

W1D5

- **Selectors**
 - **More On Protocols**
 - **NSNumber**
 - **NSNumber**
 - **Categories**
 - **Class Extensions**
-

Selectors:

- Used to identify (point to) a method.
- Selectors can be used to execute any method.
- They can be passed around.
- The selector is just the method name minus return type, and internal parameter names & types.
- This is called the "signature" of the method.
- They come up all over Cooatouch including delegetion, and the target action pattern.
- 2 common ways to get a selector:

```
1
2 // Example 1
3 - (void)fly {}
4 - (void)nameOfMethod { }
5
6 - (void)testTwoWaysToCreateSelector {
7
8     //1. compile time
9     SEL aSelector1 = @selector(fly);
10
11     //2. run time
12     NSString *name = @"name";
13     NSString *of = @"Of";
14     NSString *method = @"Method";
15     NSString *stringFromComponents = [NSString
16     stringWithFormat:@"%@@%@", name, of, method];
17     SEL aSelector2 =
18     NSSelectorFromString(stringFromComponents);
19     BOOL result1 = [W1D5Tests
20     instancesRespondToSelector: aSelector1];
21     BOOL result2 = [W1D5Tests
22     instancesRespondToSelector: aSelector2];
23     XCTAssertTrue(result1);
24     XCTAssertTrue(result2);
25 }
```

```
1
2 // Example 2
3
```

```

4 - (void)testSelectorWithOneParam {
5     // notice the colon
6     SEL mySelector = @selector(myMethodWithData:);
7     [self performSelectorOnMainThread: mySelector
withObject:[NSData new] waitUntilDone: YES];
8     XCTAssertTrue(self.wasCalled);
9 }
10
11 // Example 3
12
13 - (void)testSelectorWithTwoParams {
14     [self
performSelector:@selector(fullNameWithFirstName:
lastName:) withObject:@"Joe" withObject:@"Blow"];
15     XCTAssertTrue(self.wasCalled);
16 }
17
18 - (void)myMethodWithData:(NSData *)data {
19     self.wasCalled = YES;
20 }
21
22 - (void)fullNameWithFirstName:(NSString *)first
23     |         |         |         |         | lastName:(NSString *)last {
24     self.wasCalled = YES;
25 }
26

```

- Many framework methods expect a selector as a parameter.

- For instance, if we want to programmatically setup a target-action on a button (we'll be discussing UIButton next week) we will call the instance method:

```
1
2  // Example 4
3
4  /*
5  // definition of a method that adds a target/action
   to a button
6  - (void)addTarget:(id)target action:(SEL)action
   forControlEvents:(UIControlEvents)controlEvents;
7  */
8
9  - (void)testButtonSelectorArgument {
10     // adding it to a button
11     UIButton *button = [[UIButton alloc]
   initWithFrame:CGRectZero];
12     [button addTarget: self
   action:@selector(buttonTapped:)
   forControlEvents:UIControlEventTouchUpInside];
13     [self
   performSelectorOnMainThread:@selector(buttonTapped:)
   withObject: self waitUntilDone: YES];
14     XCTAssertTrue(self.wasCalled);
15
16 }
17
18 // Method called when the button is tapped
19 - (void)buttonTapped:(UIButton *)sender {
```

```
20     self.wasCalled = YES;
21 }
22
```

- A common use of selectors is to test whether an object can handle a message.
- You can test whether an object responds to a message; if it does, send the message.
- This is used primarily when you have optional protocol methods. (We will see lots of these next week).

```
1
2 // Protocol
3 @protocol MyProtocol <NSObject>
4 @optional
5 - (void)myOptionalMethod;
6 @end
7
8 // MyObject Class
9 @interface MyObject : NSObject <MyProtocol>
10 @end
11 @implementation MyObject
12 - (void)myOptionalMethod {
13     NSLog(@"%s was called", __PRETTY_FUNCTION__);
14 }
15 @end
16
17 // Note: The protocol can be pasted above the test
    @implementation declaration
18
```

```

19 // Example 5
20
21 - (void)testSelector {
22
23     MyObject *myObject = [MyObject new];
24
25     SEL mySelector = @selector(myOptionalMethod);
26
27     BOOL respondsToSelector = [myObject
    respondsToSelector:@selector(myOptionalMethod)];
28     XCTAssertTrue(respondsToSelector);
29
30     // since you know it responds you can send it the
    message
31     if (respondsToSelector) {
32         [myObject myOptionalMethod];
33     }
34 }
35

```

- This is a handy way of sorting an array using a selector.

```

1
2 - (void)testArraySort {
3     NSArray *unsorted = @[@"Hello", @"Light",
    @"House", @"Labs"];
4     NSArray *sorted = [unsorted
    sortedArrayUsingSelector:@selector(compare:)];
5     NSArray *expected = @[@"Hello", @"House",

```

```
@ "Labs", @"Light"];  
6     XCTAssert([sorted isEqualToArray:expected]);  
7 }  
8
```

More Protocols & Delegation

What are protocols?

- In the real world protocols consist of sets of agreed upon procedures, rules or conventions for doing stuff.
- E.g. police should follow a legally binding protocol when making an arrest.
- They read you your rights in a specific format, etc.
- Computers communicate on the internet using the *http protocol*.
- The *http protocol* defines the expected request and the expected response data and format.
- There would be no internet without a shared protocol. Actually there would be no civilization without them since they are the basis of communication.
- In iOS a protocol usually consists of a group of method signatures that any conforming class agrees to implement.
- Protocol methods can be optional or required.

- Required methods *must* be implemented.
- Optional methods *need not* be implemented. So, we always have to check whether it responds.
- Protocols are similar to interfaces in other languages.

Why are protocols important?

- Protocols are used everywhere in Cocoa and CocoaTouch especially as part of the *delegate* design pattern.
- If some class agrees to implement a protocol, then other objects can communicate with this object without needing to know any other details about the object.
- This is a good example of *loose coupling*. Why is "loose coupling" a good OO design principle?
- Identifying objects just by their conformance to a protocol is a big deal in many design patterns.

Protocol Syntax

```
1
2 #pragma mark - Protocol
3
4 // the protocol could be defined in a separate file
   and imported instead
5 // #import "AnotherProtocol.h"
6
7
```



```
8 // Protocols can inherit from other protocols
9 @protocol MyProtocol<NSObject>
10 - (void)putYourMethodsHere;
11 @end
12
13 // Optional/required
14
15 @protocol AnotherProtocol<MyProtocol>
16
17 // @required is default and this method is inherited
    from <MyProtocol> so no need to redeclare its method
18 // - (void)putYourMethodsHere;
19
20 @optional
21 - (void)optionalMethod;
22 // use @required to switch back
23
24 @required
25 - (NSString*)requiredAgain;
26 @end
27
28
29 #pragma mark - Class
30 // Conforming To A Protocol
31
32 @interface MyClass:NSObject<AnotherProtocol>
33 // don't put the signatures in the header
34 @end
35
36 @implementation MyClass
```

```
37
38 // required
39 - (NSString*)requiredAgain {
40     return @"Some result";
41 }
42
43 // required
44 - (void)putYourMethodsHere {
45     // do stuff
46 }
47 @end
48
```

```
1
2 // Testing protocol conformance
3 #pragma mark - Tests
4
5 @interface ProtocolTests : XCTestCase
6 @end
7 @implementation ProtocolTests
8
9 - (void)testProtocolConformance {
10     BOOL conforms = [MyClass
11         conformsToProtocol:@protocol(AnotherProtocol)];
12     XCTAssertTrue(conforms);
13 }
14
15 - (void)testInstanceThatDoesNotConformToOptionalProtocol {
16     MyClass *myClass = [MyClass new];
17 }
18
```

```

16     BOOL conforms = [myClass
    respondsToSelector:@selector(optionalMethod)];
17     XCTAssertFalse(conforms);
18 }
19
20 - (void)test {
21     MyClass *myClass = [MyClass new];
22     NSString *result = [myClass requiredAgain];
23     XCTAssertTrue([result isEqualToString:@"Some
    result"]);
24 }
25

```

Example Of Protocols & Polymorphism

```

1
2 // Flyable.h
3 @protocol Flyable <NSObject>
4 - (NSString *)fly;
5 @end
6
7 // Duck.h
8 //import "Flyable.h"
9 @interface Duck : NSObject<Flyable>
10 @end
11
12 // Duck.m
13 //import "Duck.h"
14 @implementation Duck
15 - (NSString *)fly {

```

```

16         return @"flyin high!";
17     }
18 @end
19
20 // RubberDuck.h
21 //import "Flyable.h"
22 @interface RubberDuck : NSObject<Flyable>
23 @end
24
25 // RubberDuck.m
26 //import "RubberDuck.h"
27 @implementation RubberDuck
28 - (NSString *)fly {
29     return @"can't fly worth beans";
30 }
31 @end
32

```

```

1
2 - (NSString *)executeFlyableObject:
   (id<Flyable>)aFlyable {
3     return [aFlyable fly];
4 }
5
6 - (void)testDucks {
7     id<Flyable>duck = [Duck new];
8     id<Flyable>rubber = [RubberDuck new];
9     NSArray *flyableArray = @[duck, rubber];
10    // loop through
11    for (id<Flyable>item in flyableArray) {
12        NSString *result = [item fly];

```

```

13     NSLog(@"%d: %@", __LINE__, result);
14 }
15 NSString *result1 = [self
    executeFlyableObject:rubber]; // ==> can't fly worth
    beans
16 XCTAssert([result1 isEqualToString:@"can't fly
    worth beans"]);
17 NSString *result2 = [self
    executeFlyableObject:duck]; // ==> flyin high!
18 XCTAssert([result2 isEqualToString:@"flyin
    high!"]);
19 }
20

```

Simple Delegation Examples

Let's review the greeter example:

Player Example:

```

1
2 // Protocol
3 @protocol PlayerDelegate <NSObject>
4 - (NSString*)play;
5 @end
6
7 // Apple Service
8 @interface AppleMusicService :
    NSObject<PlayerDelegate>

```

```
9 @end
10
11 @implementation AppleMusicService
12 - (NSString*)play {
13     return @"playing apple music playlist";
14 }
15 @end
16
17 // Spotify Service
18 @interface SpotifyService : NSObject<PlayerDelegate>
19 @end
20
21 @implementation SpotifyService
22 - (NSString*)play {
23     return @"playing spotify playlist";
24 }
25 @end
26
27 // Player
28 @interface Player : NSObject
29 @property (nonatomic, weak)
30     id<PlayerDelegate>delegate;
31 - (instancetype)initWithMusicService:
32     (id<PlayerDelegate>)service
33     NS_DESIGNATED_INITIALIZER;
34 - (NSString *)play;
35 - (void)changeServiceTo:(id<PlayerDelegate>)service;
36 @end
37
38 @implementation Player
```

```
36
37 - (instancetype)initWithMusicService:
    (id<PlayerDelegate>)service {
38     if (self = [super init]) {
39         _delegate = service;
40     }
41     return self;
42 }
43
44 - (instancetype)init {
45     return [self initWithMusicService:nil];
46 }
47
48 // player doesn't know how to play
49 - (NSString *)play {
50     return [self.delegate play];
51 }
52
53 - (void)changeServiceTo:(id<PlayerDelegate>)service {
54     if ([service isKindOfClass:[self.delegate
class]]) {
55         return;
56     }
57     self.delegate = service;
58 }
59
60 @end
61
```

```
1
```

```
2 - (void)testPlayer {
```

```
3   AppleMusicService *appleMusic = [AppleMusicService
new];
4   SpotifyService *spotify = [SpotifyService new];
5   Player *player = [[Player alloc]
initWithMusicService:appleMusic];
6   NSString *result = [player play];
7   XCTAssertEqual(result, @"playing apple music
playlist");
8   [player changeServiceTo:spotify];
9   XCTAssertEqual([player.delegate class],
[SpotifyService class]);
10  result = [player play];
11  XCTAssertEqual(result, @"playing spotify
playlist");
12 }
13
14
```

ApplicationDelegate

- AppDelegate is the class that the framework sets up in main.m.
- The UIApplication object uses the AppDelegate to call for customization information, or to give your app a chance to respond to system events.

[Working with protocols](#)

NSNumber

- Light weight wrapper around primitive integer types.
- Most often used to include number values in collections in Objective-C.
- For instance, to include integers in an NSArray convert to NSNumber.
- What would be another way to add a primitive type to a collection in Objc?

```
1 // initializing them
2 // prefer literal instantiation
3 - (void)test {
4     NSNumber *num1 = [[NSNumber alloc] initWithInt:22];
5     NSNumber *num2 = [NSNumber numberWithFloat:12.2];
6     NSNumber *num3 = @(33);
7     NSNumber *num4 = @(YES); // BOOL
8     NSNumber *num5 = @('i'); // Char
9     NSArray *array = @[num1, num2, num3, num4, num5];
10    XCTAssertTrue([array[0] integerValue] == 22);
11    XCTAssertTrue([array[1] compare:[NSNumber
numberWithFloat:22.2]] == NSOrderedAscending);
12    XCTAssertTrue([array[2] isEqual:[NSNumber
numberWithInteger:33]]);
13    XCTAssertTrue([num4 integerValue] == 1); // BOOLS
are 1 or 0 in Objc, but never do this
14    XCTAssertTrue([array[4] charValue] == 'i');
15 }
16
```

- You may need to unbox NSNumbers to use them. Do it like this:

```
1
2 - (void)test {
3     NSInteger unwrappedNum1 = [arr[0] intValue];
4     NSLog(@"%lu", unwrappedNum1);
5     float unwrappedNum2 = [arr[1] floatValue];
6     NSLog(@"%f", unwrappedNum2);
7     NSInteger unwrappedNum3 = [arr[2] intValue];
8     NSLog(@"%lu", unwrappedNum3);
9     BOOL val = [arr[3] boolValue];
10    NSLog(@"%@", val ? @"YES" : @"NO");
11
12    // char: What will these logs print?
13    NSLog(@"char value boxed %@", arr[4]); // prints
        unicode value
14    NSLog(@"char value unboxed: %c", [arr[4] charValue]);
        // prints character i
15 }
16
```

- Some Tricks

```
1
2 - (void)test {
3     // using NSNumber's literal syntax as a dictionary
        key!
4     NSDictionary *dict = @{@"1":@"One", @"2":@"Two",
        @"3":@"Three"};
```

```

5
6 // looping: dict.allKeys gets an array of keys, but
  notice it has no definite order
7 // dictionaries are unordered
8
9 for (NSNumber *key in dict.allKeys) {
10     NSLog(@"%@", dict[key]);
11 }
12
13 NSInteger num5 = 44;
14 // logging primitive integer types by wrapping them
  in an NSNumber literal syntax
15 NSLog(@"logging an NSInteger by wrapping it: %@",
  @(num5));
16
17 // this is a quick way to get the string value of an
  integer type
18 NSString *num5ToString = @(num5).stringValue;
19
20 // this is the long way of doing the same thing
21 num5ToString = [NSString stringWithFormat:@"%d", 44];
22 }
23

```

- Comparing NSNumbers

```

1
2 - (void)testEquality {
3     NSNumber *num7 = @(22);
4     NSNumber *num8 = [NSNumber numberWithInt:22];

```

```
5    XCTAssertFalse(num7 == num8); // this is a pointer
    comparison, likely not what you want!
6
7    // comparing unwrapped values
8    XCTAssertTrue([num7 intValue] == [num8 intValue]);
9
10   XCTAssertTrue([num7 isEqualToNumber:num8]);
11 }
12
```

```
1
2 // This is another way of comparing, just a FYI,
   since you may see similar "sentinels" used elsewhere
3 // Don't do this for NSNumber (it's just an
   illustration). Instead unbox and compare the
   underlying integer values
4
5 - (void)testComparison {
6     NSNumber *num7 = @(22);
7     NSNumber *num8 = [NSNumber numberWithInt:23];
8     NSComparisonResult comparisonResult = [num7
compare:num8];
9
10    NSString *expected = @"ascending";
11    NSString *result;
12
13    if (comparisonResult == NSOrderedAscending) {
14        result = @"ascending";
15    } else if (comparisonResult == NSOrderedSame) {
16        result = @"same";
17    } else if (comparisonResult == NSOrderedDescending)
```

```
{
18     result = @"descending";
19 }
20 XCTAssertTrue([expected isEqualToString:result]);
21 }
22
```

NSValue

- It's a wrapper for C structs.
- Your main contact with C structs is through the graphics layer.
- If you need to put CGRects, etc. in an array or other collection you will want to use NSValue.

```
1
2 // Box CGRect with NSValue
3 - (void)testNSValueWithRect {
4     CGRect rect1 = CGRectMake(0.0, 0.0, 200.0,
5     200.0);
6     CGRect rect2 = CGRectMake(100.0, 0.0, 200.0,
7     200.0);
8     NSValue *rect1Box = [NSValue
9     valueWithRect:rect1];
10    NSValue *rect2Box = [NSValue
11    valueWithRect:rect2];
12    NSArray *rectArr = @[rect1Box, rect2Box];
13
14    CGRect rect1Unboxed = [rectArr[0] rectValue];
15
```

```
11     NSLog(@"rect1 unboxed: %@",  
    NSStringFromRect(rect1Unboxed));  
12     CGRect rect2Unboxed = [rectArr[1] rectValue];  
13     NSLog(@"rect2 unboxed: %@",  
    NSStringFromRect(rect2Unboxed));  
14 }  
15
```

- <http://rypress.com/tutorials/objective-c/data-types/nsnumber>
 - https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSNumber_Class/
 - https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSValue_Class/
-

Objective-C Categories

What are categories:

- They're called *Extensions* in Swift.
- Add functionality to existing classes without modifying original class.
- Can modify private system classes (that you can't even see!) without subclassing.
- Can be used to break up complex classes into logical

components.

- Allows flexibility of adding functionality as needed. For instance, I could add an extension to NSString but choose to only use it in some classes and not others. So, not every NSString in my project would automatically get the next behaviour (this isn't true in Swift BTW)

Category File Naming Convention

NameOfExtendedClass+NameOfExtension.h/.m

e.g.

NSString+Utilities.h/.m

- You need to import the category to get the functionality (in Objc).

Syntax

- Categories have interface + implementation (.h/.m) just like classes.
- But the syntax is a bit different than classes.
- Notice the round brackets that specify the name of the category after the class it's extending.
- There's no superclass after a colon as in classes.
- The name of the category is inside the round brackets after the name of the class being extended.

1

2 @interface NSString(Utils)

```
3 @end
4 @implementation NSString(Utills)
5 @end
6
7 // compared to classes
8
9 @interface MyClass: NSObject
10 @end
11 @implementation MyClass
12 @end
13
```

- Xcode will automatically create the files and stubs for you if you do this:

New File >> iOS Source >> Objective-C File >> Category

Choose a template for your new file:

iOS

Source

User Interface

Core Data

Apple Watch

Resource

Other

Realm

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

Choose options for your new file:

File:

File Type:

Class:

Cancel

Previous

Next

```
1 @interface NSString(Utills)
2 - (NSString *)addStar;
3 @end
4 @implementation NSString(Utills)
5 - (NSString *)addStar {
6     // notice SELF here is used to represent the
    NSString instance that receives this message
7     return [self stringByAppendingString:@"*"];
8 }
9 @end
10
11 - (void)testAddStar {
12     NSString *result = [@"something" addStar];
13     NSString *expected = @"something*";
```

```
14     XCTAssertTrue([result isEqualToString:expected]);
15 }
16
```

```
1
2 // More advanced NSString Extension that returns the
  vowels on an NSString
3 // NSString+Vowels.h
4 @interface NSString (Vowelize)
5 - (NSString *)vowelize;
6 @end
7
8 // NSString+Vowels.m
9
10 #import "NSString+Vowels.h"
11 @implementation NSString (Vowelize)
12 - (NSString *)vowelize {
13     NSMutableString *result = [NSMutableString
string];
14     if (self.length == 0) {
15         return [result copy];
16     }
17     NSString *comparator = @"aeiou";
18     // loop through string
19     for (NSInteger i = 0; i < self.length; ++i) {
20         NSRange range = NSMakeRange(i, 1);
21         NSString *subStr = [self
substringWithRange:range];
22         if ([comparator
localizedStandardContainsString:subStr]) {
23             [result appendString:subStr];

```

```

24         }
25     }
26     return [result copy];
27 }
28 @end
29
30 - (void)testVowelize {
31     NSString *vowels = @"my vowel experiment"
vowelize];
32     NSString *result = @"ooooie";
33     XCTAssert([vowels isEqualToString:result]);
34 }
35

```

Objective C Class Extension

- This is a way to add another interface to your classes that are *not* visible to outside classes.
- They were more commonly used for methods in early versions of Objc where you had to forward declare all methods.
- Modern Objc uses Class Extensions for properties only.
- Always start by adding your properties to the class extension and only move them to the header if they need to be exposed. Why do I say this?

```

1
2 // Simple example of class extension
3

```

```
4 @import Foundation; // Notice the modern importation
  syntax
5
6 @interface Person: NSObject
7
8 // Notice age is readonly
9 @property (nonatomic, readonly) NSInteger age;
10
11 - (instancetype)initWithName:(NSString *)name age:
  (NSInteger)age;
12
13 @end
14
15
16 #import "Person.h"
17
18 @interface Person()
19
20 // privately age is readwrite but publicly it's
  readonly
21 @property (nonatomic, readwrite) NSInteger age; //
  optional way of doing this, because you
22 @property (nonatomic) NSString *name;
23 @end
24
25 @implementation Person
26
27 // this is called the designated initializer
28 - (instancetype)initWithName:(NSString *)name age:
  (NSInteger)age {
```

```
29     if (self = [super init]) {
30         _name = name;
31         _age = age;
32     }
33     return self;
34 }
35
36 @end
37
38
```

```
1 - (void)test {
2
3     Person *person2 = [[Person alloc]
initWithName:@"JJ" age:10];
4     // name is inside the class extension so is not
visible from outside!
5     // XCTAssert(person2.name == @"JJ");
6     XCTAssert(person2.age == 10);
7 }
8
```

References:

[Working With Selectors](#)

[Cocoa Core Competencies](#)

