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

Selectors:

- A name used to identify (point to) a method.
- Selectors are used to execute a method.
- They can be passed around.
- The selector is just the method name minus return type, and internal parameter names & types.
- 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
stringWithFormat:@"%@%@%@", name, of, method];
16  SEL aSelector2 =
NSSelectorFromString(stringFromComponents);
17  BOOL result1 = [W1D5Tests
instancesRespondToSelector:aSelector1];
18  BOOL result2 = [W1D5Tests
instancesRespondToSelector:aSelector2];
19  XCTAssertTrue(result1);
20  XCTAssertTrue(result2);
21 }
22
23
```

```
1
2 // Example 2
3
4 - (void)testSelectors {
5   // notice the colon
6   SEL mySelector = @selector(myMethodWithData:);
7   [self performSelectorOnMainThread:mySelector
withObject:[NSData new] waitUntilDone:YES];
8   XCTAssertTrue(self.wasCalled);
9   self.wasCalled = NO;
10  [self
```

```
performSelector:@selector(fullNameWithFirstName:lastName:) withObject:@"Joe" withObject:@"Blow"];
  XCTAssertTrue(self.wasCalled);
}

- (void)myMethodWithData:(NSData *)data {
  self.wasCalled = YES;
}

- (void)fullNameWithFirstName:(NSString *)firstlastName:(NSString *)last {
  self.wasCalled = YES;
}

```

- 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:

```

// Example 3

/*
 // definition
 - (void)addTarget:(id)target action:(SEL)action forControlEvents:(UIControlEvents)controlEvents;
 */

- (void)testButtonSelectorArgument {
```

```
  // adding it to a button
  UIButton *aButton = [[UIButton alloc] initWithFrame:CGRectZero];
  [aButton addTarget:self action:@selector(buttonTapped:) forControlEvents:UIControlEventTouchUpInside];
  XCTAssertTrue(self.wasCalled);

}
// actual method the button calls when tapped
- (void)buttonTapped:(UIButton *)sender {
  self.wasCalled = YES;
}


```

- A common use of selectors is to test whether an object can handle a message.

```

// Protocol
@protocol MyProtocol <NSObject>
@optional
- (BOOL)someMethod;
@end

// MyObject Class
@interface MyObject : NSObject <MyProtocol>
@end
@implementation MyObject
- (BOOL)someMethod {
  return YES;
```

```
}
@end

// the protocol can be pasted above the test implementation declaration

// Example 4

- (void)testSelector {
  MyObject *myObject = [MyObject new];
  if ([myObject respondsToSelector:@selector(someMethod)]) {
    BOOL result = [myObject performSelector:@selector(someMethod)];
    XCTAssertTrue(result);
    // or, since it responds, do this:
    // [myObject someMethod];
  }
}

```

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

```

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

```
    XCTAssert([sorted isEqualToArray:expected]);
}

```

[Working With Selectors]

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 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.
- In iOS a protocol usually consists of a group of method signatures (and sometimes properties) 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 need to check whether an optional protocol method is implemented before sending the message.
- 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

```

// Protocols can inherit from other protocols
@protocol MyProtocol<NSObject>
- (void)putYourMethodsHere;
@end

```

```

// Optional/required
```

```

@protocol AnotherProtocol<MyProtocol>
// @required is default
- (void)putYourMethodsHere;
// optional
@optional
- (void)optionalMethod;
// use @required to switch back
@required
- (NSString*)requiredAgain;
@end


```

```
// Conformance syntax

@interface MyClass:NSObject<AnotherProtocol>
// don't put the signatures in the header
@end

@implementation MyClass

// required
- (NSString*)requiredAgain {
    return @"Some result";
}

// required
- (void)putYourMethodsHere {
    // do stuff
}
@end

```

```
// Testing protocol conformance
- (void)testProtocol {
    BOOL conforms = [MyClass conformsToProtocol:@protocol(AnotherProtocol)];
    XCTAssertTrue(conforms);
    MyClass *myClass = [MyClass new];
    BOOL responds = [myClass respondsToSelector:@selector(optionalMethod)];
    XCTAssertFalse(responds);
}

```

Example Of Protocols & Polymorphism

```

// Flyable.h
@protocol Flyable <NSObject>
- (NSString *)fly;
@end

// Duck.h
//#import "Flyable.h"
@interface Duck : NSObject<Flyable>
@end

// Duck.m
//#import "Duck.h"
@implementation Duck
- (NSString *)fly {
    return @"flyin high!";
}
```

```
@end

// RubberDuck.h
//#import "Flyable.h"
@interface RubberDuck : NSObject<Flyable>
@end

// RebberDuck.m
//#import "RubberDuck.h"
@implementation RubberDuck
- (NSString *)fly {
    return @"can't fly worth beans";
}
@end
```

```

- (NSString *)executeFlyableObject:(id<Flyable>)aFlyable {
  return [aFlyable fly];
}

- (void)testDucks {
  id<Flyable>duck = [Duck new];
  id<Flyable>rubber = [RubberDuck new];
  NSArray *flyableArray = @[duck, rubber];
  // loop through
  for (id<Flyable>item in flyableArray) {
    __unused NSString *result = [item fly];
  }
  NSString *result1 = [self executeFlyableObject:rubber]; // ==> can't fly worth beans
```

```
  XCTAssert([result1 isEqualToString:@"can't fly worth beans"]);
  NSString *result2 = [self executeFlyableObject:duck]; // ==> flyin high!
  XCTAssert([result2 isEqualToString:@"flyin high!"]);
}

```

Simple Delegation Example

```

// Protocol
@protocol PlayerDelegate <NSObject>
- (NSString*)play;
@end

// Apple Service
@interface AppleMusicService : NSObject<PlayerDelegate>
@end

@implementation AppleMusicService
- (NSString*)play {
  return @"playing apple music playlist";
}
@end

// Spotify Service
@interface SpotifyService : NSObject<PlayerDelegate>
@end

```

```
@implementation SpotifyService
- (NSString*)play {
  return @"playing spotify playlist";
}
@end

// Player
@interface Player : NSObject
@property (nonatomic, weak) id<PlayerDelegate>delegate;
- (instancetype)initWithMusicService:(id<PlayerDelegate>)service NS_DESIGNATED_INITIALIZER;
- (NSString *)play;
- (void)changeServiceTo:(id<PlayerDelegate>)service;
@end

@implementation Player

- (instancetype)initWithMusicService:(id<PlayerDelegate>)service  {
  if (self = [super init]) {
    _delegate = service;
  }
  return self;
}

- (instancetype)init {
  return [self initWithMusicService:nil];
}

// 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 isMemberOfClass:[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
```

Simple Delegate Callback Example

```
 1
 2 // DetailViewController.h
 3 @class DetailViewController;
 4 @protocol DetailDelegate <NSObject>
 5 - (NSString *)detail:(DetailViewController *)detail
   doStuffWithData:(NSString *)data;
 6 @end
 7
 8 @interface DetailViewController : NSObject
 9 @property (nonatomic, weak)
   id<DetailDelegate>delegate;
10 - (NSString *)saveFakeUserInput:(NSString *)input;
11 @end
12
13 // DetailViewController.m
14 //#import "DetailViewController.h"
15 @implementation DetailViewController
16 - (NSString *)saveFakeUserInput:(NSString *)input {
17   return [self.delegate detail:self
   doStuffWithData:input];
18 }
19 @end
20
21 // MasterViewController.h
22 @interface MasterViewController : NSObject
23 - (void)segueToDetailButtonTapped;
```

```
@property (nonatomic, strong) DetailViewController *detail;
@end

// MasterViewController.m
//#import "MasterViewController.h"

// class extension, not really used execept for conforming to DetailDelegate
@interface MasterViewController()<DetailDelegate>
@end
@implementation MasterViewController
- (void)segueToDetailButtonTapped {
  self.detail.delegate = self;
  // do segue to Detail
}

// called by master
- (NSString *)detail:(DetailViewController *)detail doStuffWithData:(NSString *)data {
  return [NSString stringWithFormat:@"data gathered from detail: %@", data];
  // dismiss detail
}

@end

```

```

- (void)test {
  MasterViewController *master = [MasterViewController new];
  DetailViewController *detail =
```

```
    [DetailViewController new];
5     master.detail = detail;
6     [master segueToDetailButtonTapped]; // tap the
    button
7     NSString *result = [detail
    saveFakeUserInput:@"some user input"]; // save some
    data on detail
8     XCTAssertTrue([result containsString:@"some user
    input"]);
9   }
10
```

Delegation in CocoaTouch

- ApplicationDelegate 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

```
// initializing them
// prefer literal instantiation
- (void)test {
  NSNumber *num1 = [[NSNumber alloc] initWithInt:22];
  NSNumber *num2 = [NSNumber numberWithFloat:12.2];
  NSNumber *num3 = @(33);
  NSNumber *num4 = @(YES); // BOOL
  NSNumber *num5 = @('i'); // Char
  NSArray *array = @[num1, num2, num3, num4, num5];
  XCTAssertTrue([array[0] integerValue] == 22);
  XCTAssertTrue([array[1] compare:[NSNumber numberWithFloat:22.2]] == NSOrderedAscending);
  XCTAssertTrue([array[2] isEqual:[NSNumber numberWithInteger:33]]);
  XCTAssertTrue([num4 integerValue] == 1); // BOOLS are 1 or 0 in Objc, but never do this
  XCTAssertTrue([array[4] charValue] == 'i');
  // pro tip
  NSInteger num = 12;
  NSLog(@"for logging out integers wrap them using literal syntax: %@", @(num));
}

```

- 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
```

```
for (NSNumber *key in dict.allKeys) {
    NSLog(@"%@", dict[key]);
}

NSInteger num5 = 44;
// logging primitive integer types by wrapping them in an NSNumber literal syntax
NSLog(@"logging an NSInteger by wrapping it: %@", @(num5));

// this is a quick way to get the string value of an integer type
NSString *num5ToString = @(num5).stringValue;

// this is the long way of doing the same thing
num5ToString = [NSString stringWithFormat:@"%d", 44];
}

```

- Comparing NSNumbers

```

// Question: What will the statement at line 7 log out and why?
- (void)testEquality {
    NSNumber *num7 = @(22);
    NSNumber *num8 = [NSNumber numberWithInteger:22];
    BOOL value2 = num7 == num8; // this is a pointer comparison, likely not what you want!
    NSLog(@"%@ is equal to %@: %@", num7, num8,
```

```
value2 ? @"YES" : @"NO");

    // do comparisons like this for NSNumber

    // unboxing to compare
    if ([num7 intValue] == [num8 intValue]) {
        NSLog(@"they're equal");
    }

    // comparing while boxed
    if ([num7 isEqualToNumber:num8]) {
        NSLog(@"they're equal yo");
    }
}
```

```

// This is another way of comparing, just a FYI, since you may see similar "sentinels" used elsewhere
// Don't do this for NSNumber (it's just an illustration)
- (void)testComparison {
    NSNumber *num7 = @(22);
    NSNumber *num8 = [NSNumber numberWithInteger:22];
    NSComparisonResult comparisonResult = [num7 compare:num8];
    if (comparisonResult == NSOrderedAscending) {
        NSLog(@"ascending");
    } else if (comparisonResult == NSOrderedSame) {
        NSLog(@"same");
    } else if (comparisonResult == NSOrderedDescending) {
```

```
        NSLog(@"descending");
    }
}

```

NSValue:

```

- (void)testNSValue {

    // Box C struct with NSValue
    // This is just an illustration, you normally will not deal directly with C Structs outside the graphics area which provides a lot of convenience methods to work with them (see below)

    /*
     typedef struct {
        int mark;
        char name[10];
        int average;
     } Student;
     */

    struct Student {
        int mark;
        char name[10];
        int average;
    };

```

```
    struct Student report1 = { 89, "James", 79 };
    struct Student report2 = { 77, "Sonya", 70 };

    NSValue *reportValue1 = [NSValue value:&report1 withObjCType:@encode(struct Student)];

    NSValue *reportValue2 = [NSValue value:&report2 withObjCType:@encode(struct Student)];

    NSArray *arr = @[reportValue1, reportValue2];

    struct Student result1;
    [arr[0] getValue:&result1];

    NSLog(@"%@", @(result1.average));
}

```

```

// Box CGRect with NSValue
- (void)testNSValueWithRect {
    CGRect rect1 = CGRectMake(0.0, 0.0, 200.0, 200.0);
    CGRect rect2 = CGRectMake(100.0, 0.0, 200.0, 200.0);
    NSValue *rect1Box = [NSValue valueWithRect:rect1];
    NSValue *rect2Box = [NSValue valueWithRect:rect2];
    NSArray *rectArr = @[rect1Box, rect2Box];

    CGRect rect1Unboxed = [rectArr[0] rectValue];
    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:

- 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)

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

- interface + implementation just like classes

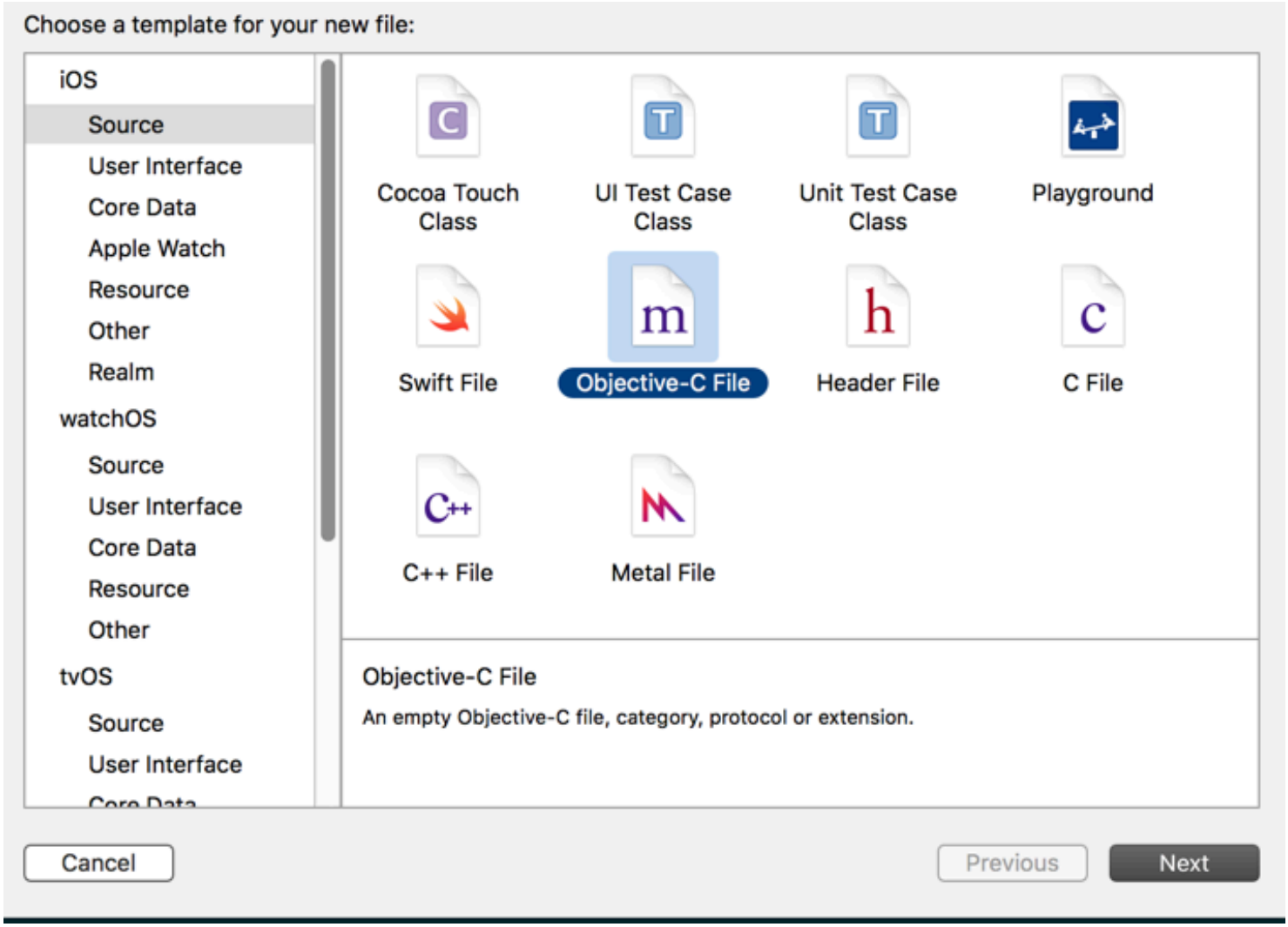
- But the syntax is a bit different than classes.
- Notice the round brackets.
- There's no superclass after a colon as in classes.
- The name of the category is inside the round brackets after the name of the class being extended.

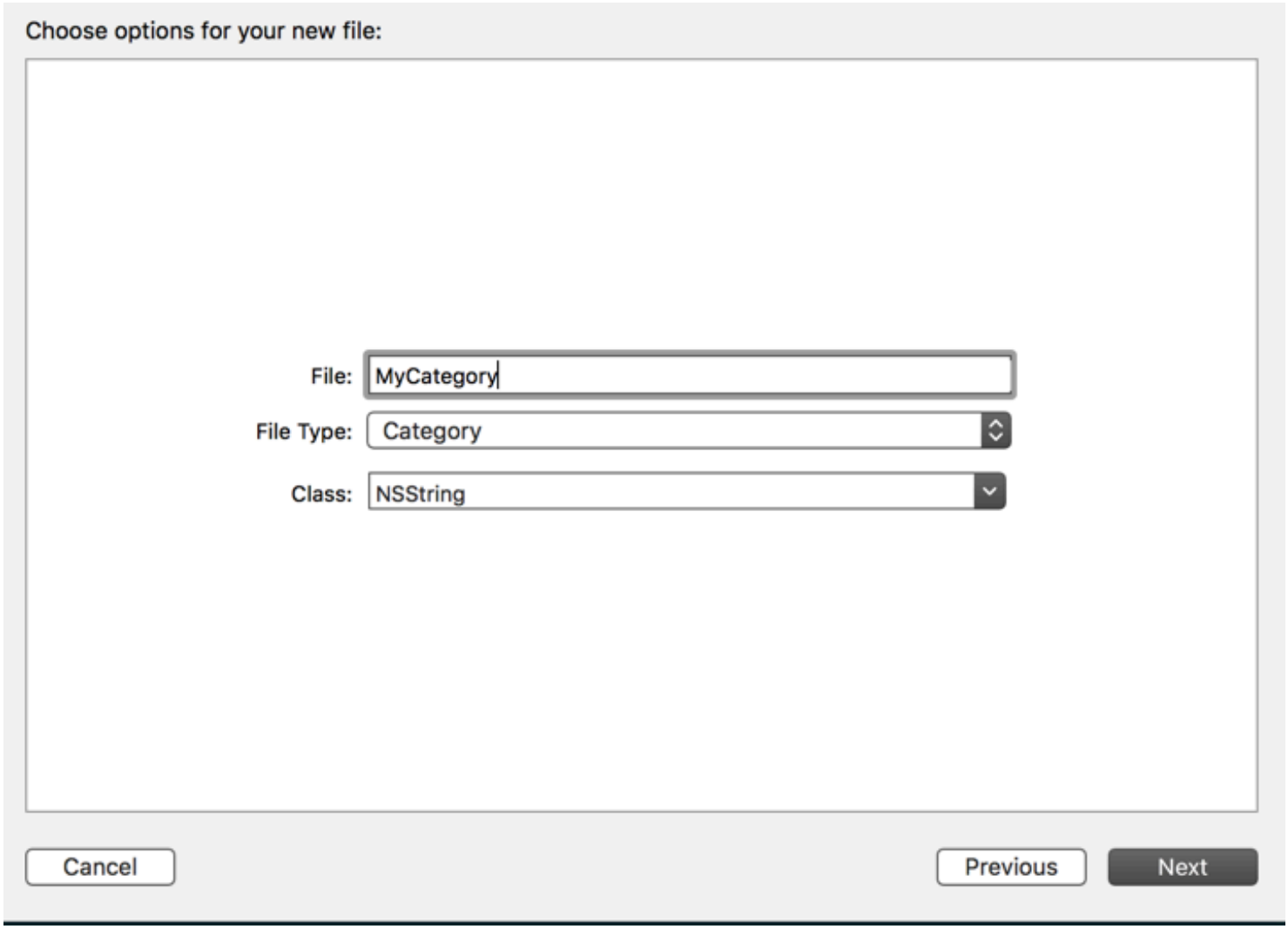
```
1
2 @interface NSString(Utils)
3 @end
4 @implementation NSString(Utils)
5 @end
```

6

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

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





- You refer to the object being acted on when you are inside the implementation as *self*.

e.g.

```
1 @interface NSString(Utils)
2 - (NSString *)addStar;
3 @end
4 @implementation NSString(Utils)
5 - (NSString *)addStar {
6     // notice SELF here to represent the NSString
  instance that receives this message
7     return [self stringByAppendingString:@"*"];
```

```
}
@end

- (void)test {
    NSString *s = [@"steve" addStar];
    XCTAssert([s isEqualToString:@"steve*"]);
}

```

```

// More advanced NSString Extension that returns the vowels on an NSString
// NSString+Vowels.h
@interface NSString (Vowelize)
- (NSString *)vowelize;
@end

// NSString+Vowels.m

#import "NSString+Vowels.h"
@implementation NSString (Vowelize)
- (NSString *)vowelize {
    NSMutableString *result = [NSMutableString string];
    if (self.length == 0) {
        return [result copy];
    }
    NSString *comparitor = @"aeiou";
    // loop through string
    for (NSInteger i = 0; i < self.length; ++i) {
        NSRange range = NSMakeRange(i, 1);
        NSString *subStr = [self substringWithRange:range];
```

```
22          if ([comparitor
    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 = @"oeeeie";
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?

```objc

// Simple example of class extension

// Person.h
@import Foundation; // Notice the modern importation syntax

@interface Person: NSObject
// Notice age is readonly
@property (nonatomic, readonly) NSInteger age;
- (instancetype)initWithName:(NSString *)name age:(NSInteger)age;
@end

// Person.m
// #import "Person.h"

// class extension. notice it's another interface on .m
@interface Person()
// privately it's readwrite but publicly it's readonly
@property (nonatomic, readwrite) NSInteger age; // optional way of doing this, because you can write to age using _age privately
@property (nonatomic) NSString *name;
@end

@implementation Person

// this is called the designated initializer
- (instancetype)initWithName:(NSString *)name age:
```

```
(NSInteger)age {
    if (self = [super init]) {
        _name = name;
        _age = age;
    }
    return self;
}

// overriding the default init and calling the designated initializer and passes in defaults
- (instancetype)init {
    return [self initWithName:nil age:0];
}
@end


```

```
- (void)test {
    Person *p1 = [[Person alloc] init];
    XCTAssert(p1.name == nil);
    XCTAssert(p1.age == 0);
    Person *p2 = [[Person alloc] initWithName:@"JJ" age:10];
    XCTAssert(p2.name = @"JJ");
    XCTAssert(p2.age == 10);
}
```

General References:

[Cocoa Core Competencies]