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

Selectors:

- A name used to refer to a method in order to execute it or pass it around
- Simply identifies a method.
- The selector of a method is just the method name minus return type, and parameters.

```
1
2 - (void)testSelectors {
3     // notice the colon
4     SEL mySelector = @selector(myMethodWithData:);
5
6     [self performSelectorOnMainThread:mySelector
  withObject:[NSData new] waitUntilDone:YES];
```

```
7
8       SEL myOtherSelector =
    @selector(myMethodWithString:value:);
9   }
10
11  - (void)myMethodWithData:(NSString *)data {}
12
13  - (NSData *)myMethodWithString:(NSString *)string
    value:(BOOL)value {
14      return [NSData new];
15  }
16
```

- 2 common ways to get a selector

```
1
2   - (void)testSelectors {
3       // compile time
4       SEL aSelector1 = @selector(fly);
5       // run time
6       SEL aSelector2 =
    NSSelectorFromString(@"nameOfMethod");
7   }
8
```

- 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  /*
3  // definition
4  - (void)addTarget:(id)target action:(SEL)action
   forControlEvents:(UIControlEvents)controlEvents;
5  */
6  - (void)testButtonSelectorArgument {
7      // adding it to a button
8      UIButton *aButton = [[UIButton alloc]
   initWithFrame:CGRectZero];
9      [aButton addTarget:self
   action:@selector(buttonTapped:)
   forControlEvents:UIControlEventTouchUpInside];
10 }
11 // actual method the button calls when tapped
12 - (void)buttonTapped:(UIButton *)sender {
13     // do stuff
14 }
15
16
```

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

```
1
2  // Protocol
3  @protocol MyProtocol <NSObject>
4  @optional
5  - (void)someMethod;
6  @end
7
8  // Class
```

```
@interface MyObj : NSObject<MyProtocol>
@end
@implementation MyObj
@end

- (void)testSelector {

    MyObj *myObj = [MyObj new];

    if ([myObj respondsToSelector:@selector(someMethod)]) {

        [myObj performSelector:@selector(someMethod)];

        // or, since it responds, do this:
        // [myObj someMethod];
    }
}

```

```

// you can perform a selector after a delay
- (void)testPerformSelectorAfterDelay {
    [self performSelector:@selector(myMethod) withObject:nil afterDelay:1.0];
}
- (void)myMethod {
    NSLog(@"works!");
}

```

- 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
```

[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

```
1
2 // Protocols can inherit from other protocols
3 @protocol MyProtocol<NSObject>
4 - (void)putYourMethodsHere;
5 @end
6
```

```
1
2 // Optional/required
3
4 @protocol AnotherProtocol<MyProtocol>
5 // @required is default
6 - (void)putYourMethodsHere;
7 // optional
8 @optional
9 - (void)optionalMethod;
10 // use @required to switch back
11 @required
12 - (NSString*)requiredAgain;
13 @end
14
15
```

```
1 // Conformance syntax
2
3 @interface MyClass:NSObject<AnotherProtocol>
4 // don't put the signatures in the header
5 @end
6
```

```
@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>bird1 = [Duck new];
    id<Flyable>bird2 = [RubberDuck new];
    NSArray *arr = @[bird1, bird2];
    for (id<Flyable>item in arr) {
        [item fly];
    }
    NSString *result1 = [self executeFlyableObject:bird2]; // ==> can't fly worth beans
    XCTAssert([result1 isEqualToString:@"can't fly worth beans"]);
    NSString *result2 = [self executeFlyableObject:bird1]; // ==> flyin high!
    XCTAssert([result2 isEqualToString:@"flyin high!"]);
}

```

Simple Delegation Example

```
// Basic Delegation Example Showing How To Get Another Object To Do Work For A Class
// This allows

#import <Foundation/Foundation.h>

// Protocol
@protocol PlayerDelegate <NSObject>
- (void)play;
```

```
@end

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

@implementation AppleMusicService
- (void)play {
    NSLog(@"playing apple music playlist");
}
@end

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

@implementation SpotifyService
- (void)play {
    NSLog(@"playing spotify playlist");
}
@end

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

```
38
39 @implementation Player
40
41 - (instancetype)initWithMusicService:
   (id<PlayerDelegate>)service {
42     if (self = [super init]) {
43         _delegate = service;
44     }
45     return self;
46 }
47
48 - (instancetype)init {
49     NSAssert(NO, @"Use designated initializer
   instead");
50     return nil;
51 }
52
53 - (void)play {
54     [self.delegate play];
55 }
56
57 - (void)changeServiceTo:(id<PlayerDelegate>)service
   {
58     if ([service isMemberOfClass:[self.delegate
   class]]) {
59         return;
60     }
61     self.delegate = service;
62 }
63
64 @end
65
```

```
1
2 - (void)testPlayer {
3     AppleMusicService *appleMusic =
  [AppleMusicService new];
4     SpotifyService *spotify = [SpotifyService new];
5     Player *player = [[Player alloc]
  initWithMusicService:appleMusic];
6     [player play];
7     [player changeServiceTo:spotify];
8     [player play];
9 }
10
```

Simple Delegate Callback Example

```
1
2 // Detail.h
3 @class Detail;
4 @protocol DetailDelegate <NSObject>
5 - (void)detail:(Detail *)detail doStuffWithData:
  (NSString *)data;
6 @end
7
8 @interface Detail : NSObject
9 @property (nonatomic, weak)
  id<DetailDelegate>delegate;
10 - (void)saveFakeUserInput:(NSString *)input;
11 @end
12
13 // Detail.m
14 //#import "Detail.h"
```

```
@implementation Detail
- (void)saveFakeUserInput:(NSString *)input {
    [self.delegate detail:self doStuffWithData:input];
}
@end

// Master.h
@interface Master : NSObject
- (void)fakeButtonTap;
@property (nonatomic, strong) Detail *detail;
@end

// Master.m
//#import "Master.h"

// class extension, not really used execept for conforming to DetailDelegate
@interface Master()<DetailDelegate>
@end
@implementation Master
- (void)fakeButtonTap {
    self.detail.delegate = self;
    // you might want to segue to detail
}

- (void)detail:(Detail *)detail doStuffWithData:(NSString *)data {
    NSLog(@"%s data: %@", __PRETTY_FUNCTION__, data);
}

```

```
@end

```

```
- (void)test {
    Master *m = [Master new];
    Detail *d = [Detail new];
    m.detail = d;
    [m fakeButtonTap];
    [d saveFakeUserInput:@"some user input"];
}
```

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

```
// 3 different ways to instantiate
// 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 *arr = @[num1, num2, num3, num4, num5];
}

```

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

```

- (void)test {
NSInteger unwrappedNum1 = [arr[0] intValue];
NSLog(@"%lu", unwrappedNum1);
float unwrappedNum2 = [arr[1] floatValue];
NSLog(@"%f", unwrappedNum2);
NSInteger unwrappedNum3 = [arr[2] intValue];
NSLog(@"%lu", unwrappedNum3);
BOOL val = [arr[3] boolValue];
NSLog(@"%@", val ? @"YES": @"NO");

// char: What will these logs print?
NSLog(@"char value boxed %@", arr[4]); // prints
unicode value
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
```

```
num5ToString = [NSString stringWithFormat:@"%d", 44];
}

```

- Comparing NSNumbers

```

// Question: What will the statement at line 7 log out and why?
- (void)test {
    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");
    }
}

```

```
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)
4 - (void)test {
5     NSNumber *num7 = @(22);
6     NSNumber *num8 = [NSNumber
  numberWithInteger:22];
7     NSComparisonResult comparisonResult = [num7
  compare:num8];
8     if (comparisonResult == NSOrderedAscending) {
9         NSLog(@"ascending");
10    } else if (comparisonResult == NSOrderedSame) {
11        NSLog(@"same");
12    } else if (comparisonResult ==
  NSOrderedDescending) {
13        NSLog(@"descending");
14    }
15 }
16
```

NSValue:

```
1
2 - (void)test {
3
4     // Box C struct with NSValue
5     // 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)test {
    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));
    CGRect rect2Unboxed = [rectArr[1] rectValue];
    NSLog(@"rect2 unboxed: %@", NSStringFromRect(rect2Unboxed));
}

// alternatively you can wrap and unwrap CG types using NSString convenience methods
- (void)test {
    CGRect rect1 = CGRectMake(0.0, 0.0, 200.0, 200.0);
    CGRect rect2 = CGRectMake(100.0, 0.0, 200.0, 200.0);
    NSString *rect1Str = NSStringFromCGRect(rect1);
    NSString *rect2Str = NSStringFromCGRect(rect2);
```

```
22      NSArray *rects = @[rect1Str, rect2Str];
23      CGRect rect3 = CGRectFromString(rects[0]);
24      CGRect rect4 = CGRectFromString(rects[1]);
25      XCTAssert(CGRectEqualToRect(rect1, rect3));
26      XCTAssert(CGRectEqualToRect(rect2, rect4));
27  }
28
```

- 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)

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

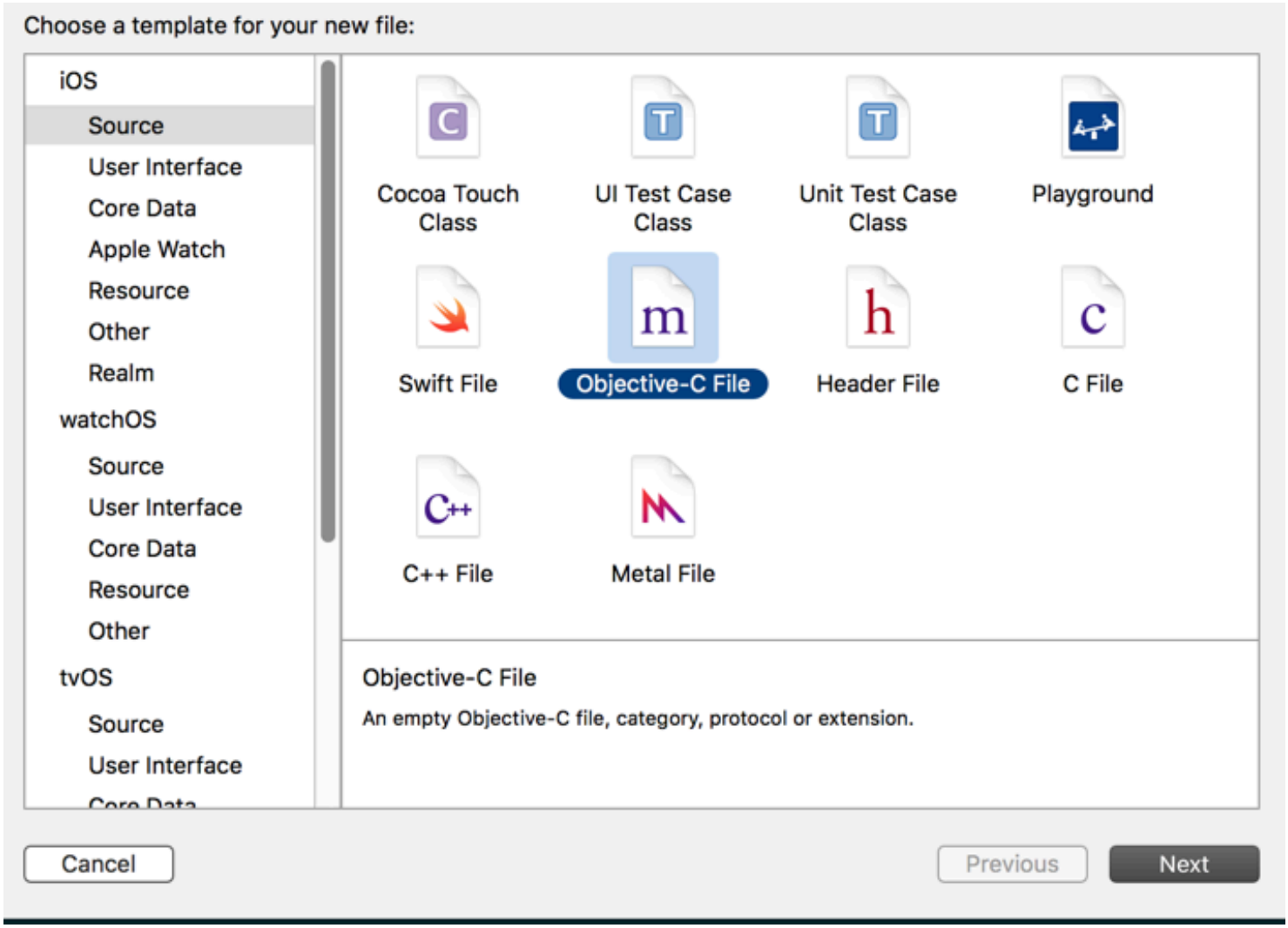
- interface + implementation just like classes

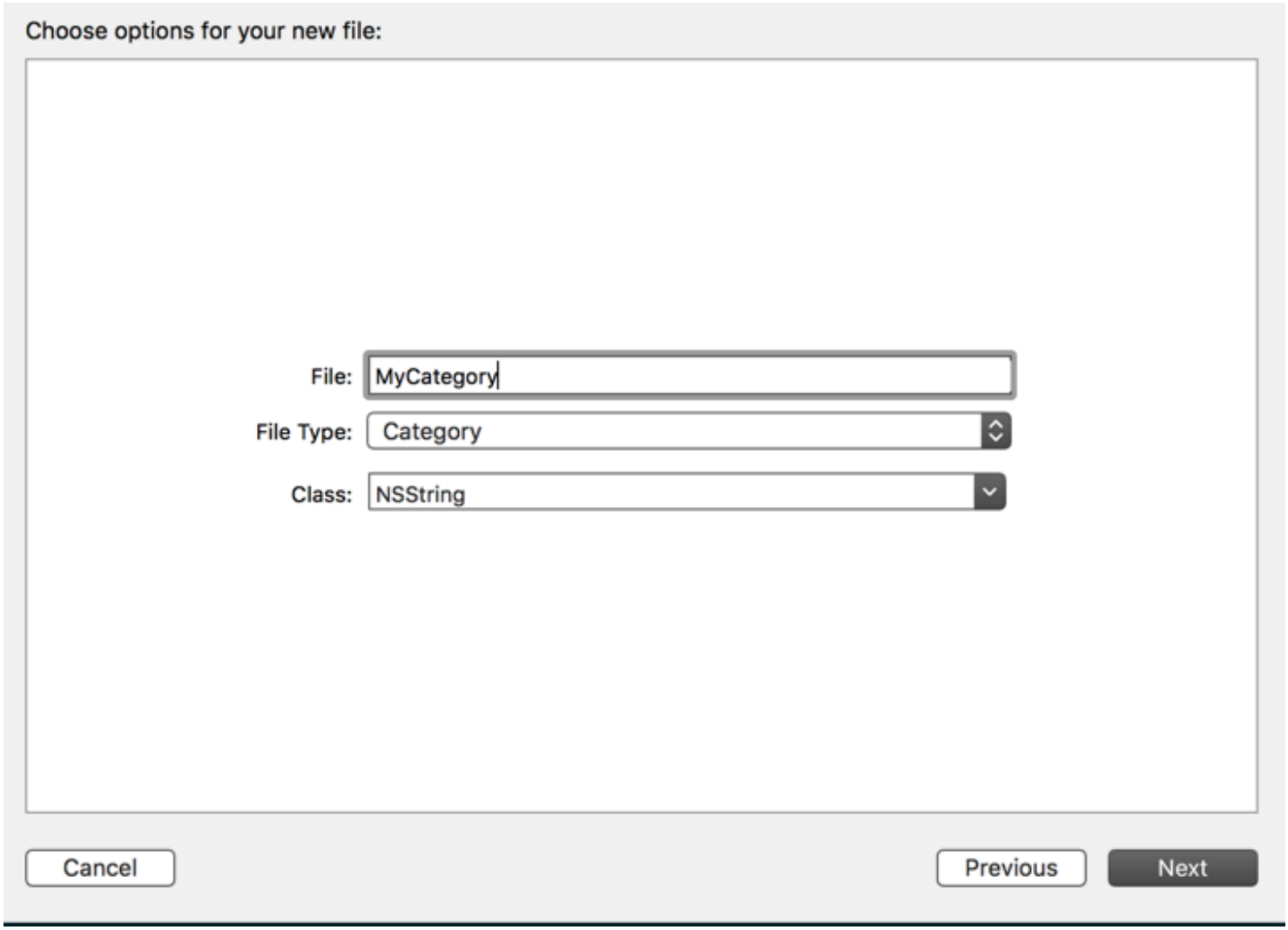
- But the syntax is a bit different than classes.
- Notice the round brackets.
- There's no superclass after a colon as in classes.
- The name of the category is inside the round brackets after the name of the class being extended.

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

- 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

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