NSHipster Obscure Topics in Cocoa and Objective C

Jia

Published with GitBook

Table of Contents

1. Introduction
2. Objective-C
    i. #pragma
    ii. nil / Nil /NULL / NSNull
    iii. BOOL / bool /Boolean / NSCFBoolean
    iv. Equality
    v. Type Encodings
    vi. C Storage Classes
    vii. at
    viii. __attribute__
    ix. NS_ENUM & NS_OPTIONS
3. Foundation & CoreFoundation
    i. Key-Value CodingCollection Operators
    ii. NSError
    iii. NSOperation
    iv. NSSortDescriptor
    v. NSPredicate
    vi. NSExpression
    vii. NSFileManager
    viii. NSValue
    ix. NSDataDetector
    x. NSCache
    xi. NSIndexSet
    xii. NSOrderedSet
    xiii. NSHashTable & NSMapTable
4. UIKit
    i. UIAppearance
5. Localization, Internationalization & Accessibility
    i. NSLocalizedString
    ii. CFStringTransform
    iii. NSLinguisticTagger

My Book

Welcome in my book!

#pragma

1.Organizing Your Code

Use #pragma mark in your @implementation to divide code into logical sections. Not only do these sections make it easier to read through the code itself, but it also adds visual cues to the Xcode source navigator.

#pragma mark declarations starting with a dash (-) are preceded with a horizontal divider.

2.Inhibiting Warnings

Especially 3rd-party code. There are few things as irksome as that one vendor library that takes forever to compile, and finishes with 200+ warnings. Even shipping code with a single warning is in poor form.

Try setting the -Weverything flag and checking the "Treat Warnings as Errors" box your build settings. This turns on Hard Mode in Xcode.

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wunused-variable"
    OSStatus status = SecItemExport(...);
    NSCAssert(status == errSecSuccess, @"%d", status);
#pragma clang diagnostic pop
```

You can read more about the LLVM's use of #pragma in the Clang Compiler User's Manual.

nil / Nil /NULL / NSNull

Symbol	Value	Meaning
NULL	(void *)0	literal null value for C pointers
nil	(id)0	literal null value for Objective-C objects
Nil	(Class)0	literal null value for Objective-C classes
NSNull	[NSNull null]	singleton object used to represent null

NULL

C represents nothing as 0 for primitive values, and NULL for pointers (which is equivalent to 0 in a pointer context).

nil

Objective-C builds on C's representation of nothing by adding nil. nil is an object pointer to nothing. Although semantically distinct from NULL, they are equivalent to one another.

NSNull

On the framework level, **Foundation defines the NSNull class, which defines a single class method, +null, which returns a singleton NSNull instance.** NSNull is different from nil or NULL, in that it is an actual object, rather than a zero value.

Nil

Additionally, in Foundation/NSObjCRuntime.h, Nil is defined as a class pointer to nothing. This lesser-known title- case cousin of nil doesn't show up much very often, but it's at least worth noting.

BOOL / bool /Boolean / NSCFBoolean

Name	Type	Header	True	False
BOOL	signed char / bool	objc.h	YES	NO
bool	_Bool (int)	stdbool.h	TRUE	FALSE
Boolean	unsigned char	MacTypes.h	TRUE	FALSE
NSNumber	__NSCFBoolean	Foundation.h	@(YES)	@(NO)

BOOL

Objective-C defines BOOL to encode truth value. ***It is a typedef of a signed char,*** with the macros YES and NO to represent true and false, respectively.

Boolean

Boolean values are used in conditionals, such as if or while statements, to conditionally perform logic or repeat execution. When evaluating a conditional statement,***the value 0 is considered "false", while any other value is considered "true".*** Because NULL and nil have a value of 0, they are considered "false" as well.

In Objective-C, use the BOOL type for parameters, properties, and instance variables dealing with truth values. When assigning literal values, use the YES and NO macros.

The Wrong Answer to the Wrong Question

1. wrong

```
if ([a isEqual:b] == YES) { ... }

static BOOL different (int a, int b)
{
    return a - b;
}

different(11, 10) == YES// YES
different(10, 11) == YES// NO (!)
different(512, 257) == YES// NO (!)
```

   However, **because BOOL is typedef 'd as a signed char on 32- bit architectures**, this will not behave as expected:

   On a 64-bit iOS, BOOL is defined as a bool, rather than signed char, which precludes the runtime from these type conversion errors.

2. right

```
if ([a isEqual:b]) { ... }

static BOOL different (int a, int b)
{
    return a != b;
}

different(11, 10) == YES// YES
different(10, 11) == YES// YES (!)
```

```
different(512, 256) == YES // YES (!)
```

Deriving truth value directly from an arithmetic operation is never a good idea. Use the `==` operator, or cast values into booleans with the `! (or !!)` operator.

The Truth About NSNumber and BOOL

```
NSLog(@"%@", [@(YES) class]);

__NSCFBoolean
```

All this time, we've been led to believe that NSNumber boxes primitives into an object representation. Any other integer or float derived NSNumber object shows its class to be __NSCFNumber. What gives?

NSCFBoolean is a private class in the NSNumber class cluster. It is a bridge to the CFBooleanRef type, which is used to wrap boolean values for Core Foundation collections and property lists. CFBoolean defines the constants kCFBooleanTrue and kCFBooleanFalse. Because CFNumberRef and CFBooleanRef are different types in Core Foundation, it makes sense that they are represented by different bridging classes in NSNumber.

Equality

Equality & Identity

1. In code, an object's identity is tied to its memory address.

2. Two objects may be equal or equivalent to one another, if they share a common set of properties. Yet, those two objects may still be thought to be distinct, each with their own identity.

isEqualTo__:

NSAttributedString	-isEqualToAttributedString:
NSData	-isEqualToData:
NSDate	-isEqualToDate:
NSDictionary	-isEqualToDictionary:
NSHashTable	-isEqualToHashTable:
NSIndexSet	-isEqualToIndexSet:
NSNumber	-isEqualToNumber:
NSOrderedSet	-isEqualToOrderedSet:
NSSet	-isEqualToSet:
NSString	-isEqualToString:
NSTimeZone	-isEqualToTimeZone:
NSValue	-isEqualToValue:

The Curious Case of NSString Equality

```
NSString *a = @"Hello";
NSString *b = @"Hello";
BOOL wtf = (a == b); // YES (!)

NSString *a = @"Hello";
NSString *b = [NSString stringWithFormat:@"%@", @"Hello"];
BOOL wtf = (a == b); // NO (!)
```

To be perfectly clear: the correct way to compare two NSString objects is -isEqualToString:. Under no circumstances should NSString objects be compared with the == operator.

So what's going on here? Why does this work, when the same code for NSArray or NSDictionary literals wouldn't do this?

It all has to do with an optimization technique known as string interning, whereby one copy of immutable string values. NSString *a and* b point to the same copy of the interned string value @"Hello".

Again, this only works for statically-defined, immutable strings. Constructing identical strings with NSString +stringWithFormat: will objects with different pointers.

Interestingly enough, Objective-C selector names are also stored as interned strings in a shared string pool.

Hashing Fundamentals

1. Object equality is commutative

   ([a isEqual:b] ⇒ [b isEqual:a])

2. If objects are equal, their hash values must also be equal

   ([a isEqual:b] ⇒ [a hash] == [b hash])

3. However, the converse does not hold: two objects need not be equal in order for their hash values to be equal

   ([a hash] == [b hash] ¬⇒ [a isEqual:b])

Implementing -isEqual: and hash in a Subclass

Type Encodings

Type Encodings.

Code	Meaning
c	A char
i	An int
s	A short
l	A longl is treated as a 32-bit quantity on 64-bit programs.
q	A long long
C	An unsigned char
i	An unsigned int
S	An unsigned short
L	An unsigned long
Q	An unsigned long long
f	A float
d	A double
B	A C++ bool or a C99 _Bool
v	A void
*	A character string (char *)
@	An object (whether statically typed or typed id)
#	A class object (Class)
:	A method selector (SEL)
[array type]	An array
{name=type...}	A structure
(name=type...)	A union
bnum	A bit field of num bits
^type	A pointer to type
?	An unknown type (among other things, this code is used for function pointers)

```
NSLog(@"int : %s", @encode(int));
NSLog(@"float: %s", @encode(float));
NSLog(@"float *: %s", @encode(float*));
NSLog(@"char: %s", @encode(char));
NSLog(@"char *: %s", @encode(char *));
NSLog(@"BOOL: %s", @encode(BOOL));
NSLog(@"void: %s", @encode(void));
NSLog(@"void *: %s", @encode(void *));

NSLog(@"NSObject * : %s", @encode(NSObject *));
NSLog(@"NSObject : %s", @encode(NSObject));
NSLog(@"[NSObject] : %s", @encode(typeof([NSObject class])));
NSLog(@"NSError ** : %s", @encode(typeof(NSError **)));
```

```
int intArray[5] = {1, 2, 3, 4, 5};
NSLog(@"int[] : %s", @encode(typeof(intArray))); !

float floatArray[3] = {0.1f, 0.2f, 0.3f};
NSLog(@"float[] : %s", @encode(typeof(floatArray))); !

typedef struct _struct
{
    short a;
    long long b;
    unsigned long long c;
} Struct;

NSLog(@"struct     : %s", @encode(typeof(Struct)));
```

```
int :i
float :f
float * :^f
char :c
char * :*
BOOL :c
void :v
void * :^v

NSObject * : @
NSObject : #
[NSObject] : {NSObject=#}
NSError ** : ^@
int[]: [5i]
float[]: [3f]
struct: {_struct=sqQ}
```

There are some interesting takeaways from this:

1. Whereas the standard encoding for pointers is a preceding ^, char *gets its own code:* . This makes sense conceptually, since C strings are thought to be entities in and of themselves, rather than a pointer to something else.

2. **BOOL is c, rather than i, as one might expect. Reason being, char is smaller than an int, and when Objective-C was originally designed in the 80's,** bits (much like the US Dollar) were more valuable than they are today.

3. **Passing NSObject directly yields #. However, passing [NSObject class] yields a struct named NSObject with a single class field. That is, of course, the isa field, which all NSObject instances have to signify their type.**

Method Encodings

As mentioned in Apple's *"Objective-C Runtime Programming Guide"*, there are a handful of type encodings that are used internally, but cannot be returned with @encode.

These are the type qualifiers for methods declared in a protocol:

Code	Meaning
r	const
n	in
N	inout
o	out
O	bycopy
R	byref
V	oneway

C Storage Classes

auto

There's a good chance you've never seen this keyword in the wild. **That's because auto is the default storage class, and therefore rarely needs to be specified explicitly.**

Automatic variables have memory automatically allocated when a program enters a block, and released when the program leaves that block. Access to automatic variables is limited to only the block in which they are declared, as well as any nested blocks.

register

Most Objective-C programmers probably aren't familiar with register either, as it's not widely used in the NS world.

register behaves just like auto, except that instead of being allocated onto the stack, they are stored in a register.

Registers offer faster access than RAM, but because of the complexities of memory management, putting variables in registers does not always guarantee a faster program—in fact, it may very well end up slowing down execution by taking up space on the register unnecessarily. As it were, using register is no more than a mere suggestion to the compiler; implementations may choose whether or not to honor this.

register's lack of popularity in Objective-C is instructive: it's probably best not to bother with. It will sooner cause a headache than any noticeable speedup.

static

extern

@

Instance Variable Visibility

1. @public: instance variable can be read and written to directly, using the notation person->age = 32"
2. @package: instance variable is public, except outside of the framework in which it is specified (64-bit architectures only)
3. @protected: instance variable is only accessible to its class and derived classes
4. @private: instance variable is only accessible to its class

Exception Handling

```
@try
{
  // attempt to execute the following statements
    ![self getValue:&value error:&error];
  // if an exception is raised, or explicitly thrown...
  if (error)
  {
    @throw exception;
  }
}
@catch(NSException *e)
{
    // ...handle the exception here
}
@finally
{
  // always execute this at the end of either the @try
or @catch block
    [self cleanup];
}
```

Objective-C Literals

1. @selector(): Returns an SEL pointer to a selector with the specified name. Used in methods like **-performSelector:withObject:.**
2. @protocol(): Returns a Protocol *pointer to the protocol with the specified name. Used in methods like *-conformsToProtocol:.*

C Literals

1. @encode(): Returns the type encoding of a type. This type value can be used as the first argument encode in NSCoder -encodeValueOfObjCType:at.
2. @defs(): Returns the layout of an Objective-C class. For example, to declare a struct with the same fields as an NSObject, you would simply do:

   ```
   struct
   {
     @defs(NSObject)
   }
   ```

@defs is unavailable in the modern Objective-C runtime.

Optimizations

1. @autoreleasepool{}: If your code contains a tight loop that creates lots of temporary objects, you can use the @autorelease directive to optimize by being more aggressive about how for these short-lived, locally-scoped objects are deallocated. @autoreleasepool replaces and improves upon the old NSAutoreleasePool, which is significantly slower, and unavailable with ARC.

2. @synchronized(){}: This directive offers a convenient way to guarantee safe execution of a particular code block within a specified context (usually self). Locking in this way is expensive, however, so for classes aiming for a particular level of thread safety, a dedicated NSLock property or the use of low-level locking functions like OSAtomicCompareAndSwap32(3) are recommended.

Compatibility

1. @compatibility_alias: **Allows existing classes to be aliased by a different name.**

   @compatibility_alias can be used to significantly improve the experience of working with classes across major OS versions, allowing developers to back-port their own custom implementations of new functionality, without changing how that class is used in the app.

attribute

attribute is a compiler directive that specifies characteristics on declarations. They allow the compiler to perform advanced optimizations and enable new kinds of warnings for the analyzer.

The syntax for this keyword is **attribute** followed by two sets of parentheses. **attribute directives are placed after function, variable, and type declarations. Inside the parentheses is a comma-delimited list of attributes.**

GCC

1. **format**

   The format attribute specifies that a function takes format arguments in the style of printf, scanf, strftime or strfmon.

   ```
   extern int my_printf (void *my_object, const char *my_format, ...)  __attribute__((format(printf, 2, 3)));
   ```

2. **nonnull**

   The nonnull attribute specifies that some function parameters should be non-null pointers.

   ```
   extern void * my_memcpy (void *dest, const void *src, size_t len)  __attribute__((nonnull (1, 2)));
   ```

   Using nonnull codifies expectations about values into an explicit contract, which can help catch any NULL pointer bugs lurking in any calling code.

3. **noreturn**

   ```
   static BOOL different (NSString *a, NSString *b)   __attribute__((noreturn))
   ```

4. **pure / const**

   i. The pure attribute specifies that a function has no effects except the return value. That is to say, the return value of a pure function depends only on the parameters and/or global variables.

   ii. The const attribute specifies that a function does not examine any values except their arguments, and have no side-effects except the return value. Note that a function with pointer arguments or calls a non-const function usually should not be const.

   ```
   int square(int n) __attribute__((const));
   ```

5. **unused**

   This attribute, when attached to a function, denotes that the function is not meant to be used. GCC will not produce a warning for this function.

   The same effect can be accomplished with the __unused keyword.

LLVM

1. availability

   Clang introduces the availability attribute, which can be placed on declarations to availability for particular operating system versions. Consider the function declaration for a hypothetical function f:

   ```
   void f(void) __attribute__((availability(macosx, introduced=10.4,deprecated=10.6,obsoleted=10.7)));
   ```

   This information is used by Clang to determine when it is safe to use f. If Clang is instructed to compile code for Mac OS X 10.5, a call to f() succeeds. If Clang is instructed to compile code for Mac OS X 10.6, the call succeeds but Clang emits a warning specifying that the function is deprecated. Finally, if Clang is instructed to compile code for Mac OS X 10.7, the call fails because f() is no longer available.

   - introduced: The first version in which this declaration was introduced.
   - deprecated: The first version in which this declaration was deprecated, meaning that users should migrate away from this API.
   - obsoleted: The first version in which this declaration was obsoleted, meaning that it was removed completely and can no longer be used.
   - unavailable: This declaration is never available on this platform.
   - message: Additional message text that Clang will provide when emitting a warning or error about use of a deprecated or obsoleted declaration. Useful for directing users to replacement APIs.

2. Supported Platforms

   - ios: Apple's iOS operating system. The minimum deployment target is specified by the -mios-version- min=*version* or -miphoneos-version-min=*version* command-line arguments.
   - macosx: Apple's Mac OS X operating system. The minimum deployment target is specified by the - mmacosx-version-min=*version* command-line argument.

3. overloadable

   Clang provides support for C++ function overloading in C with the overloadable attribute.

   ```
   #include <math.h>
   float __attribute__((overloadable)) tgsin(float x)
   {
       return sinf(x);
   }

   double __attribute__((overloadable)) tgsin(double x)
   {
       return sin(x);
   }

   long double __attribute__((overloadable)) tgsin(long double x)
   {
       return sinl(x);
   }
   ```

NS_ENUM & NS_OPTIONS

Apple had previously defined all of their enum types as:

```
typedef enum
{
    UITableViewCellStyleDefault,
    UITableViewCellStyleValue1,
    UITableViewCellStyleValue2,
    UITableViewCellStyleSubtitle
};

typedef NSInteger UITableViewCellStyle;
```

In the case of the preceding enumeration, this would simply need to be a char (1 byte)

NS_ENUM

```
typedef NS_ENUM(NSInteger, UITableViewCellStyle)
{
    UITableViewCellStyleDefault,
    UITableViewCellStyleValue1,
    UITableViewCellStyleValue2,
    UITableViewCellStyleSubtitle
};
```

The first argument for NS_ENUM is the storage type of the new type. In a 64-bit environment, UITableViewCellStyle will be 8 bytes long—same as NSInteger. If the specified size cannot fit all of the defined values, an error will be generated by the compiler. The second argument is the name of the new type. Inside the code block, the values are defined as usual.

NS_OPTIONS

```
typedef NS_OPTIONS(NSUInteger, UIViewAutoresizing)
{
    UIViewAutoresizingNone                 = 0,
    UIViewAutoresizingFlexibleLeftMargin   = 1 << 0,
    UIViewAutoresizingFlexibleWidth        = 1 << 1,
    UIViewAutoresizingFlexibleRightMargin  = 1 << 2,
    UIViewAutoresizingFlexibleTopMargin    = 1 << 3,
    UIViewAutoresizingFlexibleHeight       = 1 << 4,
    UIViewAutoresizingFlexibleBottomMargin = 1 << 5
};
```

The syntax is exactly the same as NS_ENUM, but NS_OPTIONS alerts the compiler that values can be combined with bitmask |.

Key-Value CodingCollection Operators

```
double totalSalary = 0.0;
for (Employee *employee in employees)
{
  totalSalary += [employee.salary doubleValue];
}
double averageSalary = totalSalary / [employees count];
```

Fortunately, Key-Value Coding provides a much more concise —almost Ruby-like—way to do this:

```
[employees valueForKeyPath:@"@avg.salary"];
```

Summary

Collection Operators fall into one of three different categories, according to the kind of value they return:

- Simple Collection Operators return strings, numbers, or dates, depending on the operator.
- Object Operators return an array.
- Array and Set Operators return an array or set, depending on the operator.

Simple Collection Operators

1. **@count**: Returns the number of objects in the collection.
2. **@sum**: Converts each object in the collection to a double, computes the sum, and returns the sum.
3. **@avg**: Takes the double value of each object in the collection, and returns the average value.
4. **@max**: Determines the maximum value using compare:. Objects must support mutual comparison for this to work.
5. **@min**: Same as @max, but returns the minimum value.

```
@interface Product : NSObject
@property NSString *name;
@property double price;
@property NSDate *launchedOn;
@end

[products valueForKeyPath:@"@count"]; // 4
[products valueForKeyPath:@"@sum.price"]; // 3526.00
[products valueForKeyPath:@"@avg.price"]; // 881.50
[products valueForKeyPath:@"@max.price"]; // 1699.00
[products valueForKeyPath:@"@min.launchedOn"]; //June 11, 2012
```

To get the aggregate value of an array or set of NSNumbers, simply pass self as the key path after the operator, e.g. **[@[@(1), @(2), @(3)] valueForKeyPath:@"@max.self "]**

Object Operators

1. **@unionOfObjects** / **@distinctUnionOfObjects**: Returns an array of the objects in the property specified in the key

path to the right of the operator. @distinctUnionOfObjects removes duplicates, whereas @unionOfObjects does not.

```
NSArray *inventory = @[iPhone5, iPhone5, iPhone5, iPadMini, macBookPro, macBookPro];

[inventory valueForKeyPath:@"@unionOfObjects.name"];
// "iPhone 5", "iPhone 5", "iPhone 5", "iPad mini", "MacBook Pro", "MacBook Pro"

[inventory valueForKeyPath:@"@distinctUnionOfObjects.name"];
// "iPhone 5", "iPad mini", "MacBook Pro"
```

Array and Set Operators

- **@distinctUnionOfArrays / @unionOfArrays**: Returns an array containing the combined values of each array in the collection, as specified by the key path to the right of the operator. The distinct version removes duplicate values.
- **@distinctUnionOfSets**: Similar to @distinctUnionOfArrays, but it expects an NSSet containing NSSet objects, and returns an NSSet. Because sets can't contain duplicate values anyway, there is only the distinct operator.

```
telecomStoreInventory = @[iPhone5, iPhone5, iPadMini];

[@[appleStoreInventory, telecomStoreInventory] valueForKeyPath:@"@distinctUnionOfArrays.name"];
// "iPhone 5", "MacBook Pro"
```

NSError

Introduce

NSError is the unsung hero of the Foundation framework. Passed gallantly in and out of perilous method calls, it is the messenger by which we are able to contextualize our failures.

NSError is toll-free bridged with CFError, but it's unlikely to find a reason to dip down to its Core Foundation counterpart.

1. code & domain

   These status codes are defined within a particular error domain, in order to avoid overlap and confusion. These status codes are generally defined by constants in an enum.

2. userInfo

   What gives NSError its particular charm is everyone's favorite grab bag property: userInfo. As a convention throughout Cocoa, userInfo is a dictionary that contains arbitrary key- value pairs that, whether for reasons of subclassing or schematic sparsity, are not suited to full-fledged properties.

   Here's how to construct NSError with a userInfo dictionary:

```
NSDictionary *userInfo =
@{
    NSLocalizedDescriptionKey: NSLocalizedString(@"Operation was unsuccessful.", nil),
    NSLocalizedFailureReasonErrorKey: NSLocalizedString( @"The operation timed out.", nil),
    NSLocalizedRecoverySuggestionErrorKey: NSLocalizedString(@"Have you tried turning it off and on
    again?", nil)
};

NSError *error = [NSError errorWithDomain:NSHipsterErrorDomain code:-57 userInfo:userInfo];
```

   For sake of completeness: here is a list of the standard NSError userInfo keys:

   - NSLocalizedDescriptionKey
   - NSLocalizedFailureReasonErrorKey
   - NSLocalizedRecoverySuggestionErrorKey
   - NSLocalizedRecoveryOptionsErrorKey
   - NSFilePathErrorKey
   - NSStringEncodingErrorKey
   - NSUnderlyingErrorKey
   - NSRecoveryAttempterErrorKey
   - NSHelpAnchorErrorKey

Using NSError

- Consuming

```
NSError *error = nil;
BOOL success = [[NSFileManager defaultManager]
    moveItemAtPath:@"/path/to/target"
            toPath:@"/path/to/destination"
             error:&error];
if (!success)
```

```
{
    NSLog(@"%@", error);
}
```

According to Cocoa conventions, methods returning BOOL to indicate success or failure are encouraged to have a final NSError ** parameter if there are multiple failure conditions to distinguish.

A good guideline is whether you could imagine that NSError bubbling up, and being presented to the user.

- Producing

```
- (BOOL)validateObject:(id)object error:(NSError *__autoreleasing *)error
{
    BOOL success = // ...
    if (!success & error)
    {
          *error = [NSError errorWithDomain:NSHipsterErrorDomain code:-42 userInfo:nil];

        return; NO
      }
    return YES;
}
```

NSOperation

NSSortDescriptor

```
NSMutableArray *people = [NSMutableArray array];

[firstNames enumerateObjectsUsingBlock:
^(id obj, NSUInteger idx, BOOL *stop)
{
    Person *person = [[Person alloc] init];
    person.firstName = [firstNames objectAtIndex:idx];
    person.lastName = [lastNames objectAtIndex:idx];
    person.age = [ages objectAtIndex:idx];
    [people addObject:person];
}];

NSSortDescriptor *ageSortDescriptor = [NSSortDescriptor sortDescriptorWithKey:@"age" ascending:NO];

NSLog(@"By age: %@", [people
sortedArrayUsingDescriptors:@[ageSortDescriptor]]);
```

NSPredicate

```
NSMutableArray *people = [NSMutableArray array];

[firstNames enumerateObjectsUsingBlock:^(id obj,
NSUInteger idx, BOOL *stop) {
    Person *person = [[Person alloc] init];
    person.firstName = firstNames[idx];
    person.lastName = lastNames[idx];
    person.age = ages[idx];
    [people addObject:person];
}];

NSPredicate *bobPredicate = [NSPredicate predicateWithFormat:@"firstName = 'Bob'"];

NSPredicate *smithPredicate = [NSPredicate predicateWithFormat:@"lastName = %@", @"Smith"];

NSPredicate *thirtiesPredicate = [NSPredicate predicateWithFormat:@"age >= 30"];
```

1. Using NSPredicate with Collections
2. Using NSPredicate with Core Data

Predicate Syntax

1. **Substitutions**
   - %@ is a var arg substitution for an object value—often a string, number, or date.
   - %K is a var arg substitution for a key path.

     ```
     NSPredicate *ageIs33Predicate = [NSPredicate !predicateWithFormat:@"%K = %@", @"age", @33];
     ```

   - $VARIABLE_NAME is a value that can be substituted with NSPredicate -predicateWithSubstitutionVariables:.

     ```
     NSPredicate *namesBeginningWithLetterPredicate = [NSPredicate predicateWithFormat:@"(firstName BEGINSWITH[cd

     NSLog(@"'A' Names: %@",
       [people filteredArrayUsingPredicate:
         [namesBeginningWithLetterPredicate
           predicateWithSubstitutionVariables:
             @{@"letter": @"A"}
     ]]);
     ```

2. **Basic Comparisons**
   - =, ==: The left-hand expression is equal to the right-hand expression.
   - . >=, =>: The left-hand expression is greater than or equal to the right-hand expression.
   - <=, =<: The left-hand expression is less than or equal to the right-hand expression.
   - . >: The left-hand expression is greater than the right-hand expression.
   - <: The left-hand expression is less than the right-hand expression.

- !=, <>: The left-hand expression is not equal to the right- hand expression.
- BETWEEN: The left-hand expression is between, or equal to either of, the values specified in the right-hand side. The right-hand side is a two value array (an array is required to specify order) giving upper and lower bounds. For example, 1 BETWEEN { 0 , 33 }, or $INPUT BETWEEN { $LOWER, $UPPER }.

3. **Basic Compound Predicates**
   - AND,&&:LogicalAND.
   - OR, ||: Logical OR.
   - NOT, !: Logical NOT.
4. **String Comparisons**
   - BEGINSWITH: The left-hand expression begins with the right-hand expression.
   - CONTAINS: The left-hand expression contains the right-hand expression.
   - ENDSWITH: The left-hand expression ends with the right-hand expression.
   - LIKE: The left hand expression equals the right-hand expression: ? and *are allowed as wildcard characters, where ? matches 1 character and* matches 0 or more characters.
   - MATCHES: The left hand expression equals the right hand expression using a regex-style comparison according to ICU v3 (for more details see the ICU User Guide for Regular Expressions).
5. **Relational Operations**
   - ANY, SOME: Specifies any of the elements in the following expression. For example, ANY children.age < 18.
   - ALL: Specifies all of the elements in the following expression. For example, ALL children.age < 18.
   - NONE: Specifies none of the elements in the following expression. For example, NONE children.age < 18. This is logically equivalent to NOT (ANY ...).
   - IN: Equivalent to an SQL IN operation, the left-hand side must appear in the collection specified by the right-hand side. For example, name IN { 'Ben', 'Melissa', 'Nick' }.

NSCompoundPredicate

AND & OR can be used in predicate format strings to create compound predicates. However, the same can be accomplished using an NSCompoundPredicate.

For example, the following predicates are equivalent:

```
[NSCompoundPredicate
  andPredicateWithSubpredicates:@[
    [NSPredicate predicateWithFormat:@"age > 25"],
    [NSPredicate predicateWithFormat:
        @"firstName = %@", @"Quentin"]
  ]
];

[NSPredicate predicateWithFormat:@"(age > 25) AND
  (firstName = %@)", @"Quentin"];
```

NSComparisonPredicate

Block Predicates

```
NSPredicate *shortNamePredicate = [NSPredicate predicateWithBlock:
^BOOL(id evaluatedObject, NSDictionary *bindings)
{
    return [[evaluatedObject firstName] length] <= 5;
}];

NSLog(@"Short Names: %@", [people filteredArrayUsingPredicate:shortNamePredicate]);
// ["Alice Smith", "Bob Jones"]
```

Since blocks can encapsulate any kind of calculation, there is a whole class of queries that can't be expressed with the NSPredicate format string (such as evaluating against values dynamically calculated at run-time).

NSPredicates created with predicateWithBlock: cannot be used for Core Data fetch requests backed by a SQLite store.

NSExpression

Evaluating Math

```
NSExpression *expression = [NSExpression expressionWithFormat:@"4 + 5 - 2**3"];

id value = [expression expressionValueWithObject:nil
// 1
```

Functions

1. Statistics
    - average:
    - sum:
    - count:
    - min:
    - max:
    - median:
    - mode:
    - stddev:
2. Basic Arithmetic
    - add:to:
    - from:subtract:
    - multiply:by:
    - divide:by:
    - modulus:by:
    - abs:
3. Advanced Arithmetic
    - sqrt:
    - log:
    - ln:
    - raise:toPower:
    - exp:
4. Bounding Functions
    - ceiling: - the smallest integral value not less than the value in the array
    - trunc: - the integral value nearest to but no greater than the value in the array
5. Functions Shadowing math.h Functions
    - floor:
6. Random Functions
    - random
    - random:
7. Binary Arithmetic
    - bitwiseAnd:with:

    - bitwiseOr:with:
    - bitwiseXor:with:
    - leftshift:by:
    - rightshift:by:
    - onesComplement:
8. Date Functions

    - now
9. String Functions

    - lowercase:
    - uppercase:

Custom Functions

First, define the corresponding method in a category:

```
@interface NSNumber (Factorial)
- (NSNumber *)factorial;
@end

@implementation NSNumber (Factorial)
- (NSNumber *)factorial
{
    return @(tgamma([self doubleValue] + 1));
}
@end
```

Then, use the function thusly:

```
NSExpression *expression = [NSExpression expressionWithFormat:@"FUNCTION(4.2, 'factorial')"];

id value = [expression expressionValueWithObject:nil context:nil];
// 32.578...
```

NSFileManager

Determining If A File Exists

```
NSFileManager *fileManager = ![NSFileManager defaultManager];

NSString *documentsPath =
  [NSSearchPathForDirectoriesInDomains(
NSDocumentDirectory, NSUserDomainMask, YES) ! firstObject];

NSString *filePath = [documentsPath  stringByAppendingPathComponent:@"file.txt"];

BOOL fileExists = [fileManager
  fileExistsAtPath:filePath];
```

Listing All Files In A Directory

```
NSFileManager *fileManager =
  [NSFileManager defaultManager];
NSURL *bundleURL = [[NSBundle mainBundle] bundleURL];
NSArray *contents =
  [fileManager contentsOfDirectoryAtURL:bundleURL
            includingPropertiesForKeys:@[]
          options:NSDirectoryEnumerationSkipsHiddenFiles
                                  error:nil];
NSPredicate *predicate =
  [NSPredicate predicateWithFormat:
    @"pathExtension ENDSWITH '.png'"];
for (NSString *path in
      [contents filteredArrayUsingPredicate:predicate])
{
    // Enumerate each .png file in directory
}
```

Recursively Enumerating Files In A Directory

```
NSFileManager *fileManager =
  [NSFileManager defaultManager];
NSURL *bundleURL = [[NSBundle mainBundle] bundleURL];
NSDirectoryEnumerator *enumerator =
  [fileManager enumeratorAtURL:bundleURL
    includingPropertiesForKeys:@[NSURLNameKey,
                                 NSURLIsDirectoryKey]
          options:NSDirectoryEnumerationSkipsHiddenFiles
     errorHandler:^BOOL(NSURL *url, NSError *error)
{
    NSLog(@"[Error] %@ (%@)", error, url);
}];

NSMutableArray *mutableFileURLs = ![NSMutableArray array];
for (NSURL *fileURL in enumerator)
{
    NSString *filename;
    [fileURL getResourceValue:&filename forKey:NSURLNameKey error:nil];
NSNumber *isDirectory;
    [fileURL getResourceValue:&isDirectory
    forKey:NSURLIsDirectoryKey
     error:nil];
    // Skip directories with '_' prefix, for example
    if ([isDirectory boolValue] &&
        [filename hasPrefix:@"_"])
    {
        [enumerator skipDescendants];
        continue;
```

```
    }

    if (![isDirectory boolValue])
    {
        [mutableFileURLs addObject:fileURL];
    }
}
```

Creating a Directory

```
NSFileManager *fileManager =
  [NSFileManager defaultManager];
NSString *documentsPath =
  [NSSearchPathForDirectoriesInDomains(
      NSDocumentDirectory, NSUserDomainMask, YES)
   firstObject];
NSString *imagesPath = [documentsPath
   stringByAppendingPathComponent:@"images"];
if (![fileManager fileExistsAtPath:imagesPath])
{
    [fileManager createDirectoryAtPath:imagesPath
           withIntermediateDirectories:NO
}
attributes:nil
     error:nil];
```

Deleting a File

```
NSFileManager *fileManager =
  [NSFileManager defaultManager];
NSString *documentsPath =
  [NSSearchPathForDirectoriesInDomains(
NSDocumentDirectory, NSUserDomainMask, YES)  firstObject];

NSString *filePath = [documentsPath stringByAppendingPathComponent:@"image.png"];
NSError *error = nil;
if (![fileManager removeItemAtPath:filePath
                             error:&error])
{
    NSLog(@"[Error] %@ (%@)", error, filePath);
}
```

Determine the Creation Date of a File

```
NSFileManager *fileManager =
  [NSFileManager defaultManager];
NSString *documentsPath =
  [NSSearchPathForDirectoriesInDomains(
NSDocumentDirectory, NSUserDomainMask, YES) firstObject];
NSString *filePath = [documentsPath stringByAppendingPathComponent:@"Document.pages"];
NSDate *creationDate = nil;
if ([fileManager fileExistsAtPath:filePath]) {
    NSDictionary *attributes =
      [fileManager attributesOfItemAtPath:filePath
                                    error:nil];
    creationDate = attributes[NSFileCreationDate];
}
```

NSFileManagerDelegate

- -fileManager:shouldMoveItemAtURL:toURL:
- -fileManager:shouldCopyItemAtURL:toURL:

- -fileManager:shouldRemoveItemAtURL:
- -fileManager:shouldLinkItemAtURL:toURL:

NSValue

NSValue is a container for a single C or Objective-C data values. It can hold scalars and value types, as well as pointers and object IDs.

valueWithBytes:objCType:

- value: the value for the new NSValue object.
- type: the Objective-C type of value. type should be created with the Objective-C @encode() compiler directive; it should not be hard-coded as a C string.

valueWithNonretainedObject:

In short, valueWithNonretainedObject: allows objects to be added to a collection, without the need for satisfying .

NSDataDetector

NSDataDetector is a subclass of NSRegularExpression, but instead of matching on an ICU pattern, it detects semi-structured information: dates, addresses, links, phone numbers and transit information.

```
NSError *error = nil;
NSDataDetector *detector =
[NSDataDetector dataDetectorWithTypes: NSTextCheckingTypeAddress | NSTextCheckingTypePhoneNumber error:&error];
NSString *string = @"123 Main St. / (555) 555-5555";
[detector enumerateMatchesInString:string
                           options:kNilOptions
                             range:NSMakeRange(0, [string length])
                        usingBlock:^(NSTextCheckingResult *result,
                                     NSMatchingFlags flags, BOOL *stop)
{
    if (result.resultType == NSTextCheckingTypePhoneNumber)
    {
        NSLog(@"Match: %@", result.phoneNumber);
    }
}];
```

When initializing NSDataDetector, be sure to specify only the types you're interested in. With each additional type to be checked comes a nontrivial performance cost.

NSCache

NSHashTable & NSMapTable

UIAppearance

UIAppearance allows the appearance of views and controls to be consistently customized across the entire application.

In order to have this work within the existing structure of UIKit, Apple devised a rather clever solution: **UIAppearance is a protocol that returns a proxy which forwards any configuration to every instance of a particular class.**

Why a proxy instead of a property or method on UIView directly? Because there are non-UIView objects like UIBarButtonItem that render their own composite views.

+appearance

To customize the appearance of all instances of a class, you use appearance to get the appearance proxy for the class. For example, to modify the tint color for all instances of UINavigationBar:

```
[[UINavigationBar appearance] setTintColor:myColor];
```

+appearanceWhenContainedIn:

To customize the appearances in a way that adjusts for being contained within an instance of a container class, or instances in a hierarchy, you use appearanceWhenContainedIn: to get the appearance proxy for the class:

```
[[UIBarButtonItem appearanceWhenContainedIn:[UINavigationBar class], nil] setTintColor:myNavBarColor];

[[UIBarButtonItem appearanceWhenContainedIn:[UINavigationBar class], nil] setTintColor:myPopoverNavBarColor];

[[UIBarButtonItem appearanceWhenContainedIn:[UIToolbar class], nil] setTintColor:myToolbarColor];

[[UIBarButtonItem appearanceWhenContainedIn:[UIToolbar class], [UIPopoverController class], nil] setTintColor:myPopover
```

Implementing in Custom UIView Subclasses

TODO

Determining Which Properties Work With UIAppearance

In order to find out what methods work with UIAppearance, one is forced to grep through the headers:

```
$ cd /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS*.sdk/System/Library

$ grep -H UI_APPEARANCE_SELECTOR ./* | sed 's/ __OSX_AVAILABLE_STARTING(__MAC_NA,__IPHONE_5_0) UI_APPEARANCE_SELECTOR;/
```

NSLocalizedString