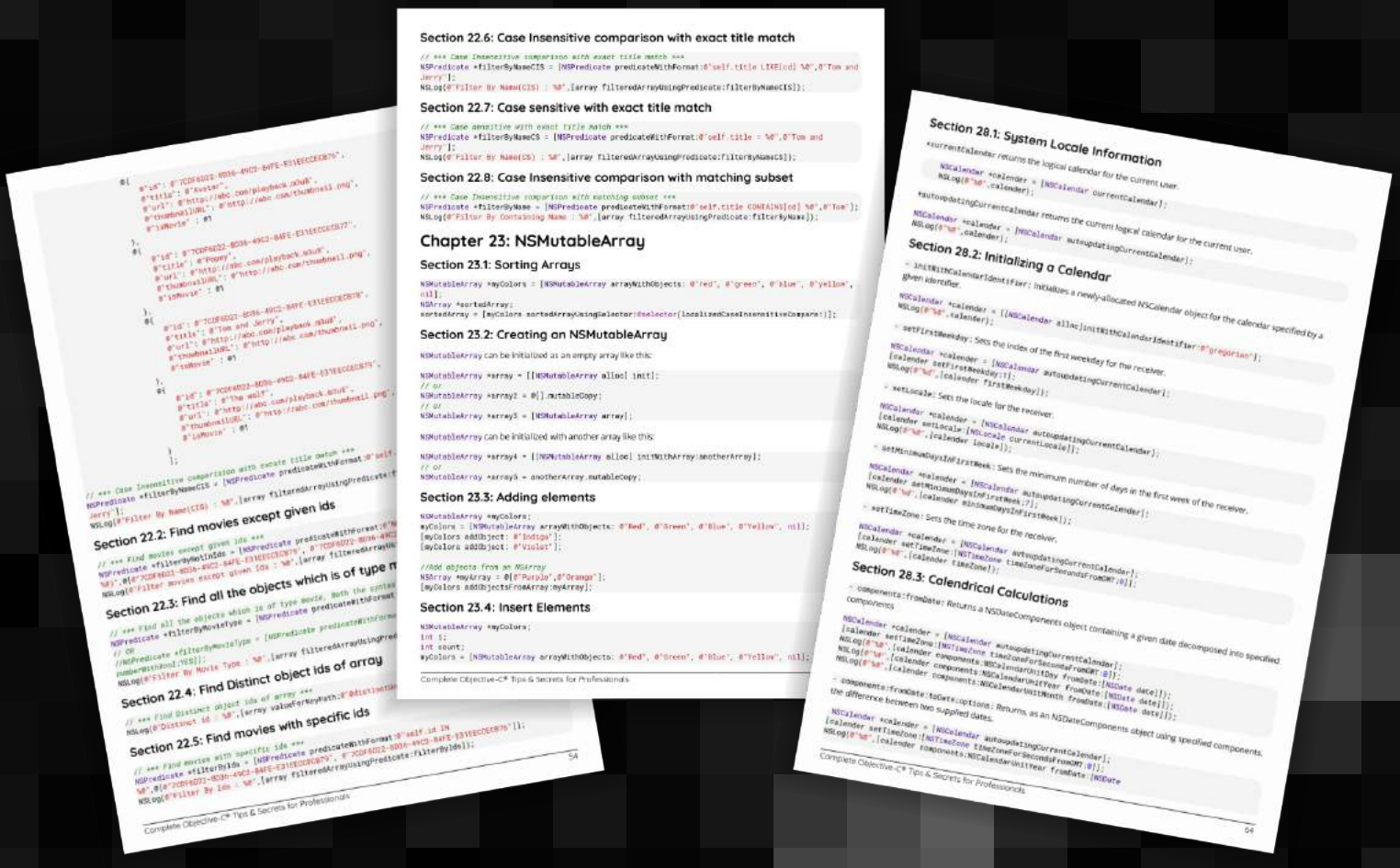
Complete Objective-C®

Tips & Secrets for Professionals



80+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
http://GoalKicker.com/ObjectiveCBook

Send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Objective-C Language

Version	Release Date
1.0	1983-01-01
2.0	2007-10-27
Modern	2014-03-10

Section 1.1: Hello World

This program will output "Hello World!"

```
#import <Foundation/Foundation.h>

int main(int argc, char * argv[]) {
    NSLog(@"Hello World!");
}
```

`#import` is a pre-processor directive, which indicates we want to *import* or include the information from that file into the program. In this case, the compiler will copy the contents of `Foundation.h` in the `Foundation` framework to the top of the file. The main difference between #import and #include is that #import is "smart" enough to not reprocess files that have already been included in other #includes.

The C Language documentation explains the `main` function.

The `NSLog()` function will print the string provided to the console, along with some debugging information. In this case, we use an Objective-C string literal: `@"Hello World!"`. In C, you would write this as `"Hello World!"`, however, Apple's Foundation Framework adds the `NSString` class which provides a lot of useful functionality, and is used by NSLog. The simplest way to create an instance of `NSString` is like this: `@"`>*string content here*".

Technically, NSLog() is part of Apple's Foundation Framework and is not actually part of the Objective-C language. However, the Foundation Framework is ubiquitous throughout Objective-C programming. Since the Foundation Framework is not open-source and cannot be used outside of Apple development, there are open-source alternatives to the framework which are associated with OPENStep and GNUStep.

Compiling the program

Assuming we want to compile our Hello World program, which consist of a single `hello.m` file, the command to compile the executable is:

```
clang -framework Foundation hello.m -o hello
```

Then you can run it:

```
./hello
```

This will output:

```
Hello World!
```

The options are:

- `-framework`: Specifies a framework to use to compile the program. Since this program uses Foundation, we include the Foundation framework.

- `-o`: This option indicate to which file we'd like to output our program. In our case `hello`. If not specified, the

default value is `a.out`.

Chapter 2: Protocols

Section 2.1: Optional and required methods

By default, all the methods declared in a protocol are required. This means that any class that conforms to this protocol must implement those methods.

It is also possible to declare *optional* methods. These method can be implemented only if needed.

You mark optional methods with the `@optional` directive.

```
@protocol NewProtocol
- (void)protocolMethod:(id)argument;
@optional
- (id)anotherMethod;
@end
```

In this case, only `anotherMethod` is marked as optional; the methods without the `@optional` directive are assumed to be required.

The `@optional` directive applies to methods that follow, until the end of the protocol definition or, until another directive is found.

```
@protocol NewProtocol
- (void)protocolMethod:(id)argument;
@optional
- (id)anotherMethod;
- (void)andAnotherMethod:(id)argument;
@required
- (void)lastProtocolMethod;
@end
```

This last example defines a protocol with two optional methods and two required methods.

Section 2.2: Checking existance of optional method implementations

```
if ([object respondsToSelector:@selector(someOptionalMethodInProtocol:)])
{
    [object someOptionalMethodInProtocol:argument];
}
```

Section 2.3: Forward Declarations

It's possible to declare protocol name without methods:

```
@protocol Person;
```

use it your code (class definition, etc):

```
@interface World : NSObject
@property (strong, nonatomic) NSArray<id<some>> *employees;
@end
```

and later define protocol's method somewhere in your code:

```
@protocol Person
- (NSString *)gender;
```

```
- (NSString *)name;
@end
```

It's useful when you don't need to know protocols details until you import that file with protocol definition. So, your class header file stays clear and contains details of the class only.

Section 2.4: Conforming to Protocols

The following syntax indicate that a class adopts a protocol, using angle brackets.

```
@interface NewClass : NSObject <NewProtocol>
...
@end
```

This means that any instance of NewClass will respond to methods declared in its interface but also it will provide an implementation for all the required methods of `NewProtocol`.

It is also possible for a class to conform to multiple protocols, by separating them with comma.

```
@interface NewClass : NSObject <NewProtocol, AnotherProtocol, MyProtocol>
...
@end
```

Like when conforming to a single protocol, the class must implement each required method of each protocols, and each optional method you choose to implement.

Section 2.5: Basic Protocol Definition

Defining a new protocol:

```
@protocol NewProtocol

- (void)protocolMethod:(id)argument;

- (id)anotherMethod;

@end
```

Section 2.6: Check conforms Protocol

Returns a Boolean indicating if the class conform the protocol:

```
[MyClass conformsToProtocol:@protocol(MyProtocol)];
```

Chapter 3: Blocks

- // Declare as a local variable:

  returnType (^blockName)(parameterType1, parameterType2, ...) = ^returnType(argument1, argument2, ...) {...};

- // Declare as a property:

  @property (nonatomic, copy, nullability) returnType (^blockName)(parameterTypes);

- // Declare as a method parameter:

- (void)someMethodThatTakesABlock:(returnType (^nullability)(parameterTypes))blockName;

- // Declare as an argument to a method call:

  [someObject someMethodThatTakesABlock:^returnType (parameters) {...}];

- // Declare as a typedef:

  typedef returnType (^TypeName)(parameterTypes);

  TypeName blockName = ^returnType(parameters) {...};

- // Declare a C function return a block object:

  BLOCK_RETURN_TYPE (^function_name(function parameters))(BLOCK_PARAMETER_TYPE);

Section 3.1: Block Typedefs

```
typedef double (^Operation)(double first, double second);
```

If you declare a block type as a typedef, you can then use the new type name instead of the full description of the arguments and return values. This defines `Operation` as a block that takes two doubles and returns a double.

The type can be used for the parameter of a method:

```
- (double)doWithOperation:(Operation)operation
                    first:(double)first
                   second:(double)second;
```

or as a variable type:

```
Operation addition = ^double(double first, double second){
    return first + second;
};

// Returns 3.0
[self doWithOperation:addition
                first:1.0
               second:2.0];
```

Without the typedef, this is much messier:

```
- (double)doWithOperation:(double (^)(double, double))operation
                    first:(double)first
                   second:(double)second;

double (^addition)(double, double) = // ...
```

Section 3.2: Blocks as Properties

```
@interface MyObject : MySuperclass

@property (copy) void (^blockProperty)(NSString *string);

@end
```

When assigning, since `self` retains `blockProperty`, block should not contain a strong reference to self. Those

mutual strong references are called a "retain cycle" and will prevent the release of either object.

```
__weak __typeof(self) weakSelf = self;
self.blockProperty = ^(NSString *string) {
    // refer only to weakSelf here.  self will cause a retain cycle
};
```

It is highly unlikely, but `self` might be deallocated inside the block, somewhere during the execution. In this case `weakSelf` becomes `nil` and all messages to it have no desired effect. This might leave the app in an unknown state. This can be avoided by retaining `weakSelf` with a `__strong` ivar during block execution and clean up afterward.

```
__weak __typeof(self) weakSelf = self;
self.blockProperty = ^(NSString *string) {
    __strong __typeof(weakSelf) strongSelf = weakSelf;
    // refer only to strongSelf here.
    // ...
    // At the end of execution, clean up the reference
    strongSelf = nil;
};
```

Section 3.3: Blocks as local variables

```
returnType (^blockName)(parameterType1, parameterType2, ...) = ^returnType(argument1, argument2,
...) {...};

float (^square)(float) = ^(float x) {return x*x;};

square(5); // resolves to 25
square(-7); // resolves to 49
```

Here's an example with no return and no parameters:

```
NSMutableDictionary *localStatus;
void (^logStatus)() = ^(void){ [MYUniversalLogger logCurrentStatus:localStatus]};

// Insert some code to add useful status information
// to localStatus dictionary

logStatus(); // this will call the block with the current localStatus
```

Section 3.4: Blocks as Method Parameters

```
- (void)methodWithBlock:(returnType (^)(paramType1, paramType2, ...))name;
```

Section 3.5: Defining and Assigning

A block that performs addition of two double precision numbers, assigned to variable `addition`:

```
double (^addition)(double, double) = ^double(double first, double second){
    return first + second;
};
```

The block can be subsequently called like so:

```
double result = addition(1.0, 2.0); // result == 3.0
```

Chapter 4: Categories

- @interface ClassName (categoryName) // ClassName is the class to be extended

- // Method and property declarations

- @end

Section 4.1: Conforming to protocol

You can add protocols to standard classes to extends their functionality:

```
@protocol EncodableToString <NSObject>
- (NSString *)toString;
@end

@interface NSDictionary (XYZExtended) <EncodableToString>
@end

@implementation NSDictionary (XYZExtended)
- (NSString *)toString {
    return self.description;
}
@end
```

where `XYZ` your project's prefix

Section 4.2: Simple Category

Interface and implementation of a simple category on NSArray, named Filter, with a single method that filters numbers.

It is good practice to add a prefix (`PF`) to the method to ensure we don't overwrite any future `NSArray` methods.

```
@interface NSArray (PFFilter)

- (NSArray *)pf_filterSmaller:(double)number;

@end

@implementation NSArray (PFFilter)

- (NSArray *)pf_filterSmaller:(double)number
{
    NSMutableArray *result = [NSMutableArray array];
    for (id val in self)
    {
        if ([val isKindOfClass:[NSNumber class] && [val doubleValue] >= number)
        {
            [result addObject:val];
        }
    }
    return [result copy];
}

@end
```

Section 4.3: Declaring a class method

Header file `UIColor+XYZPalette.h`:

```
@interface UIColor (XYZPalette)
```

```
+(UIColor *)xyz_indigoColor;

@end
```

and implementation `UIColor+XYZPalette.m`:

```
@implementation UIColor (XYZPalette)

+(UIColor *)xyz_indigoColor
{
    return [UIColor colorWithRed:75/255.0f green:0/255.0f blue:130/255.0f alpha:1.0f];
}

@end
```

Section 4.4: Adding a property with a category

Properties can be added with categories using associated objects, a feature of the Objective-C runtime.

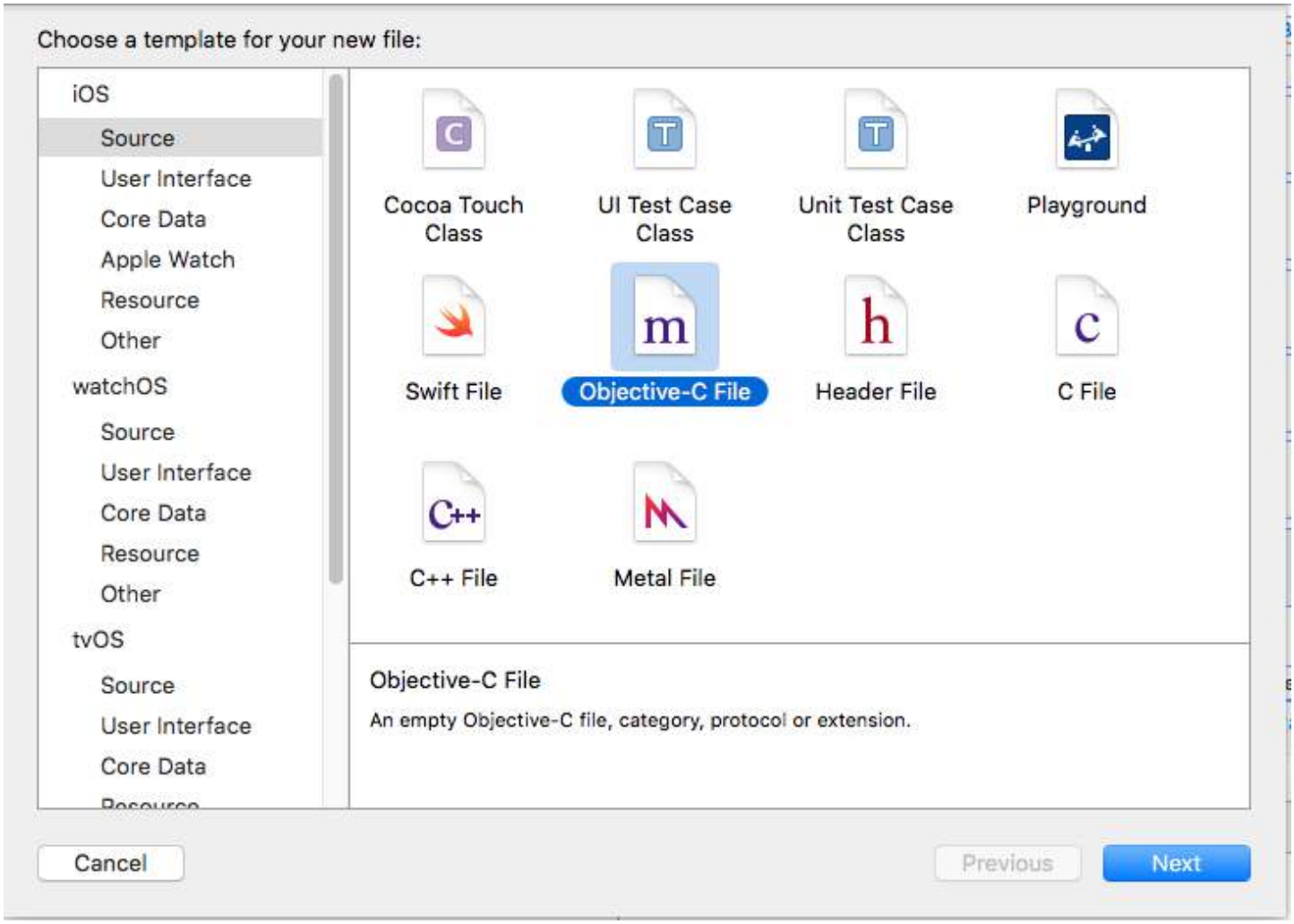
Note that the property declaration of `retain, nonatomic` matches the last argument to `objc_setAssociatedObject`. See Attach object to another existing object for explanations.

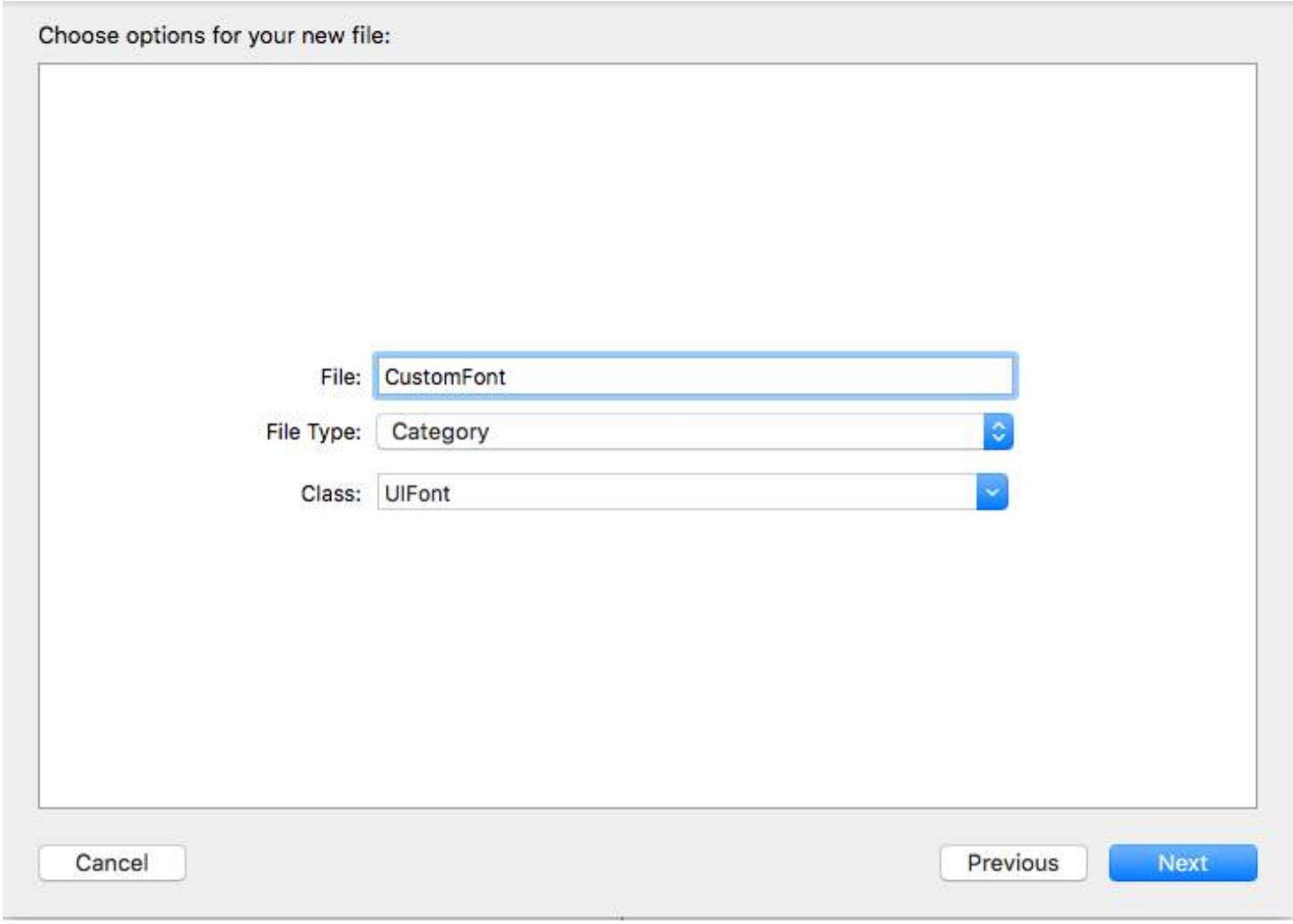
```
#import <objc/runtime.h>

@interface UIViewController (ScreenName)

@property (retain, nonatomic) NSString *screenName;

@end

@implementation UIViewController (ScreenName)

@dynamic screenName;

- (NSString *)screenName {
    return objc_getAssociatedObject(self, @selector(screenName));
}

- (void)setScreenName:(NSString *)screenName {
    objc_setAssociatedObject(self, @selector(screenName), screenName,
OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

@end
```

Section 4.5: Create a Category on XCode

Categories provide the ability to add some extra functionality to an object without subclassing or changing the actual object.

For example we want to set some custom fonts. Let's create a category that add functionality to `UIFont` class. Open your XCode project, click on `File` -> `New` -> `File` and choose `Objective-C file`, click Next enter your category name say "CustomFont" choose file type as Category and Class as UIFont then Click "Next" followed by "Create."


Choose a template for your new file:
iOS
Source
User Interface
Core Data
Apple Watch
Resource
Other
watchOS
Source
User Interface
Core Data
Resource
Other
tvOS
Source
User Interface
Core Data
Cocoa Touch Class
UI Test Case Class
Unit Test Case Class
Playground
Swift File
Objective-C File
Header File
C File
C++ File
Metal File
Objective-C File
An empty Objective-C file, category, protocol or extension.
Cancel
Previous
Next



Declare the Category Method :-

Click "UIFont+CustomFonts.h" to view the new category's header file. Add the following code to the interface to declare the method.

```
@interface UIFont (CustomFonts)

+(UIFont *)productSansRegularFontWithSize:(CGFloat)size;

@end
```

Now Implement the Category Method:-

Click "UIFont+CustomFonts.m" to view the category's implementation file. Add the following code to create a method that will set ProductSansRegular Font.

```
+(UIFont *)productSansRegularFontWithSize:(CGFloat)size{

    return [UIFont fontWithName:@"ProductSans-Regular" size:size];

}
```

Import your category

```
#import "UIFont+CustomFonts.h"
```

Now set the Label font

```
[self.label setFont:[UIFont productSansRegularFontWithSize:16.0]];
```

Chapter 5: Key Value Coding / Key Value Observing

Section 5.1: Most Common Real Life Key Value Coding Example

Key Value Coding is integrated into **NSObject** using **NSKeyValueCoding** protocol.

What this means?

It means that any id object is capable of calling valueForKey method and its various variants like valueForKeyPath etc. '

It also means that any id object can invoke setValue method and its various variants too.

Example:

```
id obj = [[MyClass alloc] init];
id value = [obj valueForKey:@"myNumber"];

int myNumberAsInt = [value intValue];
myNumberAsInt = 53;
[obj setValue:@(myNumberAsInt) forKey:@"myNumber"];
```

Exceptions:

Above example assumes that MyClass has an NSNumber Property called myNumber. If myNumber does not appear in MyClass interface definition, an NSUndefinedKeyException can be raised at possibly both lines 2 and 5 - popularly known as:

```
this class is not key value coding-compliant for the key myNumber.
```

Why this is SO powerful:

You can write code that can access properties of a class dynamically, without needing interface for that class. This means that a table view can display values from any properties of an NSObject derived object, provided its property names are supplied dynamically at runtime.

In the example above, the code can as well work without MyClass being available and id type obj being available to calling code.

Section 5.2: Querying KVC Data

```
if ([[dataObject objectForKey:@"yourVariable"] isEqualToString:"Hello World"]) {
    return YES;
} else {
    return NO;
}
```

You can query values stored using KVC quickly and easily, without needing to retrieve or cast these as local variables.

Section 5.3: Collection Operators

Collection Operators can be used in a KVC key path to perform an operation on a “collection-type” property (i.e. `NSArray`, `NSSet` and similar). For example, a common operation to perform is to count the objects in a collection. To achieve this, you use the `@count` *collection operator*:

```
self.array = @[@5, @4, @3, @2, @1];
NSNumber *count = [self.array valueForKeyPath:@"@count"];
```

```
NSNumber *countAlt = [self valueForKeyPath:@"array.@count"];
// count == countAlt == 5
```

While this is completely redundant here (we could have just accessed the `count` property), it *can* be useful on occasion, though it is rarely necessary. There are, however, some collection operators that are much more useful, namely `@max`, `@min`, `@sum`, `@avg` and the `@unionOf` family. It is important to note that these operators *also* require a separate key path *following* the operator to function correctly. Here's a list of them and the type of data they work with:

Operator	Data Type
`@count`	(none)
`@max`	`NSNumber`, `NSDate`, `int` (and related), etc.
`@min`	`NSNumber`, `NSDate`, `int` (and related), etc.
`@sum`	`NSNumber`, `int` (and related), `double` (and related), etc.
`@avg`	`NSNumber`, `int` (and related), `double` (and related), etc.
`@unionOfObjects`	`NSArray`, `NSSet`, etc.
`@distinctUnionOfObjects`	`NSArray`, `NSSet`, etc.
`@unionOfArrays`	`NSArray<NSArray*>`
`@distinctUnionOfArrays`	`NSArray<NSArray*>`
`@distinctUnionOfSets`	`NSSet<NSSet*>`

`@max` and `@min` will return the highest or lowest value, respectively, of a property of objects in the collection. For example, look at the following code:

```
// "Point" class used in our collection
@interface Point : NSObject

@property NSInteger x, y;

+ (instancetype)pointWithX:(NSInteger)x y:(NSInteger)y;

@end

...

self.points = @[[Point pointWithX:0 y:0],
                [Point pointWithX:1 y:-1],
                [Point pointWithX:5 y:-6],
                [Point pointWithX:3 y:0],
                [Point pointWithX:8 y:-4],
];

NSNumber *maxX = [self valueForKeyPath:@"points.@max.x"];
NSNumber *minX = [self valueForKeyPath:@"points.@min.x"];
NSNumber *maxY = [self valueForKeyPath:@"points.@max.y"];
NSNumber *minY = [self valueForKeyPath:@"points.@min.y"];

NSArray<NSNumber*> *boundsOfAllPoints = @[maxX, minX, maxY, minY];

...
```

In just a 4 lines of code and pure Foundation, with the power of Key-Value Coding collection operators we were able to extract a rectangle that encapsulates all of the points in our array.

It is important to note that these comparisons are made by invoking the `compare:` method on the objects, so if you ever want to make your own class compatible with these operators, you must implement this method.

`@sum` will, as you can probably guess, add up all the values of a property.

```
@interface Expense : NSObject

@property NSNumber *price;

+ (instancetype)expenseWithPrice:(NSNumber *)price;

@end

...

self.expenses = @[[Expense expenseWithPrice:@1.50],
                  [Expense expenseWithPrice:@9.99],
                  [Expense expenseWithPrice:@2.78],
                  [Expense expenseWithPrice:@9.99],
                  [Expense expenseWithPrice:@24.95]
];

NSNumber *totalExpenses = [self valueForKeyPath:@"expenses.@sum.price"];
```

Here, we used `@sum` to find the total price of all the expenses in the array. If we instead wanted to find the average price we're paying for each expense, we can use `@avg`:

```
NSNumber *averagePrice = [self valueForKeyPath:@"expenses.@avg.price"];
```

Finally, there's the `@unionOf` family. There are five different operators in this family, but they all work mostly the same, with only small differences between each. First, there's `@unionOfObjects` which will return an array of the properties of objects in an array:

```
// See "expenses" array above

NSArray<NSNumber*> *allPrices = [self valueForKeyPath:
    @"expenses.@unionOfObjects.price"];

// Equal to @[ @1.50, @9.99, @2.78, @9.99, @24.95 ]
```

`@distinctUnionOfObjects` functions the same as `@unionOfObjects`, but it removes duplicates:

```
NSArray<NSNumber*> *differentPrices = [self valueForKeyPath:
    @"expenses.@distinctUnionOfObjects.price"];

// Equal to @[ @1.50, @9.99, @2.78, @24.95 ]
```

And finally, the last 3 operators in the `@unionOf` family will go one step deeper and return an array of values found for a property contained inside dually-nested arrays:

```
NSArray<NSArray<Expense*,Expense*>*> *arrayOfArrays =
    @[
        @[ [Expense expenseWithPrice:@19.99],
           [Expense expenseWithPrice:@14.95],
           [Expense expenseWithPrice:@4.50],
           [Expense expenseWithPrice:@19.99]
         ],

        @[ [Expense expenseWithPrice:@3.75],
           [Expense expenseWithPrice:@14.95]
         ]
     ];

// @unionOfArrays
NSArray<NSNumber*> allPrices = [arrayOfArrays valueForKeyPath:
    @"@unionOfArrays.price"];
```

```
// Equal to @[ @19.99, @14.95, @4.50, @19.99, @3.75, @14.95 ];

// @distinctUnionOfArrays
NSArray<NSNumber*> allPrices = [arrayOfArrays valueForKeyPath:
    @"@distinctUnionOfArrays.price"];
// Equal to @[ @19.99, @14.95, @4.50, @3.75 ];
```

The one missing from this example is `@distinctUnionOfSets`, however this functions exactly the same as `@distinctUnionOfArrays`, but works with and returns `NSSet`s instead (there is no non-`distinct` version because in a set, every object must be distinct anyway).

And that's it! Collection operators can be really powerful if used correctly, and can help to avoid having to loop through stuff unnecessarily.

One last note: you can also use the standard collection operators on arrays of `NSNumber`s (without additional property access). To do this, you access the `self` pseudo-property that just returns the object:

```
NSArray<NSNumber*> *numbers = @[@0, @1, @5, @27, @1337, @2048];

NSNumber *largest = [numbers valueForKeyPath:@"@max.self"];
NSNumber *smallest = [numbers valueForKeyPath:@"@min.self"];
NSNumber *total = [numbers valueForKeyPath:@"@sum.self"];
NSNumber *average = [numbers valueForKeyPath:@"@avg.self"];
```

Section 5.4: Key Value Observing

Setting up key value observing.

In this case, we want to observe the `contentOffset` on an object that our observer owns

```
//
// Class to observe
//
@interface XYZScrollView: NSObject
@property (nonatomic, assign) CGPoint contentOffset;
@end

@implementation XYZScrollView
@end


//
// Class that will observe changes
//
@interface XYZObserver: NSObject
@property (nonatomic, strong) XYZScrollView *scrollView;
@end

@implementation XYZObserver

// simple way to create a KVO context
static void *XYZObserverContext = &XYZObserverContext;


// Helper method to add self as an observer to
// the scrollView's contentOffset property
- (void)addObserver {

    // NSKeyValueObservingOptions
    //
    // - NSKeyValueObservingOptionNew
```

```
    // - NSKeyValueObservingOptionOld
    // - NSKeyValueObservingOptionInitial
    // - NSKeyValueObservingOptionPrior
    //
    // can be combined:
    // (NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld)

    NSString *keyPath = NSStringFromSelector(@selector(contentOffset));
    NSKeyValueObservingOptions options = NSKeyValueObservingOptionNew;

    [self.scrollView addObserver: self
                      forKeyPath: keyPath
                         options: options
                         context: XYZObserverContext];
}

- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
change:(NSDictionary<NSString *,id> *)change context:(void *)context {

    if (context == XYZObserverContext) { // check the context

        // check the keyPath to see if it's any of the desired keyPath's.
        // You can observe multiple keyPath's
        if ([keyPath isEqualToString: NSStringFromSelector(@selector(contentOffset))]) {

            // change dictionary keys:
            // - NSKeyValueChangeKindKey
            // - NSKeyValueChangeNewKey
            // - NSKeyValueChangeOldKey
            // - NSKeyValueChangeIndexesKey
            // - NSKeyValueChangeNotificationIsPriorKey

            // the change dictionary here for a CGPoint observation will
            // return an NSPoint, so we can take the CGPointValue of it.
            CGPoint point = [change[NSKeyValueChangeNewKey] CGPointValue];

            // handle point
        }

    } else {

        // if the context doesn't match our current object's context
        // we want to pass the observation parameters to super
        [super observeValueForKeyPath: keyPath
                             ofObject: object
                               change: change
                              context: context];
    }
}

// The program can crash if an object is not removed as observer
// before it is dealloc'd
//
// Helper method to remove self as an observer of the scrollView's
// contentOffset property
- (void)removeObserver {
    NSString *keyPath = NSStringFromSelector(@selector(contentOffset));
    [self.scrollView removeObserver: self forKeyPath: keyPath];
}

@end
```

Chapter 6: Logging

- NSLog(@"text to log"); // Basic text log
- NSLog(@"data: %f - %.2f", myFloat, anotherFloat); // Logging text including float numbers.
- NSLog(@"data: %i", myInteger); // Logging text including integer number.
- NSLog(@"data: %@", myStringOrObject); // Logging text referencing another String or any NSObject derived object.

Section 6.1: Logging

```
NSLog(@"Log Message!");
NSLog(@"NSString value: %@", stringValue);
NSLog(@"Integer value: %d", intValue);
```

The first argument of `NSLog` is an `NSString` containing the log message format. The rest of the parameters are used as values to substitute in place of the format specifiers.

The formatting works exactly the same as `printf`, except for the additional format specifier `%@` for an arbitrary Objective-C object. This:

```
NSLog(@"%@", object);
```

is equivalent to:

```
NSLog(@"%s", [object description].UTF8String);
```

Section 6.2: NSLog Output Format

```
NSLog(@"NSLog message");
```

The message that gets printed by calling `NSLog` has the following format when viewed in Console.app:

Date	Time	Program name	Process ID	Thread ID	Message
2016-07-16	08:58:04.681	test	[46259	:1244773]	NSLog message

Section 6.3: Removing Log Statements from Release Builds

Messages printed from `NSLog` are displayed on Console.app even in the release build of your app, which doesn't make sense for printouts that are only useful for debugging. To fix this, you can use this macro for debug logging instead of `NSLog`.

```
#ifdef DEBUG
#define DLog(...) NSLog(__VA_ARGS__)
#else
#define DLog(...)
#endif
```

To use:

```
NSString *value = @"value 1";
DLog(@"value = %@", value);
// little known fact: programmers look for job postings in Console.app
NSLog(@"We're hiring!");
```

In debug builds, `DLog` will call `NSLog`. In release builds, `DLog` will do nothing.

Section 6.4: Logging Variable Values

You shouldn't call `NSLog` without a literal format string like this:

```
NSLog(variable);    // Dangerous code!
```

If the variable is not an `NSString`, the program will crash, because `NSLog` expects an `NSString`.

If the variable is an `NSString`, it will work unless your string contains a `%`. `NSLog` will parse the `%` sequence as a format specifier and then read a garbage value off the stack, causing a crash or even executing arbitrary code.

Instead, always make the first argument a format specifier, like this:

```
NSLog(@"%@", anObjectVariable);
NSLog(@"%d", anIntegerVariable);
```

Section 6.5: Empty message is not printed

When `NSLog` is asked to print empty string, it omits the log completely.

```
NSString *name = @"";
NSLog(@"%@", name);  // Resolves to @""
```

The above code will print **nothing**.

It is a good practice to prefix logs with labels:

```
NSString *name = @"";
NSLog(@"Name: %@", name);  // Resolves to @"Name: "
```

The above code will print:

2016-07-21 14:20:28.623 App[87711:6153103] Name:

Section 6.6: Using __FUNCTION __

```
NSLog(@"%s %@",__FUNCTION__, @"etc etc");
```

Inserts the class and method name into the output:

```
2016-07-22 12:51:30.099 loggingExample[18132:2971471] -[ViewController viewDidLoad] etc etc
```

Section 6.7: NSLog vs printf

```
NSLog(@"NSLog message");
printf("printf message\n");
```

Output:

```
2016-07-16 08:58:04.681 test[46259:1244773] NSLog message
printf message
```

`NSLog` outputs the date, time, process name, process ID, and thread ID in addition to the log message. `printf` just outputs the message.

`NSLog` requires an `NSString` and automatically adds a newline at the end. `printf` requires a C string and does not automatically add a newline.

`NSLog` sends output to `stderr`, `printf` sends output to `stdout`.

Some `format-specifiers` in `printf` vs `NSLog` are different. For example when including a nested string, the following differences incur:

```
NSLog(@"My string: %@", (NSString *)myString);
```

```
printf("My string: %s", [(NSString *)myString UTF8String]);
```

Section 6.8: Logging NSLog meta data

```
NSLog(@"%s %d %s, yourVariable: %@", __FILE__, __LINE__, __PRETTY_FUNCTION__, yourVariable);
```

Will log the file, line number and function data along with any variables you want to log. This can make the log lines much longer, particularly with verbose file and method names, however it can help to speed up error diagnostics.

You can also wrap this in a Macro (store this in a Singleton or where you'll need it most);

```
#define ALog(fmt, ...) NSLog((@"%s [Line %d] " fmt), __PRETTY_FUNCTION__, __LINE__, ##__VA_ARGS__);
```

Then when you want to log, simply call

```
ALog(@"name: %@", firstName);
```

Which will give you something like;

```
-[AppDelegate application:didFinishLaunchingWithOptions:] [Line 27] name: John
```

Section 6.9: NSLog and BOOL type

There is no format specifier to print boolean type using NSLog. One way to print boolean value is to convert it to a string.

```
BOOL boolValue = YES;
NSLog(@"Bool value %@", boolValue ? @"YES" : @"NO");
```

Output:

```
2016-07-30 22:53:18.269 Test[4445:64129] Bool value YES
```

Another way to print boolean value is to cast it to integer, achieving a binary output (1=yes, 0=no).

```
BOOL boolValue = YES;
NSLog(@"Bool value %i", boolValue);
```

Output:

```
2016-07-30 22:53:18.269 Test[4445:64129] Bool value 1
```

Section 6.10: Logging by Appending to a File

NSLog is good, but you can also log by appending to a file instead, using code like:

```
NSFileHandle* fh = [NSFileHandle fileHandleForWritingAtPath:path];
if ( !fh ) {
    [[NSFileManager defaultManager] createFileAtPath:path contents:nil attributes:nil];
    fh = [NSFileHandle fileHandleForWritingAtPath:path];
}
if ( fh ) {
    @try {
        [fh seekToEndOfFile];
        [fh writeData:[self dataUsingEncoding:enc]];
    }
    @catch (...) {
    }
    [fh closeFile];
}
```

Chapter 7: NSArray

- NSArray *words; // Declaring immutable array
- NSMutableArray *words; // Declaring mutable array
- NSArray *words = [NSArray arrayWithObjects: @"one", @"two", nil]; // Array initialization syntax
- NSArray *words = @[@"list", @"of", @"words", @123, @3.14]; // Declaring array literals
- NSArray *stringArray = [NSArray arrayWithObjects: [NSMutableArray array], [NSMutableArray array], [NSMutableArray array], nil]; // Creating multidimentional arrays

Section 7.1: Creating Arrays

Creating immutable arrays:

```
NSArray *myColors = [NSArray arrayWithObjects: @"Red", @"Green", @"Blue", @"Yellow", nil];

// Using the array literal syntax:
NSArray *myColors = @[@"Red", @"Green", @"Blue", @"Yellow"];
```

For mutable arrays, see NSMutableArray.

Section 7.2: Accessing elements

```
NSArray *myColors = @[@"Red", @"Green", @"Blue", @"Yellow"];
// Preceding is the preferred equivalent to [NSArray arrayWithObjects:...]
```

Getting a single item

The `objectAtIndex:` method provides a single object. The first object in an `NSArray` is index 0. Since an `NSArray` can be homogenous (holding different types of objects), the return type is `id` ("any object"). (An `id` can be assigned to a variable of any other object type.) Importantly, `NSArray`s can only contain objects. They cannot contain values like `int`.

```
NSUInteger idx = 2;
NSString *color = [myColors objectAtIndex:idx];
// color now points to the string @"Green"
```

Clang provides a better subscript syntax as part of its array literals functionality:

```
NSString *color = myColors[idx];
```

Both of these throw an exception if the passed index is less than 0 or greater than `count - 1`.

First and Last Item

```
NSString *firstColor = myColors.firstObject;
NSString *lastColor = myColors.lastObject;
```

The `firstObject` and `lastObject` are computed properties and return `nil` rather than crashing for empty arrays. For single element arrays they return the same object. Although, the `firstObject` method was not introduced to `NSArray` until iOS 4.0.

```
NSArray *empty = @[]
id notAnObject = empty.firstObject;    // Returns `nil`
id kaboom = empty[0];    // Crashes; index out of bounds
```

Section 7.3: Using Generics

For added safety we can define the type of object that the array contains:

```
NSArray<NSString *> *colors = @[@"Red", @"Green", @"Blue", @"Yellow"];
```

```
NSMutableArray<NSString *> *myColors = [NSMutableArray arrayWithArray:colors];
[myColors addObject:@"Orange"]; // OK
[myColors addObject:[UIColor purpleColor]]; // "Incompatible pointer type" warning
```

It should be noted that this is checked during compilation time only.

Section 7.4: Reverse an Array

```
NSArray *reversedArray = [myArray.reverseObjectEnumerator allObjects];
```

Section 7.5: Converting between Sets and Arrays

```
NSSet *set = [NSSet set];
NSArray *array = [NSArray array];

NSArray *fromSet = [set allObjects];
NSSet *fromArray = [NSSet setWithArray:array];
```

Section 7.6: Converting NSArray to NSMutableArray to allow modification

```
NSArray *myColors = [NSArray arrayWithObjects: @"Red", @"Green", @"Blue", @"Yellow", nil];

// Convert myColors to mutable
NSMutableArray *myColorsMutable = [myColors mutableCopy];
```

Section 7.7: Looping through

```
NSArray *myColors = @[@"Red", @"Green", @"Blue", @"Yellow"];

// Fast enumeration
// myColors cannot be modified inside the loop
for (NSString *color in myColors) {
    NSLog(@"Element %@", color);
}

// Using indices
for (NSUInteger i = 0; i < myColors.count; i++) {
    NSLog(@"Element %d = %@", i, myColors[i]);
}

// Using block enumeration
[myColors enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL * stop) {
    NSLog(@"Element %d = %@", idx, obj);

    // To abort use:
    *stop = YES
}];

// Using block enumeration with options
[myColors enumerateObjectsWithOptions:NSEnumerationReverse usingBlock:^(id obj, NSUInteger idx,
BOOL * stop) {
    NSLog(@"Element %d = %@", idx, obj);
}];
```

Section 7.8: Enumerating using blocks

```
NSArray *myColors = @[@"Red", @"Green", @"Blue", @"Yellow"];
[myColors enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
    NSLog(@"enumerating object %@ at index %lu", obj, idx);
}];
```

By setting the `stop` parameter to `YES` you can indicate that further enumeration is not needed. to do this simply set

`&stop = YES`.

NSEnumerationOptions

You can enumerate the array in reverse and / or concurrently :

```
[myColors enumerateObjectsWithOptions:NSEnumerationConcurrent | NSEnumerationReverse
                              usingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
                                  NSLog(@"enumerating object %@ at index %lu", obj, idx);
                              }];
```

Enumerating subset of array

```
NSIndexSet *indexSet = [NSIndexSet indexSetWithIndexesInRange:NSMakeRange(1, 1)];
[myColors enumerateObjectsAtIndexes:indexSet
                            options:kNilOptions
                         usingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
                              NSLog(@"enumerating object %@ at index %lu", obj, idx);
                            }];
```

Section 7.9: Comparing arrays

Arrays can be compared for equality with the aptly named **isEqualToArray:** method, which returns **YES** when both arrays have the same number of elements and every pair pass an **isEqual:** comparison.

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
NSArray *sameGermanMakes = [NSArray arrayWithObjects:@"Mercedes-Benz",
                            @"BMW", @"Porsche", @"Opel",
                            @"Volkswagen", @"Audi", nil];

if ([germanMakes isEqualToArray:sameGermanMakes]) {
    NSLog(@"Oh good, literal arrays are the same as NSArrays");
}
```

The important thing is every pair must pass the isEqual: test. For custom objects this method should be implemented.It exists in the NSObject protocol.

Section 7.10: Filtering Arrays With Predicates

```
NSArray *array = [NSArray arrayWithObjects:@"Nick", @"Ben", @"Adam", @"Melissa", nil];

NSPredicate *aPredicate = [NSPredicate predicateWithFormat:@"SELF beginswith[c] 'a'"];
NSArray *beginWithA = [array filteredArrayUsingPredicate:bPredicate];
     // beginWithA contains { @"Adam" }.

NSPredicate *ePredicate = [NSPredicate predicateWithFormat:@"SELF contains[c] 'e'"];
[array filterUsingPredicate:ePredicate];
    // array now contains { @"Ben", @"Melissa" }
```

More about

NSPredicate:

Apple doc : NSPredicate

Section 7.11: Sorting array with custom objects

Compare method

Either you implement a compare-method for your object:

```
- (NSComparisonResult)compare:(Person *)otherObject {
    return [self.birthDate compare:otherObject.birthDate];
}

NSArray *sortedArray = [drinkDetails sortedArrayUsingSelector:@selector(compare:)];
```

NSSortDescriptor

```
NSSortDescriptor *sortDescriptor;
sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"birthDate"
                                             ascending:YES];
NSArray *sortDescriptors = [NSArray arrayWithObject:sortDescriptor];
NSArray *sortedArray = [drinkDetails sortedArrayUsingDescriptors:sortDescriptors];
```

You can easily sort by multiple keys by adding more than one to the array. Using custom comparator-methods is possible as well. Have a look at the documentation.

Blocks

```
NSArray *sortedArray;
sortedArray = [drinkDetails sortedArrayUsingComparator:^NSComparisonResult(id a, id b) {
    NSDate *first = [(Person*)a birthDate];
    NSDate *second = [(Person*)b birthDate];
    return [first compare:second];
}];
```

Performance

The `-compare:` and block-based methods will be quite a bit faster, in general, than using `NSSortDescriptor` as the latter relies on KVC. The primary advantage of the `NSSortDescriptor` method is that it provides a way to define your sort order using data, rather than code, which makes it easy to e.g. set things up so users can sort an `NSTableView` by clicking on the header row.

Section 7.12: Add objects to NSArray

```
NSArray *a = @[@1];
a = [a arrayByAddingObject:@2];
a = [a arrayByAddingObjectsFromArray:@[@3, @4, @5]];
```

These methods are optimized to recreate the new array very efficiently, usually without having to destroy the original array or even allocate more memory.

Section 7.13: Finding out the Number of Elements in an Array

```
NSArray *myColors = [NSArray arrayWithObjects: @"Red", @"Green", @"Blue", @"Yellow", nil];
NSLog (@"Number of elements in array = %lu", [myColors count]);
```

Chapter 8: Classes and Objects

- Cat *cat = [[Cat alloc] init]; // Create cat object of type Cat
- Dog *dog = [[Dog alloc] init]; // Create dog object of type Dog
- NSObject *someObject = [NSObject alloc]; [someObject init]; // **don't do this**
- XYZObject *object = [XYZObject new]; // Use new to create objects if NO arguments are needed for initialization
- NSString *someString = @"Hello, World!"; // Creating an NSString with *literal syntax*
- NSNumber *myFloat = @3.14f; // Another example to create a NSNumber using literal syntax
- NSNumber *myInt = @(84 / 2); // Create an object using a boxed expression

Section 8.1: Difference between allocation and initialization

In most object oriented languages, allocating memory for an object and initializing it is an atomic operation:

```
// Both allocates memory and calls the constructor
MyClass object = new MyClass();
```

In Objective-C, these are separate operations. The class methods `alloc` (and its historic sibling `allocWithZone:`) makes the Objective-C runtime reserve the required memory and clears it. Except for a few internal values, all properties and variables are set to 0/`NO`/`nil`.

The object then is already "valid" but we always want to call a method to actually set up the object, which we call an *initializer*. These serve the same purpose as *constructors* in other languages. By convention, these methods start with `init`. From a language point of view, they are just normal methods.

```
// Allocate memory and set all properties and variables to 0/NO/nil.
MyClass *object = [MyClass alloc];
// Initialize the object.
object = [object init];

// Shorthand:
object = [[MyClass alloc] init];
```

Section 8.2: Creating classes with initialization values

```
#import <Foundation/Foundation.h>
@interface Car:NSObject {
    NSString *CarMotorCode;
    NSString *CarChassisCode;
}

- (instancetype)initWithMotorValue:(NSString *) motorCode andChassisValue:(NSInteger)chassisCode;
- (void) startCar;
- (void) stopCar;

@end

@implementation Car

- (instancetype)initWithMotorValue:(NSString *) motorCode andChassisValue:(NSInteger)chassisCode{
    CarMotorCode = motorCode;
    CarChassisCode = chassisCode;
    return self;
}

- (void) startCar {...}
- (void) stopCar {...}

@end
```

The method `initWithMotorValue: type andChassisValue: type` will be used to initialize the Car objects.

Section 8.3: Specifying Generics

You can enhance your own classes with *generics* just like `NSArray` or `NSDictionary`.

```
@interface MyClass<__covariant T>

@property (nonnull, nonatomic, strong, readonly) NSArray<T>* allObjects;

- (void) addObject:(nonnull T)obj;

@end
```

Section 8.4: Singleton Class

What is a Singleton Class?

A singleton class returns the same instance no matter how many times an application requests it. Unlike a regular class, A singleton object provides a global point of access to the resources of its class.

When to Use Singleton Classes?

Singletons are used in situations where this single point of control is desirable, such as with classes that offer some general service or resource.

How to Create Singleton Classes

First, create a New file and subclass it from `NSObject`. Name it anything, we will use `CommonClass` here. Xcode will now generate CommonClass.h and CommonClass.m files for you.

In your `CommonClass.h` file:

```
#import <Foundation/Foundation.h>

@interface CommonClass : NSObject {
}
+ (CommonClass *)sharedObject;
@property NSString *commonString;
@end
```

In your `CommonClass.m` File:

```
#import "CommonClass.h"

@implementation CommonClass

+ (CommonClass *)sharedObject {
    static CommonClass *sharedClass = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedClass = [[self alloc] init];
    });
    return sharedClass;
}

- (id)init {
    if (self = [super init]) {
        self.commonString = @"this is string";
    }
    return self;
}

@end
```

How to Use Singleton Classes

The Singleton Class that we created earlier will be accessible from anywhere in the project as long as you have imported `CommonClass.h` file in the relevant module. To modify and access the shared data in Singleton Class, you will have to access the shared Object of that class which can be accessed by using `sharedObject` method like following:

```
[CommonClass sharedObject]
```

To read or modify the elements in Shared Class, do the following:

```
NSString *commonString = [[CommonClass sharedObject].commonString; //Read the string in singleton
class

NSString *newString = @"New String";
[CommonClass sharedObject].commonString = newString;//Modified the string in singleton class
```

Section 8.5: The "instancetype" return type

Objective-C supports a special type called `instancetype that can only be used as type returned by a method. It evaluates to the class of the receiving object.

Consider the following class hierarchy:

```
@interface Foo : NSObject

- (instancetype)initWithString:(NSString *)string;

@end

@interface Bar : Foo
@end
```

When `[[Foo alloc] initWithString:@"abc"]` is called, the compiler can infer that the return type is `Foo *`. The `Bar` class derived from `Foo` but did not override the declaration of the initializer. Yet, thanks to `instancetype`, the compiler can infer that `[[Bar alloc] initWithString:@"xyz"]` returns a value of type `Bar *`.

Consider the return type of `-[Foo initWithString:]` being `Foo *` instead: if you would call `[[Bar alloc] initWithString:]`, the compiler would infer that a `Foo *` is returned, not a `Bar *` as is the intention of the developer. The `instancetype` solved this issue.

Before the introduction of `instancetype`, initializers, static methods like singleton accessors and other methods that want to return an instance of the receiving class needed to return an `id`. The problem is that `id` means *"an object of any type"*. The compiler is thus not able to detect that `NSString *wrong = [[Foo alloc] initWithString:@"abc"];` is assigning to a variable with an incorrect type.

Due to this issue, **initializers should always use `instancetype` instead of `id`** as the return value.

Chapter 9: NSString

The *NSString* class is a part of Foundation framework to work with strings (series of characters). It also includes methods for comparing, searching and modifying strings.

Section 9.1: Encoding and Decoding

```
// decode
NSString *string = [[NSString alloc] initWithData:utf8Data
                                         encoding:NSUTF8StringEncoding];

// encode
NSData *utf8Data = [string dataUsingEncoding:NSUTF8StringEncoding];
```

Some supported encodings are:

- `NSASCIIStringEncoding`
- `NSUTF8StringEncoding`
- `NSUTF16StringEncoding` (== `NSUnicodeStringEncoding`)

Note that `utf8Data.bytes` does not include a terminating null character, which is necessary for C strings. If you need a C string, use `UTF8String`:

```
const char *cString = [string UTF8String];
printf("%s", cString);
```

Section 9.2: String Length

NSString has a `length` property to get the number of characters.

```
NSString *string = @"example";
NSUInteger length = string.length;       // length equals 7
```

As in the Splitting Example, keep in mind that `NSString` uses UTF-16 to represent characters. The length is actually just the number of UTF-16 code units. This can differ from what the user perceives as characters.

Here are some cases that might be surprising:

```
@"é".length == 1   // LATIN SMALL LETTER E WITH ACUTE (U+00E9)
@"é".length == 2   // LATIN SMALL LETTER E (U+0065) + COMBINING ACUTE ACCENT (U+0301)
@"❤️".length == 2  // HEAVY BLACK HEART (U+2764) + VARIATION SELECTOR-16 (U+FE0F)
@"🇩🇪".length == 4  // REGIONAL INDICATOR SYMBOL LETTER I (U+1F1EE) + REGIONAL INDICATOR SYMBOL
LETTER T (U+1F1F9)
```

In order to get the number of user-perceived characters, known technically as "grapheme clusters", you must iterate over the string with `-enumerateSubstringsInRange:options:usingBlock:` and keep a count. This is demonstrated in an answer by Nikolai Ruhe on Stack Overflow.

Section 9.3: Comparing Strings

Strings are compared for equality using `isEqualToString:`

The `==` operator just tests for object identity and does not compare the logical values of objects, so it can't be used:

```
NSString *stringOne = @"example";
NSString *stringTwo = [stringOne mutableCopy];

BOOL objectsAreIdentical = (stringOne == stringTwo);          // NO
BOOL stringsAreEqual = [stringOne isEqualToString:stringTwo]; // YES
```

The expression `(stringOne == stringTwo)` tests to see if the memory addresses of the two strings are the same, which is usually not what we want.

If the string variables can be `nil` you have to take care about this case as well:

```
BOOL equalValues = stringOne == stringTwo || [stringOne isEqualToString:stringTwo];
```

This condition returns `YES` when strings have equal values or both are `nil`.

To order two strings alphabetically, use `compare:`.

```
NSComparisonResult result = [firstString compare:secondString];
```

`NSComparisonResult` can be:

- `NSOrderedAscending`: The first string comes before the second string.
- `NSOrderedSame`: The strings are equal.
- `NSOrderedDescending`: The second string comes before the first string.

To compare two strings equality, use `isEqualToString:`.

```
BOOL result = [firstString isEqualToString:secondString];
```

To compare with the empty string (`@""`), better use `length`.

```
BOOL result = string.length == 0;
```

Section 9.4: Splitting

You can split a string into an array of parts, divided by **a separator character**.

```
NSString * yourString = @"Stack,Exchange,Network";
NSArray * yourWords = [yourString componentsSeparatedByString:@","];
// Output: @[@"Stack", @"Exchange", @"Network"]
```

If you need to split on a set of **several different delimiters**, use `-[NSString componentsSeparatedByCharactersInSet:]`.

```
NSString * yourString = @"Stack Overflow+Documentation/Objective-C";
NSArray * yourWords = [yourString componentsSeparatedByCharactersInSet:
                       [NSCharacterSet characterSetWithCharactersInString:@"+/"]];
// Output: @[@"Stack Overflow", @"Documentation", @"Objective-C"]`
```

If you need to break a string into its **individual characters**, loop over the length of the string and convert each character into a new string.

```
NSMutableArray * characters = [[NSMutableArray alloc] initWithCapacity:[yourString length]];
for (int i = 0; i < [myString length]; i++) {
    [characters addObject: [NSString stringWithFormat:@"%C",
                                       [yourString characterAtIndex:i]];
}
```

As in the Length Example, keep in mind that a "character" here is a UTF-16 code unit, not necessarily what the user sees as a character. If you use this loop with `@"????????"`, you'll see that it's split into four pieces.

In order to get a list of the user-perceived characters, use `-enumerateSubstringsInRange:options:usingBlock:`.

```
NSMutableArray * characters = [NSMutableArray array];
[yourString enumerateSubstringsInRange:(NSRange){0, [yourString length]}
                               options:NSStringEnumerationByComposedCharacterSequences
                            usingBlock:^(NSString * substring, NSRange r, NSRange s, BOOL * b){
                                [characters addObject:substring];
                            }];
```

This preserves grapheme clusters like the Italian flag as a single substring.

Section 9.5: Searching for a Substring

To search if a String contains a substring, do the following:

```
NSString *myString = @"This is for checking substrings";
NSString *subString = @"checking";

BOOL doesContainSubstring = [myString containsString:subString];  // YES
```

If targeting iOS 7 or OS X 10.9 (or earlier):

```
BOOL doesContainSubstring = ([myString rangeOfString:subString].location != NSNotFound);  // YES
```

Section 9.6: Creation

Simple:

```
NSString *newString = @"My String";
```

From multiple strings:

```
NSString *stringOne = @"Hello";
NSString *stringTwo = @"world";
NSString *newString = [NSString stringWithFormat:@"My message: %@ %@",
                       stringOne, stringTwo];
```

Using Mutable String

```
NSString *stringOne = @"Hello";
NSString *stringTwo = @"World";
NSMutableString *mutableString = [NSMutableString new];
[mutableString appendString:stringOne];
[mutableString appendString:stringTwo];
```

From NSData:

When initializing from `NSData`, an explicit encoding must be provided as `NSString` is not able to guess how characters are represented in the raw data stream. The most common encoding nowadays is UTF-8, which is even a requirement for certain data like JSON.

Avoid using `+[NSString stringWithUTF8String:]` since it expects an explicitly NULL-terminated C-string, which `-[NSData bytes]` does *not* provide.

```
NSString *newString = [[NSString alloc] initWithData:myData encoding:NSUTF8StringEncoding];
```

From NSArray:

```
NSArray *myArray = [NSArray arrayWithObjects:@"Apple", @"Banana", @"Strawberry", @"Kiwi", nil];
NSString *newString = [myArray componentsJoinedByString:@" "];
```

Section 9.7: Changing Case

To convert a String to uppercase, use `uppercaseString`:

```
NSString *myString = @"Emphasize this";
NSLog(@"%@", [myString uppercaseString]; // @"EMPHASIZE THIS"
```

To convert a String to lowercase, use `lowercaseString`:

```
NSString *myString = @"NORMALIZE this";
NSLog(@"%@", [myString lowercaseString]; // @"normalize this"
```

To capitalize the first letter character of each word in a string, use `capitalizedString`:

```
NSString *myString = @"firstname lastname";
NSLog(@"%@", [myString capitalizedString]); // @"Firstname Lastname"
```

Section 9.8: Removing Leading and Trailing Whitespace

```
NSString *someString = @"   Objective-C Language  \n";
NSString *trimmedString = [someString stringByTrimmingCharactersInSet:[NSCharacterSet
whitespaceAndNewlineCharacterSet]];
//Output will be - "Objective-C Language"
```

Method stringByTrimmingCharactersInSet returns a new string made by removing from both ends of the String characters contained in a given character set.

We can also just remove only whitespace or newline

```
// Removing only WhiteSpace
NSString *trimmedWhiteSpace = [someString stringByTrimmingCharactersInSet:[NSCharacterSet
whitespaceCharacterSet]];
//Output will be - "Objective-C Language  \n"

// Removing only NewLine
NSString *trimmedNewLine = [someString stringByTrimmingCharactersInSet:[NSCharacterSet
newlineCharacterSet]];
//Output will be - "   Objective-C Language  "
```

Section 9.9: Joining an Array of Strings

To combine an `NSArray` of `NSString` into a new `NSString`:

```
NSArray *yourWords = @[@"Objective-C", @"is", @"just", @"awesome"];
NSString *sentence = [yourWords componentsJoinedByString:@" "];

// Sentence is now: @"Objective-C is just awesome"
```

Section 9.10: Formatting

The `NSString` formatting supports all the format strings available on the `printf` ANSI-C function. The only addition made by the language is the `%@` symbol used for formatting all the Objective-C objects.

It is possible to format integers

```
int myAge = 21;
NSString *formattedAge = [NSString stringWithFormat:@"I am %d years old", my_age];
```

Or any object subclassed from NSObject

```
NSDate *now = [NSDate date];
NSString *formattedDate = [NSString stringWithFormat:@"The time right now is: %@", now];
```

For a complete list of Format Specifiers, please see: Objective-C, Format Specifiers, Syntax

Section 9.11: Working with C Strings

To convert `NSString` to `const char` use `-[NSString UTF8String]`:

```
NSString *myNSString = @"Some string";
const char *cString = [myNSString UTF8String];
```

You could also use `-[NSString cStringUsingEncoding:]` if your string is encoded with something other than UTF-8.

For the reverse path use `-[NSString stringWithUTF8String:]`:

```
const *char cString = "Some string";
NSString *myNSString = [NSString stringWithUTF8String:cString];
myNSString = @(cString); // Equivalent to the above.
```

Once you have the `const char *`, you can work with it similarly to an array of `chars`:

```
printf("%c\n", cString[5]);
```

If you want to modify the string, make a copy:

```
char *cpy = calloc(strlen(cString)+1, 1);
strncpy(cpy, cString, strlen(cString));
// Do stuff with cpy
free(cpy);
```

Section 9.12: Reversing a NSString Objective-C

```
// myString is "hi"
NSMutableString *reversedString = [NSMutableString string];
NSInteger charIndex = [myString length];
while (charIndex > 0) {
    charIndex--;
    NSRange subStrRange = NSMakeRange(charIndex, 1);
    [reversedString appendString:[myString substringWithRange:subStrRange]];
}
NSLog(@"%@", reversedString); // outputs "ih"
```

Chapter 10: NSDictionary

Section 10.1: Create

```
NSDictionary *dict = [[NSDictionary alloc] initWithObjectsAndKeys:@"value1", @"key1", @"value2",
@"key2", nil];
```

or

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

or using appropriate literal syntax

```
NSDictionary *dict = @{@"key": @"value", @"nextKey": @"nextValue"};
```

Section 10.2: Fast Enumeration

`NSDictionary` can be enumerated using fast enumeration, just like other collection types:

```
NSDictionary stockSymbolsDictionary = @{
                                       @"AAPL": @"Apple",
                                       @"GOOGL": @"Alphabet",
                                       @"MSFT": @"Microsoft",
                                       @"AMZN": @"Amazon"
                                      };

for (id key in stockSymbolsDictionary)
{
    id value = dictionary[key];
    NSLog(@"Key: %@, Value: %@", key, value);
}
```

Because `NSDictionary` is inherently unordered, the order of keys that in the for loop is not guaranteed.

Section 10.3: NSDictionary to NSArray

```
NSDictionary *myDictionary = [[NSDictionary alloc] initWithObjectsAndKeys:@"value1", @"key1",
@"value2", @"key2", nil];
NSArray *copiedArray = myDictionary.copy;
```

Get keys:

```
NSArray *keys = [myDictionary allKeys];
```

Get values:

```
NSArray *values = [myDictionary allValues];
```

Section 10.4: NSDictionary to NSData

```
NSDictionary *myDictionary = [[NSDictionary alloc] initWithObjectsAndKeys:@"value1", @"key1",
@"value2", @"key2", nil];
NSData *myData = [NSKeyedArchiver archivedDataWithRootObject:myDictionary];
```

Reserve path:

```
NSDictionary *myDictionary = (NSDictionary*) [NSKeyedUnarchiver unarchiveObjectWithData:myData];
```

Section 10.5: NSDictionary to JSON

```
NSDictionary *myDictionary = [[NSDictionary alloc] initWithObjectsAndKeys:@"value1", @"key1",
@"value2", @"key2", nil];

NSMutableDictionary *mutableDictionary = [myDictionary mutableCopy];
NSData *data = [NSJSONSerialization dataWithJSONObject:myDictionary
options:NSJSONWritingPrettyPrinted error:nil];
NSString *jsonString = [[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];
```

Section 10.6: Block Based Enumeration

Enumerating dictionaries allows you to run a block of code on each dictionary key-value pair using the method `enumerateKeysAndObjectsUsingBlock:(void (^)(id key, id obj, BOOL *stop))block`

Example:

```
NSDictionary stockSymbolsDictionary = @{
                                        @"AAPL": @"Apple",
                                        @"GOOGL": @"Alphabet",
                                        @"MSFT": @"Microsoft",
                                        @"AMZN": @"Amazon"
                                      };
NSLog(@"Printing contents of dictionary via enumeration");
[stockSymbolsDictionary enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL *stop) {
    NSLog(@"Key: %@, Value: %@", key, obj);
}];
```

Chapter 11: NSDictionary

- @{ key: value, ... }
- [NSDictionary dictionaryWithObjectsAndKeys: value, key, ..., nil];
- dict[*key*] = *value*;
- id *value* = dict[*key*];

Section 11.1: Creating using literals

```
NSDictionary *inventory = @{
    @"Mercedes-Benz SLK250" : @(13),
    @"BMW M3 Coupe" : @(self.BMWM3CoupeInventory.count),
    @"Last Updated" : @"Jul 21, 2016",
    @"Next Update"  : self.nextInventoryUpdateString
};
```

Section 11.2: Creating using dictionaryWithObjectsAndKeys:

```
NSDictionary *inventory = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt:13], @"Mercedes-Benz SLK250",
    [NSNumber numberWithInt:22], @"Mercedes-Benz E350",
    [NSNumber numberWithInt:19], @"BMW M3 Coupe",
    [NSNumber numberWithInt:16], @"BMW X6",
    nil];
```

`nil` must be passed as the last parameter as a sentinel signifying the end.

It's important to remember that when instantiating dictionaries this way the values go first and the keys second. In the example above the strings are the keys and the numbers are the values. The method's name reflects this too: `dictionaryWithObjectsAndKeys`. While this is not incorrect, the more modern way of instantiating dictionaries (with literals) is prefered.

Section 11.3: Creating using plists

```
NSString *pathToPlist = [[NSBundle mainBundle] pathForResource:@"plistName"
    ofType:@"plist"];
NSDictionary *plistDict = [[NSDictionary alloc] initWithContentsOfFile:pathToPlist];
```

Section 11.4: Setting a Value in NSDictionary

There are multiple ways to set a key's object in an NSDictionary, corresponding to the ways you get a value. For instance, to add a lamborghini to a list of cars

Standard

```
[cars setObject:lamborghini forKey:@"Lamborghini"];
```

Just like any other object, call the method of NSDictionary that sets an object of a key, `objectForKey:`. Be careful not to confuse this with `setValue:forKey:`; that's for a completely different thing, Key Value Coding

Shorthand

```
cars[@"Lamborghini"] = lamborghini;
```

This is the syntax that you use for dictionaries in most other languages, such as C#, Java, and Javascript. It's much more convenient than the standard syntax, and arguably more readable (especially if you code in these other languages), but of course, it isn't *standard*. It's also only available in newer versions of Objective C

Section 11.5: Getting a Value from NSDictionary

There are multiple ways to get an object from an NSDictionary with a key. For instance, to get a lamborghini from a list of cars

Standard

```
Car * lamborghini = [cars objectForKey:@"Lamborghini"];
```

Just like any other object, call the method of NSDictionary that gives you an object for a key, `objectForKey:`. Be careful not to confuse this with `valueForKey:`; that's for a completely different thing, Key Value Coding

Shorthand

```
Car * lamborghini = cars[@"Lamborghini"];
```

This is the syntax that you use for dictionaries in most other languages, such as C#, Java, and Javascript. It's much more convenient than the standard syntax, and arguably more readable (especially if you code in these other languages), but of course, it isn't *standard*. It's also only available in newer versions of Objective C

Section 11.6: Check if NSDictionary already has a key or not

Objective c:

```
  //this is the dictionary you start with.
 NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:@"name1", @"Sam",@"name2",
@"Sanju",nil];

//check if the dictionary contains the key you are going to modify.  In this example, @"Sam"
if (dict[@"name1"] != nil) {
    //there is an entry for Key name1
}
else {
    //There is no entry for name1
}
```

Chapter 12: Low-level Runtime Environment

Section 12.1: Augmenting methods using Method Swizzling

The Objective-C runtime allows you to change the implementation of a method at runtime. This is called *method swizzling* and is often used to exchange the implementations of two methods. For example, if the methods `foo` and `bar` are exchanged, sending the message `foo` will now execute the implementation of `bar` and vice versa.

This technique can be used to augment or "patch" existing methods which you cannot edit directly, such as methods of system-provided classes.

In the following example, the `-[NSUserDefaults synchronize]` method is augmented to print the execution time of the original implementation.

IMPORTANT: Many people try to do swizzling using `method_exchangeImplementations`. However, this approach is dangerous if you need to call the method you're replacing, because you'll be calling it using a different selector than it is expecting to receive. As a result, your code can break in strange and unexpected ways—particularly if multiple parties swizzle an object in this way. Instead, you should always do swizzling using `setImplementation` in conjunction with a C function, allowing you to call the method with the original selector.

```
#import "NSUserDefaults+Timing.h"
#import <objc/runtime.h> // Needed for method swizzling

static IMP old_synchronize = NULL;

static void new_synchronize(id self, SEL _cmd);

@implementation NSUserDefaults(Timing)

+ (void)load
{
    Method originalMethod = class_getInstanceMethod([self class], @selector(synchronize:));
    IMP swizzleImp = (IMP)new_synchronize;
    old_synchronize = method_setImplementation(originalMethod, swizzleImp);
}
```

```
@end

static void new_synchronize(id self, SEL _cmd);
{
    NSDate *started;
    BOOL returnValue;

    started = [NSDate date];

    // Call the original implementation, passing the same parameters
    // that this function was called with, including the selector.
    returnValue = old_synchronize(self, _cmd);


    NSLog(@"Writing user defaults took %f seconds.", [[NSDate date]
timeIntervalSinceDate:started]);

    return returnValue;
}

@end
```

If you need to swizzle a method that takes parameters, you just add them as additional parameters to the function. For example:

```
static IMP old_viewWillAppear_animated = NULL;
static void new_viewWillAppear_animated(id self, SEL _cmd, BOOL animated);

...

Method originalMethod = class_getClassMethod([UIViewController class], @selector(viewWillAppear:));
IMP swizzleImp = (IMP)new_viewWillAppear_animated;
old_viewWillAppear_animated = method_setImplementation(originalMethod, swizzleImp);

...

static void new_viewWillAppear_animated(id self, SEL _cmd, BOOL animated)
{
    ...

    old_viewWillAppear_animated(self, _cmd, animated);

    ...
}
```

Section 12.2: Attach object to another existing object (association)

It's possible to attach an object to an existing object as if there was a new property. This is called *association* and allows one to extend existing objects. It can be used to provide storage when adding a property via a class extension or otherwise add additional information to an existing object.

The associated object is automatically released by the runtime once the target object is deallocated.

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

// "Key" for association. Its value is never used and doesn't
// matter. The only purpose of this global static variable is to
// provide a guaranteed unique value at runtime: no two distinct
// global variables can share the same address.
static char key;
```

```
id target = ...;
id payload = ...;
objc_setAssociateObject(target, &key, payload, OBJC_ASSOCIATION_RETAIN);
// Other useful values are OBJC_ASSOCIATION_COPY
// and OBJ_ASSOCIATION_ASSIGN

id queryPayload = objc_getAssociatedObject(target, &key);
```

Section 12.3: Calling methods directly

If you need to call an Objective-C method from C code, you have two ways: using `objc_msgSend`, or obtaining the `IMP` (method implementation function pointer) and calling that.

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

@implementation Example

- (double)negate:(double)value {
    return -value;
}

- (double)invert:(double)value {
    return 1 / value;
}

@end

// Calls the selector on the object. Expects the method to have one double argument and return a
double.
double performSelectorWithMsgSend(id object, SEL selector, double value) {
    // We declare pointer to function and cast `objc_msgSend` to expected signature.
    // WARNING: This step is important! Otherwise you may get unexpected results!
    double (*msgSend)(id, SEL, double) = (typeof(msgSend)) &objc_msgSend;

    // The implicit arguments of self and _cmd need to be passed in addition to any explicit
arguments.
    return msgSend(object, selector, value);
}

// Does the same as the above function, but by obtaining the method's IMP.
double performSelectorWithIMP(id object, SEL selector, double value) {
    // Get the method's implementation.
    IMP imp = class_getMethodImplementation([self class], selector);

    // Cast it so the types are known and ARC can work correctly.
    double (*callableImp)(id, SEL, double) = (typeof(callableImp)) imp;

    // Again, you need the explicit arguments.
    return callableImp(object, selector, value);
}

int main() {
    Example *e = [Example new];

    // Invoke negation, result is -4
    double x = performSelectorWithMsgSend(e, @selector(negate:), 4);

    // Invoke inversion, result is 0.25
    double y = performSelectorWithIMP(e, @selector(invert:), 4);
}
```

`objc_msgSend` works by obtaining the IMP for the method and calling that. The `IMP`s for the last several methods

called are cached, so if you're sending an Objective-C message in a very tight loop you can get acceptable performance. In some cases, manually caching the IMP can give slightly better performance, although this is a last resort optimization.

Chapter 13: NSArray

Section 13.1: Sorting Arrays

The most flexible ways to sort an array is with the sortedArrayUsingComparator: method. This accepts an **^NSComparisonResult(id obj1, id obj2) block**.

```
Return Value            Description
NSOrderedAscending      obj1 comes before obj2
NSOrderedSame           obj1 and obj2 have no order
NSOrderedDescending     obj1 comes after obj2
```

Example:

```
   NSArray *categoryArray = @[@"Apps", @"Music", @"Songs",
                      @"iTunes", @"Books", @"Videos"];

    NSArray *sortedArray = [categoryArray sortedArrayUsingComparator:
^NSComparisonResult(id obj1, id obj2) {
    if ([obj1 length] < [obj2 length]) {
        return NSOrderedAscending;
    } else if ([obj1 length] > [obj2 length]) {
        return NSOrderedDescending;
    } else {
        return NSOrderedSame;
    }
  }];

 NSLog(@"%@", sortedArray);
```

Section 13.2: Filter NSArray and NSMutableArray

```
NSMutableArray *array =
    [NSMutableArray arrayWithObjects:@"Ken", @"Tim", @"Chris", @"Steve",@"Charlie",@"Melissa",
nil];

NSPredicate *bPredicate =
    [NSPredicate predicateWithFormat:@"SELF beginswith[c] 'c'"];
NSArray *beginWithB =
    [array filteredArrayUsingPredicate:bPredicate];
// beginWith "C" contains { @"Chris", @"Charlie" }.

NSPredicate *sPredicate =
    [NSPredicate predicateWithFormat:@"SELF contains[c] 'a'"];
[array filterUsingPredicate:sPredicate];
// array now contains { @"Charlie", @"Melissa" }
```

Section 13.3: Creating NSArray instances

```
NSArray *array1 = [NSArray arrayWithObjects:@"one", @"two", @"three", nil];
NSArray *array2 = @[@"one", @"two", @"three"];
```

Chapter 14: NSURL

Section 14.1: Create

From NSString:

```
NSString *urlString = @"https://www.stackoverflow.com";
NSURL *myUrl = [NSURL URLWithString: urlString];
```

You can also use the following methods:

```
- initWithString:
+ URLWithString:relativeToURL:
- initWithString:relativeToURL:
+ fileURLWithPath:isDirectory:
- initFileURLWithPath:isDirectory:
+ fileURLWithPath:
- initFileURLWithPath:
 Designated Initializer
+ fileURLWithPathComponents:
+ URLByResolvingAliasFileAtURL:options:error:
+ URLByResolvingBookmarkData:options:relativeToURL:bookmarkDataIsStale:error:
- initByResolvingBookmarkData:options:relativeToURL:bookmarkDataIsStale:error:
+ fileURLWithFileSystemRepresentation:isDirectory:relativeToURL:
- getFileSystemRepresentation:maxLength:
- initFileURLWithFileSystemRepresentation:isDirectory:relativeToURL:
```

Section 14.2: Compare NSURL

```
NSString *urlString = @"https://www.stackoverflow.com";

NSURL *myUrl = [NSURL URLWithString: urlString];
NSURL *myUrl2 = [NSURL URLWithString: urlString];

if ([myUrl isEqual:myUrl2]) return YES;
```

Section 14.3: Modifying and Converting a File URL with removing and appending path

1. URLByDeletingPathExtension:

If the receiver represents the root path, this property contains a copy of the original URL. If the URL has multiple path extensions, only the last one is removed.

2. URLByAppendingPathExtension:

Returns a new URL made by appending a path extension to the original URL.

Example:

```
    NSUInteger count = 0;
        NSString *filePath = nil;
        do {
            NSString *extension = ( NSString *)UTTypeCopyPreferredTagWithClass((
CFStringRef)AVFileTypeQuickTimeMovie, kUTTagClassFilenameExtension);
            NSString *fileNameNoExtension = [[asset.defaultRepresentation.url
URLByDeletingPathExtension] lastPathComponent];//Delete is used
            NSString *fileName = [NSString stringWithFormat:@"%@-%@-%u",fileNameNoExtension ,
AVAssetExportPresetLowQuality, count];
            filePath = NSTemporaryDirectory();
            filePath = [filePath stringByAppendingPathComponent:fileName];//Appending is used
            filePath = [filePath stringByAppendingPathExtension:extension];
            count++;
```

```
        } while ([[NSFileManager defaultManager] fileExistsAtPath:filePath]);

        NSURL *outputURL = [NSURL fileURLWithPath:filePath];
```

Chapter 15: Methods

- `-` or `+`: The type of method. Instance or class?
- (): Where the return type goes. Use void if you don't want to return anything!
- Next is the name of the method. Use camelCase and make the name easy to remember an understand.
- If your method needs parameters, now is the time! The first parameter come right after the name of the function like this `:(type)parameterName`. All the other parameters are done this way `parameterLabel:(type)parameterName`
- What does your method do? Put it all here, in the curly braces {}!

Section 15.1: Class methods

A class method is called on the class the method belongs to, not an instance of it. This is possible because Objective-C classes are also objects. To denote a method as a class method, change the `-` to a `+`:

```
+ (void)hello {
  NSLog(@"Hello World");
}
```

Section 15.2: Pass by value parameter passing

In pass by value of parameter passing to a method, actual parameter value is copied to formal parameter value. So actual parameter value will not change after returning from called function.

```
@interface SwapClass : NSObject

-(void) swap:(NSInteger)num1 andNum2:(NSInteger)num2;

@end

@implementation SwapClass

-(void) num:(NSInteger)num1 andNum2:(NSInteger)num2{
    int temp;
    temp = num1;
    num1 = num2;
    num2 = temp;
}
@end
```

Calling the methods:

```
NSInteger a = 10, b =20;
SwapClass *swap = [[SwapClass alloc]init];
NSLog(@"Before calling swap: a=%d,b=%d",a,b);
[swap num:a andNum2:b];
```

```
NSLog(@"After calling swap: a=%d,b=%d",a,b);
```

Output:

```
2016-07-30 23:55:41.870 Test[5214:81162] Before calling swap: a=10,b=20
2016-07-30 23:55:41.871 Test[5214:81162] After calling swap: a=10,b=20
```

Section 15.3: Pass by reference parameter passing

In pass by reference of parameter passing to a method, address of actual parameter is passed to formal parameter. So actual parameter value will be changed after returning from called function.

```
@interface SwapClass : NSObject

-(void) swap:(int)num1 andNum2:(int)num2;

@end

@implementation SwapClass

-(void) num:(int*)num1 andNum2:(int*)num2{
    int temp;
    temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
@end
```

Calling the methods:

```
int a = 10, b =20;
SwapClass *swap = [[SwapClass alloc]init];
NSLog(@"Before calling swap: a=%d,b=%d",a,b);
[swap num:&a andNum2:&b];
NSLog(@"After calling swap: a=%d,b=%d",a,b);
```

Output:

```
2016-07-31 00:01:47.067 Test[5260:83491] Before calling swap: a=10,b=20
2016-07-31 00:01:47.070 Test[5260:83491] After calling swap: a=20,b=10
```

Section 15.4: Method parameters

If you want to pass in values to a method when it is called, you use parameters:

```
- (int)addInt:(int)intOne toInt:(int)intTwo {
  return intOne + intTwo;
}
```

The colon (`:`) separates the parameter from the method name.

The parameter type goes in the parentheses `(int)`.

The parameter name goes after the parameter type.

Section 15.5: Create a basic method

This is how to create a basic method that logs 'Hello World" to the console:

```
- (void)hello {
  NSLog(@"Hello World");
```

```
}
```

The `-` at the beginning denotes this method as an instance method.

The `(void)` denotes the return type. This method doesn't return anything, so you enter `void`.

The 'hello' is the name of the method.

Everything in the `{}` is the code run when the method is called.

Section 15.6: Return values

When you want to return a value from a method, you put the type you want to return in the first set of parentheses.

```
- (NSString)returnHello {
  return @"Hello World";
}
```

The value you want to return goes after the `return` keyword;

Section 15.7: Calling methods

Calling an instance method:

```
[classInstance hello];

 @interface Sample
 -(void)hello; // exposing the class Instance method
 @end

 @implementation Sample
     -(void)hello{
        NSLog(@"hello");
      }
 @end
```

Calling an instance method on the current instance:

```
[self hello];

@implementation Sample

     -(void)otherMethod{
       [self hello];
     }

     -(void)hello{
        NSLog(@"hello");
      }
 @end
```

Calling a method that takes arguments:

```
[classInstance addInt:1 toInt:2];

 @implementation Sample
     -(void)add:(NSInteger)add to:(NSInteger)to
        NSLog(@"sum = %d",(add+to));
      }
 @end
```

Calling a class method:

```
[Class hello];

 @interface Sample
 +(void)hello; // exposing the class method
 @end

 @implementation Sample
     +(void)hello{
        NSLog(@"hello");
      }
 @end
```

Section 15.8: Instance methods

An instance method is a method that's available on a particular instance of a class, after the instance has been instantiated:

```
MyClass *instance = [MyClass new];
[instance someInstanceMethod];
```

Here's how you define one:

```
@interface MyClass : NSObject

- (void)someInstanceMethod; // "-" denotes an instance method

@end

@implementation MyClass

- (void)someInstanceMethod {
    NSLog(@"Whose idea was it to have a method called \"someInstanceMethod\"?");
}

@end
```

Chapter 16: Error Handling

- NSAssert(condition, fmtMessage, *arg1, arg2, ...*) (args in italics are optional) -- Asserts that *condition* evaluates to a true value. If it doesn't than the assertion will raise an exception (NSAssertionException), with the *fmtMessage* formatted with the args provided

Section 16.1: Error & Exception handling with try catch block

Exceptions represent programmer-level bugs like trying to access an array element that doesn't exist.

Errors are user-level issues like trying load a file that doesn't exist. Because errors are expected during the normal execution of a program.

Example:

```
    NSArray *inventory = @[@"Sam",
                           @"John",
                           @"Sanju"];
    int selectedIndex = 3;
    @try {
        NSString * name = inventory[selectedIndex];
        NSLog(@"The selected Name is: %@", name);
```

```
    } @catch(NSException *theException) {
        NSLog(@"An exception occurred: %@", theException.name);
        NSLog(@"Here are some details: %@", theException.reason);
    } @finally {
        NSLog(@"Executing finally block");
    }
```

OUTPUT:

An exception occurred: NSRangeException

Here are some details: *** -[__NSArrayI objectAtIndex:]: index 3 beyond bounds [0 .. 2]

Executing finally block

Section 16.2: Asserting

```
@implemenetation Triangle

...

-(void)setAngles:(NSArray *)_angles {
    self.angles = _angles;

    NSAssert((self.angles.count == 3), @"Triangles must have 3 angles. Array '%@' has %i",
self.angles, (int)self.angles.count);

    CGFloat angleA = [self.angles[0] floatValue];
    CGFloat angleB = [self.angles[1] floatValue];
    CGFloat angleC = [self.angles[2] floatValue];
    CGFloat sum = (angleA + angleB + angleC);
    NSAssert((sum == M_PI), @"Triangles' angles must add up to pi radians (180°). This triangle's
angles add up to %f radians (%f°)", (float)sum, (float)(sum * (180.0f / M_PI)));
}
```

These assertions make sure that you don't give a triangle incorrect angles, by throwing an exception if you do. If they didn't throw an exception than the triangle, not being a true triangle at all, might cause some bugs in later code.

Chapter 17: Enums

- typedef NS_ENUM(type, name) {...} -- *type* is the type of enumeration and *name* is the name of the enum. values are in "...". This creates a basic enum and a type to go with it; programs like Xcode will assume a variable with the enum type has one of the enum values

Section 17.1: typedef enum declaration in Objective-C

A enum declares a set of ordered values - the typedef just adds a handy name to this. The 1st element is 0 etc.

```
typedef enum {
        Monday=1,
        Tuesday,
        Wednesday

    } WORKDAYS;

    WORKDAYS today = Monday;//value 1
```

Section 17.2: Converting C++ std::vector<Enum> to an Objective-C Array

Many C++ libraries use enums and return/receive data using vectors that contain enums. As C enums are not Objective-C objects, Objective-C collections cannot be used directly with C enums. The example below deals with this by using a combination of an NSArray and generics and a wrapper object for the array. This way, the collection can be explicit about the data type and there is no worry about possible memory leaks with C arrays Objective-C objects are used.

Here is the C enum & Objective-C equivalent object:

```
typedef enum
{
  Error0 = 0,
  Error1 = 1,
  Error2 = 2
} MyError;

@interface ErrorEnumObj : NSObject

@property (nonatomic) int intValue;

+ (instancetype) objWithEnum:(MyError) myError;
- (MyError) getEnumValue;

@end

@implementation ErrorEnumObj

+ (instancetype) objWithEnum:(MyError) error
{
    ErrorEnumObj * obj = [ErrorEnumObj new];
    obj.intValue = (int)error;
    return obj;
}

- (MyError) getEnumValue
{
    return (MyError)self.intValue;
}

@end
```

And here is a possible use of it in Objective-C++ (the resulting NSArray can be used in Objective-C only files as no C++ is used).

```
class ListenerImpl : public Listener
{
public:
    ListenerImpl(Listener* listener) : _listener(listener) {}
    void onError(std::vector<MyError> errors) override
    {
        NSMutableArray<ErrorEnumObj *> * array = [NSMutableArray<ErrorEnumObj *> new];
        for (auto&& myError : errors)
        {
            [array addObject:[ErrorEnumObj objWithEnum:myError]];
        }
        [_listener onError:array];
    }

private:
    __weak Listener* _listener;
```

```
}
```

If this kind of solution is to be used on multiple enums, the creation of the EnumObj (declaration & implementation) can be done using a macro (to create a template like solution).

Section 17.3: Defining an enum

Enums are defined by the following the syntax above.

```
typedef NS_ENUM(NSUInteger, MyEnum) {
    MyEnumValueA,
    MyEnumValueB,
    MyEnumValueC,
};
```

You also can set your own raw-values to the enumeration types.

```
typedef NS_ENUM(NSUInteger, MyEnum) {
    MyEnumValueA = 0,
    MyEnumValueB = 5,
    MyEnumValueC = 10,
};
```

You can also specify on the first value and all the following will use it with increment:

```
typedef NS_ENUM(NSUInteger, MyEnum) {
    MyEnumValueA = 0,
    MyEnumValueB,
    MyEnumValueC,
};
```

Variables of this enum can be created by `MyEnum enumVar = MyEnumValueA`.

Chapter 18: NSData

Section 18.1: Create

From NSString:

```
NSString *str = @"Hello world";
NSData *data = [str dataUsingEncoding:NSUTF8StringEncoding];
```

From Int:

```
int i = 1;
NSData *data = [NSData dataWithBytes: &i length: sizeof(i)];
```

You can also use the following methods:

```
+ dataWithContentsOfURL:
+ dataWithContentsOfURL:options:error:
+ dataWithData:
- initWithBase64EncodedData:options:
- initWithBase64EncodedString:options:
- initWithBase64Encoding:
- initWithBytesNoCopy:length:
- initWithBytesNoCopy:length:deallocator:
- initWithBytesNoCopy:length:freeWhenDone:
- initWithContentsOfFile:
- initWithContentsOfFile:options:error:
- initWithContentsOfMappedFile:
```

```
- initWithContentsOfURL:
- initWithContentsOfURL:options:error:
- initWithData:
```

Section 18.2: NSData and Hexadecimal String

Get NSData from Hexadecimal String

```
+ (NSData *)dataFromHexString:(NSString *)string
{
    string = [string lowercaseString];
    NSMutableData *data= [NSMutableData new];
    unsigned char whole_byte;
    char byte_chars[3] = {'\0','\0','\0'};
    int i = 0;
    int length = (int) string.length;
    while (i < length-1) {
        char c = [string characterAtIndex:i++];
        if (c < '0' || (c > '9' && c < 'a') || c > 'f')
            continue;
        byte_chars[0] = c;
        byte_chars[1] = [string characterAtIndex:i++];
        whole_byte = strtol(byte_chars, NULL, 16);
        [data appendBytes:&whole_byte length:1];
    }
    return data;
}
```

Get Hexadecimal String from data:

```
+ (NSString *)hexStringForData:(NSData *)data
{
    if (data == nil) {
        return nil;
    }

    NSMutableString *hexString = [NSMutableString string];

    const unsigned char *p = [data bytes];

    for (int i=0; i < [data length]; i++) {
        [hexString appendFormat:@"%02x", *p++];
    }

    return hexString;
}
```

Section 18.3: Get NSData lengh

```
NSString *filePath = [[NSFileManager defaultManager] pathForRessorce: @"data" ofType:@"txt"];
NSData *data = [NSData dataWithContentsOfFile:filePath];
int len = [data length];
```

Section 18.4: Encoding and decoding a string using NSData Base64

Encoding

```
//Create a Base64 Encoded NSString Object
NSData *nsdata = [@"iOS Developer Tips encoded in Base64" dataUsingEncoding:NSUTF8StringEncoding];

// Get NSString from NSData object in Base64
NSString *base64Encoded = [nsdata base64EncodedStringWithOptions:0];
```

```
// Print the Base64 encoded string
NSLog(@"Encoded: %@", base64Encoded);
```

Decoding:

```
// NSData from the Base64 encoded str
NSData *nsdataFromBase64String = [[NSData alloc]initWithBase64EncodedString:base64Encoded
options:0];

// Decoded NSString from the NSData
NSString *base64Decoded = [[NSString alloc] initWithData:nsdataFromBase64String
encoding:NSUTF8StringEncoding];
NSLog(@"Decoded: %@", base64Decoded);
```

Chapter 19: Random Integer

Section 19.1: Basic Random Integer

The `arc4random_uniform()` function is the simplest way to get high-quality random integers. As per the manual:

arc4random_uniform(upper_bound) will return a uniformly distributed random number less than upper_bound.

arc4random_uniform() is recommended over constructions like "arc4random() % upper_bound" as it avoids "modulo bias" when the upper bound is not a power of two.

```
uint32_t randomInteger = arc4random_uniform(5); // A random integer between 0 and 4
```

Section 19.2: Random Integer within a Range

The following code demonstrates usage of `arc4random_uniform()` to generate a random integer between 3 and 12:

```
uint32_t randomIntegerWithinRange = arc4random_uniform(10) + 3; // A random integer between 3 and 12
```

This works to create a range because `arc4random_uniform(10)` returns an integer between 0 and 9. Adding 3 to this random integer produces a range between `0 + 3` and `9 + 3`.

Chapter 20: Properties

- @property (*optional_attributes, ...*) *type identifier*;
- @synthesize *identifier* = *optional_backing_ivar*;
- @dynamic *identifier*;

Attribute	Description
`atomic`	*Implicit.* Enables synchronization in synthesized accessor methods.
`nonatomic`	Disables synchronization in the synthesized accessor methods.
`readwrite`	*Implicit.* Synthesizes getter, setter and backing ivar.
`readonly`	Synthesizes only the getter method and backing ivar, which can be assigned directly.
`getter=`*name*	Specifies the name of getter method, implicit is `propertyName`.
`setter=`*name*	Specifies the name of setter method, implicity is `setPropertyName:`. Colon `:` must be a part of the name.
`strong`	*Implicit for objects under ARC*. The backing ivar is synthesized using `__strong`, which prevents deallocation of referenced object.
`retain`	Synonym for `strong`.
`copy`	Same as `strong`, but the synthesized setter also calls `-copy` on the new value.
`unsafe_unretained`	*Implicit, except for objects under ARC.* The backing ivar is synthesized using `__unsafe_unretained`, which (for obejcts) results in dangling pointer once the referenced object deallocates.
`assign`	Synonym for `unsafe_unretained`. Suitable for non-object types.

`weak`	Backing ivar is synthesized using `__weak`, so the value will be nullified once the referenced object is deallocated.
`class`	Property accessors are synthesized as class methods, instead of instance methods. No backing storage is synthesized.
`nullable`	The property accepts `nil` values. Mainly used for Swift bridging.
`nonnull`	The property doesn't accept `nil` values. Mainly used for Swift bridging.
`null_resettable`	The property accepts `nil` values in setter, but never returns `nil` values from getter. Your custom implementation of getter or setter must ensure this behavior. Mainly used for Swift bridging.
`null_unspecified`	*Implicit.* The property doesn't specify handling of `nil` values. Mainly used for Swift bridging.

Section 20.1: Custom getters and setters

The default property getters and setters can be overridden:

```
@interface TestClass

@property NSString *someString;

@end

@implementation TestClass

// override the setter to print a message
- (void)setSomeString:(NSString *)newString {
    NSLog(@"Setting someString to %@", newString);
    // Make sure to access the ivar (default is the property name with a _
    // at the beginning) because calling self.someString would call the same
    // method again leading to an infinite recursion
    _someString = newString;
}

- (void)doSomething {
    // The next line will call the setSomeString: method
    self.someString = @"Test";
}

@end
```

This can be useful to provide, for example, lazy initialization (by overriding the getter to set the initial value if it has not yet been set):

```
- (NSString *)someString {
    if (_someString == nil) {
        _someString = [self getInitialValueForSomeString];
    }
    return _someString;
}
```

You can also make a property that computes its value in the getter:

```
@interface Circle : NSObject

@property CGPoint origin;
@property CGFloat radius;
@property (readonly) CGFloat area;

@end

@implementation Circle
```

```
- (CGFloat)area {
    return M_PI * pow(self.radius, 2);
}

@end
```

Section 20.2: Properties that cause updates

This object, `Shape` has a property `image` that depends on `numberOfSides` and `sideWidth`. If either one of them is set, than the `image` has to be recalculated. But recalculation is presumably long, and only needs to be done once if both properties are set, so the `Shape` provides a way to set both properties and only recalculate once. This is done by setting the property ivars directly.

In `Shape.h`

```
@interface Shape {
    NSUInteger numberOfSides;
    CGFloat sideWidth;

    UIImage * image;
}

// Initializer that takes initial values for the properties.
- (instancetype)initWithNumberOfSides:(NSUInteger)numberOfSides withWidth:(CGFloat)width;

// Method that allows to set both properties in once call.
// This is useful if setting these properties has expensive side-effects.
// Using a method to set both values at once allows you to have the side-
// effect executed only once.
- (void)setNumberOfSides:(NSUInteger)numberOfSides andWidth:(CGFloat)width;

// Properties using default attributes.
@property NSUInteger numberOfSides;
@property CGFloat sideWidth;

// Property using explicit attributes.
@property(strong, readonly) UIImage * image;

@end
```

In `Shape.m`

```
@implementation AnObject

// The variable name of a property that is auto-generated by the compiler
// defaults to being the property name prefixed with an underscore, for
// example "_propertyName". You can change this default variable name using
// the following statement:
// @synthesize propertyName = customVariableName;

- (id)initWithNumberOfSides:(NSUInteger)numberOfSides withWidth:(CGFloat)width {
    if ((self = [self init])) {
       [self setNumberOfSides:numberOfSides andWidth:width];
    }

    return self;
}

- (void)setNumberOfSides:(NSUInteger)numberOfSides {
    _numberOfSides = numberOfSides;
```

```
    [self updateImage];
}

- (void)setSideWidth:(CGFloat)sideWidth {
    _sideWidth = sideWidth;

    [self updateImage];
}

- (void)setNumberOfSides:(NSUInteger)numberOfSides andWidth:(CGFloat)sideWidth {
    _numberOfSides = numberOfSides;
    _sideWidth = sideWidth;

    [self updateImage];
}

// Method that does some post-processing once either of the properties has
// been updated.
- (void)updateImage {
    ...
}

@end
```

When properties are assigned to (using `object.property = value`), the setter method `setProperty:` is called. This setter, even if provided by `@synthesize`, can be overridden, as it is in this case for `numberOfSides` and `sideWidth`. However, if you set an property's ivar directly (through `property` if the object is self, or `object->property`), it doesn't call the getter or setter, allowing you to do things like multiple property sets that only call one update or bypass side-effects caused by the setter.

Section 20.3: What are properties?

Here is an example class which has a couple of instance variables, without using properties:

```
@interface TestClass : NSObject {
    NSString *_someString;
    int _someInt;
}

-(NSString *)someString;
-(void)setSomeString:(NSString *)newString;

-(int)someInt;
-(void)setSomeInt:(NSString *)newInt;

@end


@implementation TestClass

-(NSString *)someString {
    return _someString;
}

-(void)setSomeString:(NSString *)newString {
    _someString = newString;
}

-(int)someInt {
    return _someInt;
}
```

```
-(void)setSomeInt:(int)newInt {
    _someInt = newInt;
}

@end
```

This is quite a lot of boilerplate code to create a simple instance variable. You have to create the instance variable & create accessor methods which do nothing except set or return the instance variable. So with Objective-C 2.0, Apple introduced properties, which auto-generate some or all of the boilerplate code.

Here is the above class rewritten with properties:

```
@interface TestClass

@property NSString *someString;
@property int someInt;

@end


@implementation testClass

@end
```

A property is an instance variable paired with auto-generated getters and setters. For a property called `someString`, the getter and setter are called `someString` and `setSomeString:` respectively. The name of the instance variable is, by default, the name of the property prefixed with an underscore (so the instance variable for `someString` is called `_someString`, but this can be overridden with an `@synthesize` directive in the `@implementation` section:

```
@synthesize someString=foo;    //names the instance variable "foo"
@synthesize someString;    //names it "someString"
@synthesize someString=_someString;        //names it "_someString"; the default if
                                           //there is no @synthesize directive
```

Properties can be accessed by calling the getters and setters:

```
[testObject setSomeString:@"Foo"];
NSLog(@"someInt is %d", [testObject someInt]);
```

They can also be accessed using dot notation:

```
testObject.someString = @"Foo";
NSLog(@"someInt is %d", testObject.someInt);
```

Chapter 21: NSDate

Section 21.1: Convert NSDate that is composed from hour and minute (only) to a full NSDate

There are many cases when one has created an NSDate from only an hour and minute format, i.e: 08:12

The downside for this situation is that your NSDate is almost completely "naked" and what you need to do is to create: day, month, year, second and time zone in order to this object to "play along" with other NSDate types.

For the sake of the example let's say that hourAndMinute is the NSDate type that is composed from hour and minute format:

```
NSDateComponents *hourAndMintuteComponents = [calendar components:NSCalendarUnitHour |
NSCalendarUnitMinute
```

```
                                                        fromDate:hourAndMinute];
NSDateComponents *componentsOfDate = [[NSCalendar currentCalendar] components:NSCalendarUnitDay |
NSCalendarUnitMonth | NSCalendarUnitYear
                                                                    fromDate:[NSDate date]];

NSDateComponents *components = [[NSDateComponents alloc] init];
[components setDay: componentsOfDate.day];
[components setMonth: componentsOfDate.month];
[components setYear: componentsOfDate.year];
[components setHour: [hourAndMintuteComponents hour]];
[components setMinute: [hourAndMintuteComponents minute]];
[components setSecond: 0];
[calendar setTimeZone: [NSTimeZone defaultTimeZone]];

NSDate *yourFullNSDateObject = [calendar dateFromComponents:components];
```

Now your object is the total opposite of being "naked".

Section 21.2: Converting NSDate to NSString

If ww have NSDate object, and we want to convert it into NSString. There are different types of Date strings. How we can do that?, It is very simple. Just 3 steps.

1. Create NSDateFormatter Object

```
NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
```

2. Set the date format in which you want your string.

```
dateFormatter.dateFormat = @"yyyy-MM-dd 'at' HH:mm";
```

3. Now, get the formatted string

```
NSDate *date = [NSDate date]; // your NSDate object
NSString *dateString = [dateFormatter stringFromDate:date];
```

This will give output something like this: `2001-01-02 at 13:00`

Note:

Creating an `NSDateFormatter` instance is an expensive operation, so it is recommended to create it once and reuse when possible.

Section 21.3: Creating an NSDate

The `NSDate` class provides methods for creating `NSDate` objects corresponding to a given date and time. An `NSDate` can be initialized using the designated initializer, which:

Returns an `NSDate` object initialized relative to 00:00:00 UTC on 1 January 2001 by a given number of seconds.

```
NSDate *date = [[NSDate alloc] initWithTimeIntervalSinceReferenceDate:100.0];
```

`NSDate` also provides an easy way to create an `NSDate` equal to the current date and time:

```
NSDate *now = [NSDate date];
```

It is also possible to create an `NSDate` a given amount of seconds from the current date and time:

```
NSDate *tenSecondsFromNow = [NSDate dateWithTimeIntervalSinceNow:10.0];
```

Section 21.4: Date Comparison

There are 4 methods for comparing `NSDate`s in Objective-C:

- `- (BOOL)isEqualToDate:(NSDate *)anotherDate`
- `- (NSDate *)earlierDate:(NSDate *)anotherDate`
- `- (NSDate *)laterDate:(NSDate *)anotherDate`
- `- (NSComparisonResult)compare:(NSDate *)anotherDate`

Consider the following example using 2 dates, `NSDate date1 = July 7, 2016` and `NSDate date2 = July 2, 2016`:

```
NSDateComponents *comps1 = [[NSDateComponents alloc]init];
comps.year = 2016;
comps.month = 7;
comps.day = 7;

NSDateComponents *comps2 = [[NSDateComponents alloc]init];
    comps.year = 2016;
    comps.month = 7;
    comps.day = 2;

NSDate* date1 = [calendar dateFromComponents:comps1]; //Initialized as July 7, 2016
NSDate* date2 = [calendar dateFromComponents:comps2]; //Initialized as July 2, 2016
```

Now that the `NSDate`s are created, they can be compared:

```
if ([date1 isEqualToDate:date2]) {
    //Here it returns false, as both dates are not equal
}
```

We can also use the `earlierDate:` and `laterDate:` methods of the `NSDate` class:

```
NSDate *earlierDate = [date1 earlierDate:date2];//Returns the earlier of 2 dates. Here earlierDate
will equal date2.
NSDate *laterDate = [date1 laterDate:date2];//Returns the later of 2 dates. Here laterDate will
equal date1.
```

Lastly, we can use `NSDate`'s `compare:` method:

```
NSComparisonResult result = [date1 compare:date2];
    if (result == NSOrderedAscending) {
        //Fails
        //Comes here if date1 is earlier than date2. In our case it will not come here.
    }else if (result == NSOrderedSame){
        //Fails
        //Comes here if date1 is the same as date2. In our case it will not come here.
    }else{//NSOrderedDescending

        //Succeeds
        //Comes here if date1 is later than date2. In our case it will come here
    }
```

Chapter 22: NSPredicate

- CONTAINS operator : It allows to filter objects with matching subset.

  `NSPredicate *filterByName = [NSPredicate predicateWithFormat:@"self.title CONTAINS[cd] %@",@"Tom"];`

- LIKE : Its simple comparison filter.

```
NSPredicate *filterByNameCIS = [NSPredicate predicateWithFormat:@"self.title LIKE[cd]
%@",@"Tom and Jerry"];
```

- = operator : It returns all the objects with matching filter value.

```
NSPredicate *filterByNameCS = [NSPredicate predicateWithFormat:@"self.title = %@",@"Tom and
Jerry"];
```

- IN operator : It allows you to filter objects with specific filter set.

```
NSPredicate *filterByIds = [NSPredicate predicateWithFormat:@"self.id IN
%@",@[@"7CDF6D22-8D36-49C2-84FE-E31EECCECB79", @"7CDF6D22-8D36-49C2-84FE-E31EECCECB76"]];
```

- NOT IN operator : It allows you to find Inverse objects with specific set.

```
NSPredicate *filterByNotInIds = [NSPredicate predicateWithFormat:@"NOT (self.id IN
%@)",@[@"7CDF6D22-8D36-49C2-84FE-E31EECCECB79", @"7CDF6D22-8D36-49C2-84FE-E31EECCECB76"]];
```

Section 22.1: Filter By Name

```
NSArray *array = @[
                    @{
                        @"id": @"7CDF6D22-8D36-49C2-84FE-E31EECCECB71",
                        @"title": @"Jackie Chan Strike Movie",
                        @"url": @"http://abc.com/playback.m3u8",
                        @"thumbnailURL": @"http://abc.com/thumbnail.png",
                        @"isMovie" : @1
                    },
                    @{
                        @"id": @"7CDF6D22-8D36-49C2-84FE-E31EECCECB72",
                        @"title": @"Sherlock homes",
                        @"url": @"http://abc.com/playback.m3u8",
                        @"thumbnailURL": @"http://abc.com/thumbnail.png",
                        @"isMovie" : @0
                    },
                    @{
                        @"id": @"7CDF6D22-8D36-49C2-84FE-E31EECCECB73",
                        @"title": @"Titanic",
                        @"url": @"http://abc.com/playback.m3u8",
                        @"thumbnailURL": @"http://abc.com/thumbnail.png",
                        @"isMovie" : @1
                    },
                    @{
                        @"id": @"7CDF6D22-8D36-49C2-84FE-E31EECCECB74",
                        @"title": @"Star Wars",
                        @"url": @"http://abc.com/playback.m3u8",
                        @"thumbnailURL": @"http://abc.com/thumbnail.png",
                        @"isMovie" : @1
                    },
                    @{
                        @"id": @"7CDF6D22-8D36-49C2-84FE-E31EECCECB75",
                        @"title": @"Pokemon",
                        @"url": @"http://abc.com/playback.m3u8",
                        @"thumbnailURL": @"http://abc.com/thumbnail.png",
                        @"isMovie" : @0
                    },
```

```
                    @{
                        @"id": @"7CDF6D22-8D36-49C2-84FE-E31EECCECB76",
                        @"title": @"Avatar",
                        @"url": @"http://abc.com/playback.m3u8",
                        @"thumbnailURL": @"http://abc.com/thumbnail.png",
                        @"isMovie" : @1
                    },
                    @{
                        @"id": @"7CDF6D22-8D36-49C2-84FE-E31EECCECB77",
                        @"title": @"Popey",
                        @"url": @"http://abc.com/playback.m3u8",
                        @"thumbnailURL": @"http://abc.com/thumbnail.png",
                        @"isMovie" : @1
                    },
                    @{
                        @"id": @"7CDF6D22-8D36-49C2-84FE-E31EECCECB78",
                        @"title": @"Tom and Jerry",
                        @"url": @"http://abc.com/playback.m3u8",
                        @"thumbnailURL": @"http://abc.com/thumbnail.png",
                        @"isMovie" : @1
                    },
                    @{
                        @"id": @"7CDF6D22-8D36-49C2-84FE-E31EECCECB79",
                        @"title": @"The wolf",
                        @"url": @"http://abc.com/playback.m3u8",
                        @"thumbnailURL": @"http://abc.com/thumbnail.png",
                        @"isMovie" : @1
                    }
                    ];

// *** Case Insensitive comparision with excate title match ***
NSPredicate *filterByNameCIS = [NSPredicate predicateWithFormat:@"self.title LIKE[cd] %@",@"Tom and
Jerry"];
NSLog(@"Filter By Name(CIS) : %@",[array filteredArrayUsingPredicate:filterByNameCIS]);
```

Section 22.2: Find movies except given ids

```
// *** Find movies except given ids ***
NSPredicate *filterByNotInIds = [NSPredicate predicateWithFormat:@"NOT (self.id IN
%@)",@[@"7CDF6D22-8D36-49C2-84FE-E31EECCECB79", @"7CDF6D22-8D36-49C2-84FE-E31EECCECB76"]];
NSLog(@"Filter movies except given Ids : %@",[array filteredArrayUsingPredicate:filterByNotInIds]);
```

Section 22.3: Find all the objects which is of type movie

```
// *** Find all the objects which is of type movie, Both the syntax are valid ***
NSPredicate *filterByMovieType = [NSPredicate predicateWithFormat:@"self.isMovie = %@",@1];
// OR
//NSPredicate *filterByMovieType = [NSPredicate predicateWithFormat:@"self.isMovie = %@",[NSNumber
numberWithBool:YES]];
NSLog(@"Filter By Movie Type : %@",[array filteredArrayUsingPredicate:filterByMovieType]);
```

Section 22.4: Find Distinct object ids of array

```
// *** Find Distinct object ids of array ***
NSLog(@"Distinct id : %@",[array valueForKeyPath:@"@distinctUnionOfObjects.id"]);
```

Section 22.5: Find movies with specific ids

```
// *** Find movies with specific ids ***
NSPredicate *filterByIds = [NSPredicate predicateWithFormat:@"self.id IN
%@",@[@"7CDF6D22-8D36-49C2-84FE-E31EECCECB79", @"7CDF6D22-8D36-49C2-84FE-E31EECCECB76"]];
NSLog(@"Filter By Ids : %@",[array filteredArrayUsingPredicate:filterByIds]);
```

Section 22.6: Case Insensitive comparison with exact title match

```
// *** Case Insensitive comparison with exact title match ***
NSPredicate *filterByNameCIS = [NSPredicate predicateWithFormat:@"self.title LIKE[cd] %@",@"Tom and
Jerry"];
NSLog(@"Filter By Name(CIS) : %@",[array filteredArrayUsingPredicate:filterByNameCIS]);
```

Section 22.7: Case sensitive with exact title match

```
// *** Case sensitive with exact title match ***
NSPredicate *filterByNameCS = [NSPredicate predicateWithFormat:@"self.title = %@",@"Tom and
Jerry"];
NSLog(@"Filter By Name(CS) : %@",[array filteredArrayUsingPredicate:filterByNameCS]);
```

Section 22.8: Case Insensitive comparison with matching subset

```
// *** Case Insensitive comparison with matching subset ***
NSPredicate *filterByName = [NSPredicate predicateWithFormat:@"self.title CONTAINS[cd] %@",@"Tom"];
NSLog(@"Filter By Containing Name : %@",[array filteredArrayUsingPredicate:filterByName]);
```

Chapter 23: NSMutableArray

Section 23.1: Sorting Arrays

```
NSMutableArray *myColors = [NSMutableArray arrayWithObjects: @"red", @"green", @"blue", @"yellow",
nil];
NSArray *sortedArray;
sortedArray = [myColors sortedArrayUsingSelector:@selector(localizedCaseInsensitiveCompare:)];
```

Section 23.2: Creating an NSMutableArray

`NSMutableArray` can be initialized as an empty array like this:

```
NSMutableArray *array = [[NSMutableArray alloc] init];
// or
NSMutableArray *array2 = @[].mutableCopy;
// or
NSMutableArray *array3 = [NSMutableArray array];
```

`NSMutableArray` can be initialized with another array like this:

```
NSMutableArray *array4 = [[NSMutableArray alloc] initWithArray:anotherArray];
// or
NSMutableArray *array5 = anotherArray.mutableCopy;
```

Section 23.3: Adding elements

```
NSMutableArray *myColors;
myColors = [NSMutableArray arrayWithObjects: @"Red", @"Green", @"Blue", @"Yellow", nil];
[myColors addObject: @"Indigo"];
[myColors addObject: @"Violet"];

//Add objects from an NSArray
NSArray *myArray = @[@"Purple",@"Orange"];
[myColors addObjectsFromArray:myArray];
```

Section 23.4: Insert Elements

```
NSMutableArray *myColors;
int i;
int count;
myColors = [NSMutableArray arrayWithObjects: @"Red", @"Green", @"Blue", @"Yellow", nil];
```

```
[myColors insertObject: @"Indigo" atIndex: 1];
[myColors insertObject: @"Violet" atIndex: 3];
```

Section 23.5: Deleting Elements

Remove at specific index:

```
[myColors removeObjectAtIndex: 3];
```

Remove the first instance of a specific object:

```
[myColors removeObject: @"Red"];
```

Remove all instances of a specific object:

```
[myColors removeObjectIdenticalTo: @"Red"];
```

Remove all objects:

```
[myColors removeAllObjects];
```

Remove last object:

```
[myColors removeLastObject];
```

Section 23.6: Move object to another index

Move *Blue* to the beginning of the array:

```
NSMutableArray *myColors = [NSMutableArray arrayWithObjects: @"Red", @"Green", @"Blue", @"Yellow",
nil];

NSUInteger fromIndex = 2;
NSUInteger toIndex = 0;

id blue = [[[self.array objectAtIndex:fromIndex] retain] autorelease];
[self.array removeObjectAtIndex:fromIndex];
[self.array insertObject:blue atIndex:toIndex];
```

`myColors` is now `[@"Blue", @"Red", @"Green", @"Yellow"]`.

Section 23.7: Filtering Array content with Predicate

Using **filterUsingPredicate:** This Evaluates a given predicate against the arrays content and return objects that match.

Example:

```
    NSMutableArray *array = [NSMutableArray array];
    [array setArray:@[@"iOS",@"macOS",@"tvOS"]];
    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"SELF beginswith[c] 'i'"];
    NSArray *resultArray = [array filteredArrayUsingPredicate:predicate];
    NSLog(@"%@",resultArray);
```

Chapter 24: NSRegularExpression

- NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:PATTERN options:OPTIONS error:ERROR];
- NSArray<NSTextCheckingResult *> *results = [regex matchesInString:STRING options:OPTIONS range:RANGE_IN_STRING];

- NSInteger numberOfMatches = [regex numberOfMatchesInString:STRING options:OPTIONS range:RANGE_IN_STRING];

Section 24.1: Check whether a string matches a pattern

```
NSString *testString1 = @"(555) 123-5678";
NSString *testString2 = @"not a phone number";

NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:@"^\\(\\d{3}\\)
\\d{3}\\-\\d{4}$"

 options:NSRegularExpressionCaseInsensitive error:&error];

NSInteger result1 = [regex numberOfMatchesInString:testString1 options:0 range:NSMakeRange(0,
testString1.length)];
NSInteger result2 = [regex numberOfMatchesInString:testString2 options:0 range:NSMakeRange(0,
testString2.length)];

NSLog(@"Is string 1 a phone number? %@", result1 > 0 ? @"YES" : @"NO");
NSLog(@"Is string 2 a phone number? %@", result2 > 0 ? @"YES" : @"NO");
```

The output will show that the first string is a phone number and the second one isn't.

Section 24.2: Find all the numbers in a string

```
NSString *testString = @"There are 42 sheep and 8672 cows.";
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:@"(\\d+)"

 options:NSRegularExpressionCaseInsensitive
                                                                          error:&error];

NSArray *matches = [regex matchesInString:testString
                                  options:0
                                    range:NSMakeRange(0, testString.length)];

for (NSTextCheckingResult *matchResult in matches) {
    NSString* match = [testString substringWithRange:matchResult.range];
    NSLog(@"match: %@", match);
}
```

The output will be `match: 42` and `match: 8672`.

Chapter 25: NSJSONSerialization

- (id)JSONObjectWithData:(NSData *)data options:(NSJSONReadingOptions)opt error:(NSError * _Nullable *)error

Operator	Description
data	A data object containing JSON data
opt	Options for reading the JSON data and creating the Foundation objects.
error	If an error occurs, upon return contains an NSError object that describes the problem.

Section 25.1: JSON Parsing using NSJSONSerialization Objective c

```
NSError *e = nil;
NSString *jsonString = @"[{\"id\": \"1\", \"name\":\"sam\"}]";
NSData *data = [jsonString dataUsingEncoding:NSUTF8StringEncoding];
```

```
NSArray *jsonArray = [NSJSONSerialization JSONObjectWithData: data options:
 NSJSONReadingMutableContainers error: &e];

if (!jsonArray) {
    NSLog(@"Error parsing JSON: %@", e);
} else {
    for(NSDictionary *item in jsonArray) {
        NSLog(@"Item: %@", item);
    }
}
```

Output:

```
Item: {
    id = 1;
    name = sam;
}
```

Example 2:Using contents of url:

```
//Parsing:

NSData *data = [NSData dataWithContentsOfURL:@“URL HERE”];
NSError *error;
NSDictionary *json = [NSJSONSerialization JSONObjectWithData:data options:kNilOptions
error:&error];
NSLog(@“json :%@”,json);
```

Sample response:

```
json: {
    MESSAGE = “Test Message";
    RESPONSE =(
                {
            email = "test@gmail.com";
            id = 15;
            phone = 1234567890;
            name = Staffy;
        }
    );
    STATUS = SUCCESS;
}


 NSMutableDictionary *response = [[[json valueForKey:@"RESPONSE"] objectAtIndex:0]mutableCopy];
 NSString *nameStr = [response valueForKey:@"name"];
 NSString *emailIdStr = [response valueForKey:@"email"];
```

Chapter 26: Memory Management

Section 26.1: Memory management rules when using manual reference counting.

These rules apply only if you use manual reference counting!

1. **You own any object you create**

   By calling a method whose name begins with `alloc`, `new`, `copy` or `mutableCopy`. For example:

   ```
   NSObject *object1 = [[NSObject alloc] init];
   ```

```
NSObject *object2 = [NSObject new];
NSObject *object3 = [object2 copy];
```

That means that you are responsible for releasing these objects when you are done with them.

2. **You can take ownership of an object using retain**

   To take ownership for an object you call the retain method.

   For example:

   ```
   NSObject *object = [NSObject new]; // object already has a retain count of 1
   [object retain]; // retain count is now 2
   ```

   This makes only sense in some rare situations.

   For example when you implement an accessor or an init method to take ownership:

   ```
   - (void)setStringValue:(NSString *)stringValue {
       [_privateStringValue release]; // Release the old value, you no longer need it
       [stringValue retain]; // You make sure that this object does not get deallocated outside of
   your scope.
       _privateStringValue = stringValue;
   }
   ```

3. **When you no longer need it, you must relinquish ownership of an object you own**

   ```
   NSObject* object = [NSObject new]; // The retain count is now 1
   [object performAction1]; // Now we are done with the object
   [object release]; // Release the object
   ```

4. **You must not relinquish ownership of an object you do not own**

   That means when you didn't take ownership of an object you don't release it.

5. **Autoreleasepool**

   The autoreleasepool is a block of code that releases every object in the block that received an autorelease message.

   Example:

   ```
   @autoreleasepool {
       NSString* string = [NSString stringWithString:@"We don't own this object"];
   }
   ```

   We have created a string without taking ownership. The `NSString` method `stringWithString:` has to make sure that the string is correctly deallocated after it is no longer needed. Before the method returns the newly created string calls the autorelease method so it does not have to take ownership of the string.

   This is how the `stringWithString:` is implemented:

   ```
   + (NSString *)stringWithString:(NSString *)string {
       NSString *createdString = [[NSString alloc] initWithString:string];
       [createdString autorelease];
       return createdString;
   }
   ```

It is necessary to use autoreleasepool blocks because you sometimes have objects that you don't own (the fourth rules does not always apply).

Automatic reference counting takes automatically care of the rules so you don't have to.

Section 26.2: Automatic Reference Counting

With automatic reference counting (ARC), the compiler inserts `retain`, `release`, and `autorelease` statements where they are needed, so you don't have to write them yourself. It also writes `dealloc` methods for you.

The sample program from Manual Memory Management looks like this with ARC:

```
@interface MyObject : NSObject {
    NSString *_property;
}
@end

@implementation MyObject
@synthesize property = _property;

- (id)initWithProperty:(NSString *)property {
    if (self = [super init]) {
        _property = property;
    }
    return self;
}

- (NSString *)property {
    return property;
}

- (void)setProperty:(NSString *)property {
    _property = property;
}

@end
int main() {
    MyObject *obj = [[MyObject alloc] init];

    NSString *value = [[NSString alloc] initWithString:@"value"];
    [obj setProperty:value];

    [obj setProperty:@"value"];
}
```

You are still able to override the dealloc method to clean up resources not handled by ARC. Unlike when using manual memory management you do not call `[super dealloc]`.

```
-(void)dealloc {
   //clean up
}
```

Section 26.3: Strong and weak references

```
Version=Modern
```

A weak reference looks like one of these:

```
@property (weak) NSString *property;
NSString *__weak variable;
```

If you have a weak reference to an object, then under the hood:

- You're not retaining it.
- When it gets deallocated, every reference to it will automatically be set to `nil`

Object references are always strong by default. But you can explicitly specify that they're strong:

```
@property (strong) NSString *property;
NSString *__strong variable;
```

A strong reference means that while that reference exists, you are retaining the object.

Section 26.4: Manual Memory Management

This is an example of a program written with manual memory management. You really shouldn't write your code like this, unless for some reason you can't use ARC (like if you need to support 32-bit). The example avoids `@property` notation to illustrate how you used to have to write getters and setters.

```
@interface MyObject : NSObject {
    NSString *_property;
}
@end

@implementation MyObject
@synthesize property = _property;

- (id)initWithProperty:(NSString *)property {
    if (self = [super init]) {
        // Grab a reference to property to make sure it doesn't go away.
        // The reference is released in dealloc.
        _property = [property retain];
    }
    return self;
}

- (NSString *)property {
    return [[property retain] autorelease];
}

- (void)setProperty:(NSString *)property {
    // Retain, then release. So setting it to the same value won't lose the reference.
    [property retain];
    [_property release];
    _property = property;
}

- (void)dealloc {
    [_property release];
    [super dealloc]; // Don't forget!
}

@end
int main() {
    // create object
    // obj is a reference that we need to release
    MyObject *obj = [[MyObject alloc] init];

    // We have to release value because we created it.
    NSString *value = [[NSString alloc] initWithString:@"value"];
    [obj setProperty:value];
```

```
    [value release];

    // However, string constants never need to be released.
    [obj setProperty:@"value"];
    [obj release];
}
```

Chapter 27: Singletons

Just make sure you read this thread (What is so bad about singletons?) before using it.

Section 27.1: Using Grand Central Dispatch (GCD)

GCD will guarantee that your singleton only gets instantiated once, even if called from multiple threads. Insert this into any class for a singleton instance called `shared`.

```
+ (instancetype)shared {

    // Variable that will point to the singleton instance. The `static`
    // modifier makes it behave like a global variable: the value assigned
    // to it will "survive" the method call.
    static id _shared;

    static dispatch_once_t _onceToken;
    dispatch_once(&_onceToken, ^{

        // This block is only executed once, in a thread-safe way.
        // Create the instance and assign it to the static variable.
        _shared = [self new];
    });

    return _shared;
}
```

Section 27.2: Creating Singleton and also preventing it from having multiple instance using alloc/init, new.

```
//MySingletonClass.h
@interface MYSingletonClass : NSObject

+ (instancetype)sharedInstance;

-(instancetype)init NS_UNAVAILABLE;

-(instancetype)new NS_UNAVAILABLE;

@end

//MySingletonClass.m

@implementation MySingletonClass

+ (instancetype)sharedInstance
{
    static MySingletonClass *_sharedInstance = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^{
        _sharedInstance = [[self alloc]init];
    });

    return _sharedInstance;
```

```
}
-(instancetype)init
{
    self = [super init];
    if(self)
    {
        //Do any additional initialization if required
    }
    return self;
}
@end
```

Section 27.3: Creating Singleton class and also preventing it from having multiple instances using alloc/init.

We can create Singleton class in such a way that developers are forced to used the shared instance (singleton object) instead of creating their own instances.

```
@implementation MySingletonClass

+ (instancetype)sharedInstance
{
    static MySingletonClass *_sharedInstance = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^{
        _sharedInstance = [[self alloc] initClass];
    });

    return _sharedInstance;
}

-(instancetype)initClass
{
    self = [super init];
    if(self)
    {
        //Do any additional initialization if required
    }
    return self;
}

- (instancetype)init
{
    @throw [NSException exceptionWithName:@"Not designated initializer"
                                   reason:@"Use [MySingletonClass sharedInstance]"
                                 userInfo:nil];
    return nil;
}
@end



/*Following line will throw an exception
  with the Reason:"Use [MySingletonClass sharedInstance]"
  when tried to alloc/init directly instead of using sharedInstance */
MySingletonClass *mySingletonClass = [[MySingletonClass alloc] init];
```

Chapter 28: NSCalendar

Section 28.1: System Locale Information

`+currentCalendar` returns the logical calendar for the current user.

```
    NSCalendar *calender = [NSCalendar currentCalendar];
    NSLog(@"%@",calender);
```

`+autoupdatingCurrentCalendar` returns the current logical calendar for the current user.

```
NSCalendar *calender = [NSCalendar autoupdatingCurrentCalendar];
NSLog(@"%@",calender);
```

Section 28.2: Initializing a Calendar

`- initWithCalendarIdentifier:` Initializes a newly-allocated NSCalendar object for the calendar specified by a given identifier.

```
NSCalendar *calender = [[NSCalendar alloc]initWithCalendarIdentifier:@"gregorian"];
NSLog(@"%@",calender);
```

`- setFirstWeekday:` Sets the index of the first weekday for the receiver.

```
NSCalendar *calender = [NSCalendar autoupdatingCurrentCalendar];
[calender setFirstWeekday:1];
NSLog(@"%d",[calender firstWeekday]);
```

`- setLocale:` Sets the locale for the receiver.

```
NSCalendar *calender = [NSCalendar autoupdatingCurrentCalendar];
[calender setLocale:[NSLocale currentLocale]];
NSLog(@"%@",[calender locale]);
```

`- setMinimumDaysInFirstWeek:` Sets the minimum number of days in the first week of the receiver.

```
NSCalendar *calender = [NSCalendar autoupdatingCurrentCalendar];
[calender setMinimumDaysInFirstWeek:7];
NSLog(@"%d",[calender minimumDaysInFirstWeek]);
```

`- setTimeZone:` Sets the time zone for the receiver.

```
NSCalendar *calender = [NSCalendar autoupdatingCurrentCalendar];
[calender setTimeZone:[NSTimeZone timeZoneForSecondsFromGMT:0]];
NSLog(@"%@",[calender timeZone]);
```

Section 28.3: Calendrical Calculations

`- components:fromDate:` Returns a NSDateComponents object containing a given date decomposed into specified components

```
NSCalendar *calender = [NSCalendar autoupdatingCurrentCalendar];
[calender setTimeZone:[NSTimeZone timeZoneForSecondsFromGMT:0]];
NSLog(@"%@",[calender components:NSCalendarUnitDay fromDate:[NSDate date]]);
NSLog(@"%@",[calender components:NSCalendarUnitYear fromDate:[NSDate date]]);
NSLog(@"%@",[calender components:NSCalendarUnitMonth fromDate:[NSDate date]]);
```

`- components:fromDate:toDate:options:` Returns, as an NSDateComponents object using specified components, the difference between two supplied dates.

```
 NSCalendar *calender = [NSCalendar autoupdatingCurrentCalendar];
[calender setTimeZone:[NSTimeZone timeZoneForSecondsFromGMT:0]];
NSLog(@"%@",[calender components:NSCalendarUnitYear fromDate:[NSDate
```

```
dateWithTimeIntervalSince1970:0] toDate:[NSDate dateWithTimeIntervalSinceNow:18000]
options:NSCalendarWrapComponents]);
```

`- dateByAddingComponents:toDate:options:` Returns a new NSDate object representing the absolute time calculated by adding given components to a given date.

```
NSCalendar *calender = [NSCalendar autoupdatingCurrentCalendar];
NSDateComponents *dateComponent = [[NSDateComponents alloc]init];
[dateComponent setYear:10];
NSLog(@"%@",[calender dateByAddingComponents:dateComponent toDate:[NSDate
dateWithTimeIntervalSinceNow:0] options:NSCalendarWrapComponents] );
```

`- dateFromComponents:` Returns a new NSDate object representing the absolute time calculated from given components.

```
NSCalendar *calender = [NSCalendar autoupdatingCurrentCalendar];
NSDateComponents *dateComponent = [[NSDateComponents alloc]init];
[dateComponent setYear:2020];
NSLog(@"%@",[calender dateFromComponents:dateComponent]);
```

Chapter 29: NSMutableDictionary

objects	keys
An array containing the values for the new CellAn dictionary.	array containing the keys for the new dictionary. Each key is copied and the copy is added to the dictionary.

Section 29.1: NSMutableDictionary Example

+ dictionaryWithCapacity:

Creates and returns a mutable dictionary, initially giving it enough allocated memory to hold a given number of entries.

```
NSMutableDictionary *dict =  [NSMutableDictionary dictionaryWithCapacity:1];
NSLog(@"%@",dict);
```

- init

Initializes a newly allocated mutable dictionary.

```
NSMutableDictionary *dict =  [[NSMutableDictionary alloc] init];
NSLog(@"%@",dict);
```

+ dictionaryWithSharedKeySet:

Creates a mutable dictionary which is optimized for dealing with a known set of keys.

```
id sharedKeySet = [NSDictionary sharedKeySetForKeys:@[@"key1", @"key2"]]; // returns NSSharedKeySet
NSMutableDictionary *dict = [NSMutableDictionary dictionaryWithSharedKeySet:sharedKeySet];
dict[@"key1"] = @"Easy";
dict[@"key2"] = @"Tutorial";
//We can an object thats not in the shared keyset
dict[@"key3"] = @"Website";
NSLog(@"%@",dict);
```

OUTPUT

```
{
    key1 = Eezy;
    key2 = Tutorials;
```

```
    key3 = Website;
}
```

Adding Entries to a Mutable Dictionary

- setObject:forKey:

Adds a given key-value pair to the dictionary.

```
NSMutableDictionary *dict =  [NSMutableDictionary dictionary];
[dict setObject:@"Easy" forKey:@"Key1"];
NSLog(@"%@",dict);
```

OUTPUT

```
{
    Key1 = Eezy;
}
```

- setObject:forKeyedSubscript:

Adds a given key-value pair to the dictionary.

```
NSMutableDictionary *dict =  [NSMutableDictionary dictionary];
[dict setObject:@"Easy" forKeyedSubscript:@"Key1"];
NSLog(@"%@",dict);
```

OUTPUT { Key1 = Easy; }

Section 29.2: Removing Entries From a Mutable Dictionary

- removeObjectForKey:

Removes a given key and its associated value from the dictionary.

```
NSMutableDictionary *dict =  [NSMutableDictionary
dictionaryWithDictionary:@{@"key1":@"Easy",@"key2": @"Tutorials"}];
[dict removeObjectForKey:@"key1"];
NSLog(@"%@",dict);
```

OUTPUT

```
{
    key2 = Tutorials;
}
```

- removeAllObjects

Empties the dictionary of its entries.

```
NSMutableDictionary *dict =  [NSMutableDictionary
dictionaryWithDictionary:@{@"key1":@"Eezy",@"key2": @"Tutorials"}];
[dict removeAllObjects];
NSLog(@"%@",dict);
```

OUTPUT

```
{
}
```

- removeObjectsForKeys:

Removes from the dictionary entries specified by elements in a given array.

```
NSMutableDictionary *dict =  [NSMutableDictionary
dictionaryWithDictionary:@{@"key1":@"Easy",@"key2": @"Tutorials"}];
[dict removeObjectsForKeys:@[@"key1"]];
NSLog(@"%@",dict);
```

OUTPUT

```
{
    key2 = Tutorials;
}
```

Chapter 30: Multi-Threading

Section 30.1: Creating a simple thread

The most simple way to create a thread is by calling a selector "in the background". This means a new thread is created to execute the selector. The receiving object can be any object, not just `self`, but it needs to respond to the given selector.

```
- (void)createThread {
    [self performSelectorInBackground:@selector(threadMainWithOptionalArgument:)
                           withObject:someObject];
}

- (void)threadMainWithOptionalArgument:(id)argument {
    // To avoid memory leaks, the first thing a thread method needs to do is
    // create a new autorelease pool, either manually or via "@autoreleasepool".
    @autoreleasepool {
        // The thread code should be here.
    }
}
```

Section 30.2: Create more complex thread

Using a subclass of `NSThread` allows implementation of more complex threads (for example, to allow passing more arguments or to encapsulate all related helper methods in one class). Additionally, the `NSThread` instance can be saved in a property or variable and can be queried about its current state (whether it's still running).

The `NSThread` class supports a method called `cancel` that can be called from any thread, which then sets the `cancelled` property to `YES` in a thread-safe way. The thread implementation can query (and/or observe) the `cancelled` property and exit its `main` method. This can be used to gracefully shut down a worker thread.

```
// Create a new NSThread subclass
@interface MyThread : NSThread

// Add properties for values that need to be passed from the caller to the new
// thread. Caller must not modify these once the thread is started to avoid
// threading issues (or the properties must be made thread-safe using locks).
@property NSInteger someProperty;

@end

@implementation MyThread

- (void)main
```

```
{
    @autoreleasepool {
        // The main thread method goes here
        NSLog(@"New thread. Some property: %ld", (long)self.someProperty);
    }
}

@end


MyThread *thread = [[MyThread alloc] init];
thread.someProperty = 42;
[thread start];
```

Section 30.3: Thread-local storage

Every thread has access to a mutable dictionary that is local to the current thread. This allows to cache informations in an easy way without the need for locking, as each thread has its own dedicated mutable dictionary:

```
NSMutableDictionary *localStorage = [NSThread currentThread].threadDictionary;
localStorage[someKey] = someValue;
```

The dictionary is automatically released when the thread terminates.

Chapter 31: NSAttributedString

Section 31.1: Using Enumerating over Attributes in a String and underline part of string

```
 NSMutableDictionary *attributesDictionary = [NSMutableDictionary dictionary];
 [attributesDictionary setObject:[UIFont systemFontOfSize:14] forKey:NSFontAttributeName];
 //[attributesDictionary setObject:[UIColor redColor] forKey:NSForegroundColorAttributeName];
 NSMutableAttributedString *attributedString = [[NSMutableAttributedString
alloc]initWithString:@"Google www.google.com link" attributes:attributesDictionary];

 [attributedString enumerateAttribute:(NSString *) NSFontAttributeName
                              inRange:NSMakeRange(0, [attributedString length])
                              options:NSAttributedStringEnumerationLongestEffectiveRangeNotRequired
                           usingBlock:^(id value, NSRange range, BOOL *stop) {
                               NSLog(@"Attribute: %@, %@", value, NSStringFromRange(range));
                              }];

  NSMutableAttributedString *attributedStr = [[NSMutableAttributedString alloc]
initWithString:@"www.google.com "];

  [attributedString addAttribute:NSUnderlineStyleAttributeName
                           value:[NSNumber numberWithInt:NSUnderlineStyleDouble]
                           range:NSMakeRange(7, attributedStr.length)];

  [attributedString addAttribute:NSForegroundColorAttributeName
                           value:[UIColor blueColor]
                           range:NSMakeRange(6,attributedStr.length)];

  _attriLbl.attributedText = attributedString;//_attriLbl (of type UILabel) added in storyboard
```

Output:

Section 31.2: Creating a string that has custom kerning (letter spacing) editshare

`NSAttributedString` (and its mutable sibling `NSMutableAttributedString`) allows you to create strings that are complex in their appearance to the user.

A common application is to use this to display a string and adding custom kerning / letter-spacing.

This would be achieved as follows (where label is a `UILabel`), giving a different kerning for the word "kerning"

```
NSMutableAttributedString *attributedString;
attributedString = [[NSMutableAttributedString alloc] initWithString:@"Apply kerning"];
[attributedString addAttribute:NSKernAttributeName value:@5 range:NSMakeRange(6, 7)];
[label setAttributedText:attributedString];
```

Section 31.3: Create a string with text struck through

```
NSMutableAttributedString *attributeString = [[NSMutableAttributedString alloc]
initWithString:@"Your String here"];
[attributeString addAttribute:NSStrikethroughStyleAttributeName
                    value:@2
                    range:NSMakeRange(0, [attributeString length])];
```

Section 31.4: How you create a tri-color attributed string.

```
 NSMutableAttributedString * string = [[NSMutableAttributedString alloc]
initWithString:@"firstsecondthird"];
[string addAttribute:NSForegroundColorAttributeName value:[UIColor redColor]
range:NSMakeRange(0,5)];
[string addAttribute:NSForegroundColorAttributeName value:[UIColor greenColor]
range:NSMakeRange(5,6)];
[string addAttribute:NSForegroundColorAttributeName value:[UIColor blueColor]
range:NSMakeRange(11,5)];
```

Range : start to end string

Here we have firstsecondthird string so in first we have set range (0,5) so from starting first character to fifth character it will display in green text color.

Chapter 32: NSTimer

Section 32.1: Storing information in the Timer

When creating a timer, you can set the `userInfo` parameter to include information that you want to pass to the function you call with the timer.

By taking a timer as a parameter in said function, you can access the `userInfo` property.

```
NSDictionary *dictionary = @{
                              @"Message" : @"Hello, world!"
                             }; //this dictionary contains a message
[NSTimer scheduledTimerWithTimeInterval:5.0
     target:self
     selector:@selector(doSomething)
     userInfo:dictionary
     repeats:NO]; //the timer contains the dictionary and later calls the function

...

- (void) doSomething:(NSTimer*)timer{
```

```
    //the function retrieves the message from the timer
    NSLog("%@", timer.userInfo["Message"]);
}
```

Section 32.2: Creating a Timer

This will create a timer to call the `doSomething` method on `self` in `5.0` seconds.

```
[NSTimer scheduledTimerWithTimeInterval:5.0
         target:self
         selector:@selector(doSomething)
         userInfo:nil
         repeats:NO];
```

Setting the `repeats` parameter to `false`/`NO` indicates that we want the timer to fire only once. If we set this to `true`/`YES`, it would fire every five seconds until manually invalidated.

Section 32.3: Invalidating a timer

```
[timer invalidate];
timer = nil;
```

This will stop the timer from firing. **Must be called from the thread the timer was created in,** see Apple's notes:

You must send this message from the thread on which the timer was installed. If you send this message from another thread, the input source associated with the timer may not be removed from its run loop, which could prevent the thread from exiting properly.

Setting `nil` will help you next to check whether it's running or not.

```
if(timer) {
    [timer invalidate];
    timer = nil;
}

//Now set a timer again.
```

Section 32.4: Manually firing a timer

```
[timer fire];
```

Calling the `fire` method causes an NSTimer to perform the task it would have usually performed on a schedule.

In a **non-repeating timer**, this will automatically invalidate the timer. That is, calling `fire` before the time interval is up will result in only one invocation.

In a **repeating timer**, this will simply invoke the action without interrupting the usual schedule.

Chapter 33: Structs

- typedef struct { *typeA propertyA*; *typeB propertyB*; ... } *StructName*

Section 33.1: Defining a Structure and Accessing Structure Members

The format of the struct statement is this:

```
struct [structure tag]
{
   member definition;
```

```
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

Example: declare the ThreeFloats structure:

```
   typedef struct {
    float x, y, z;
} ThreeFloats;

@interface MyClass
- (void)setThreeFloats:(ThreeFloats)threeFloats;
- (ThreeFloats)threeFloats;
@end
```

Sending an instance of MyClass the message valueForKey: with the parameter @"threeFloats" will invoke the MyClass method threeFloats and return the result wrapped in an NSValue.

Section 33.2: CGPoint

One really good example of a struct is `CGPoint`; it's a simple value that represents a 2-dimensional point. It has 2 properties, `x` and `y`, and can be written as

```
typedef struct {
    CGFloat x;
    CGFloat y;
} CGPoint;
```

If you used Objective-C for Mac or iOS app development before, you've almost certainly come across `CGPoint`; `CGPoint`s hold the position of pretty much everything on screen, from views and controls to objects in a game to changes in a gradient. This means that `CGPoint`s are used a lot. This is even more true with really performance-heavy games; these games tend to have a lot of objects, and all of these objects need positions. These positions are often either `CGPoint`s, or some other type of struct that conveys a point (such as a 3-dimensional point for 3d games).

Points like `CGPoint` could easily be represented as objects, like

```
@interface CGPoint {
    CGFloat x;
    CGFloat y;
}

... //Point-related methods (e.g. add, isEqualToPoint, etc.)

@property(nonatomic, assign)CGFloat x;
@property(nonatomic, assign)CGFloat y;

@end

@implementation CGPoint

@synthesize x, y;

...

@end
```

However, if `CGPoint` was used in this way it would take a lot longer to create and manipulate points. In smaller, faster programs this wouldn't really cause a difference, and in those cases it would be OK or maybe even better to

use object points. But in large programs where points are be used a lot, using objects as points can really hurt performance, making the program slower, and also waste memory, which could force the program to crash.

Chapter 34: Subscripting

Section 34.1: Subscripts with NSArray

Subscripts can be used to simplify retrieving and setting elements in an array. Given the following array

```
NSArray *fruit = @[@"Apples", @"Bananas", @"Cherries"];
```

This line

```
[fruit objectAtIndex: 1];
```

Can be replaced by

```
fruit[1];
```

They can also be used to set an element in a mutable array.

```
NSMutableArray *fruit = [@[@"Apples", @"Bananas", @"Cherries"] mutableCopy];
fruit[1] = @"Blueberries";
NSLog(@"%@", fruit[1]); //Blueberries
```

If the index of the subscript equals the count of the array, the element will be appended to the array.

Repeated subscripts may be used to access elements of nested arrays.

```
NSArray *fruit = @[@"Apples", @"Bananas", @"Cherries"];
NSArray *vegetables = @[@"Avocado", @"Beans", @"Carrots"];
NSArray *produce = @[fruit, vegetables];

NSLog(@"%@", produce[0][1]); //Bananas
```

Section 34.2: Custom Subscripting

You can add subscripting to your own classes by implementing the required methods.

For indexed subscripting (like arrays):

```
- (id)objectAtIndexedSubscript:(NSUInteger)idx
- (void)setObject:(id)obj atIndexedSubscript:(NSUInteger)idx
```

For keyed subscripting (like dictionaries):

```
- (id)objectForKeyedSubscript:(id)key
- (void)setObject:(id)obj forKeyedSubscript:(id <NSCopying>)key
```

Section 34.3: Subscripts with NSDictionary

Subscripts can also be used with NSDictionary and NSMutableDictionary. The following code:

```
NSMutableDictionary *myDictionary = [@{@"Foo": @"Bar"} mutableCopy];
[myDictionary setObject:@"Baz" forKey:@"Foo"];
NSLog(@"%@", [myDictionary objectForKey:@"Foo"]); // Baz
```

Can be shortened to:

```
NSMutableDictionary *myDictionary = [@{@"Foo": @"Bar"} mutableCopy];
```

```
myDictionary[@"Foo"] = @"Baz";
NSLog(@"%@", myDictionary[@"Foo"]); // Baz
```

Chapter 35: Basic Data Types

- BOOL havePlutonium = YES; // Direct assigment
- BOOL fastEnough = (car.speedInMPH >= 88); // Comparison expression
- BOOL fluxCapacitorActive = (havePlutonium && fastEnough); // Boolean expression
-
- id somethingWicked = [witchesCupboard lastObject]; // Retrieve untyped object
- id powder = prepareWickedIngredient(somethingWicked); // Pass and return
- if ([ingredient isKindOfClass:[Toad class]]) { // Test runtime type

Section 35.1: SEL

Selectors are used as method identifiers in Objective-C.

In the example below, there are two selectors. `new` and `setName:`

```
Person* customer = [Person new];
[customer setName:@"John Doe"];
```

Each pair of brackets corresponds to a message send. On the first line we send a message containing the `new` selector to the `Person` class and on the second line we send a message containing the `setName:` selector and a string. The receiver of these messages uses the selector to look up the correct action to perform.

Most of the time, message passing using the bracket syntax is sufficient, but occasionally you need to work with the selector itself. In these cases, the `SEL` type can be used to hold a reference to the selector.

If the selector is available at compile time, you can use `@selector()` to get a reference to it.

```
SEL s = @selector(setName:);
```

And if you need to find the selector at runtime, use NSSelectorFromString.

```
SEL s NSSelectorFromString(@"setName:");
```

When using NSSelectorFromString, make sure to wrap the selector name in a NSString.

It is commonly used to check if a delegate implements an optional method.

```
if ([self.myDelegate respondsToSelector:@selector(doSomething)]) {
    [self.myDelegate doSomething];
}
```

Section 35.2: BOOL

The `BOOL` type is used for boolean values in Objective-C. It has two values, `YES`, and `NO`, in contrast to the more common "true" and "false".

Its behavior is straightforward and identical to the C language's.

```
BOOL areEqual = (1 == 1);    // areEqual is YES
BOOL areNotEqual = !areEqual    // areNotEqual is NO
NSCAssert(areEqual, "Mathematics is a lie");    // Assertion passes

BOOL shouldFlatterReader = YES;
if (shouldFlatterReader) {
```

```
    NSLog(@"Only the very smartest programmers read this kind of material.");
}
```

A `BOOL` is a primitive, and so it cannot be stored directly in a Foundation collection. It must be wrapped in an `NSNumber`. Clang provides special syntax for this:

```
NSNumber * yes = @YES;    // Equivalent to [NSNumber numberWithBool:YES]
NSNumber * no = @NO;    // Equivalent to [NSNumber numberWithBool:NO]
```

The `BOOL` implementation is directly based on C's, in that it is a typedef of the C99 standard type `bool`. The `YES` and `NO` values are defined to `__objc_yes` and `__objc_no`, respectively. These special values are compiler builtins introduced by Clang, which are translated to `(BOOL)1` and `(BOOL)0`. If they are not available, `YES` and `NO` are defined directly as the cast-integer form. The definitions are found in the Objective-C runtime header objc.h

Section 35.3: id

`id` is the generic object pointer, an Objective-C type representing "any object". An instance of any Objective-C class can be stored in an `id` variable. An `id` and any other class type can be assigned back and forth without casting:

```
id anonymousSurname = @"Doe";
NSString * surname = anonymousSurname;
id anonymousFullName = [NSString stringWithFormat:@"%@, John", surname];
```

This becomes relevant when retrieving objects from a collection. The return types of methods like `objectAtIndex:` are `id` for exactly this reason.

```
DataRecord * record = [records objectAtIndex:anIndex];
```

It also means that a method or function parameter typed as `id` can accept any object.

When an object is typed as `id`, any known message can be passed to it: method dispatch does not depend on the compile-time type.

```
NSString * extinctBirdMaybe =
               [anonymousSurname stringByAppendingString:anonymousSurname];
```

A message that the object does not actually respond to will still cause an exception at runtime, of course.

```
NSDate * nope = [anonymousSurname addTimeInterval:10];
// Raises "Does not respond to selector" exception
```

Guarding against exception.

```
NSDate * nope;
if([anonymousSurname isKindOfClass:[NSDate class]]){
    nope = [anonymousSurname addTimeInterval:10];
}
```

The `id` type is defined in objc.h

```
typedef struct objc_object {
    Class isa;
} *id;
```

Section 35.4: IMP (implementation pointer)

IMP is a C type referring to the implementation of a method, also known as an implementation pointer. It is a pointer to the start of a method implementation.

Syntax:

```
id (*IMP)(id, SEL, …)
```

IMP is defined by:

```
typedef id (*IMP)(id self,SEL _cmd,…);
```

To access this IMP, the message **"methodForSelector"** can be used.

Example 1:

```
IMP ImpDoSomething = [myObject methodForSelector:@selector(doSomething)];
```

The method adressed by the IMP can be called by dereferencing the IMP.

```
ImpDoSomething(myObject, @selector(doSomething));
```

So these calls are equal:

```
 myImpDoSomething(myObject, @selector(doSomething));
[myObject doSomething]
[myObject performSelector:mySelector]
[myObject performSelector:@selector(doSomething)]
[myObject performSelector:NSSelectorFromString(@"doSomething")];
```

Example :2:

```
SEL otherWaySelector = NSSelectorFromString(@"methodWithFirst:andSecond:andThird:");

IMP methodImplementation  = [self methodForSelector:otherWaySelector];

result = methodImplementation( self,
                          betterWaySelector,
                          first,
                          second,
                          third );

NSLog(@"methodForSelector : %@", result);
```

Here, we call [NSObject methodForSelector which returns us a pointer to the C function that actually implements the method, which we can the subsequently call directly.

Section 35.5: NSInteger and NSUInteger

The NSInteger is just a typedef for either an int or a long depending on the architecture. The same goes for a NSUInteger which is a typedef for the unsigned variants. If you check the NSInteger you will see the following:

```
#if __LP64__ || (TARGET_OS_EMBEDDED && !TARGET_OS_IPHONE) || TARGET_OS_WIN32 || NS_BUILD_32_LIKE_64
typedef long NSInteger;
typedef unsigned long NSUInteger;
#else
typedef int NSInteger;
typedef unsigned int NSUInteger;
#endif
```

The difference between an signed and an unsigned int or long is that a signed int or long can contain negative values. The range of the int is -2 147 483 648 to 2 147 483 647 while the unsigned int has a range of 0 to 4 294 967 295. The value is doubled because the first bit isn't used anymore to say the value is negative or not. For a long and NSInteger on 64-bit architectures, the range is much wider.

Most methods Apple provides are returning an NS(U)Integer over the normal int. You'll get a warning if you try to

cast it to a normal int because you will lose precision if you are running on a 64-bit architecture. Not that it would matter in most cases, but it is easier to use NS(U)Integer. For example, the count method on a array will return an NSUInteger.

```
NSNumber *iAmNumber = @0;

NSInteger iAmSigned = [iAmNumber integerValue];
NSUInteger iAmUnsigned = [iAmNumber unsignedIntegerValue];

NSLog(@"%ld", iAmSigned); // The way to print a NSInteger.
NSLog(@"%lu", iAmUnsigned); // The way to print a NSUInteger.
```

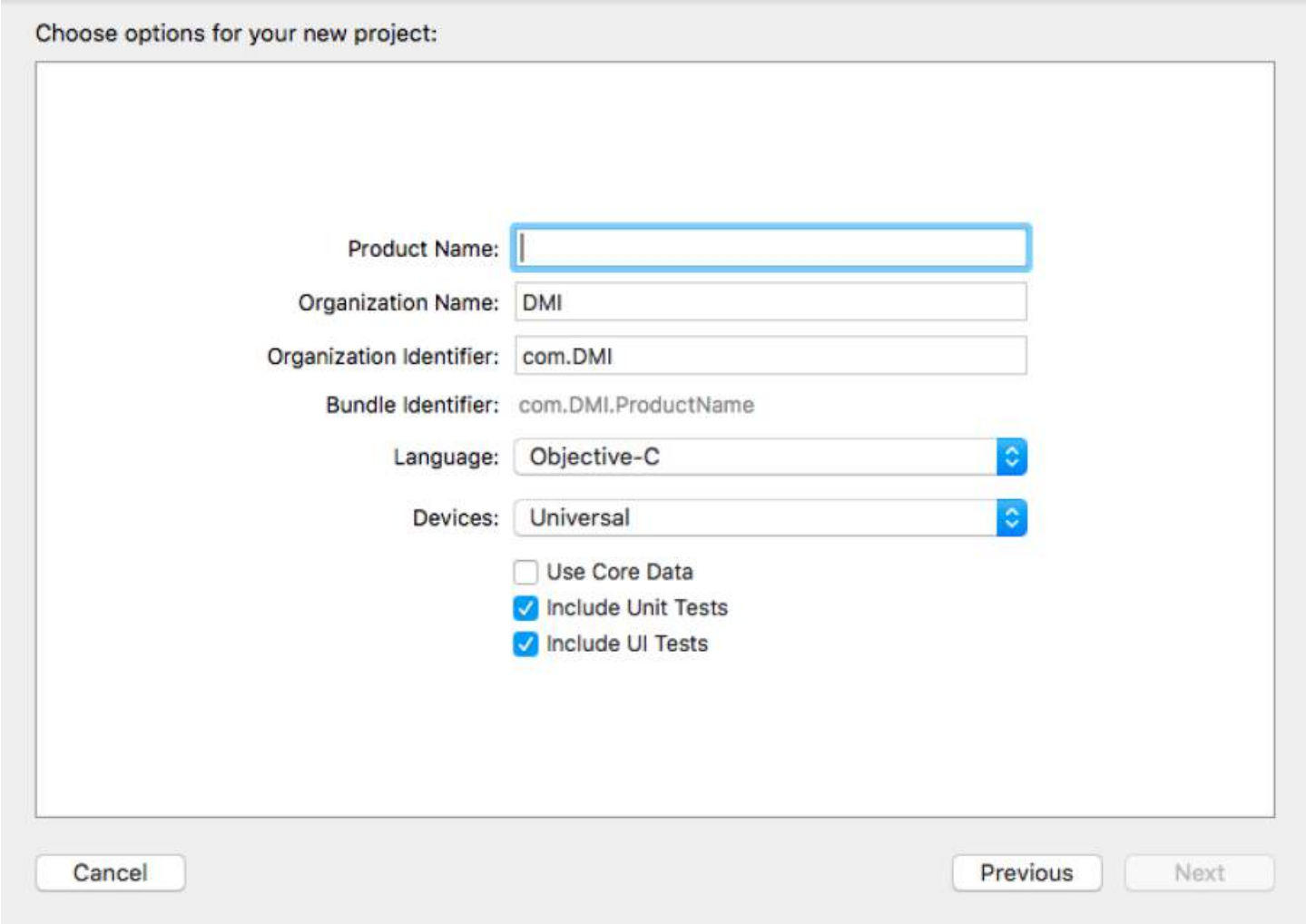
Just like a BOOL, the NS(U)Integer is a primitive datatype, so you sometimes need to wrap it in a NSNumber you can use the @ before the integer to cast it like above and retrieve it using the methods below. But to cast it to NSNumber, you could also use the following methods:

```
[NSNumber numberWithInteger:0];
[NSNumber numberWithUnsignedInteger:0];
```

Chapter 36: Unit testing using Xcode

Section 36.1: Note:

Make sure that include unit test case box is checked when creating a new project as shown below:



Section 36.2: Testing a block of code or some method:

- Import the class, which contains the method to be tested.
- Perform the operation with dummy data.

- Now compare the result of operation with expected result.

```
- (void)testReverseString{
NSString *originalString = @"hi_my_name_is_siddharth";
NSString *reversedString = [self.someObject reverseString:originalString];
NSString *expectedReversedString = @"htrahddis_si_eman_ym_ih";
XCTAssertEqualObjects(expectedReversedString, reversedString, @"The reversed string did not match
the expected reverse");
}
```

Feed the dummy data to the method under test if required & then compare the expected & actual results.

Section 36.3: Testing asynchronous block of code:

```
- (void)testDoSomethingThatTakesSomeTime{
XCTestExpectation *completionExpectation = [self expectationWithDescription:@"Long method"];
[self.someObject doSomethingThatTakesSomeTimesWithCompletionBlock:^(NSString *result) {
    XCTAssertEqualObjects(@"result", result, @"Result was not correct!");
    [completionExpectation fulfill];
}];
[self waitForExpectationsWithTimeout:5.0 handler:nil];
}
```

- Feed the dummy data to the method under test if required.
- The test will pause here, running the run loop, until the timeout is hit or all expectations are fulfilled.
- Timeout is the expected time for the asynchronous block to response.

Section 36.4: Measuring Performance of a block of code:

1. For Synchronous methods :

```
- (void)testPerformanceReverseString {
    NSString *originalString = @"hi_my_name_is_siddharth";
    [self measureBlock:^{
        [self.someObject reverseString:originalString];
    }];
}
```

2. For Asynchronous methods :

```
- (void)testPerformanceOfAsynchronousBlock {
   [self measureMetrics:@[XCTPerformanceMetric_WallClockTime] automaticallyStartMeasuring:YES
forBlock:^{

    XCTestExpectation *expectation = [self
expectationWithDescription:@"performanceTestWithResponse"];

    [self.someObject doSomethingThatTakesSomeTimesWithCompletionBlock:^(NSString *result) {
        [expectation fulfill];
    }];
    [self waitForExpectationsWithTimeout:5.0 handler:^(NSError *error) {
    }];
}];
}
```

- These performance measure block gets executed for 10 times consecutively & then the average is calculated, & on the basis of this average performance result gets created & baseline is accepted for further evaluation.
- The performance result is compared with the previous test results & baseline with a customizable max standard deviation.

Section 36.5: Running Test Suits:

Run all tests by choosing Product > Test. Click the Test Navigator icon to view the status and results of the tests. You can add a test target to a project (or add a class to a test) by clicking the Add (plus) button in the bottom-left corner of the test navigator. To view the source code for a particular test, select it from the test list. The file opens in the source code editor.

Chapter 37: Fast Enumeration

Section 37.1: Fast enumeration of an NSArray with index.

This example shows how to use fast enumeration in order to traverse through an NSArray. With this way you can also track current object's index while traversing.

Suppose you have an array,

```
NSArray *weekDays = @[@"Monday", @"Tuesday", @"Wednesday", @"Thursday", @"Friday", @"Saturday",
@"Sunday"];
```

Now you can traverse through the array like below,

```
[weekDays enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {

    //... Do your usual stuff here

    obj  // This is the current object
    idx  // This is the index of the current object
    stop // Set this to true if you want to stop

}];
```

Section 37.2: Fast enumeration of an NSArray

This example shows how to use fast enumeration in order to traverse through an NSArray.

When you have an array, such as

```
NSArray *collection = @[@"fast", @"enumeration", @"in objc"];
```

You can use the `for ... in` syntax to go through each item of the array, automatically starting with the first at index `0` and stopping with the last item:

```
for (NSString *item in collection) {
    NSLog(@"item: %@", item);
}
```

In this example, the output generated would look like

```
// item: fast
// item: enumeration
// item: in objc
```

Chapter 38: NSSortDescriptor

Section 38.1: Sorted by combinations of NSSortDescriptor

```
NSArray *aryFName = @[ @"Alice", @"Bob", @"Charlie", @"Quentin" ];
NSArray *aryLName = @[ @"Smith", @"Jones", @"Smith", @"Alberts" ];
NSArray *aryAge = @[ @24, @27, @33, @31 ];
```

```
//Create a Custom class with properties for firstName & lastName of type NSString *,
//and age, which is an NSUInteger.

NSMutableArray *aryPerson = [NSMutableArray array];
[firstNames enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
    Person *person = [[Person alloc] init];
    person.firstName = [aryFName objectAtIndex:idx];
    person.lastName = [aryLName objectAtIndex:idx];
    person.age = [aryAge objectAtIndex:idx];
    [aryPerson addObject:person];
}];

NSSortDescriptor *firstNameSortDescriptor = [NSSortDescriptor sortDescriptorWithKey:@"firstName"
                                             ascending:YES
                                             selector:@selector(localizedStandardCompare:)];

NSSortDescriptor *lastNameSortDescriptor = [NSSortDescriptor sortDescriptorWithKey:@"lastName"
                                            ascending:YES
                                            selector:@selector(localizedStandardCompare:)];

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

NSLog(@"By age: %@", [aryPerson sortedArrayUsingDescriptors:@[ageSortDescriptor]]);
// "Charlie Smith", "Quentin Alberts", "Bob Jones", "Alice Smith"


NSLog(@"By first name: %@", [aryPerson sortedArrayUsingDescriptors:@[firstNameSortDescriptor]]);
// "Alice Smith", "Bob Jones", "Charlie Smith", "Quentin Alberts"


NSLog(@"By last name, first name: %@", [aryPerson
sortedArrayUsingDescriptors:@[lastNameSortDescriptor, firstNameSortDescriptor]]);
// "Quentin Alberts", "Bob Jones", "Alice Smith", "Charlie Smith"
```

Chapter 39: Inheritance

1. @interface derived-class-Name: base-class-Name

Section 39.1: Car is inherited from Vehicle

Consider a base class **Vehicle** and its derived class **Car** as follows:

```
#import <Foundation/Foundation.h>

@interface Vehicle : NSObject

{
    NSString *vehicleName;
    NSInteger vehicleModelNo;
}

- (id)initWithName:(NSString *)name andModel:(NSInteger)modelno;
- (void)print;
@end

@implementation Vehicle

- (id)initWithName:(NSString *)name andModel:(NSInteger)modelno{
    vehicleName = name;
```

```
    vehicleModelNo = modelno;
    return self;
}

- (void)print{
    NSLog(@"Name: %@", vehicleName);
    NSLog(@"Model: %ld", vehicleModelNo);
}

@end

@interface Car : Vehicle

{
    NSString *carCompanyName;
}

- (id)initWithName:(NSString *)name andModel:(NSInteger)modelno
  andCompanyName:(NSString *)companyname;
- (void)print;

@end


@implementation Car

- (id)initWithName:(NSString *)name andModel:(NSInteger) modelno
  andCompanyName: (NSString *) companyname
  {
    vehicleName = name;
    vehicleModelNo = modelno;
    carCompanyName = companyname;
    return self;
}
- (void)print
{
    NSLog(@"Name: %@", vehicleName);
    NSLog(@"Model: %ld", vehicleModelNo);
    NSLog(@"Company: %@", carCompanyName);
}

@end

int main(int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSLog(@"Base class Vehicle Object");
    Vehicle *vehicle = [[Vehicle alloc]initWithName:@"4Wheeler" andModel:1234];
    [vehicle print];
    NSLog(@"Inherited Class Car Object");
    Car *car = [[Car alloc]initWithName:@"S-Class"
    andModel:7777 andCompanyName:@"Benz"];
    [car print];
    [pool drain];
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

2016-09-29 18:21:03.561 Inheritance[349:303] Base class Vehicle Object

2016-09-29 18:21:03.563 Inheritance[349:303] Name: 4Wheeler

2016-09-29 18:21:03.563 Inheritance[349:303] Model: 1234

2016-09-29 18:21:03.564 Inheritance[349:303] Inherited Class Car Object

2016-09-29 18:21:03.564 Inheritance[349:303] Name: S-Class

2016-09-29 18:21:03.565 Inheritance[349:303] Model: 7777

2016-09-29 18:21:03.565 Inheritance[349:303] Company: Benz

Chapter 40: NSTextAttachment

1. NSTextAttachment *attachmentName = [[NSTextAttachment alloc] init];

Section 40.1: NSTextAttachment Example

```
NSTextAttachment *attachment = [[NSTextAttachment alloc] init];
attachment.image = [UIImage imageNamed:@"imageName"];
attachment.bounds = CGRectMake(0, 0, 35, 35);
NSAttributedString *attachmentString = [NSAttributedString
attributedStringWithAttachment:attachment];
```

Chapter 41: NSUrl send a post request

Section 41.1: Simple POST request

```
// Create the request.
NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:[NSURL
URLWithString:@"http://google.com"]];

// Specify that it will be a POST request
request.HTTPMethod = @"POST";

// This is how we set header fields
[request setValue:@"application/xml; charset=utf-8" forHTTPHeaderField:@"Content-Type"];

// Convert your data and set your request's HTTPBody property
NSString *stringData = @"some data";
NSData *requestBodyData = [stringData dataUsingEncoding:NSUTF8StringEncoding];
request.HTTPBody = requestBodyData;

// Create url connection and fire request
NSURLConnection *conn = [[NSURLConnection alloc] initWithRequest:request delegate:self];
```

Section 41.2: Simple Post Request With Timeout

```
// Create the request.
NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:[NSURL
URLWithString:@"http://google.com"]];

// Specify that it will be a POST request
request.HTTPMethod = @"POST";

// Setting a timeout
request.timeoutInterval = 20.0;
// This is how we set header fields
[request setValue:@"application/xml; charset=utf-8" forHTTPHeaderField:@"Content-Type"];

// Convert your data and set your request's HTTPBody property
NSString *stringData = @"some data";
```

```
NSData *requestBodyData = [stringData dataUsingEncoding:NSUTF8StringEncoding];
request.HTTPBody = requestBodyData;

// Create url connection and fire request
NSURLConnection *conn = [[NSURLConnection alloc] initWithRequest:request delegate:self];
```

Chapter 42: BOOL / bool / Boolean / NSCFBoolean

Section 42.1: BOOL/Boolean/bool/NSCFBoolean

1. bool is a datatype defined in C99.
2. 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 are defined as 0, conditional statements on these nonexistent values are also evaluated as “false”.
3. BOOL is an Objective-C type defined as signed char with the macros YES and NO to represent true and false

From the definition in objc.h:

```
#if (TARGET_OS_IPHONE && __LP64__)  ||  TARGET_OS_WATCH
typedef bool BOOL;
#else
typedef signed char BOOL;
// BOOL is explicitly signed so @encode(BOOL) == "c" rather than "C"
// even if -funsigned-char is used.
#endif

#define YES ((BOOL)1)
#define NO  ((BOOL)0)
```

4. 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 property lists and collections. 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.

Section 42.2: BOOL VS Boolean

BOOL

- Apple's Objective-C frameworks and most Objective-C/Cocoa code uses BOOL.
- Use BOOL in objective-C, when dealing with any CoreFoundation APIs

Boolean

- Boolean is an old Carbon keyword , defined as an unsigned char

Chapter 43: Protocols and Delegates

Section 43.1: Implementation of Protocols and Delegation mechanism.

Suppose you have two views `ViewA` and `ViewB`

Instance of `ViewB` is created inside `ViewA`, so `ViewA` can send message to `ViewB's` instance, but for the reverse to happen we need to implement delegation (so that using delegate `ViewB's` instance could send message to `ViewA`)

Follow these steps to implement the delegation

1. In `ViewB` create protocol as

```
@protocol ViewBDelegate

-(void) exampleDelegateMethod;

@end
```

2. Declare the delegate in the sender class

```
@interface ViewB : UIView
@property (nonatomic, weak) id< ViewBDelegate > delegate;
@end
```

3. Adopt the protocol in Class ViewA

```
@interfac ViewA: UIView < ViewBDelegate >
```

4. Set the delegate

```
-(void) anyFunction
{
    // create Class ViewB's instance and set the delegate
    [viewB setDelegate:self];
}
```

5. Implement the delegate method in class `ViewA`

```
-(void) exampleDelegateMethod
{
    // will be called by Class ViewB's instance
}
```

6. Use the method in class `ViewB` to call the delegate method as

```
-(void) callDelegateMethod
{
    [delegate exampleDelegateMethod];
    //assuming the delegate is assigned otherwise error
}
```

Chapter 44: XML parsing

Section 44.1: XML Parsing

```
<?xml version="1.0" encoding="UTF-8"?>
<GeocodeResponse>
 <status>OK</status>
 <result>
  <type>premise</type>
  <formatted_address>4201 Oak Lawn Ave, Dallas, TX 75219, USA</
      formatted_address>
  <address_component>
   <long_name>4201</long_name>
   <short_name>4201</short_name>
   <type>street_number</type>
  </address_component>
  <address_component>
   <long_name>Oak Lawn Avenue</long_name>
   <short_name>Oak Lawn Ave</short_name>
   <type>route</type>
  </address_component>
  <address_component>
   <long_name>Oak Lawn</long_name>
   <short_name>Oak Lawn</short_name>
   <type>neighborhood</type>
   <type>political</type>
  </address_component>
  <address_component>
   <long_name>Dallas</long_name>
   <short_name>Dallas</short_name>
   <type>locality</type>
   <type>political</type>
  </address_component>
 </result>
</GeocodeResponse>
```

We will parse the highlighted tag data through `NSXMLParser`

We have declared few properties as follows

```
@property(nonatomic, strong)NSMutableArray *results;
@property(nonatomic, strong)NSMutableString *parsedString;
@property(nonatomic, strong)NSXMLParser *xmlParser;

//Fetch xml data
NSURLSession *session=[NSURLSession sessionWithConfiguration:[NSURLSessionConfiguration
defaultSessionConfiguration]];

NSURLSessionDataTask *task=[session dataTaskWithRequest:[NSURLRequest requestWithURL:[NSURL
URLWithString:YOUR_XMLURL]] completionHandler:^(NSData * _Nullable data, NSURLResponse * _Nullable
response, NSError * _Nullable error) {

self.xmlParser=[[NSXMLParser alloc] initWithData:data];
self.xmlParser.delegate=self;
if([self.xmlParser parse]){
   //If parsing completed successfully

        NSLog(@"%@",self.results);

}
```

```
}];

[task resume];
```

Then we define the `NSXMLParserDelegate`

```
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName namespaceURI:(nullable
NSString *)namespaceURI qualifiedName:(nullable NSString *)qName attributes:(NSDictionary<NSString
*, NSString *> *)attributeDict{

    if([elementName isEqualToString:@"GeocodeResponse"]){
        self.results=[[NSMutableArray alloc] init];
    }

    if([elementName isEqualToString:@"formatted_address"]){
        self.parsedString=[[NSMutableString alloc] init];
    }

}


- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string{

    if(self.parsedString){
        [self.parsedString appendString:[string stringByTrimmingCharactersInSet:[NSCharacterSet
whitespaceAndNewlineCharacterSet]]];
    }


}

- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName namespaceURI:(nullable
NSString *)namespaceURI qualifiedName:(nullable NSString *)qName{

    if([elementName isEqualToString:@"formatted_address"]){
        [self.results addObject:self.parsedString];

        self.parsedString=nil;
    }

}
```

Chapter 45: Declare class method and instance method

1. -(void)testInstanceMethod; //Class methods declare with "+" sign
2. (void)classMethod;//instance methods declare with "-" sign

Instance method are methods that are specific to particular classes. Instance methods are declared and defined followed by - (minus) symbol.

Class methods can be called by class name itself .Class methods are declared and defined by using + (plus)sign .

Section 45.1: How to declare class method and instance method.

instance methods use an instance of a class.

```
@interface MyTestClass : NSObject

- (void)testInstanceMethod;
```

```
@end
```

They could then be used like so:

```
MyTestClass *object = [[MyTestClass alloc] init];
[object testInstanceMethod];
```

Class method can be used with just the class name.

```
@interface MyClass : NSObject

+ (void)aClassMethod;

@end
```

They could then be used like so:

```
[MyClass aClassMethod];
```

class methods are the convenience methods on many Foundation classes like [NSString's +stringWithFormat:] or NSArray's +arrayWithArray

Chapter 46: Predefined Macros

1. **DATE** The current date as a character literal in "MMM DD YYYY" format
2. **TIME** The current time as a character literal in "HH:MM:SS" format
3. **FILE** This contains the current filename as a string literal.
4. **LINE** This contains the current line number as a decimal constant.
5. **STDC** Defined as 1 when the compiler complies with the ANSI standard.

ANSI C defines a number of macros. Although each one is available for your use in programming, the predefined macros should not be directly modified.

Section 46.1: Predefined Macros

```
#import <Foundation/Foundation.h>

int main()
{
   NSLog(@"File :%s\n", __FILE__ );
   NSLog(@"Date :%s\n", __DATE__ );
   NSLog(@"Time :%s\n", __TIME__ );
   NSLog(@"Line :%d\n", __LINE__ );
   NSLog(@"ANSI :%d\n", __STDC__ );

   return 0;
}
```

When the above code in a file main.m is compiled and executed, it produces the following result:

```
2013-09-14 04:46:14.859 demo[20683] File :main.m
2013-09-14 04:46:14.859 demo[20683] Date :Sep 14 2013
2013-09-14 04:46:14.859 demo[20683] Time :04:46:14
2013-09-14 04:46:14.859 demo[20683] Line :8
2013-09-14 04:46:14.859 demo[20683] ANSI :1
```

Chapter 47: NSCache

Section 47.1: NSCache

You use it the same way you would use NSMutableDictionary. The difference is that when NSCache detects excessive memory pressure (i.e. it's caching too many values) it will release some of those values to make room.

If you can recreate those values at runtime (by downloading from the Internet, by doing calculations, whatever) then NSCache may suit your needs. If the data cannot be recreated (e.g. it's user input, it is time-sensitive, etc.) then you should not store it in an NSCache because it will be destroyed there.

Chapter 48: Grand Central Dispatch

Grand Central Dispatch (GCD) In iOS, Apple provides two ways to do multitasking: The Grand Central Dispatch (GCD) and NSOperationQueue frameworks.We will discuss here about GCD. GCD is a lightweight way to represent units of work that are going to be executed concurrently You don't schedule these units of work; the system takes care of scheduling for you. Adding dependency among blocks can be a headache. Canceling or suspending a block creates extra work for you as a developer!

Section 48.1: What is Grand central dispatch.

What is Concurrency?

- Doing multiple things at the same time.
- Taking advantage of number of cores available in multicore CPUs.
- Running multiple programs in parallel.

Objectives of Concurrency

- Running program in background without hogging CPU.
- Define Tasks, Define Rules and let the system take the responsibility of performing them.
- Improve responsiveness by ensuring that the main thread is free to respond to user events.

DISPATCH QUEUES

Grand central dispatch – dispatch queues allows us to execute arbitrary blocks of code either asynchronously or synchronously All Dispatch Queues are first in – first out All the tasks added to dispatch queue are started in the order they were added to the dispatch queue.

Chapter 49: Continue and Break!

Section 49.1: Continue and Break Statement

The continue statement in Objective-C programming language works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.

For the for loop, continue statement causes the conditional test and increment portions of the loop to execute. For the while and do...while loops, continue statement causes the program control pass to the conditional tests.

```
#import <Foundation/Foundation.h>
```

```
int main ()
{
   /* local variable definition */
   int a = 10;

   /* do loop execution */
   do
   {
      if( a == 15)
      {
         /* skip the iteration */
         a = a + 1;
         continue;
      }
      NSLog(@"value of a: %d\n", a);
      a++;

   }while( a < 20 );

   return 0;
}
```

Output:

```
2013-09-07 22:20:35.647 demo[29998] value of a: 10
2013-09-07 22:20:35.647 demo[29998] value of a: 11
2013-09-07 22:20:35.647 demo[29998] value of a: 12
2013-09-07 22:20:35.647 demo[29998] value of a: 13
2013-09-07 22:20:35.647 demo[29998] value of a: 14
2013-09-07 22:20:35.647 demo[29998] value of a: 16
2013-09-07 22:20:35.647 demo[29998] value of a: 17
2013-09-07 22:20:35.647 demo[29998] value of a: 18
2013-09-07 22:20:35.647 demo[29998] value of a: 19
```

Refer to this link for more information.

Chapter 50: Format-Specifiers

- %@ //String
- %d //Signed 32-bit integer
- %D //Signed 32-bit integer
- %u //Unsigned 32-bit integer
- %U //Unsigned 32-bit integer
- %x //Unsigned 32-bit integer in lowercase hexadecimal format
- %X //Unsigned 32-bit integer in UPPERCASE hexadecimal format
- %o //Unsigned 32-bit integer in octal format
- %O //Unsigned 32-bit integer in octal format
- %f //64-bit floating-point number
- %F //64-bit floating-point number printed in decimal notation
- %e //64-bit floating-point number in lowercase scientific notation format
- %E //64-bit floating-point number in UPPERCASE scientific notation format
- %g //special case %e which uses %f when less than 4 sig-figs are available, else %e
- %G //special case %E which uses %f when less than 4 sig-figs are available, else %E
- %c //8-bit unsigned character
- %C //16-bit UTF-16 code unit
- %s //UTF8 String
- %S //16-bit variant of %s

- %p //Void Pointer in lowercase hexidecmial format with leading '0x'
- %zx //special case %p which removes leading '0x' (For use with no-type cast)
- %a //64-bit floating-point number in scientific notation with leading '0x' and one hexadecimal digit before the decimal point using a 'p' to notate the exponent.
- %A //64-bit floating-point number in scientific notation with leading '0x' and one hexadecimal digit before the decimal point using an 'P' to notate the exponent.

Format-Specifiers are used in Objective-c to implant object-values into a string.

Section 50.1: Integer Example - %i

```
int highScore = 57;
NSString *scoreBoard = [NSString stringWithFormat:@"HighScore: %i", (int)highScore];

NSLog(scoreBoard);//logs "HighScore: 57"
```

Chapter 51: NSObject

- self
- superclass
- init
- alloc
- new
- isEqual
- isKindOfClass
- isMemberOfClass
- description

`NSObject` is the root class of `Cocoa`, however the `Objective-C` language itself does not define any root classes at all its define by `Cocoa`, Apple's Framework.This root class of most Objective-C class hierarchies, from which subclasses inherit a basic interface to the runtime system and the ability to behave as Objective-C objects.

This class have all basic property of `Objective'C` class object like:

self.

class (name of the class).

superclass (superclass of current class).

Section 51.1: NSObject

`@interface NSString : NSObject` (`NSObject` is a base class of NSString class).

You can use below methods for allocation of string class:

```
- (instancetype)init

+ (instancetype)new

+ (instancetype)alloc
```

For Copy any object :

```
- (id)copy;

- (id)mutableCopy;
```

For compare objects :

```
- (BOOL)isEqual:(id)object
```

To get superclass of current class :

```
superclass
```

To check which kind of class is this ?

```
- (BOOL)isKindOfClass:(Class)aClass
```

Some property of NON-ARC classes:

```
- (instancetype)retain OBJC_ARC_UNAVAILABLE;

- (oneway void)release OBJC_ARC_UNAVAILABLE;

- (instancetype)autorelease OBJC_ARC_UNAVAILABLE;

- (NSUInteger)retainCount
```

Chapter 52: Modern Objective-C

Section 52.1: Literals

Modern Objective C provides ways to reduce amount of code you need to initialize some common types. This new way is very similar to how NSString objects are initialized with constant strings.

NSNumber

Old way:

```
NSNumber *number = [NSNumber numberWithInt:25];
```

Modern way:

```
NSNumber *number = @25;
```

Note: you can also store `BOOL` values in `NSNumber` objects using `@YES`, `@NO` or `@(someBoolValue)`;

NSArray

Old way:

```
NSArray *array = [[NSArray alloc] initWithObjects:@"One", @"Two", [NSNumber numberWithInt:3],
@"Four", nil];
```

Modern way:

```
NSArray *array = @[@"One", @"Two", @3, @"Four"];
```

NSDictionary

Old way:

```
NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys: array, @"Object", [NSNumber
numberWithFloat:1.5], @"Value", @"ObjectiveC", @"Language", nil];
```

Modern way:

```
NSDictionary *dictionary = @{@"Object": array, @"Value": @1.5, @"Language": @"ObjectiveC"};
```

Section 52.2: Container subscripting

In modern Objective C syntax you can get values from `NSArray` and `NSDictionary` containers using container subscripting.

Old way:

```
NSObject *object1 = [array objectAtIndex:1];
NSObject *object2 = [dictionary objectForKey:@"Value"];
```

Modern way:

```
NSObject *object1 = array[1];
NSObject *object2 = dictionary[@"Value"];
```

You can also insert objects into arrays and set objects for keys in dictionaries in a cleaner way:

Old way:

```
// replacing at specific index
[mutableArray replaceObjectAtIndex:1 withObject:@"NewValue"];
// adding a new value to the end
[mutableArray addObject:@"NewValue"];

[mutableDictionary setObject:@"NewValue" forKey:@"NewKey"];
```

Modern way:

```
mutableArray[1] = @"NewValue";
mutableArray[[mutableArray count]] = @"NewValue";

mutableDictionary[@"NewKey"] = @"NewValue";
```

Chapter 53: NSUserDefaults

Section 53.1: Simple example

For example:

FOR SAVING:

```
 NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];

 // saving an NSString
 [prefs setObject:txtUsername.text forKey:@"userName"];
 [prefs setObject:txtPassword.text forKey:@"password"];

 [prefs synchronize];
```

FOR RETRIEVING

```
 NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];

 // getting an NSString
NSString *savedUsername = [prefs stringForKey:@"userName"];
NSString *savedPassword = [prefs stringForKey:@"password"];
```

Section 53.2: Clear NSUserDefaults

```
NSString *appDomain = [[NSBundle mainBundle] bundleIdentifier];
[[NSUserDefaults standardUserDefaults] removePersistentDomainForName:appDomain];
```

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Adam	Chapter 13
Adriana Carelli	Chapter Chapter 54
Albert Renshaw	Chapters 6, 9 and 50
Ali Riahipour	Chapter 1
Amit Kalghatgi	Chapter 27
aniket.ghode	Chapter 23
AnthoPak	Chapter 7
Arc676	Chapter 32
atroutt	Chapter 4
BB9z	Chapter 3
Bharath	Chapter 7
BKO	Chapters 39 and 40
byJeevan	Chapter 28
Caleb Kleveter	Chapter 15
Chris Prince	Chapter 6
CodeChanger	Chapter 51
connor	Chapters 6, 34 and 35
Cory Wilhite	Chapter 5
danh	Chapter 3
Daniel Bocksteger	Chapter 17
DarkDust	Chapters 4, 6, 8, 9, 12, 20, 27 and 30
Darshan Kunjadiya	Chapter 9
DavidA	Chapters 6 and 23
dgatwood	Chapter 12
Dipen Panchasara	Chapter 22
Doc	Chapters 11, 16, 17, 20 and 33
Doron Yakovlev	Chapter 17
Ekta Padaliya	Chapter 7
Fantini	Chapters 3 and 10
Faran Ghani	Chapter 4
ff10	Chapter 37
gadu	Chapter 11
Håvard	Chapters 2 and 4
HCarrasko	Chapters 6, 7, 8 and 23
Hemang	Chapter 32
il Malvagio Dottor Prosciutto	Chapter 9
insys	Chapters 1, 2, 3, 4, 5 and 10
iphonic	Chapter 44
J F	Chapters 1, 3 and 20
j.f.	Chapter 7
James P	Chapters 7, 9 and 26
Jason McDermott	Chapters 5 and 6
Jeff Wolski	Chapters 1, 3, 7 and 9
Jens Meder	Chapter 8
Johannes Fahrenkrug	Chapters 7 and 24
Johnny Rockex	Chapter 9
Jon Schneider	Chapter 9
Joost	Chapter 23
Josh Brown	Chapter 1

Josh Caswell	Chapters 3, 7, 9 and 35
Joshua	Chapters 7, 9 and 15
jsondwyer	Chapters 7, 15, 19 and 21
Kote	Chapter 3
Losiowaty	Chapter 7
lostInTransit	Chapter 17
malicedShade	Chapter 11
Md. Ibrahim Hassan	Chapters 9, 41, 42 and 49
Mikhail Larionov	Chapter 17
mrtnf	Chapters 7 and 15
mszaro	Chapter 7
Muhammad Zohaib Ehsan	Chapters 7 and 35
Mykola Denysyuk	Chapters 2 and 4
Nef10	Chapter 20
Nicolas Miari	Chapter 6
Nikolai Ruhe	Chapters 9 and 21
Nirav Bhatt	Chapter 5
njuri	Chapter 7
NobodyNada	Chapters 6 and 20
NSNoob	Chapters 8 and 9
Nullify	Chapter 11
ok404	Chapter 26
Orlando	Chapters 4 and 9
Patrick	Chapters 2, 9, 10, 11, 14, 18, 21, 31 and 32
Paulo Fierro	Chapters 4 and 7
pckill	Chapter 52
Peter DeWeese	Chapter 27
Peter N Lewis	Chapter 6
phi	Chapter 4
Rahul	Chapter 38
RamenChef	Chapter 9
Ravi Dhorajiya	Chapter 29
Sanjay Mohnani	Chapter 43
shuvo	Chapter 37
Siddharth Sunil	Chapter 36
Sietse	Chapter 35
Spidy	Chapter 23
Stephen Leppik	Chapter 44
StrAbZ	Chapters 1 and 2
Sujania	Chapters 10, 13, 14, 15, 16, 18, 23, 25, 31, 33 and 35
Sunil Sharma	Chapter 9
Tamarous	Chapter 26
Tapan Prakash	Chapters 6 and 15
tbodt	Chapters 3, 6, 7, 9, 12, 15, 23 and 26
ThatsJustCheesy	Chapter 5
Thomas Tempelmann	Chapter 6
Tricertops	Chapters 6, 9, 12 and 20
user459460	Chapters 13, 17, 31, 42, 45, 46, 47 and 48
william205	Chapter 23
Yevhen Dubinin	Chapter 3
Zoltán	Chapter 11

You may also like