston is a simple tool for sending Apple Push Notifications. Pass credentials, construct a message, and send it to a device.

75 TestFlight's parent company, Burstly, was acquired by Apple in February of 2014.

76 Named for Houston, TX, the metonymical home of NASA's Johnson Space Center, as in "Houston, We Have Liftoff!".

```
$ apn push "<token>" \
    -c /path/to/apple_push_notification.pem \
    -m "Hello from the command line!"
```

This tool is especially useful for testing remote notifications—especially when implementing the feature in a new app.

Venice [77]

In-app purchases have, for better or worse, become the most profitable business model for app developers. With so much on the line, ensuring the validity of these purchases is paramount to one's livelihood.

Venice is a CLI for verifying Apple In-App Purchase receipts, and retrieving the information associated with receipt data.

```
$ iap verify /path/to/receipt

+-----------------------------+-------------------------------+
|                           Receipt                           |
+-----------------------------+-------------------------------+
| app_item_id                 |                               |
| bid                         | com.foo.bar                   |
| bvrs                        | 20120427                      |
```

77 Venice is named for Venice, Italy—or more specifically, Shakespeare's *The Merchant of Venice*.

```
| original_purchase_date      | Sun, 01 Jan 2013 12:00:00 GMT |
| original_transaction_id     | 1000000000000001              |
| product_id                  | com.example.product           |
| purchase_date               | Sun, 01 Jan 2013 12:00:00 GMT |
| quantity                    | 1                             |
| transaction_id              | 1000000000000001              |
| version_external_identifier |                               |
+-----------------------------+-------------------------------+
```

Like Houston, Venice has a client library component to it, allowing it to be deployed in a Rails or Sinatra application. Verifying receipts on the server allows one to keep their own records of past purchases, which is useful for up-to-the-minute metrics and historical analysis. As such, it is recommended practice for anyone serious about IAPs.

Dubai [78]

Passbook manages boarding passes, movie tickets, retail coupons, & loyalty cards. Using the PassKit API, developers can register web services to automatically update content on the pass, such as gate changes on a boarding pass, or adding credit to a loyalty card.

Dubai makes it easy to generate `.pkpass` file from a script or the command line, allowing one to rapidly iterate on the design and content of your passes, or generate one-offs on the fly.

78 Dubai is named for Dubai, UAE, a center of commerce and trade

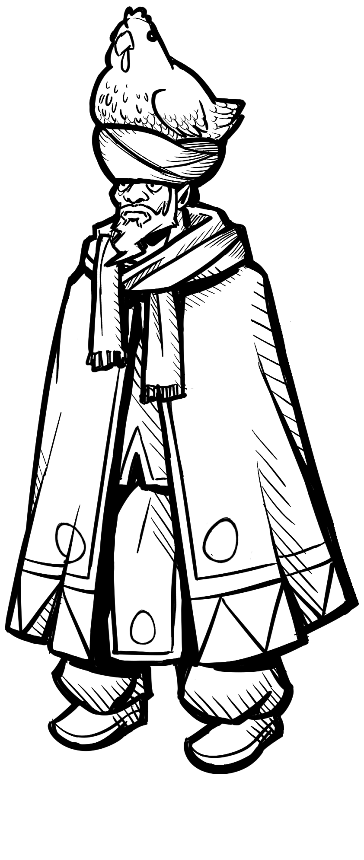
```
$ pk generate Example.pass -T boarding-pass
```

Once a pass is generated, it can be served locally over HTTP with Dubai, allowing for passes to be previewed live in the iOS Simulator:

```
$ pk serve Example.pass -c /path/to/certificate.p12
$ open http://localhost:4567/pass.pkpass
```

Chapter 16

CocoaPods



Civilization is built on infrastructure: roads, bridges, canals, sewers, pipes, wires, fiber. When well thought-out and implemented, infrastructure is a multiplying force that drives growth and development. But when such formative structures are absent or ad hoc, it feels as if progress is made *in spite of* the situation.

It all has to do with solving the problem of scale.

No matter what the medium, whether it's accommodating millions of families into a region, or integrating a large influx of developers into a language ecosystem, the challenges are the same.

In the case of Objective-C, CocoaPods provided a much-needed tool for channeling and organizing open source participation, and served as a rallying point for the community at a time of rapid growth and evolution.

A Look Back

For the first twenty or so years of its existence, Objective-C was not a widely known language—NeXT and later OS X were marginal platforms, with a comparatively small user base and developer community. Like any community, there were local user groups and mailing lists and websites, but open source collaboration was not a widespread phenomenon. Granted, Open Source was only just starting to pick up steam at that time, but there was no contemporary Objective-C equivalent to, for example, CPAN, the Comprehensive Perl Archive Network. Everyone took SDKs from Redwood City and Cupertino as far as they could, (maybe sprinkling in some code salvaged from a forum thread), but ultimately rolling their own solutions to pretty much everything else.

Objective-C and the iPhone

This went on until the summer of 2008, when iPhone OS was first opened up to third party developers. Almost overnight, Objective-C went from being an obscure C++/C# also-ran to the one of the most sought-after programmer qualifications. Millions of developers flocked from all walks of code, bringing an influx of new ideas and influences to the language.

Around this same time, GitHub had just launched, and was starting to change the way we thought about open source by enabling a new distributed, collaborative workflow.

In those early years of iPhone OS, we started to see the first massively adopted open source projects, like ASIHTTPRequest and Facebook's

Three20. These first libraries and frameworks were built to fill in the gaps of app development on iPhone OS 2.0 and 3.0, and although largely made obsolete by subsequent OS releases or other projects, they demonstrated a significant break from the tradition of "every developer for themselves".

Of this new wave of developers, those coming from a Ruby background had a significant influence on the code and culture of Objective-C. Ruby, a spiritual successor to Perl, had its own package manager similar to CPAN: RubyGems.

As open source contributions in Objective-C began to get some traction, the pain points of code distribution were starting to become pretty obvious:

Lacking frameworks, code for iOS could be packaged as a static library, but getting that set up and keeping code and static distributions in sync was an arduous process.

Another approach was to use Git submodules, and include the source directly in the project. But getting everything working, with linked frameworks and build flags configured, was not great either—especially at a time when the body of code was split between ARC and non-ARC.

Enter CocoaPods

CocoaPods was created by Eloy Durán on August 12, 2011.

Taking inspiration from Bundler and RubyGems, CocoaPods was designed to resolve a list of dependencies, download the required sources, and configure the existing project in such a way to be able to use them. Considering the challenges of working with a sparsely documented Xcode project format and build system, it's pretty amazing that this exists at all.

Another notable decision made early on was to use a central Git repository as the database for all of the available libraries. Although there were certain logistical considerations with this approach, bootstrapping on GitHub provided a stable infrastructure, that allowed the team to iterate on building out the tool chain.

Since its initial proof-of-concept, the project has grown to include over a dozen core contributors along with over 100 additional contributors. There are thousands of open source projects available for anyone to add to their project.

A significant portion of these prolific contributions from the open source community for Objective-C has been directly enabled and encouraged by increased ownership around tooling. Everyone involved should be commended for their hard work and dedication.

Using CocoaPods

CocoaPods is easy to get started with both as a consumer and a library author. It should only take a few minutes to get set up.

Installing CocoaPods

CocoaPods is installed through RubyGems, the Ruby package manager, which comes with a standard OS X install.

To install, open Terminal.app and enter the following command:

```
$ sudo gem install cocoapods
```

Now you should have the `pod` command available in the terminal.

Managing Dependencies

A dependency manager resolves a list of software requirements into a list of specific tags to download and integrate into a project.

Declaring requirements in such a way allows for project setup to be automated, which is general best practice for software development practice, no matter what the language. Even if you don't include third-party libraries, CocoaPods is still an invaluable tool for managing code dependencies across projects.

Podfile

A `Podfile` is where the dependencies of a project are listed. It is equivalent to `Gemfile` for Ruby projects using Bundler, or `package.json` for JavaScript projects using npm.

To create a Podfile, `cd` into the directory of your `.xcodeproj` file and enter the command:

```
$ pod init
```

Podfile

```
platform :ios, '7.0'

target "AppName" do

end
```

Dependencies can have varying levels of specificity. For most libraries, pegging to a minor or patch version is the safest and easiest way to include them in your project.

```
pod 'X', '~> 1.1'
```

To include a library not included in the public specs database, a Git, Mercurial, or SVN repository can be used instead, for which a `commit`, `branch`, or `tag` can be specified.

```
pod 'Y', :git => 'https://github.com/NSHipster/Y.git', :commit => 'b4dc0ffee'
```

Once all of the dependencies have been specified, they can be installed with:

```
$ pod install
```

When this is run, CocoaPods will recursively analyze the dependencies of each project, resolving them into a dependency graph, and serializing into a `Podfile.lock` file.

CocoaPods will create a new Xcode project that creates static library targets for each dependency, and then links them all together into a `libPods.a` target. This static library becomes a dependency for your original application target. An `xcworkspace` file is created, and should be used from that point onward. This allows the original `xcodeproj` file to remain unchanged.

Subsequent invocations of `pod install` will add new pods or remove old pods according to the locked dependency graph. To update the individual dependencies of a project to the latest version, do the following:

```
$ pod update
```

Trying Out a CocoaPod

One great, but lesser-known, feature of CocoaPods is the `try` command, which allows you to test-drive a library before you add it to your project.

Invoking `$ pod try` with the name of a project in the public specs database opens up any example projects for the library:

```
$ pod try Ono
```

Creating a CocoaPod

Being the de facto standard for Objective-C software distribution, CocoaPods is pretty much a requirement for open source projects with the intention of being used by others

Yes, it raises the barrier to entry for sharing your work, but the effort is minimal, and more than justifies itself. Taking a couple minutes to create a `.podspec` file saves every user at least that much time attempting to integrate it into their own projects.

Remember: raising the bar for contribution within a software ecosystem lowers the bar for participation.

Specification

A `.podspec` file is the atomic unit of a CocoaPods dependency. It specifies the name, version, license, and source files for a library, along with other metadata.

NSHipsterKit.podspec

```
Pod::Spec.new do |s|
  s.name     = 'NSHipsterKit'
  s.version  = '1.0.0'
  s.license  = 'MIT'
  s.summary  = "A pretty obscure library.
                You've probably never heard of it."
  s.homepage = 'http://nshipster.com'
  s.authors  = { 'Mattt Thompson' =>
                 'mattt@nshipster.com' }
  s.social_media_url = "https://twitter.com/mattt"
  s.source   = { :git => 'https://github.com/nshipster/NSHipsterKit.git', :tag ↩
     => '1.0.0' }
  s.source_files = 'NSHipsterKit'
end
```

Once published to the public specs database, anyone could add it to their project, specifying their Podfile thusly:

Podfile

```
pod 'NSHipsterKit', '~> 1.0'
```

A `.podspec` file can be useful for organizing internal or private dependencies as well:

```
pod 'Z', :path => 'path/to/directory/with/podspec'
```

Publishing a CocoaPod

Although it worked brilliantly at first, the process of using Pull Requests on GitHub for managing new pods became something of a chore, both for library authors and spec organizers. Sometimes podspecs would be submitted without passing `$ pod lint`, causing the specs repo build to break. Other times, rogue commits from people other than the original library author would break things unexpectedly.

The CocoaPods Trunk service, introduced in CocoaPods 0.33, solves a lot of this, making the process nicer for everyone involved. Being a centralized service, it also has the added benefit of being able to get analytics for library usage, and other metrics.

To get started, you must first register your machine with the Trunk service. This is easy enough, just specify your email address (the one you use for committing library code) along with your name.

```
$ pod trunk register mattt@nshipster.com "Mattt Thompson"
```

Now, all it takes to publish your code to CocoaPods is a single command. The same command works for creating a new library or adding a new version to an existing one:

```
$ pod trunk push NAME.podspec
```

A Look Forward

CocoaPods exemplifies the compounding effect of infrastructure on a community. In a few short years, the Objective-C community has turned into something that we can feel proud to be part of.

CocoaPods is a good thing for Objective-C. And it's only getting better.

About NSHipster

NSHipster is a journal of the overlooked bits in Swift, Objective-C, and Cocoa. Updated weekly.

Launched in the Summer of 2012, NSHipster has become an essential resource for iOS and Mac developers around the world.

Colophon

The text is set in Minion Pro, by Robert Slimbach, with code excerpts set in Source Code Pro, by Paul D. Hunt.

The cover is set in Open Sans by Steve Matteson, with illustrations by Conor Heelan, in homage to *Structure and Interpretation of Computer Programs,* by Harold Abelson, Gerald Jay Sussman, and Julie Sussman.

About the Author

Mattt Thompson is the creator & maintainer of AFNetworking and other popular open-source projects, including Postgres.app, ASCIIwwdc, and Nomad.

Previously, Mattt has worked as a software engineer at Panic, Heroku, and Gowalla.

His work has taken him across the United States and around the world, to speak at conferences and meetups about topics in Swift, Objective-C, Ruby, Javascript, web development, design, linguistics, and philosophy.

Mattt holds a Bachelor's degree in Philosophy and Linguistics from Carnegie Mellon University.