```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSArray *myTodoList = @[@"Clean the house",
                            @"Feed the dog",
                            @"Take out the trash",
                            @"Fight crime"];
    NSString *firstThing = [myTodoList objectAtIndex:0];
    NSLog(@"First thing I need to do: %@", firstThing);
    return YES;
}
```

output

```
2014-06-16 09:46:25.594 Test[821:303] First thing I need to do: Clean the house
```

Similar to how **NSString** literals are established by placing an **@** symbol followed by a pair of quotes, a hard-coded **NSArray** can be instantiate by proceeding the **@** symbol with a couple of square brackets: **@[object0, object1, …, lastObject]**.

NSArrays are immutable; once you've established an **NSArray** it can no longer be altered. You cannot remove, add, or re-assign the elements of an **NSArray**. However...

NSMutableArray

NSMutableArray is perfect for when you expect to alter an array after it's been instantiated. This array type supports all the features of **NSArray** while also being modifiable, adjustable and all around malleable. It's far more likely that an application supporting todo-list functionality would employ an **NSMutableArray** rather than an **NSArray**. Use **NSArray**s when you're certain that the contents will never need t change.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                           @"Feed the dog",
                                           @"Take out the trash",
                                           @"Fight crime"] mutableCopy];
    // Whoops, forgot one!
    [myMutableTodoList addObject:@"Solve world hunger"];
    // I cleaned the house last week, no need to do it again ;)
    [myMutableTodoList removeObjectAtIndex:0];
    NSLog(@"First thing I need to do: %@", [myMutableTodoList objectAtIndex:0]);
    return YES;
}
```

output

```
2014-06-16 10:02:13.677 Test[887:303] First thing I need to do: Feed the dog
```

After removing the item at index **0**, each remaining item is shifted down by 1. What was previously at index **1** is now at index **0**, index **2** is now at index **1**, etc. Therefore when we print the contents at index **0** we see that to **Feed the dog** is now our first task.

Sorting

Another useful feature belonging to mutable arrays is the ability to sort their contents. You may sort an array by several means. The first of the two methods you will learn is to employ an **NSSortDescriptor**. Let's sort the to-do list alphabetically:

AppDelegate.m

```
{
    NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                           @"Feed the dog",
                                           @"Take out the trash",
                                           @"Fight crime"] mutableCopy];
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:nil ascending:YES selector:@selector(localizedCaseInsensi
    [myMutableTodoList sortUsingDescriptors:@[sortDescriptor]];
    NSLog(@"Sorted Array Object 0: %@", myMutableTodoList[0]);
    NSLog(@"Sorted Array Object 1: %@", myMutableTodoList[1]);
    NSLog(@"Sorted Array Object 2: %@", myMutableTodoList[2]);
    NSLog(@"Sorted Array Object 3: %@", myMutableTodoList[3]);
    return YES;
}
```

output

```
2014-06-16 10:39:48.054 Test[978:303] Sorted Array Object 0: Clean the house
2014-06-16 10:39:48.059 Test[978:303] Sorted Array Object 1: Feed the dog
2014-06-16 10:39:48.059 Test[978:303] Sorted Array Object 2: Fight crime
2014-06-16 10:39:48.060 Test[978:303] Sorted Array Object 3: Take out the trash
```

As you can see in the output, **`Take out the trash`** is now properly located after **`Fight crime`**. Let's dig deeper into how **`NSSortDescriptor`** works. Here's the line that instantiated it:

AppDelegate.m

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:nil ascending:YES selector:@selector(localizedCaseInsensitive
```

Note that **`[[ObjectType alloc] init…]`** is a common pattern for creating new objects. It'll be discussed more in a later checkpoint.

According to Apple's Objective-C documentation:

> *"An instance of* **`NSSortDescriptor`** *describes a basis for ordering objects by specifying the property to use to compare the objects, the method to use to compare the properties, and whether the comparison should be ascending or descending."*

We set the initial **`key`** property to **`nil`**, therefore, no key specified. We want the comparison to operate in ascending order so we've assigned **`YES`** to **`ascending`**. Lastly, we want to supply a sorting method. **`[NSString -localizedCaseInsensitiveCompare:]`** returns exactly what we want, an **`NSComparisonResult`** between two **`NSStrings`** localized to the appropriate language.

Sorting Numbers

Strings aren't the only objects worth sorting; let's arrange some lucky lottery numbers in *descending* order:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *luckyLotto = [@[@(38), @(21), @(42), @(13), @(6), @(29), @(11)] mutableCopy];
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:nil ascending:NO];
    [luckyLotto sortUsingDescriptors:@[sortDescriptor]];
    NSLog(@"Lucky lotto number 1: %ld", [luckyLotto[0] longValue]);
    NSLog(@"Lucky lotto number 2: %ld", [luckyLotto[1] longValue]);
    NSLog(@"Lucky lotto number 3: %ld", [luckyLotto[2] longValue]);
    NSLog(@"Lucky lotto number 4: %ld", [luckyLotto[3] longValue]);
    NSLog(@"Lucky lotto number 5: %ld", [luckyLotto[4] longValue]);
    NSLog(@"Lucky lotto number 6: %ld", [luckyLotto[5] longValue]);
    NSLog(@"Lucky lotto number 7: %ld", [luckyLotto[6] longValue]);
    return YES;
}
```

```
2014-06-16 11:55:06.967 Test[1238:303] Lucky lotto number 1: 42
2014-06-16 11:55:06.979 Test[1238:303] Lucky lotto number 2: 38
2014-06-16 11:55:06.980 Test[1238:303] Lucky lotto number 3: 29
2014-06-16 11:55:06.981 Test[1238:303] Lucky lotto number 4: 21
2014-06-16 11:55:06.981 Test[1238:303] Lucky lotto number 5: 13
2014-06-16 11:55:06.982 Test[1238:303] Lucky lotto number 6: 11
2014-06-16 11:55:06.982 Test[1238:303] Lucky lotto number 7: 6
```

In addition to specifying a selector built in to the objects being sorted, we can also sort an array using code blocks.

Blocks

Blocks are special kinds of objects that represent methods. Instead of the object being a piece of data, it's an operation. You can pass blocks i as object parameters to other methods. Let's re-write our lottery example using a block and the **`NSMutableArray`** method, **`sortUsingComparator:`**:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *luckyLotto = [@[@(38), @(21), @(42), @(13), @(6), @(29), @(11)] mutableCopy];
    [luckyLotto sortUsingComparator:^NSComparisonResult(id obj1, id obj2) {
        NSNumber *numberA = (NSNumber *)obj1;
        NSNumber *numberB = (NSNumber *)obj2;

        int intValueA = [numberA intValue];
        int intValueB = [numberB intValue];

        if (intValueA > intValueB) {
            return NSOrderedAscending;
        } else if (intValueA < intValueB) {
            return NSOrderedDescending;
        }
        return NSOrderedSame;
    }];
    NSLog(@"Lucky lotto number 1: %ld", [luckyLotto[0] longValue]);
    NSLog(@"Lucky lotto number 2: %ld", [luckyLotto[1] longValue]);
    NSLog(@"Lucky lotto number 3: %ld", [luckyLotto[2] longValue]);
    NSLog(@"Lucky lotto number 4: %ld", [luckyLotto[3] longValue]);
    NSLog(@"Lucky lotto number 5: %ld", [luckyLotto[4] longValue]);
    NSLog(@"Lucky lotto number 6: %ld", [luckyLotto[5] longValue]);
    NSLog(@"Lucky lotto number 7: %ld", [luckyLotto[6] longValue]);
    return YES;
}
```

output

```
2014-06-16 12:00:53.143 Test[1278:303] Lucky lotto number 1: 42
2014-06-16 12:00:53.145 Test[1278:303] Lucky lotto number 2: 38
2014-06-16 12:00:53.146 Test[1278:303] Lucky lotto number 3: 29
2014-06-16 12:00:53.146 Test[1278:303] Lucky lotto number 4: 21
2014-06-16 12:00:53.147 Test[1278:303] Lucky lotto number 5: 13
2014-06-16 12:00:53.147 Test[1278:303] Lucky lotto number 6: 11
2014-06-16 12:00:53.147 Test[1278:303] Lucky lotto number 7: 6
```

> *Note that we converted the* **`NSNumber`** *object to an* **`int`** *before comparing. This is commonly called "unboxing".*
>
> *We can't compare two* **`NSNumber`** *objects using the* **`<`** *and* **`>`** *operators. The reason is the same reason (discussed in the Equality checkpoint) that we can't compare two* **`NSString`**s *using the* **`==`** *operator: these simple comparisons erroneously use the memory address, not the actual data stored in the object.*

returns an **NSComparisonResult** and passes in two objects: **id obj1** and **id obj2**. Blocks can be provided in-line as above. Xcode will auto-complete a stub block for you when you tab over to that parameter and press the return key.

```
int main(int argc, const char * argv[])
{
    NSMutableArray *luckyLotto = [@[@(38), @(21), @(42), @(13), @(6), @(29), @(11)] mutableCopy];
    
    return 0;
}
```

To inform the block's caller that **obj1** should come before **obj2**, the block must return **NSOrderedAscending**. If **obj1** comes *after* **obj2**, the block must return **NSOrderedDescending**. However, when the objects may occupy the same place in the list – in our case when two of them are identical numbers – then the block should return **NSOrderedSame**.

Here's another example of a comparator block used to order our tasks by the number of characters in each string:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                           @"Feed the dog",
                                           @"Take out the trash",
                                           @"Fight crime"] mutableCopy];
    [myMutableTodoList sortUsingComparator:^NSComparisonResult(id obj1, id obj2) {
        NSString *string1 = (NSString *)obj1;
        NSString *string2 = (NSString *)obj2;
        if (string1.length < string2.length) {
            return NSOrderedAscending;
        } else if (string2.length < string1.length) {
            return NSOrderedDescending;
        }
        return NSOrderedSame;
    }];
    NSLog(@"Sorted Array Object 0: %@", myMutableTodoList[0]);
    NSLog(@"Sorted Array Object 1: %@", myMutableTodoList[1]);
    NSLog(@"Sorted Array Object 2: %@", myMutableTodoList[2]);
    NSLog(@"Sorted Array Object 3: %@", myMutableTodoList[3]);
    return YES;
}
```

output

```
2014-06-16 14:13:34.206 Test[1577:303] Sorted Array Object 0: Fight crime
2014-06-16 14:13:34.208 Test[1577:303] Sorted Array Object 1: Feed the dog
2014-06-16 14:13:34.208 Test[1577:303] Sorted Array Object 2: Clean the house
2014-06-16 14:13:34.209 Test[1577:303] Sorted Array Object 3: Take out the trash
```

Iteration

The best part of having an ordered list of items is looping through them. You've learned how loops can help us reduce the size of our codebase by eliminating the need to copy and paste. Let's see how we can iterate over an array using a **for** loop:

AppDelegate.m

```
{
    NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                           @"Feed the dog",
                                           @"Take out the trash",
                                           @"Fight crime"] mutableCopy];
    for (NSInteger idx = 0; idx < myMutableTodoList.count; idx++) {
        NSLog(@"Task %ld: %@", idx, myMutableTodoList[idx]);
    }
    return YES;
}
```

output

```
2014-06-16 14:45:55.631 Test[1630:303] Task 0: Clean the house
2014-06-16 14:45:55.633 Test[1630:303] Task 1: Feed the dog
2014-06-16 14:45:55.634 Test[1630:303] Task 2: Take out the trash
2014-06-16 14:45:55.634 Test[1630:303] Task 3: Fight crime
```

That's easy enough but what if we don't care to have a counting variable like **idx**? **Fast enumeration** allows us to focus on just the objects inside of the array:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                           @"Feed the dog",
                                           @"Take out the trash",
                                           @"Fight crime"] mutableCopy];
    for (NSString* task in myMutableTodoList) {
        NSLog(@"Do this: %@", task);
    }
    return YES;
}
```

output

```
2014-06-16 14:48:22.650 Test[1655:303] Do this: Clean the house
2014-06-16 14:48:22.652 Test[1655:303] Do this: Feed the dog
2014-06-16 14:48:22.652 Test[1655:303] Do this: Take out the trash
2014-06-16 14:48:22.653 Test[1655:303] Do this: Fight crime
```

Fast enumeration pulls each element out of the array ahead of time so we don't have to. It hides the basic **for** loop logic and gives us a cleaner, simpler way to loop.

Let's say we didn't want people knowing about our secret identity, the one that fights crime at night. Using **[NSArray -enumerateObjectsUsingBlock:]** we can stop the loop prematurely. This method will loop through the **NSArray** and run the block of code we provide on each index until it reaches the end *or* we tell it to stop:

AppDelegate.m

```
{
    NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                           @"Feed the dog",
                                           @"Take out the trash",
                                           @"Fight crime"] mutableCopy];
    [myMutableTodoList enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
        NSString *task = (NSString *)obj;
        NSLog(@"Just doing a mild-mannered chore: %@", task);
        if (idx == 2) {
            // Uh-oh, I don't want anyone to know my secret identity!
            *stop = YES;
        }
    }];
    return YES;
}
```

output

```
2014-06-16 14:54:28.013 Test[1679:303] Just doing a mild-mannered chore: Clean the house
2014-06-16 14:54:28.023 Test[1679:303] Just doing a mild-mannered chore: Feed the dog
2014-06-16 14:54:28.024 Test[1679:303] Just doing a mild-mannered chore: Take out the trash
```

> **`BOOL *stop`** *points to a primitive* **`BOOL`** *which is read by the* **`NSArray`** *after each loop. We de-reference the pointer by placing an* **`*`** *in front of it. Once we've done so, we may treat it just like a regular* **`BOOL`**. *Read more about* ***pointers here***.

As you can see, only three tasks were printed. We stopped the enumeration once we reached index **2**, the second to last index. However, it is just as easy to stop looping in a **`for`** or **`while`** loop by employing a **`break`** statement. Here's the same enumeration in a basic **`for`** loop:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                           @"Feed the dog",
                                           @"Take out the trash",
                                           @"Fight crime"] mutableCopy];
    for (NSInteger idx = 0; idx < myMutableTodoList.count; idx++) {
        NSLog(@"Just doing a mild-mannered chore: %@", myMutableTodoList[idx]);
        if (idx == 2) {
            // Uh-oh, I don't want anyone to know I fight crime!
            break;
        }
    }
    return YES;
}
```

output

```
2014-06-16 16:44:29.719 Test[1847:303] Just doing a mild-mannered chore: Clean the house
2014-06-16 16:44:29.721 Test[1847:303] Just doing a mild-mannered chore: Feed the dog
2014-06-16 16:44:29.722 Test[1847:303] Just doing a mild-mannered chore: Take out the trash
```

Recap

Let's recap the three ways we can iterate over an **`NSArray`**'s contents.

Basic for loop

AppDelegate.m

```
    // code
}
```

Fast for loop

AppDelegate.m

```
for (NSNumber *number in someArrayOfNumbers) {
    // code
}
```

Enumeration with a block

AppDelegate.m

```
[someArray enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
    // code
}];
```

Method	Pros	Cons
Basic **for** loop	It's familiar and easy to use	You have to instantiate a counter variable
Fast **for** loop	The objects are recovered for you and it's easy to write the loop	No index variable
Enumeration with a block	Object and index created for you, easy to write	Ugly-looking

Filtering

Now that you can establish arrays and iterate through them, let's learn how to search for specific elements within them. Let's go back to our lottery numbers example. We've picked our winning numbers, yet Uncle Clark wants to have his say. He has a superstitious belief that choosing *numbers larger than 30* is a bad omen. To appease Uncle Clark we'll remove them:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *luckyLotto = [@[@(38), @(21), @(42), @(13), @(6), @(29), @(11)] mutableCopy];
    NSMutableArray *clarksNumbers = [[NSMutableArray alloc] init];
    for (NSNumber *number in luckyLotto) {
        if (number.longValue <= 30) {
            NSLog(@"Adding %ld to Clark's collection", number.longValue);
            [clarksNumbers addObject:number];
        }
    }
    return YES;
}
```

output

```
2014-06-17 09:12:03.623 Test[619:303] Adding 21 to Clark's collection
2014-06-17 09:12:03.626 Test[619:303] Adding 13 to Clark's collection
2014-06-17 09:12:03.626 Test[619:303] Adding 6 to Clark's collection
2014-06-17 09:12:03.626 Test[619:303] Adding 29 to Clark's collection
```

We took the looping/adding approach which is perfectly fine, but Objective-C provides a handy tool to shorten this process. Using **predicates** and the **`filterUsingPredicate:`** method we can arrive at the same conclusion *in style.*

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *luckyLotto = [@[@(38), @(21), @(42), @(13), @(6), @(29), @(11)] mutableCopy];
    NSPredicate *lessThan30Predicate = [NSPredicate predicateWithFormat:@"SELF <= 30"];
    [luckyLotto filterUsingPredicate:lessThan30Predicate];
    for (NSNumber *number in luckyLotto) {
        NSLog(@"Remaining number: %ld", number.longValue);
    }
    return YES;
}
```

output

```
2014-06-17 09:37:04.632 Test[723:303] Remaining number: 21
2014-06-17 09:37:04.634 Test[723:303] Remaining number: 13
2014-06-17 09:37:04.634 Test[723:303] Remaining number: 6
2014-06-17 09:37:04.635 Test[723:303] Remaining number: 29
2014-06-17 09:37:04.635 Test[723:303] Remaining number: 11
```

`NSPredicate` lets us specify conditions for which to filter a collection. In the example above, our **`NSPredicate`** is defined as such: **`NSPredicate *lessThan30Predicate = [NSPredicate predicateWithFormat:@"SELF <= 30"];`**. In this line, **`SELF`** refers to the object in the array. This predicate is a condition which is tested on each item in the array and if the condition passes, that item is included in the resulting filter.

`NSPredicate` allows other comparators for numbers as well: **`!=`**, **`>`**, **`<`**, **`>=`**, and **`==`**. However, **`NSPredicate`** also works with **`NSString`** objects. Let's filter our todo-list by tasks which include the word **`the`**:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                           @"Feed the dog",
                                           @"Take out the trash",
                                           @"Fight crime"] mutableCopy];
    NSPredicate *containsThe = [NSPredicate predicateWithFormat:@"SELF CONTAINS[c] 'the'"];
    [myMutableTodoList filterUsingPredicate:containsThe];
    for (NSString *task in myMutableTodoList) {
        NSLog(@"Remaining task: %@", task);
    }
    return YES;
}
```

output

```
2014-06-17 09:54:54.440 Test[805:303] Remaining task: Clean the house
2014-06-17 09:54:54.442 Test[805:303] Remaining task: Feed the dog
2014-06-17 09:54:54.442 Test[805:303] Remaining task: Take out the trash
```

> **`[c]`** *indicates a case-insensitive comparison.*

Our **`NSPredicate`** declaration, **`NSPredicate *containsThe = [NSPredicate predicateWithFormat:@"SELF CONTAINS[c] 'the'"];`** needs to be explained. There are a handful of keywords you may use for comparing **`NSStrings`** to others within predicates. Here's a handy table of them:

Keyword	Purpose	Example

CONTAINS	characters, even within a word	**like it", @"FaceBook"]** matches "Pit", "Sit" and "I don't like it"
MATCHES	Matches the characters *exactly*	**@"SELF MATCHES 'FaceBook'"** among **@[@"Pit", @"Sit", @"I don't like it", @"FaceBook"]** matches "FaceBook"
LIKE	Similar to MATCHES but allows wildcards, ***** and **?** where ***** matches 0 or more characters and **?** matches 1 character	**@"SELF LIKE '*o*'"** among **@[@"Pit", @"Sit", @"I don't like it", @"FaceBook"]** matches anything with an **o** in it, "I don't like it" and "FaceBook"
BEGINSWITH	The string must begin with the expression	**@"SELF BEGINSWITH[c] 'p'"** among **@[@"Pit", @"Sit", @"I don't like it", @"FaceBook"]** matches "Pit"
ENDSWITH	Similar to BEGINSWITH, the string must end with the expression	**@"SELF ENDSWITH 'it'"** among **@[@"Pit", @"Sit", @"I don't like it", @"FaceBook"]** matches "Pit", "Sit" and "I don't like it"

Array To String

We know you're tired, this is the longest checkpoint in the history of checkpoints. This is the last portion, we promise. For convenience, **NSArray** allows us to convert the values in our array to one large string for easy printing; check it out:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                           @"Feed the dog",
                                           @"Take out the trash",
                                           @"Fight crime"] mutableCopy];
    NSString *allMyTasks = [myMutableTodoList componentsJoinedByString:@", "];
    NSLog(@"What do I have to do today? %@", allMyTasks);
    return YES;
}
```

output

```
2014-06-17 10:46:51.877 Test[1047:303] What do I have to do today? Clean the house, Feed the dog, Take out the trash, Fight crime
```

As you can see, **allMyTasks** is a string made by taking each element in the array and placing them one after the other separated only by the string literal we suppled, **@", "** a comma and a space. We can put anything we want there, it doesn't have to make sense:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                           @"Feed the dog",
                                           @"Take out the trash",
                                           @"Fight crime"] mutableCopy];
    NSString *allMyTasks = [myMutableTodoList componentsJoinedByString:@", do I have to? "];
    NSLog(@"What do I have to do today? %@", allMyTasks);
    return YES;
}
```

output

```
2014-06-17 10:52:18.303 Test[1095:303] What do I have to do today? Clean the house, do I have to? Feed the dog, do I have to? Take out
```

Here's an example using our lottery numbers:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *luckyLotto = [@[@(38), @(21), @(42), @(13), @(6), @(29), @(11)] mutableCopy];
    NSString *lottoNumbers = [luckyLotto componentsJoinedByString:@", "];
    NSLog(@"I'd like to play these numbers, please: %@.", lottoNumbers);
    return YES;
}
```

output

```
2014-06-17 10:54:41.103 Test[1113:303] I'd like to play these numbers, please: 38, 21, 42, 13, 6, 29, 11.
```

String To Array

It's also possible to go from character-separated **NSString**s like those created earlier to an **NSArray** of strings. Let's convert our tasks from a string to an array:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSString *allTasks = @"Clean the house, Feed the dog, Take out the trash, Fight crime";
    NSArray *myTaskArray = [allTasks componentsSeparatedByString:@", "];
    [myTaskArray enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
        NSLog(@"Task #%ld: %@", idx + 1, obj);
    }];
    return YES;
}
```

output

```
2014-06-17 11:03:50.568 Test[1230:303] Task #1: Clean the house
2014-06-17 11:03:50.570 Test[1230:303] Task #2: Feed the dog
2014-06-17 11:03:50.571 Test[1230:303] Task #3: Take out the trash
2014-06-17 11:03:50.571 Test[1230:303] Task #4: Fight crime
```

Unfortunately, the **NSString** class is only capable of creating an **NSArray** of other **NSStrings**, it cannot convert a string of numbers, such as **@"853,422,62,12,7884,223"** into an array of **NSNumber** objects. The resulting **NSArray** will look like this, **@[@"853", @"422", @"62", @"12", @"7884", @"223"]**.

Open the BlocExercises project in Xcode. Open the folder that corresponds to this checkpoint and implement solutions to make the tests pass. Exercise descriptions are found the the header (`.h`) file, while you should implement your solutions in the implementation (`.m`) file. The test file will also give you some clues for implementing a proper solution, so be sure to read the tests as well.

Run the tests that correspond with the implementation file. Press ⌘**5** to view the **Test Navigator** and click the "play" button to the right of the corresponding test group. You can also run individual tests by clicking the "play" button corresponding to a specific test.

Commit and push your changes to GitHub:

Terminal

```
$ git status #=> you should see the exercise file you just updated
$ git add . #=> stage the changes
$ git commit -m "Completed arrays"
$ git push origin master #=> send your changes to your remote GitHub repo
```

When you've completed the corresponding exercises, submit this assignment with the relevant repo and commit links.

assignment completed

COURSES

Full Stack Web Development
Frontend Web Development
UX Design
Android Development
iOS Development

ABOUT BLOC

Our Team | Jobs
Bloc Veterans Program
Employer Sponsored
FAQ
Blog
Engineering Blog
Refer-a-Friend
Privacy Policy
Terms of Service

MADE BY BLOC

Tech Talks & Resources
Programming Bootcamp Comparison
Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
Swiftris: Build Your First iOS Game with Swift
Webflow Tutorial: Design Responsive Sites with Webflow
Ruby Warrior
Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io
Considering enrolling? (404) 480-2562

COPYRIGHT © 2015 BLOC

Dictionaries

> *"Always do your best. What you plant now, you will harvest later."*
>
> *- Og Mandino*

Much like arrays, dictionaries are collections of data. However, unlike arrays their elements are not arranged linearly. If an array can be thought of as a list of your favorite movies, a dictionary is something which can store that list under the title, "FaveMovies." It can also hold your favorite TV shows, books, poems, authors and actors in similar lists:

Favorite-Movie-Array

```
[Thing, It, Nightmare On Elm Street, Sharknado]
```

Favorite-Stuff-Dictionary

```
{
    "FaveMovies" : [Thing, It, Nightmare On Elm Street, Sharknado],
    "FaveMovie" : "Monty Python And The Holy Grail",
    "Number of Movies Watched" : 896,
    "FaveBooks" : [The Count Of Monte Cristo, The Giver, Ender's Game],
    "FaveBook" : "The Catcher In The Rye",
    "Number of Books Read" : 724,
    "FaveBeliefs" : {
                        "Santa" : YES,
                        "Dragons" : YES,
                        "Flying Spaghetti Mons    YES,
                        "Sasquatch" : NO
                       }
}
```

Dictionaries use a key-value mapping dynamic. The **key** is a unique object which provides an identifier for a piece of data. Each key is mapped directly to a single **value**–a single piece of data. Let's break down the example above by its keys and values:

Key	Value Type
FaveMovies	**NSArray** of **NSString**s
FaveMovie	**NSString**
Number of Movies Watched	**NSNumber**
FaveBooks	**NSArray** of **NSString**s
FaveBook	**NSString**
Number of Books Read	**NSNumber**

As you can see, a dictionary can hold a wide variety of elements, including other dictionaries. A dictionary may have any type of object as its *keys* or *values.*

> *The dictionary above adheres to the **JSON protocol** where each key must be a string and each piece of data either a boolean, number, string, array or another dictionary. You will learn more about JSON when we start building iOS apps.*

NSDictionary

You've certainly presumed by now that the Objective-C class which represents a dictionary is **`NSDictionary`**. Here's how you may instantiate a dictionary which matches the one detailed above:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSArray *faveMovies = @[@"Thing", @"It", @"Nightmare On Elm Street", @"Sharknado"];
    NSString *faveMovie = @"Monty Python And The Holy Grail";
    NSNumber *numbMoviesWatched = @(896);
    NSArray *faveBooks = @[@"The Count Of Monte Cristo", @"The Giver", @"Ender's Game"];
    NSString *faveBook = @"The Catcher In The Rye";
    NSNumber *numbBooksRead = @(724);
    NSDictionary *faveBeliefs = @{@"Santa" : @YES,
                                  @"Dragons" : @YES,
                                  @"Flying Spaghetti Monster" : @YES,
                                  @"Sasquatch" : @NO};

    NSDictionary *myFavoriteThings = @{@"FaveMovies" : faveMovies,
                                       @"FaveMovie" : faveMovie,
                                       @"Number of Movies Watched" : numbMoviesWatched,
                                       @"FaveBooks" : faveBooks,
                                       @"FaveBook" : faveBook,
                                       @"Number of Books Read" : numbBooksRead,
                                       @"FaveBeliefs" : faveBeliefs};
    return YES;
}
```

We agree, that looks a bit messy. However, once it's created it's done. We can iterate over the keys in our dictionary much like we would with the objects in an array:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // DICTIONARY SETUP OMITTED FOR BREVITY
    [myFavoriteThings enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL *stop) {
        NSLog(@"In our dictionary, the key \"%@\" corresponds with %@", key, obj);
    }];
    return YES;
}
```

output

```
    "The Count Of Monte Cristo",
    "The Giver",
    "Ender's Game"
)
2014-06-17 14:59:38.145 Test[2733:303] In our dictionary, the key "FaveBook" corresponds with The Catcher In The Rye
2014-06-17 14:59:38.147 Test[2733:303] In our dictionary, the key "Number of Movies Watched" corresponds with 896
2014-06-17 14:59:38.148 Test[2733:303] In our dictionary, the key "FaveMovies" corresponds with (
    Thing,
    It,
    "Nightmare On Elm Street",
    Sharknado
)
2014-06-17 14:59:38.145 Test[2733:303] In our dictionary, the key "FaveMovie" corresponds with Monty Python And The Holy Grail
2014-06-17 14:59:38.148 Test[2733:303] In our dictionary, the key "Number of Books Read" corresponds with 724
2014-06-17 14:59:38.149 Test[2733:303] In our dictionary, the key "FaveBeliefs" corresponds with {
    Dragons = 1;
    "Flying Spaghetti Monster" = 1;
    Santa = 1;
    Sasquatch = 0;
}
```

Unlike arrays, dictionaries are **unordered**. Therefore, the order of enumeration is unpredictable and should not be relied upon. For example, the first key-value added to a dictionary may not necessarily be the first encountered during the enumeration.

You may use the **`enumerateKeysAndObjectsUsingBlock:`** exactly as you use the **`enumerateObjectsUsingBlock:`** method of **`NSArray`**. Following the pattern we've seen several times before, **`NSDictionary`** is immutable. If we'd like to alter our dictionary after instantiating it, we need to use **`NSMutableDictionary`**.

NSMutableDictionary

Creating a mutable dictionary is similar to the process we have seen before. Create an immutable version and send it the **`mutableCopy`** method call to create a mutable version.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // DICTIONARY SETUP OMITTED FOR BREVITY
    NSDictionary *myFavorites = @{@"FaveMovies" : faveMovies,
                                  @"FaveMovies" : faveMovie,
                                  @"Number of Movies Watched" : numbMoviesWatched,
                                  @"FaveBooks" : faveBooks,
                                  @"FaveBook" : faveBook,
                                  @"Number of Books Read" : numbBooksRead,
                                  @"FaveBeliefs" : faveBeliefs};
    NSMutableDictionary *myChangingFavorites = [myFavorites mutableCopy];
    [myChangingFavorites setObject:@(6000) forKey:@"Number of Books Read"];
    NSLog(@"I've read %ld books", [myChangingFavorites[@"Number of Books Read"] longValue]);
    return YES;
}
```

output

```
2014-06-17 15:20:55.445 Test[2819:303] I've read 6000 books
```

You may also remove objects entirely using the **`removeObjectForKey:`** or **`removeObjectForKeys`** method:

AppDelegate.m

```
{
    // DICTIONARY SETUP OMITTED FOR BREVITY
    NSMutableDictionary *myChangingFavorites = [@{@"FaveMovies" : faveMovies,
                                                  @"FaveMovie" : faveMovie,
                                                  @"Number of Movies Watched" : numbMoviesWatched,
                                                  @"FaveBooks" : faveBooks,
                                                  @"FaveBook" : faveBook,
                                                  @"Number of Books Read" : numbBooksRead,
                                                  @"FaveBeliefs" : faveBeliefs} mutableCopy];
    [myChangingFavorites removeObjectsForKeys:@[@"FaveMovies", @"FaveMovie", @"Number of Movies Watched"]];
    [myChangingFavorites enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL *stop) {
        NSLog(@"Remaining key %@", key);
    }];
    return YES;
}
```

output

```
2014-06-17 15:28:58.604 Test[2863:303] Remaining key FaveBooks
2014-06-17 15:28:58.606 Test[2863:303] Remaining key FaveBeliefs
2014-06-17 15:28:58.606 Test[2863:303] Remaining key FaveBook
2014-06-17 15:28:58.606 Test[2863:303] Remaining key Number of Books Read
```

Dictionaries From The Web

In later checkpoints you'll be downloading dictionary objects from the Internet. However, there is a caveat to those dictionaries: *you don't kno what's in them.* Therefore, before attempting to manipulate the data it's best to check what kind of data it is. Let's try to recover the "Number of Movies Watched" from our previous example dictionary *as if* we had downloaded it from the Internet:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // DOWNLOADED THE downloadedFavorites DICTIONARY
    id numberOfBooksRead = downloadedFavorites[@"Number of Books Read"];
    if (numberOfBooksRead != nil && [numberOfBooksRead isKindOfClass:[NSNumber class]]) {
        NSLog(@"The Number of Books Read: %ld", [numberOfBooksRead longValue]);
    }
    return YES;
}
```

output

```
2014-06-17 15:39:53.448 Test[2915:303] The Number of Books Read: 724
```

We performed a couple of important checks above with our **`if`** condition. The first, **`numberOfBooksRead != nil`**, should *always* be present when recovering objects from a downloaded dictionary. As a mobile developer you cannot update your application as quickly as the web service which supports it. Therefore, you must be prepared for unexpected changes. If you're not prepared, unexpected changes may harm the user experience or crash the application.

For example, if someone changed the key sent down from "Number of Books Read" to "number-books-read", your application wouldn't be able to update itself to reflect the change. To keep your application from crashing we check if the object we expected to be there *is actually there.*

Second, we checked for the *right type.* Another change that can occur on the server is a change of data type. Previously, the type may have been an **`NSNumber`** but now it can be an **`NSString`** representing the number written out literally. An incorrect type could crash our application. To check for correct type, use the **`isKindOfClass:`** method. Call it on the object type you need to check and provide the class to compare it to Here's an example of string type verification:

AppDelegate.m

```
{
    // DOWNLOADED THE downloadedFavorites DICTIONARY
    id favoriteMovie = downloadedFavorites[@"FaveMovie"];
    if (favoriteMovie != nil && [favoriteMovie isKindOfClass:[NSString class]]) {
        NSLog(@"Favorite Movie: %@", favoriteMovie);
    }
    return YES;
}
```

output

```
2014-06-17 15:53:34.298 Test[2942:303] Favorite Movie: Monty Python And The Holy Grail
```


Open the BlocExercises project in Xcode. Open the folder that corresponds to this checkpoint and implement solutions to make the tests pass. Exercise descriptions are found the the header (`.h`) file, while you should implement your solutions in the implementation (`.m`) file. The test file will also give you some clues for implementing a proper solution, so be sure to read the tests as well.

Run the tests that correspond with the implementation file. Press ⌘**5** to view the **Test Navigator** and click the "play" button to the right of the corresponding test group. You can also run individual tests by clicking the "play" button corresponding to a specific test.

Commit and push your changes to GitHub:

Terminal

```
$ pwd #=> you should be in your forked ios-exercises directory. If not, cd into it.
$ git status #=> you should see the exercise file you just updated
$ git add . #=> stage the changes
$ git commit -m "Completed dictionaries"
$ git push origin master #=> send your changes to your remote GitHub repo
```

When you've completed the corresponding exercises, submit this assignment with the relevant repo and commit links.

assignment completed

COPYRIGHT © 2015 BLOC

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Simple Data Types and Objects

Simple Data Types and Objects

Objective-C is called an "object-oriented language". What does this mean? Steve Jobs once gave an excellent explanation of what it means in this **Rolling Stones interview**.

> ***Steve Jobs:*** *Objects are like people. They're living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we're doing right here.*
>
> *Here's an example: If I'm your laundry object, you can give me your dirty clothes and send me a message that says, "Can you get my clothes laundered, please." I happen to know where the best laundry place in San Francisco is. And I speak English, and I have dollars in my pockets. So I go out and hail a taxicab and tell the driver to take me to this place in San Francisco. I go get your clothes laundered, I jump back in the cab, I get back here. I give you your clean clothes and say, "Here are your clean clothes."*
>
> *You have no idea how I did that. You have no knowledge of the laundry place. Maybe you speak French, and you can't even hail a taxi. You can't pay for one, you don't have dollars in your pocket. Yet I knew how to do all of that. And you didn't have to know any of it. All that complexity was hidden inside of me, and we were able to interact at a very high level of abstraction. That's what objects are. They encapsulate complexity, and the interfaces to that complexity are high level.*

To put it another way: **an object is something that hol ? a and has behavior.**

You've already encountered objects before in the checkpoint on **Arrays**.

AppDelegate.m

```
NSArray *myTodoList = @[@"Clean the house",
                        @"Feed the dog",
                        @"Take out the trash",
                        @"Fight crime"];
```

Arrays are objects. They hold data. Arrays also have behavior.

We can ask the array to tell us information about itself such as, how many items are in it?

AppDelegate.m

```
[myTodoList count]; // returns 4
```

Or we can ask what its last item is.

AppDelegate.m

```
[myTodoList lastObject]; // returns @"Fight Crime"
```

AppDelegate.m

```
NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                       @"Feed the dog",
                                       @"Take out the trash",
                                       @"Fight crime"] mutableCopy];
// Whoops, forgot one!
[myMutableTodoList addObject:@"Solve world hunger"];
```

Object vs Class

When learning about objects you may encounter the term **Class** a lot. It is often used interchangebly with **Object**. What's the difference?

Classes are like blueprints from which objects can be created. For example:

`NSArray` is a class. An array created from a class, **`myTodoList`**, is an object. In object oriented programming we often say that **`myTodoList`** is ar **instance** of the **`NSArray`** class.

Different instances of a class will have the same *behavior* but they could all have different *data*.

To use an analogy:

The **2004 Volkswagen Beetle** is a **class** of car that Volkswagen once made. They made a lot of these cars. The particular car that they sold to me, and that I drive around is an instance of the **2004 Volkswagen Beetle** class.

All other **2004 Volkswagen Beetles** might drive similar to my **2004 Volkswagen Beetle** but my car (instance) is green and your car (instance) is red.

Class vs. Instance Methods

Classes have *class methods*, which are typically (but not always) used for creating a new instance of that class. When describing them, they're written with a plus (**+**). Here are some examples you've seen:

- **`[NSNumber +numberWithInt:]`**
- **`[NSMutableString +stringWithFormat:]`**
- **`[NSSortDescriptor +alloc]`**

Once an instance of a class is created, you can call *instance methods* on these objects, which are written with a minus (**-**). Here are some examples you've seen:

- **`[NSString -length]`**
- **`[NSArray -count]`**
- **`[NSArray -enumerateObjectsWithOptions:usingBlock:]`**

The most common *class method* you call is **`alloc`**. It creates a new instance of that class. It is almost always followed by an init method, like so

```
SomeClass *myInstance = [[SomeClass alloc] init];
```

Many classes have shorthand constructors which call this code for you. For example, these two lines are equivalent:

```
NSString *myString = [NSString stringWithFormat:@"I have %lu pets", pets.count];
NSString *myString = [[NSString alloc] initWithFormat:@"I have %lu pets", pets.count];
```

You can optionally use the class method **`new`** instead of calling **`alloc-init`**. For example, these three lines of code are equivalent:

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

If you'd like to learn more, we encourage you to read **this Stack Overflow question** for more examples, sample code, and analogies.

NSNumber: the number class

In the checkpoint **Numbers and Variables** we discussed different types of number values used in Objective-C. We discussed the C primitives **`int`** and **`long`** as well as the more advanced Objective-C primitives **`NSInteger`** and **`CGFloat`**.

Objective-C also defines a *class* for storing numbers called **`NSNumber`**.

NSNumber is the catch-all class for storing numeric values. When we were dealing with **`int`** and **`double`** variables we had to be careful to use the correct type. NSNumber can hold both of these.

AppDelegate.m

```
NSNumber *myInt = [NSNumber numberWithInt:42];
NSNumber *myDouble = [NSNumber numberWithDouble:42.1223432522423423];
```

This process of converting a primitive value into an object is often called **wrapping** or **boxing**.

Since objects have behaviors, one benefit of wrapping the values in objects is that they gain extra abilities. Much like Tony Stark gets extra powers when he puts on his Iron Man suit:

One ability that **`NSNumber`** has is the ability to convert between different number types.

AppDelegate.m

```
NSNumber *myDouble = [NSNumber numberWithDouble:42.1223432522423423];
```

Here we used the **`intValue`** to get the integer value back out of the object. This is referred to as **unboxing**.

`NSNumber` also has the ability to compare two numbers to determine if they are equal.

AppDelegate.m

```
NSNumber *intOne = [NSNumber numberWithInt:42];
NSNumber *intTwo = [NSNumber numberWithInt:36];

[intOne isEqualToNumber:intTwo]; //NO
```

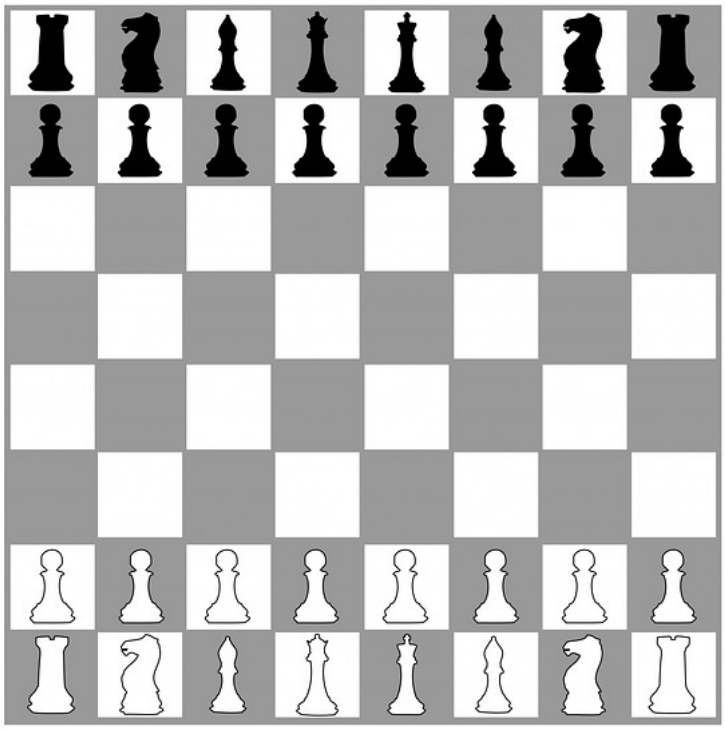
`NSNumber` is a basic class, so it doesn't have many abilities beyond this.

Using **`NSValue`** to store primitives

One useful aspect of wrapping **`NSNumber`** is that it allows us to store number values in collections such as an **`NSArray`** of an **`NSDictionary`**.

This is great for numeric primitives, but what about other types? For instance, in Objective-C we often use a type called **`CGRect`** for representing rectangles.

If we wanted to represent a chess board:



... we could use an array of CGRect values.

AppDelegate.m

```
NSArray *chessBoard = @[CGRectMake(0, 0, 100, 100),
                        CGRectMake(100, 0, 100, 100),
                        CGRectMake(200, 0, 100, 100),
                        CGRectMake(300, 0, 100, 100),
                        CGRectMake(400, 0, 100, 100),
                        CGRectMake(500, 0, 100, 100),
                        CGRectMake(600, 0, 100, 100),
                        CGRectMake(700, 0, 100, 100),
                        CGRectMake(800, 0, 100, 100),
                        CGRectMake(0, 100, 100, 100),
                        CGRectMake(0, 200, 100, 100)
                        //...
                       ];
```

Unfortunately this will give us a nasty warning:

```
    // insert code here...
    NSArray *chessBoard = @[CGRectMake(0, 0, 100, 100),     Collection element of type 'CGRect' (aka 'struct CGRect') is not an Objective-C objec
                            CGRectMake(100, 0, 100, 100),
                            CGRectMake(200, 0, 100, 100),
                            CGRectMake(300, 0, 100, 100),
                            CGRectMake(400, 0, 100, 100),
                            CGRectMake(500, 0, 100, 100),
                            CGRectMake(600, 0, 100, 100),
```

It seems that we are only allowed to store Objective-C objects inside of an **`NSArray`**.

This is where **`NSValue`** comes in handy. In order to create our array we must wrap our **`CGRect`** values in **`NSValue`** objects.

AppDelegate.m

```
//One CGRect for each space on the board
-   NSArray *chessBoard = @[CGRectMake(0, 0, 100, 100),
-                           CGRectMake(100, 0, 100, 100),
-                           CGRectMake(200, 0, 100, 100),
-                           CGRectMake(300, 0, 100, 100),
-                           CGRectMake(400, 0, 100, 100),
-                           CGRectMake(500, 0, 100, 100),
-                           CGRectMake(600, 0, 100, 100),
-                           CGRectMake(700, 0, 100, 100),
-                           CGRectMake(800, 0, 100, 100),
-                           CGRectMake(0, 100, 100, 100),
-                           CGRectMake(0, 200, 100, 100)
+   NSArray *chessBoard = @[[NSValue valueWithCGRect:CGRectMake(0, 0, 100, 100)],
+                           [NSValue valueWithCGRect:CGRectMake(100, 0, 100, 100)],
+                           [NSValue valueWithCGRect:CGRectMake(200, 0, 100, 100)],
+                           [NSValue valueWithCGRect:CGRectMake(300, 0, 100, 100)],
+                           [NSValue valueWithCGRect:CGRectMake(400, 0, 100, 100)],
+                           [NSValue valueWithCGRect:CGRectMake(500, 0, 100, 100)],
+                           [NSValue valueWithCGRect:CGRectMake(600, 0, 100, 100)],
+                           [NSValue valueWithCGRect:CGRectMake(700, 0, 100, 100)],
+                           [NSValue valueWithCGRect:CGRectMake(800, 0, 100, 100)],
+                           [NSValue valueWithCGRect:CGRectMake(0, 100, 100, 100)],
+                           [NSValue valueWithCGRect:CGRectMake(0, 200, 100, 100)]
                           //...
                          ];
```

When we want to use one of the rectangles again, we must unbox it.

AppDelegate.m

```
CGRect firstSpace =  [[chessBoard objectAtIndex:0] CGRectValue];
```

typedef NS_ENUM

`typedef NS_ENUM` is used to give human-readable names to numbers. For example, here's a list of different animation transitions.

```
    UIViewAnimationTransitionNone,
    UIViewAnimationTransitionFlipFromLeft,
    UIViewAnimationTransitionFlipFromRight,
    UIViewAnimationTransitionCurlUp,
    UIViewAnimationTransitionCurlDown,
};
```

This allows you to type (and allows Xcode to autocomplete) **`UIViewAnimationTransitionFlipFromRight`** instead of having to remember the number **2**.

For example, in a web browser app, we could declare button types in the header file.

AppDelegate.m

```
typedef NS_ENUM(NSInteger, BLCBrowserButton) {
    BLCBrowserButtonBack,
    BLCBrowserButtonForward,
    BLCBrowserButtonStop,
    BLCBrowserButtonRefresh
};
```

This would allow us to add a property.

AppDelegate.m

```
@property (nonatomic, assign) BLCBrowserButton buttonType;
```

Once we set this property, we can use it to run code for certain buttons.

AppDelegate.m

```
if (self.buttonType == BLCBrowserButtonRefresh) {
    NSLog(@"This is the refresh button.");
}
```


Open the BlocExercises project in Xcode. Open the folder that corresponds to this checkpoint and implement solutions to make the tests pass. Exercise descriptions are found in the header (`.h`) file, while you should implement your solutions in the implementation (`.m`) file. The test file will also give you some clues for implementing a proper solution, so be sure to read the tests as well.

Run the tests that correspond with the implementation file. Press ⌘**5** to view the **Test Navigator** and click the "play" button to the right of the corresponding test group. You can also run individual tests by clicking the "play" button corresponding to a specific test.

Commit and push your changes to GitHub:

Terminal

```
$ pwd #=> you should be in your forked ios-exercises directory. If not, cd into it.
$ git status #=> you should see the exercise file you just updated
$ git add . #=> stage the changes
$ git commit -m "Completed simple data types and objects"
$ git push origin master #=> send your changes to your remote GitHub repo
```

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

- Our Team | Jobs
- Bloc Veterans Program
- Employer Sponsored
- FAQ
- Blog
- Engineering Blog
- Refer-a-Friend
- Privacy Policy
- Terms of Service

MADE BY BLOC

- Tech Talks & Resources
- Programming Bootcamp Comparison
- Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
- Swiftris: Build Your First iOS Game with Swift
- Webflow Tutorial: Design Responsive Sites with Webflow
- Ruby Warrior
- Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

- hello@bloc.io
- Considering enrolling? (404) 480-2562
- Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Properties and Scope

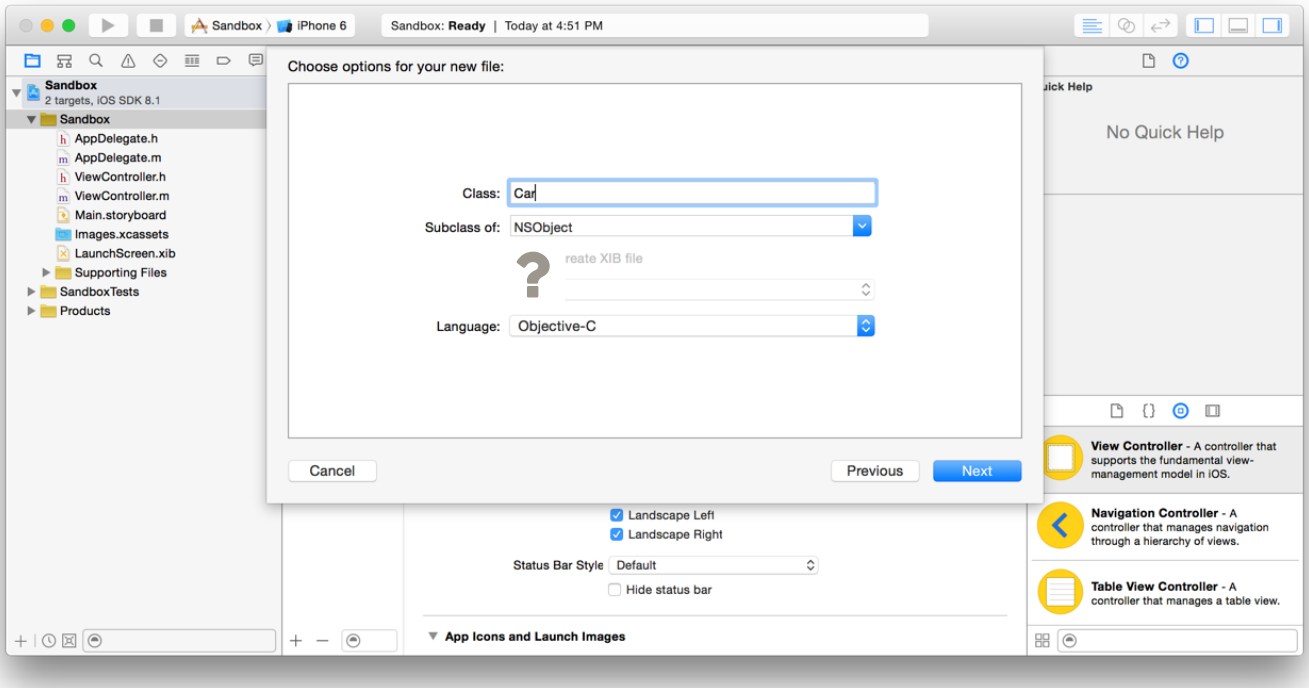
> *"Who'd have thought a nuclear reactor would be so complicated!?"*
>
> *- Homer Simpson*

The time has come to write our own class. We've learned about classes and objects and discussed how they are similar to a model of a car. In this checkpoint we'll expand on that analogy and create a class called **`Car`**.

Create a single view iOS project. Click the **File** -> **New** -> **File...**, or press ⌘**N**.

Select **Cocoa Touch Class** and click **Next**.

For the class name enter the word **Car**, make sure it's an **`NSObject`** subclass, and click **Next** again.



Xcode will ask us where to save the file. Select the same folder as the **`AppDelegate.m`** and **`AppDelegate.h`** files (this is the default) and click **Create**. This will create two new files called **`Car.h`** and **`Car.m`**.

Interface vs Implementation

Let's first look at the **`Car.h`** file. This file is called a *Header* file. In it we define the **Interface** for our class.

An interface works similar to a building's directory. A building's directory lists all the areas and functions of a building.

A class interface lists all the abilities of the class and tells other classes how to use it.

Now open the other file **`Car.m`**

In this file we define the **implementation** for our class. If the interface is where we list what our app can do, the implementation is where we write the code that does those things.

Our car currently can't do anything, so let's give it the ability to honk.

Implementing the Honk method

When Honk is called it should print out the word "Beep" to the application log.

Open **`Car.h`** and add the following code inside the **`@interface`**

Car.h

```
@interface Car : NSObject
+ -(void)honk;
@end
```

This may look similar to how we've defined methods before. The only difference is that the method has no body. We call this part of the method that we've written the **Method Signature**.

In the implementation file:

Car.m

```
@implementation Car
+ -(void)honk
+ {
+     NSLog(@"Beep! Beep!");
+ }
@end
```

We use the same method signature as in the header file, but this time we write the body as well.

We've now implemented a basic car class. Next we'll use it from the **`AppDelegate.m`** file.

Using the Car Class

Before we can use our new class, we have to import it into the **`AppDelegate.m`** file.

Add the following code:

AppDelegate.m

```
#import "AppDelegate.h"
+ #import "Car.h"

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
    return YES;
}
```

This line will give the AppDelegate access to the **`Car`** class. Use it to make a new car instance.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
+     Car *myCar = [Car new];
    return YES;
}
```

We just made a new instance of the car class. This process is referred to as **instantiation**.

Now that we have a car we can make it honk:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
    Car *myCar = [Car new];
+     [myCar honk];
    return YES;
}
```

When we run the application it should print **`"Beep! Beep!"`**

Adding a property

When you buy a car you often have customization options. You can pick the color, the interior, and add upgrades. Let's say that we want to be

A property is **a variable that is attached to a class**. When a class defines a property it allows other code to change attributes about that class. Let's add a property to **`Car`** to store our horn sound:

Car.h

```
@interface Car : NSObject
 + @property NSString *hornSound;
-(void)honk;
@end
```

Like methods, properties are declared in a class's interface file. Unlike methods, there is no implementation necessary. We write a property similar to how we write a variable. We have to declare the type for the property-in this case **`NSString`**.

The only difference here is that we use the **`@property`** keyword.

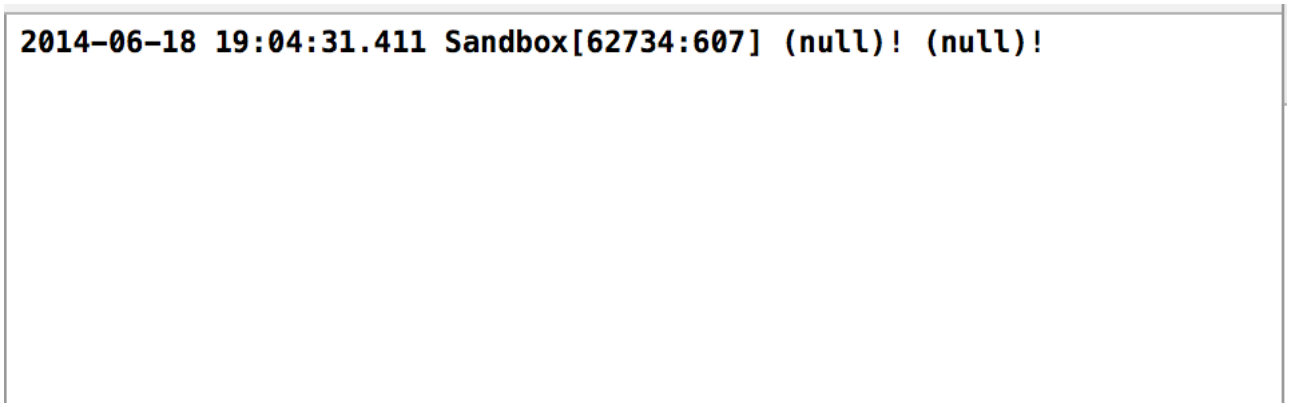
Now let's update the implementation of the **`honk`** method to use the new property.

Car.m

```
@implementation Car
-(void)honk{
 -     NSLog(@"Beep! Beep!");
 +     NSLog(@"%@! %@!", self.hornSound, self.hornSound);
}
@end
```

Here we use string formatting to print our horn sound twice. To access a property from inside a class we use **`self.`** followed by the property name. (In this case we use **`self.hornSound`**.)

Now see what happens when we run the app:



Uh Oh! It's printing **`(null)! (null)!`**. That's because we haven't set **`hornSound`** yet.

In the AppDelegate set the horn sound on the **`myCar`** object.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
    Car *myCar = [Car new];
 +     myCar.hornSound = @"Awoooga";
    [myCar honk];
    return YES;
}
```

Now when we run the app it should print our custom horn sound, **`Awoooga! Awoooga!`**.

Scope

Scope refers to different levels of an application. When code exists inside the body of a method it is said to be inside that method's **scope**. Th code has access to anything that is defined inside that scope.

For example, inside the **`application:didFinishLaunchingWithOptions`** method we define the variable **`myCar`**. Since this variable was define inside the scope of this method it can only be accessed inside this method. We often refer to this type of variable as being **local to the scope of this method** or as a **local variable**.

As an experiment, let's see what happens when we try to access this variable from outside the method.

In the second method of the **`AppDelegate.m`** file add the following code:

AppDelegate.m

```
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
    Car *myCar = [Car new];
    myCar.hornSound = @"Beep";
    [myCar honk];
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
    // Sent when the application is about to move from active to inactive state. This can occur for certain types of temporary interru
    // For example, use this method to pause ongoing tasks or disable timers. Games should use this method to pause the game.

+     [myCar honk];
}
```

This causes the following error:

```
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
    Car *myCar = [Car new];
    myCar.hornSound = @"Beep";
    [myCar honk];
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
    // Sent when the application is about to move from active to inactive state. This can occur for certain
        types of temporary interruptions (such as an incoming phone call or SMS message) or when the user
        quits the application and it begins the transition to the background state.
    // Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame rates.
        Games should use this method to pause the game.

    [myCar honk];                                                    Use of undeclared identifier 'myCa
}
```

The reason is that we tried to access a variable that is **out of scope** for the second method.

If we want to have something accessible from any method in a class we have to define it as a **`@property`**.

Properties are defined on the *class scope*. All instance methods of a class have access to the class scope.

Memory Management

Automatic Reference Counting

Automatic Reference Counting will work well for you, provided you understand how it works. Here are the basics:

- All objects will stay around as long as they have one **`strong`** reference to them. If you care about an object's continued existence, maintain at least one **`strong`** reference to it.
- If you *don't* care about an object's continued existence, maintain a **`weak`** reference.

> ***Note:*** *It takes people a while to grasp the concepts between the different properties. Don't be afraid to move on, and refer back to this checkpoint as you use properties in your apps.*

Consider these three examples:

A **`strong`** property

SomeClass.h

```
@interface SomeClass
@property (nonatomic, strong) NSString *someString;
@end
```

SomeClass.m

```
- (void) doSomething {
    self.someString = @"Hello";
    NSLog(@"%@", self.someString);  // Prints Hello
}
```

The sample above behaves as we expect. We assigned **`@"Hello"`** to an **`NSString`** property and when we print it out, it works. However...

A **`weak`** property

SomeClass.h

```
@interface SomeClass
@property (nonatomic, weak) NSString *someString;
@end
```

SomeClass.m

```
- (void) doSomething {
    self.someString = @"Hello";
    NSLog(@"%@", self.someString);  // Prints (null)
}
```

In this case, **`self.someString`** is immediately deallocated after assignment because there are no **`strong`** pointers to keep the object around.

Combining a **`strong`** reference with a **`weak`** property

Local variables are **`strong`** by default even though you wouldn't explicitly type this:

SomeClass.h

```
@property (nonatomic, weak) NSString *someString;
```

```objc
- (void) doSomething {
    NSString *helloString = @"Hello";  // A "local", implicitly strong variable
    self.someString = helloString
    NSLog(@"%@", self.someString);  // Prints Hello
}
```

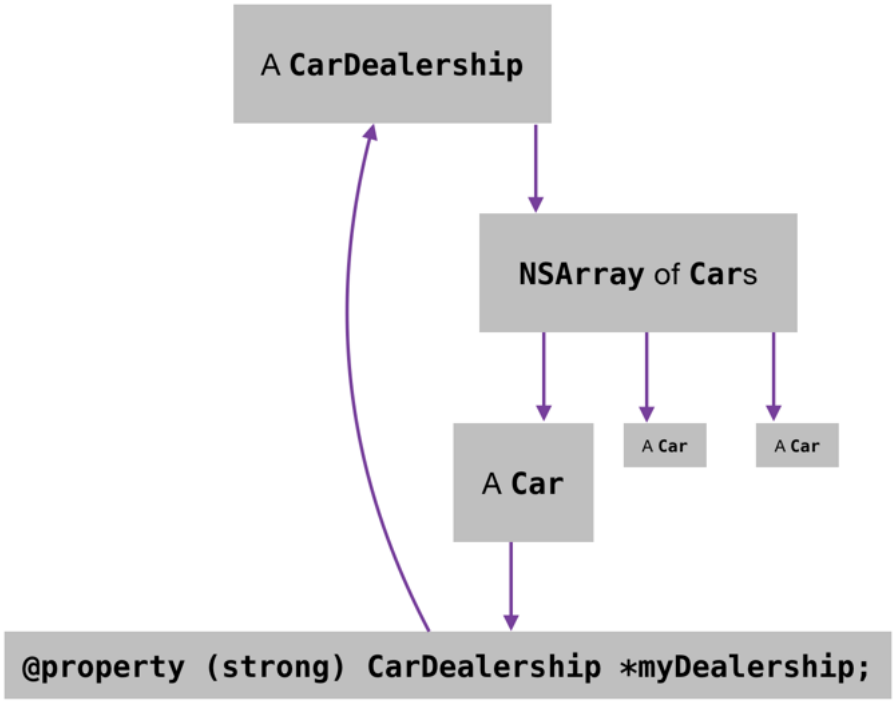
helloString is a strong reference, and exists until the **doSomething** method returns. Even though **self.someString** is a **weak** reference, **helloString** maintains a **strong** reference to **@"Hello"**.

Strong Reference Cycles

A strong reference cycle, also called a "retain" cycle, is a somewhat common memory management bug.

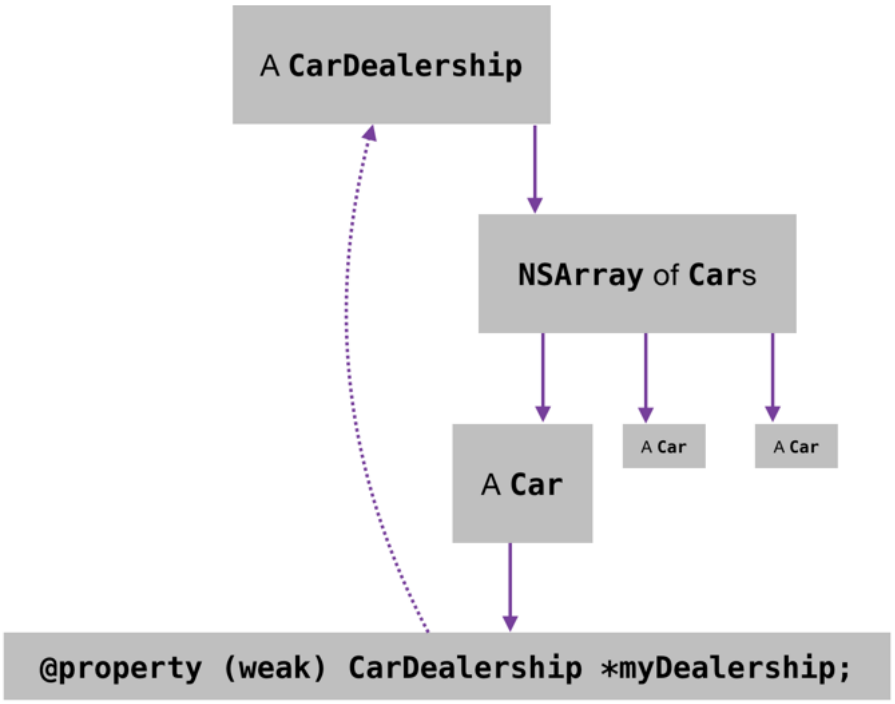
Objects are deallocated when there are no longer any strong references to it. If two objects have a strong reference to each other (either directly, or through other objects), then they will never be deallocated.

To continue our **Car** example, imagine you're writing a Car Dealership app, and you have some objects arranged like this:



None of these objects will ever get deallocated, because they all have strong pointers to each other.

To resolve this, switch one of the references from **strong** to **weak**:



Now if the **CarDealership** gets deallocated, all of its cars will get deallocated as we'd expect.

Apple's **Practical Memory Management** guide gives you a guideline for when to use **weak**:

> *Cocoa establishes a convention, therefore, that a "parent" object should maintain* **strong** *references to its "children," and that the children should have* **weak** *references to their parents.*

copy properties

Sometimes, properties are declared as **copy** instead of **weak** or **strong**.

When the property is set, the object being set is sent a **copy** message and a copy of the original object is stored.

This is helpful in avoiding errors with mutable data. For example, consider a **Politician** class with the property **fullName** declared like this:

Politician.h

```
@property (nonatomic, strong) NSString *fullName;
```

Consider this code used to create three politicians:

Election.m

```
    Politician *hillaryClinton = [[Politician alloc] init];
    Politician *randPaul = [[Politician alloc] init];
    Politician *joeBiden = [[Politician alloc] init];

    NSMutableString *currentName = [[@"Hillary Clinton"] mutableCopy];
    hillaryClinton.fullName = currentName;

    [currentName setString:@"Rand Paul"];
    randPaul.fullName = currentName;

    [currentName setString:@"Joe Biden"];
    joeBiden.fullName = currentName;

    self.politicians = @[hillaryClinton, randPaul, joeBiden];
}
```

> **`[[Politician alloc] init]`** *is equivalent to* **`[Politician new]`**. *Each creates a new instance of the* **`Politician`** *class.*

After this code runs, all three **`Politician`** objects will identify their **`fullName`** as "Joe Biden". This is because they're all pointing to the *same* **`NSMutableString`** instance.

If we update the property to **`copy`**:

Politician.h

```
- @property (nonatomic, strong) NSString *fullName;
+ @property (nonatomic, copy) NSString *fullName;
```

Then the issue will be resolved: each **`Politician`** will have a *copy* of the string, instead of all sharing the same instance.

`assign` properties

When you want to maintain basic data types–such as **`BOOL`**, **`NSInteger`**, **`CGFloat`**, etc.– as properties, all you need is the **`assign`** modifier:

Politician.h

```
+ @property (assign) NSInteger numberOfVotesRecevied;
+ @property (assign) BOOL isProDecapitation;
```

`assign` should be used *exclusively* with primitive data types.

Let's use these new properties:

Election.m

```
- (Politician *) electWinner
{
    Politician* winner;
    for (Politician *politician in self.politicians) {
        if (politician.isProDecapitation) {
            continue;
        }
        if (politician.numberOfVotesRecevied > winner.numberOfVotesRecevied) {
            winner = politician;
        }
    }
    return winner;
}
```

Open the BlocExercises project in Xcode. Open the folder that corresponds to this checkpoint and implement solutions to make the tests pass. Exercise descriptions are found the the header (`.h`) file, while you should implement your solutions in the implementation (`.m`) file. The test file will also give you some clues for implementing a proper solution, so be sure to read the tests as well.

Run the tests that correspond with the implementation file. Press ⌘**5** to view the **Test Navigator** and click the "play" button to the right of the corresponding test group. You can also run individual tests by clicking the "play" button corresponding to a specific test.

Commit and push your changes to GitHub:

Terminal

```
$ pwd #=> you should be in your forked ios-exercises directory. If not, cd into it.
$ git status #=> you should see the exercise file you just updated
$ git add . #=> stage the changes
$ git commit -m "Completed properties and scope"
$ git push origin master #=> send your changes to your remote GitHub repo
```

When you've completed the corresponding exercises, submit this assignment with the relevant repo and commit links.

assignment completed

COURSES

Full Stack Web Development
Frontend Web Development
UX Design
Android Development
iOS Development

ABOUT BLOC

Our Team | Jobs
Bloc Veterans Program
Employer Sponsored
FAQ
Blog
Engineering Blog
Refer-a-Friend
Privacy Policy

MADE BY BLOC

Tech Talks & Resources

Programming Bootcamp Comparison

Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css

Swiftris: Build Your First iOS Game with Swift

Webflow Tutorial: Design Responsive Sites with Webflow

Ruby Warrior

Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

Introduction to Swift

> *"When a true genius appears, you can know him by this sign: that all the dunces are in a confederacy against him."*
>
> *- Jonathan Swift, satirist, essayist, and political pamphleteer*

Objective-C, with its roots in Smalltalk, is decades old. Like the famous wizard Albus Dumbledore, it's powerful, but it's by no means *modern*. It's battle-tested and reliable, but it's got some problems that modern languages like Ruby and Python don't.

In June 2014, Apple announced a new programming language called Swift. Swift has similarities to modern languages like Ruby and Python and therefore is more accessible for people learning to build apps for iOS.

This checkpoint reviews some issues with Objective-C, gives you a general idea of how Swift solves these problems, and introduces some new features along the way.

In the next checkpoint we'll review snippets of Objective-C code and discuss their Swift equivalents.

> ***Note:*** *Most iOS apps are still written in Objective-C, and most hiring managers still want Objective-C developers. For these reasons, Bloc's iOS course is taught primarily in Objective-C. You are strongly encouraged to focus most of your learning on Objective-C. The two Swift checkpoints included in this course are intended to give you a broad overview of the Swift language and its advantages.*
>
> *If you want to learn more Swift after completing t[?]heckpoints, you'll be able to build Swift apps with your mentor during the Project Phase of this course.*

Safety

Objective-C can be unsafe at times. One reason is that it doesn't pay attention to the types of objects at runtime.

For example, consider a class like this:

```
@property (nonatomic, strong) NSMutableArray *presidentialCandidates;

@end

@implementation 1844PresidentialElection

- (void) addCandidates {
    if (!self.presidentialCandidates) {
        self.presidentialCandidates = [NSMutableArray array];
    }

    [self.presidentialCandidates addObject:@{@"name" : @"Martin Van Buren", @"info" : @"Spoke English as a second language"}];
    [self.presidentialCandidates addObject:@{@"name" : @"Lewis Cass", @"info" : @"A U.S. Army Brigadier General"}];
    [self.presidentialCandidates addObject:@{@"name" : @"James Buchanan", @"info" : @"A moderate"}];
}

- (void) printCandidates {
    for (NSDictionary *candidate in self.presidentialCandidates) {
        NSString *name = candidate[@"name"];
        NSLog(@"%@ has %ld letters in his name", name, (long)[name length]);
    }
}

@end
```

If we run **`printCandidates`**, all three 1844 US Presidential candidates will be printed to the console.

But let's say someone mischievous comes along and adds this to your mutable array:

```
@{"name" : @1, @"info" : @"number one"}
```

(Note that the value for **`@"name"`** is now an **`NSNumber`** and not an **`NSString`**.)

This won't immediately cause an error, but one will emerge as this line in **`printCandidates`** is executed:

```
NSString *name = candidate[@"name"];
```

Here, we're *assuming* the value stored under "name" in the **`NSDictionary`** is an **`NSString`**, but we're wrong: it's actually an **`NSNumber`**.

Because Objective-C doesn't check to make sure, this doesn't cause an error - the number **`@1`** is happily assigned to **`name`**.

```
NSLog(@"%@ has %ld letters in his name", name, (long)[name length]);
```

Here we're trying to ask the string for its **`length`**. However, it's actually a number, and **`NSNumber`** doesn't have a property called **`length`**. This is an error, and will cause the app to crash.

This sort of problem is way less likely to happen in Swift, because of a feature called **type safety**.

Brevity

Objective-C is openly mocked for its verbose method names. For example, in Objective-C you combine two arrays like this:

```
NSArray *newArray = [firstArray arrayByAddingObjectsFromArray:secondArray];
```

By comparison, the same thing in Swift looks like this:

```
let newArray = firstArray + secondArray
```

Another benefit is that Swift doesn't require separate header (`.h`) and implementation (`.m`) files: all code is placed in one file with a `.swift` extension.

Swift removes much of the boiler-plate and redundant code found in Objective-C. This results in cleaner code that's easier to read and write.

Modern and Powerful Features

Swift adds many modern features that are new to iOS and Mac development. We'll review a few.

Functions & Methods

Functions are similar to methods in Objective-C, with some improvements:

- Parameters can have default values
- Functions can return more than one value (via Tuples; see below)
- Functions are themselves objects, which can be passed in to other functions

A **function** doesn't have to be part of a specific class.

A **method** is a specific function that *is* associated with a specific class.

Optionals

In Objective-C, objects either exist, or they're `nil`. But for enumerations and simple types like `NSInteger`, there's no way to check if they exist.

In Swift, a new feature called *optionals* helps you handle situations in which a value may or may not be present. Optionals let you know either

- A value is present, and what the value is, or
- A value is not present

We'll see some examples of optionals in the next checkpoint.

Tuples

In Objective-C, you can only return one variable from a method:

```
- (NSString *) firstName;
```

In Swift, you can create a Tuple, which groups multiple values into one:

```
let fullName = ("James", "Polk")
```

`fullName` could be returned from a method, but would contain two separate values.

Tuples can contain any combination of types:

```
let emergencyNumber = (911, "Police, Fire, or Ambulance")
let threeLittlePigHouseMaterials = ("straw", "wood", "brick")
```

There's more coverage of tuples in the next checkpoint.

Playground

In all of the exercises you've completed so far, you've had to compile and run your project to test your code.

compilation.

To get started:

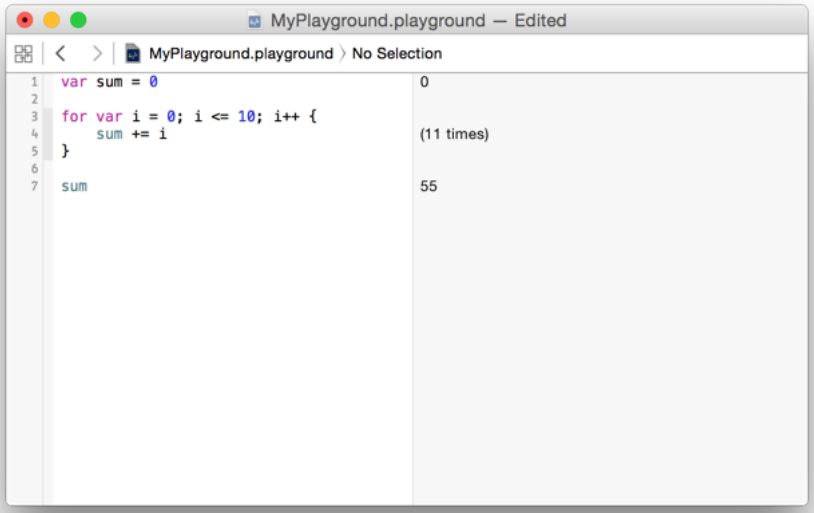
1. In Xcode, press **File** > **New Playground...**.
2. Select a filename and location

Your playground is now open. On the left, you can type code. On the right, you can see the result of each line.

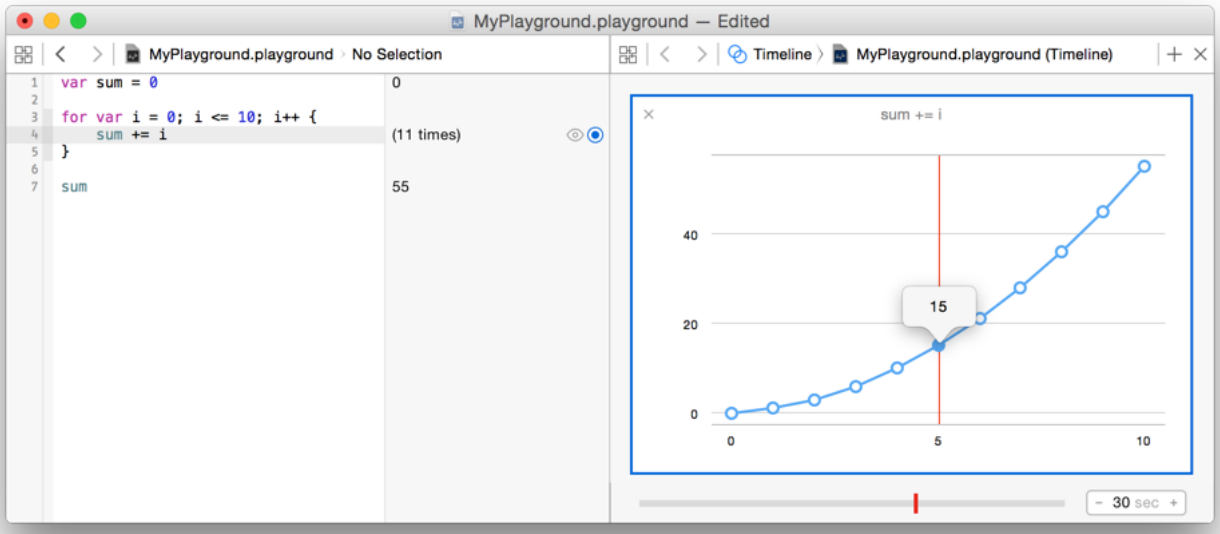
Let's write a basic loop as an example. (You'll learn more about Swift loops in the next checkpoint, but this will look pretty familiar.)

```
var sum = 0

for var i = 0; i <= 10; i++ {
    sum += i
}

sum
```

You'll notice that the playground shows you how many times the loop ran, and the values at the end:



Move your mouse over the phrase "(11 times)". The Value History button ( ) will appear. Press it to show interim values for a line of code that's executed repeatedly:



The x-axis shows execution time in seconds. The y-axis shows the value(s) logged for that line of code.

You can click on a single point to see the value at that point in time.

Playgrounds will make it easy to experiment with different algorithms as you write code. Playgrounds also have more advanced features thar the ones shown here. These are outlined in Apple's **Exploring and Evaluating Swift Code in a Playground** document.

Roadmap · **Previous Checkpoint** **Jump to Submission** · **Next Checkpoir**

Your assignment	Ask a question	Submit your work

Send a message to your mentor outlining the questions you have about Swift, and how it compares to the code you've written with Objective-C.

assignment completed

COURSES

- **Full Stack Web Development**
- **Frontend Web Development**
- **UX Design**
- **Android Development**
- **iOS Development**

Bloc Veterans Program

Employer Sponsored

FAQ

Blog

Engineering Blog

Refer-a-Friend

Privacy Policy

Terms of Service

MADE BY BLOC

Tech Talks & Resources

Programming Bootcamp Comparison

Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css

Swiftris: Build Your First iOS Game with Swift

Webflow Tutorial: Design Responsive Sites with Webflow

Ruby Warrior

Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Swift vs Objective-C

In this checkpoint, we'll review some of your Objective-C knowledge, and translate it to Swift.

Declaring Properties

Creating two properties looks like this in Objective-C:

Objective-C

```
property (nonatomic, strong) NSString *fullName;
property (nonatomic, strong) NSMutableString *mostRecentlyEatenMeal
```

If we wanted default values, we'd set them later:

Objective-C

```
self.fullName = @"Dumbledore";
self.mostRecentlyEatenMeal = [NSMutableString stringWithString:@"Owl Burger"];
```

Here's the same thing in Swift:

Swift

```
let fullName = "Dumbledore"
var mostRecentlyEatenMeal = "Owl Burger"
```

A few things are going on here:

- The *mutability* is defined by the use of **`let`** vs. **`var`**. **`let`** is used for variables that will never change again, and **`var`** for variables that will. S in Swift, we never need to specify a class with "Mutable" in its name.
- The *type* - String - is inferred by Swift based on the default values we specify. In fact, we don't need to specify a class at all. If you *don't* want to specify a default value, then you *do* need to specify a type, like so:

  Swift

  ```
  var fullName : String
  ```

- The default values are automatically assigned, so you don't need to set them inside a method.
- The property types (**`nonatomic`**, **`strong`**) are inferred.
- No **`@`**, **`*`** or **`;`** is needed.

This reduces the amount of code you need to write.

Strings and String Interpolation

Objective-C

```
NSInteger a = 3;
NSInteger b = 5;
NSString *streetName = @"Privet Drive";
NSString *uppercaseStreetName = [streetName uppercaseString];
NSString *familyDescription = @"TERRIBLE";
NSString *fullSentence = [NSString stringWithFormat:@"HARRY POTTER LIVES ON %ld %@ WITH HIS %@ FAMILY." , ((long)a + (long)b), upperCa

// Output: HARRY POTTER LIVES ON 8 PRIVET DRIVE WITH HIS TERRIBLE FAMILY.
```

In Swift, this is a bit simpler:

Swift

```
let a = 3
let b = 5
let streetName = "Privet Drive"
let uppercaseStreetName = streetName.uppercaseString
let familyDescription = "TERRIBLE"
let fullSentence = "HARRY POTTER LIVES ON \(a + b) \(uppercaseStreetName) WITH HIS \(familyDescription) FAMILY."
```

A few things to note:

- The type of **a** and **b** is inferred, so there's no need for casting.
- The string interpolation is much shorter and easier to read.

As mentioned earlier, there are no separate classes for a string vs. a mutable string in Swift. Instead, an object's mutability is determined by the use of **`let`** or **`var`**. So if we wanted to be even meaner to Harry's terrible, awful family, we might try:

Swift

```
let familyDescription = "TERRIBLE"
familyDescription += ", AWFUL"
```

However, this code would cause an error, because you're attempting to modify a constant.

Instead, changing **`let`** to **`var`** would work:

Swift

```
var familyDescription = "TERRIBLE"
familyDescription += ", AWFUL"
```

Arrays and Dictionaries

Objective-C

```
NSMutableArray *badGuys = [NSMutableArray arrayWithArray:@[@"Draco", @"Severus", @"Sirius"]];

NSMutableDictionary *ronWeasley = [NSMutableDictionary dictionaryWithDictionary:@{@"name" : @"ron", @"hair color" : @"red"}];
```

Swift

```
var badGuys = ["Draco", "Severus", "Sirius"]

var ronWeasley = ["name" : "ron", "hair color" : "red"]
```

Aside from syntax, Arrays and Dictionaries are pretty similar in Objective-C and Swift. There are, however, two differences you should know

First, collections in Objective-C can only hold *objects* like **`NSString`** and **`NSNumber`**. Collections in Swift can hold objects *or* primitive types like **`CGFloat`**.

Second, arrays in Objective-C can hold multiple different types of objects:

Objective-C

```
NSArray *randomStuff = @[@1000, @"Hello!", [UIColor greenColor]];
```

This is *possible* in Swift, but by default these objects only hold one particular type. Consider the array we made earlier:

Swift

```
var badGuys = ["Draco", "Severus", "Sirius"]
```

It's likely that we're only going to want to add strings to this array, and if we were to add a number it would likely be a mistake. Swift will catch this mistake; Objective-C won't.

Loops

Loops in Swift work the same as in Objective-C, but without the parentheses.

For example, this code in Objective-C:

Objective-C

```
for (int i = 1; i < 4; i++) {
    // do something with i
}
```

Looks like this in Swift:

Swift

```
for var i = 1; i < 4; i++ {
    // do something with i
}
```

For-In Loops in Swift

We've seen how you can iterate over every object in an **`NSArray`** using a loop like:

Objective-C

```
for (NSString *badGuy in badGuys) {
    [self haveDramaticConfrontationWith:badGuy];
}
```

Swift adds some new types of for-in loops. You can specify a range:

Swift

```
for diagonAlleyShopNumber in 100...200 {
    println("I heard you can get some really sweet wands at \(diagonAlleyShopNumber) Diagon Alley")
}
```

Here are a couple of example ranges:

- **`1..<10`** (two dots and a less-than sign). Half-closed range, includes 1 but not 10 (only goes up to 9)

Looping over Dictionaries

In Objective-C, when you loop over a dictionary, each object you receive is the *key*:

Objective-C

```
NSDictionary *ronWeasley = @{@"name" : @"ron", @"hair color" : @"red"};
for (NSString *key in ronWeasley) {
    NSLog(@"His %@ is %@.", key, ronWeasley[key]);
}

// Example Output:
// His name is ron.
// His hair color is red.

/* The order in which these items appear is arbitrary (because NSDictionary is unordered.) */
```

In Swift, you get the key and the value together:

Swift

```
let ronWeasley = ["name" : "ron", "hair color" : "red"]
for (characteristic, descriptionOfCharacteristic) in ronWeasley {
    println("His \(characteristic) is \(descriptionOfCharacteristic).")
}

// Output:
// His name is ron.
// His hair color is red.

/* (The output's order is similarly arbitrary.) */
```

This pairing of key and value is an example of the **tuple** that you learned about in the last session. (More on that later).

Functions

The difference between Swift functions and Objective-C methods is covered in the prior checkpoint, but the syntax is important.

Here's a function to print Harry's name:

Swift

```
func printHarrysName() {
    println("Harry Potter")
}

// Example Output:
// Harry Potter
```

If you want to print a full movie title:

Swift

```
func printHarrysMovieTitle(name: String) {
    println("Harry Potter and \(name)")
}

// Example Output for printHarrysMovieTitle("the Goblet of Fire"):
// Harry Potter and the Goblet of Fire
```

Swift

```swift
func printHarrysMovieTitle(name: String = "the Sorcerer's Stone") {
    println("Harry Potter and \(name)")
}


// printHarrysMovieTitle("the Goblet of Fire"):
// Harry Potter and the Goblet of Fire
//
// printHarrysMovieTitle():
// Harry Potter and the Sorcerer's Stone
```

If you want to return the movie title instead of print it:

Swift

```swift
func createHarrysMovieTitle(name: String = "the Sorcerer's Stone") -> String {
    return "Harry Potter and " + name
}

// Example Usage:
let title = createHarrysMovieTitle()
println(title)
// Output:
// Harry Potter and the Sorcerer's Stone
```

You can also return multiple values instead of one using a tuple.

Tuples

A tuple is a lightweight data structure that compounds multiple values into one.

Objective-C doesn't have tuples.

When more than one piece of data needs to be returned, it's common to return them in an array. Another common approach is returning them by reference:

Objective-C

```objc
-(void) makeUpperAndLowerCaseStrings
{
     self.startingString = @"HeRmOiNe";
     NSString *lowercaseString;
     NSString *uppercaseString = [self getUppercaseStringAndLowercaseString:&lowercaseString];
 }

-(NSString *) getUppercaseStringAndLowercaseString:(NSString**)string
{
     *string = [self.startingString lowercaseString];
     return [self.startingString uppercaseString]
}
```

This is ugly and unintuitive.

As outlined in the last checkpoint, a tuple generally looks like this:

Swift

```swift
let emergencyNumber = (911, "Police, Fire, or Ambulance")
```

Swift

```
func createHarrysMovieTitle(name: String = "the Sorcerer's Stone") -> (Int, String) {
    let ratingStars = 5
    let title = "Harry Potter and " + name
    return (ratingStars, title)
}

// Example usage
let (rating, fullTitle) = createHarrysMovieTitle()
println("\(fullTitle) got \(rating) stars. Let's see it!")
```

To increase clarity, you can also name the values:

Swift

```
func createHarrysMovieTitle(name: String = "the Sorcerer's Stone") -> (starRating: Int, title: String) {
    let ratingStars = 5
    let title = "Harry Potter and " + name
    return (ratingStars, title)
}

// Example usage
let movie = createHarrysMovieTitle()
println("\(movie.title) got \(movie.starRating) stars. Let's see it!")
```

Closures

A closure is a block of code that can be passed around or executed. If you think that sounds a lot like a function, you're right: a function is a special type of closure. Functions are convenient for naming code that will be reused multiple times, or will be called from other objects. But sometimes you just want to separate a chunk of code while keeping it inline with the related stuff.

Consider code that does this:

1. Gets a list of all Hogwarts students from the database
2. Sorts them by their grade in *Defense Against the Dark Arts*
3. Prints the top five to the console

You might want your sorting algorithm separate from your database reads and your logging code. A closure offers the opportunity to separat out your sorting code without the cognitive overhead of creating a separate function.

Example using **`sorted`**

From the Swift documentation, here's how **`sorted`** works:

The **`sorted`** function takes two arguments:

- An array of values of a known type.
- A closure that takes two arguments of the same type as the array's contents, and returns a **`Bool`** value to say whether the first value should appear before or after the second value once the values are sorted. The sorting closure needs to return **`true`** if the first value should appear *before* the second value, and **`false`** otherwise.

Since a function is a closure, we could write a function and pass it in:

Swift

```
let wizardsFromDatabase = …

func darkArtsGradeSort(student1: HogwartsStudent, student2: HogwartsStudent) -> Bool {
    return student1.darkArtsGrade > student2.darkArtsGrade
}
```

A closure is defined like a function, but:

- there's no name,
- the curly braces are on the outside, and
- the word **`in`** separates the definition of the closure from its body:

Swift

```
{ (parameters) -> (return type) in
    // statements
}
```

In our example, it could look like this:

Swift

```
let wizardsFromDatabase = …

let wizardsSortedByDarkArtsGrade = sorted(wizardsFromDatabase, {(student1: HogwartsStudent, student2: HogwartsStudent) -> Bool in
    return student1.darkArtsGrade > student2.darkArtsGrade
})
```

Swift is actually pretty smart. Swift knows that:

- our **`wizardsFromDatabase`** array is filled with **`HogwartsStudent`**s
- the **`sorted`** function is expecting a **`Bool`** to be returned.
- something is going to be returned at the end

Because we know this is implied, we can remove this stuff and write:

Swift

```
let wizardsSortedByDarkArtsGrade = sorted(wizardsFromDatabase, {student1, student2 in student1.darkArtsGrade > student2.darkArtsGrade})
```

To make this shorter we can use Swift's **Operator Overloading**, which would allow us to write:

Swift

```
let wizardsSortedByDarkArtsGrade = sorted(wizardsFromDatabase, >)
```

This works by redefining what the **>** comparison operator means when comparing **`HogwartsStudent`** objects.

1. Using Xcode 6, open **`SwiftExercises.playground`** from your fork of the **`ios-exercises`** repo.
2. Follow the instructions inline to complete the exercises.
3. Commit your code, push it to your repo, and send a link to your mentor for review.

assignment completed

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

- Our Team | Jobs
- Bloc Veterans Program
- Employer Sponsored
- FAQ
- Blog
- Engineering Blog
- Refer-a-Friend
- Privacy Policy
- Terms of Service

MADE BY BLOC

- Tech Talks & Resources
- Programming Bootcamp Comparison
- Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
- Swiftris: Build Your First iOS Game with Swift
- Webflow Tutorial: Design Responsive Sites with Webflow
- Ruby Warrior
- Bloc's Diversity Scholarship

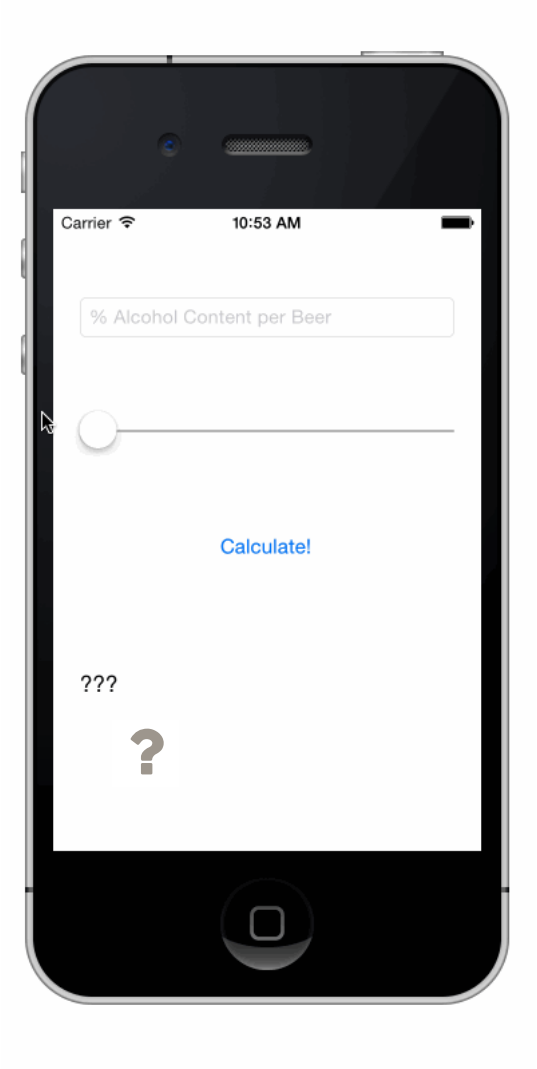
SIGN UP FOR OUR MAILING LIST

Send

- hello@bloc.io
- Considering enrolling? (404) 480-2562
- Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Storyboarding Alcolator



> *"History has demonstrated that the most notable winners usually encountered heartbreaking obstacles before they triumphed. They won because they refused to become discouraged by their defeats."*
>
> *- B.C. Forbes*

Introduction

It's the moment you've been waiting for - time to make your first app! In this checkpoint, you'll start working on an alcohol calculator (or "alcolator") that will help you manage your alcohol intake. You'll learn how to add different controls to your iOS app, including buttons, labels, and sliders.

Write Your Questions Down

We're going to throw a lot at you in this checkpoint. When you don't understand something, *don't stop.* Instead, take a note and keep moving. Every concept in this checkpoint will be discussed in more detail later, and you'll have a chance to go through everything with your mentor.

Creating Views

Planning how users will interact with your app is the heart of iOS development. Although iOS offers many ways to interact with users, the touchscreen is the most important because it is the most expressive.

Our calculator will get user input and display responses on the touchscreen. To build a touchscreen interface on iOS, we use *views.*

Introducing Views & View Controllers

We'll give you a brief introduction to views and view controllers, and then we'll get started on our calculator so you can see them in action.

On iOS, *views* have two fundamental purposes:

1. Display content to the user
2. Process touch events from the user

Some common view types are buttons, text fields and images.

Views are created, configured, and controlled by view controllers. One view controller typically controls many views. Think of a view controller as an intermediary between views and the outside world. Here are some things a view controller might do:

- Resize a button when the iPhone is rotated from portrait to landscape
- Save text typed by a user to disk
- Show the user a country's flag based on their current location

Creating View Controllers

View Controllers can be created three ways:

1. Entirely in code
2. Visually, one at a time, in separate files (.XIB files)
3. Visually, interconnected, in one file (Storyboard files)

Generally, the code approach is more configurable but more difficult to learn. Visual building is easier to learn, but it's less flexible. Storyboards can manage animations between view controllers (called *transitions*), but they can become cumbersome in large apps and are more difficult to work with in larger teams of people.

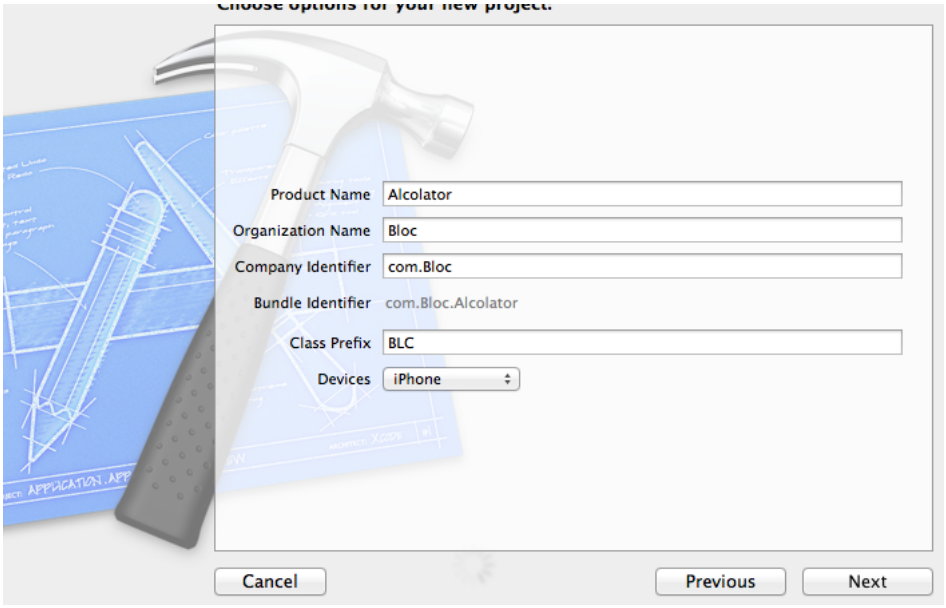
Eventually, you'll have experience with all three approaches and you'll gain an intuition for which approach is best for any given project. Our calculator app will be fairly simple, so we'll use Storyboards.

Making a Storyboard

Let's make a project and storyboard for our sample app, which we'll call Alcolator.

We'll start by launching Xcode. Press **File** > **New** > **Project**. On the left, select **iOS Application**, and on the right, select **Single View Application**. Press **Next**.

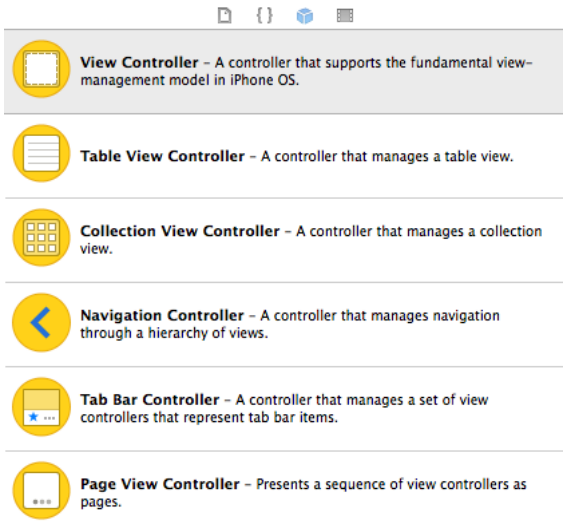
Configure your project like so: (Feel free to use the name “Calcohol” if you don't like ”Alcolator”.)



Press **Next** again and select a location on your computer. Make sure **Create git repository on My Mac** is checked, and press **Create**.

> *Make sure you don't save this project in your* `ios-exercises` *directory.*

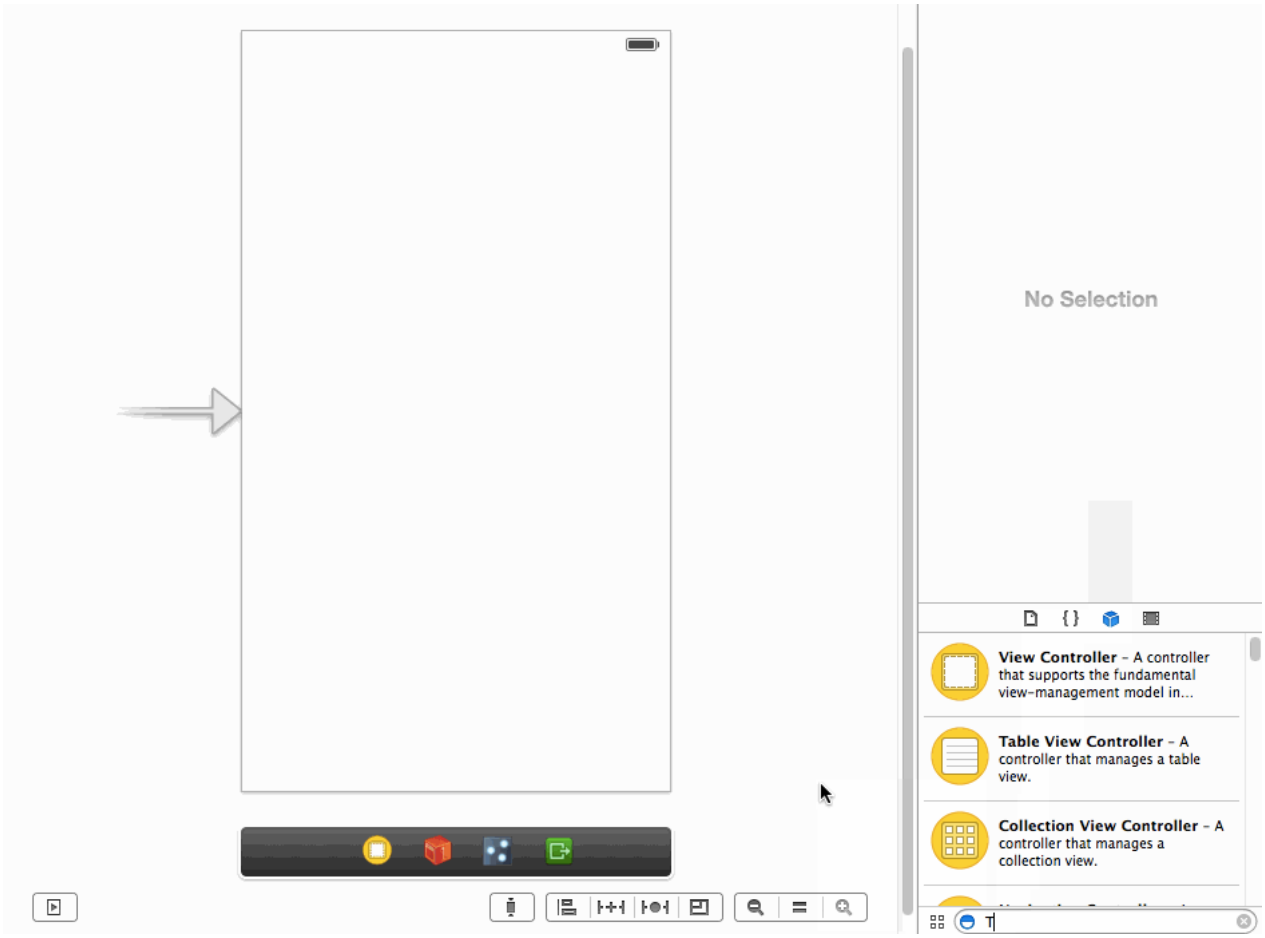
You should now see a list of files on the left. (If you don't, press **View** > **Navigators** > **Show Project Navigator**). Select your storyboard, `main.Storyboard` and on the bottom right, you should see a list of objects: (If you don't, press **View** > **Utilities** > **Show Object Library**)



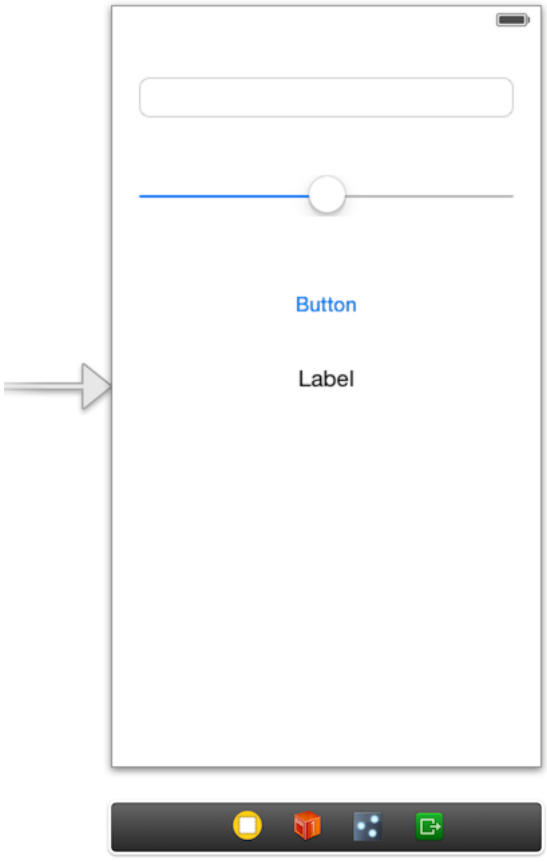
For this exercise, we'll need five objects on our View Controller:

- Button (`UIButton`)
- Label (`UILabel`)
- Text Field (`UITextField`)
- Slider (`UISlider`)
- Tap Gesture Recognizer (`UITapGestureRecognizer`)

Drag them from the library onto your View Controller. You can search for them in the search box underneath the library:

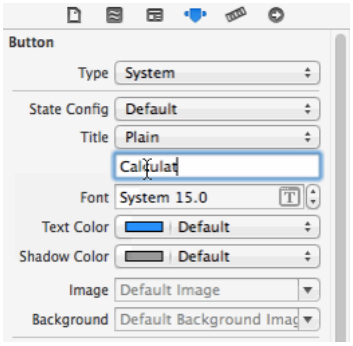


Your View Controller should look something like this:

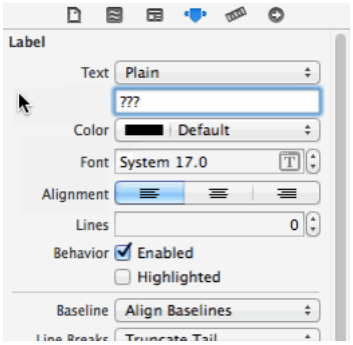


Click the button, then the Attributes Inspector icon: .

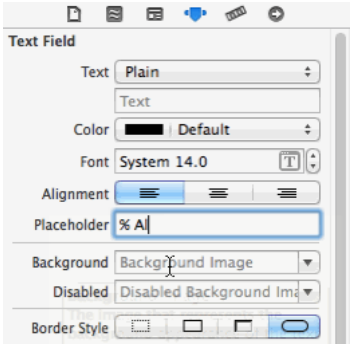
Change the button's title from "Button" to "Calculate!" You may need to resize the button so the text will fit.



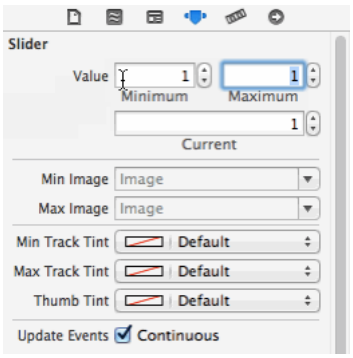
Similarly, change the label's **`text`** to say "???", and changes lines to 0.



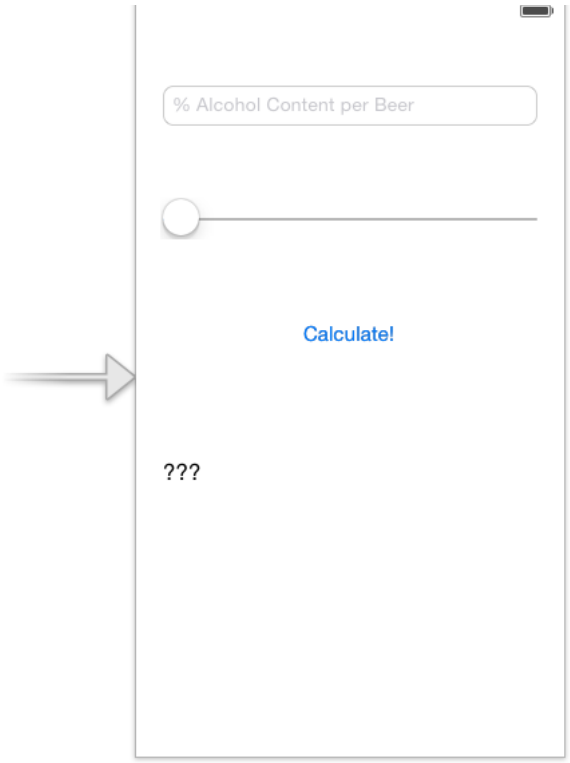
Change the text field's placeholder to say "% Alcohol Content Per Beer"



Change the slider's minimum to 1 and maximum to 10.



Congratulations! You've created your first View Controller using a storyboard!



It doesn't do anything yet, but it will soon.

Outlets and Actions

We want our calculator app to be able to:

- read a number (the % alcohol content) from the text box
- get the number of beers from the slider
- figure out how much alcohol that is
- calculate the equivalent in wine
- display the result in the label

In order to write code that does this, we'll need to add some *outlets* and *actions.* Before we get started, here's an overview of what those are:

Outlets

Outlets (**`IBOutlet`**) are portals from your code to your visual interface. They are only ever used with XIB and Storyboard files. Without them, your code will have a hard time telling your views to do anything.

Actions

In addition to simply looking good, views (**`UIView`**) are responsible for handling touch events like tapping, pinching, and swiping. Views use actions (**`IBAction`**) to communicate when something has happened. Some commonly used actions:

Event Name	Description	Example Uses
`UIControlEventTouchUpInside`	A touch ends inside the bounds of a view, e.g. a finger lifted off of a button.	Handle button taps
`UIControlEventEditingChanged`	A touch which edits the content of an **`UITextField`** object.	Handle text entered, cut, pasted, or deleted in a text field

it to emit a series of different values.

(Read about more events in `UIControl` **Class Reference**.)

Views communicate these actions to their **targets**. Nearly all of the time, a view's target will be the view controller it belongs to.

Outlets and Actions can go in the *Interface* or the *Implementation*

Outlets and Actions from our views will go in our code. Our code is split up into two sections: *interface* and *implementation*.

We need to place our calculator's Outlets and Actions correctly. Technically speaking, this will require us to understand the difference betwee interface and implementation. Let's re-cap some terms:

An **object** is the general name for most things you'll work with. Just like humans, bears, and dinosaurs are all *creatures*, buttons, targets and view controllers are all *objects.*

A **class** describes a particular object (`UILabel` is a *class*, just like "bear" is a kind of *creature*).

An **instance** is one specific object of a particular class (one specific label is an *instance,* just like "**Smokey the Bear**" is a specific bear).

Each class usually has two source code files ending in `.h` and `.m`:

- The `.h` file contains only interface code
- The `.m` file can contain both interface and implementation code

Interface code describes how one class communicates with another class.

Implementation code provides the instructions to follow when it's time to do something.

> ***Real-world example:*** *A dimmer switch is an* ***interface*** *for controlling your lights, containing two elements: a slider and a switch. The wiring that connects this switch to your overhead lights and your electricity is the* ***implementation****. You don't need to understand the implementation to use the interface.*

In terms of Outlets and Actions:

	represent	**so they belong in your**
Outlets	a connection to another class	interface
Actions	some type of event that your code needs to handle	implementation

Now that you have a general understanding of the difference, let's solidify that by making use of both outlets and actions.

Adding Outlets and Actions

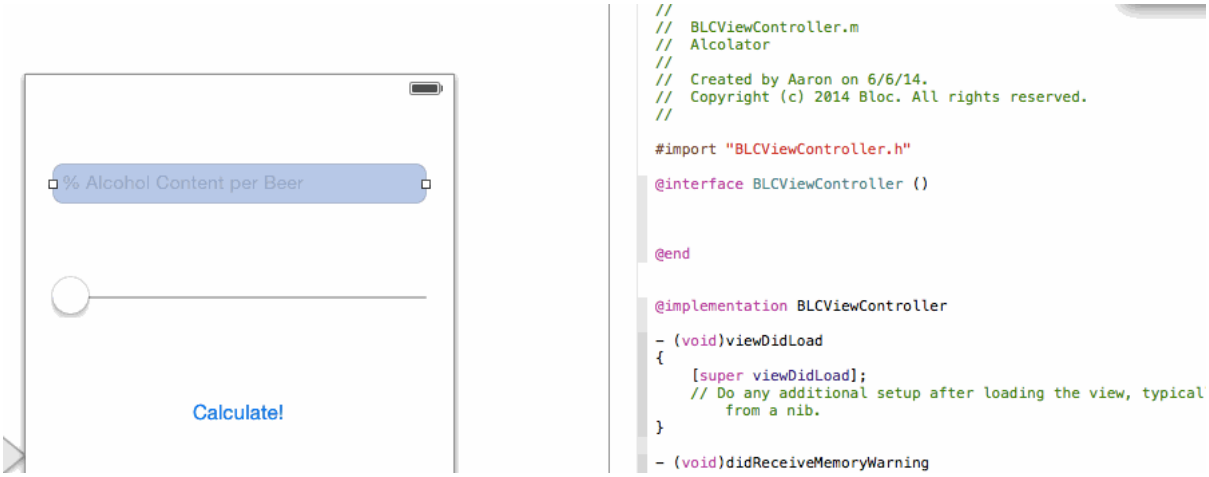
When you're working with a Storyboard and code at the same time, use the Assistant Editor, to see both together.

Click the Assistant Editor icon  .

You should now see your storyboard on your left, and your code (`BLCViewController.m`) on the right.

> ***Tip:*** *If you're working on a smaller screen, press* ***View*** *>* ***Navigators*** *>* ***Hide Navigator*** *and* ***View*** *>* ***Utilities*** *>* ***Hide Utilities*** *to get some more screen real estate.*

`beerPercentTextField`, and press **connect**:



Let's review the fields in the modal you just updated:

- **Connection:** Usually either an Outlet or an Action. It can also (rarely) be an "Outlet Collection", for example a group of 6 labels.
- **Object:** The object you're connecting *to*.
- **Name:** a name describing the object you connected, usually **`capitalizedLikeThis`**. This capitalization pattern is known as **camel case**, and is a common convention in many programming languages.
- **Type:** a hint for your code about what type of object you're connecting. Your selection here doesn't *change* the object, it just *describes* it. If you lie and say your **`UITextField`** is a **`UIButton`**, you might cause a crash or other bug.
- **Storage:** **`weak`** or **`strong`**, depending on whether *Object* above "owns" the object you're connecting. (This is a complex topic which is covered extensively later.) **`weak`** storage is usually appropriate for outlets.

Now hold down the control button and drag from your text field to the **implementation** and press **connect**:

Let's review the fields in the modal you just updated:

- **Name:** describe the action that you want to write code for
- **Event:** the name of the event (Editing Changed = **`UIControlEventEditingChanged`** above)
- **Arguments:** any additional information you want provided to your code. Usually "sender". (The "sender" is the object sending the event in this case the text field.)

Now we'll add the following:

- An **action** from the slider's Value Changed event called **`sliderValueDidChange`** (include the **`sender`** argument)
- An **action** from the button's Touch Up Inside event called **`buttonPressed`** (include the **`sender`** argument)
- An **action** from the tap gesture recognizer called **`tapGestureDidFire`**.

Your code should now look similar to this:

BLCViewController.m

```
#import "BLCViewController.h"

@interface BLCViewController ()
@property (weak, nonatomic) IBOutlet UITextField *beerPercentTextField;
@property (weak, nonatomic) IBOutlet UISlider *beerCountSlider;
@property (weak, nonatomic) IBOutlet UILabel *resultLabel;

@end

@implementation BLCViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
  // Do any additional setup after loading the view, typically from a nib.
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (IBAction)textFieldDidChange:(UITextField *)sender {

}

- (IBAction)sliderValueDidChange:(UISlider *)sender {

}

- (IBAction)buttonPressed:(UIButton *)sender {

}

- (IBAction)tapGestureDidFire:(UITapGestureRecognizer *)sender {

}

@end
```

Wrapping things up

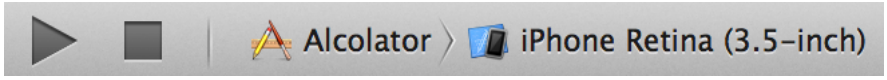
We're almost done! All we need to do now is write some code to handle the different events. You'll soon learn about the code we're using below, but for now just read it and try to make sense of what it's doing. In other words, don't get caught up with the syntax yet, just think about the logic. Add this code:

> *Note that lines starting with "//" are comments, and used to provide context for code. Commented lines are ignored by the compiler.*

BLCViewController.m

```
    // Make sure the text is a number
+     NSString *enteredText = sender.text;
+     float enteredNumber = [enteredText floatValue];

+     if (enteredNumber == 0) {
+         // The user typed 0, or something that's not a number, so clear the field
+         sender.text = nil;
+     }
}

- (IBAction)sliderValueDidChange:(UISlider *)sender {
+     NSLog(@"Slider value changed to %f", sender.value);
+     [self.beerPercentTextField resignFirstResponder];
}

- (IBAction)buttonPressed:(UIButton *)sender {

+     [self.beerPercentTextField resignFirstResponder];

      // first, calculate how much alcohol is in all those beers...

+     int numberOfBeers = self.beerCountSlider.value;
+     int ouncesInOneBeerGlass = 12;  //assume they are 12oz beer bottles

+     float alcoholPercentageOfBeer = [self.beerPercentTextField.text floatValue] / 100;
+     float ouncesOfAlcoholPerBeer = ouncesInOneBeerGlass * alcoholPercentageOfBeer;
+     float ouncesOfAlcoholTotal = ouncesOfAlcoholPerBeer * numberOfBeers;

     // now, calculate the equivalent amount of wine...

+     float ouncesInOneWineGlass = 5;  // wine glasses are usually 5oz
+     float alcoholPercentageOfWine = 0.13;  // 13% is average

+     float ouncesOfAlcoholPerWineGlass = ouncesInOneWineGlass * alcoholPercentageOfWine;
+     float numberOfWineGlassesForEquivalentAlcoholAmount = ouncesOfAlcoholTotal / ouncesOfAlcoholPerWineGlass;

   // decide whether to use "beer"/"beers" and "glass"/"glasses"

+     NSString *beerText;

+     if (numberOfBeers == 1) {
+         beerText = NSLocalizedString(@"beer", @"singular beer");
+     } else {
+         beerText = NSLocalizedString(@"beers", @"plural of beer");
+     }

+     NSString *wineText;

+     if (numberOfWineGlassesForEquivalentAlcoholAmount == 1) {
+         wineText = NSLocalizedString(@"glass", @"singular glass");
+     } else {
+         wineText = NSLocalizedString(@"glasses", @"plural of glass");
+     }

      // generate the result text, and display it on the label

+     NSString *resultText = [NSString stringWithFormat:NSLocalizedString(@"%d %@ contains as much alcohol as %.1f %@ of wine.", nil)
+     self.resultLabel.text = resultText;
}

- (IBAction)tapGestureDidFire:(UITapGestureRecognizer *)sender {
+     [self.beerPercentTextField resignFirstResponder];
}
```

Now, select an iPhone simulator in the toolbar and press the play button to try out your app:



- Type a percentage in the box
- Tap on a blank part of the view to dismiss the keyboard
- Adjust the slider to pick a number between **1** and **10** beers
- Press **Calculate** to see the equivalent number of wine glasses

Assignment

Add another label so the user can see the number of beers as the slider moves.

Modify the project so the label automatically updates when you adjust the slider (without tapping the button.)

Send a message to your mentor describing in your own words:

- the difference between a View and a View Controller
- the difference between an Outlet and an Action
- the difference between a Class and an Instance

Finally, we'll want to commit our code and push it to a GitHub repo. In your terminal:

Terminal

```
$ pwd #=> should be in /alcolator. If not, cd into it
$ git status #=> you should see your xcodeproj, BLCViewController.m and storyboard files with changes
$ git add .
$ git commit -m 'Storyboarding Alcolator'
```

Sign into your GitHub account and create a new repo named "alcolator". In your terminal:

Terminal

```
$ git remote add origin https://github.com/<username>/alcolator.git #=> replace <username> with your GitHub username
$ git push -u origin master
```

Refresh your new repo's home page, and you should see your project in GitHub. Submit this checkpoint's assignment with links for your repo and commit.

Add another label so the user can see the number of beers as the slider moves.

Modify the project so the label automatically updates when you adjust the slider (without tapping the button.)

Send a message to your mentor describing in your own words:

- the difference between a View and a View Controller
- the difference between an Outlet and an Action
- the difference between a Class and an Instance

Finally, we'll want to commit our code and push it to a Github repo. In your terminal:

Terminal

```
$ git status #=> you should see your xcodeproj, BLCViewController.m and storyboard files with changes
$ git add .
$ git commit -m 'Storyboarding Alcolator'
```

Sign into your Github account and create a new repo named "alcolator". In your terminal:

Terminal

```
$ git remote add origin https://github.com/<username>/alcolator.git #=> replace <username> with your Github username
$ git push -u origin master
```

Refresh your new repo's home page, and you should see your project in Github. Submit this checkpoint's assignment with links for your repo and commit.

assignment completed

COURSES

Full Stack Web Development

Frontend Web Development

UX Design

Android Development

iOS Development

ABOUT BLOC

Our Team | Jobs

Bloc Veterans Program

Employer Sponsored

FAQ

Blog

Engineering Blog

Refer-a-Friend

Privacy Policy

Terms of Service

MADE BY BLOC

Tech Talks & Resources

Programming Bootcamp Comparison

Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css

Swiftris: Build Your First iOS Game with Swift

Webflow Tutorial: Design Responsive Sites with Webflow

Ruby Warrior

Bloc's Diversity Scholarship

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Sizing and Styling

View Controllers mediate between views (like buttons) and your app's data. In our last checkpoint, the view controller handled the input from the text box, slider, button, and displayed the output on a label.

Introduction

In this checkpoint we'll modify our view controller from the last checkpoint so that it no longer relies on a Storyboard or Interface Builder. The resulting app will look the same as it did in the prior checkpoint, but it will be quite different under the hood.

The view controller in the previous checkpoint, **`BLCViewController`** was generated for us and is a **subclass** of **`UIViewController`**.

What is subclassing?

- A **MINI Cooper Convertible** is a type of Mini Cooper.
- A MINI Cooper is a type of car.
- A car is a type of vehicle.
- A vehicle is a type of machine.
- A machine is a type of tool.

Et cetera. A **`Vehicle`** class might have these methods:

Vehicle.h

```
- (NSUInteger) numberOfWheels;
- (NSUInteger) numberOfWings;
- (NSUInteger) engineMileagePerGallon;
```

`Car`, as a specific type of **`Vehicle`**, it may implement some defaults:

Car.m

```
- (NSUInteger) numberOfWheels {
    return 4;
}
- (NSUInteger) numberOfWings {
    return 2;
}
- (NSUInteger) engineMileagePerGallon {
    return 20;
}
```

`MiniCooper`, as a specific **`Car`**, would override anything that makes it different from **`Car`**:

MiniCooper.m

```
- (NSUInteger) engineMileagePerGallon {
    return 31;
}
```

`MiniCooper` doesn't need to override **`numberOfWings`** or **`numberOfWheels`**, because **`MiniCooper`** and **`Car`**'s implementation of those is identical

Identifying a Subclass

In the **`.h`** file, a subclass is indicated like so:

MiniCooper.h

```
// Imports
@interface MiniCooper : Car
// Properties
// Methods
@end
```

Where **`MiniCooper`** is the name of the file as well as the first string which follows **`@interface`**. The **`:`** indicates that the following string of characters represents the class we extend from or *inherit* from, **`Car`**. **`MiniCooper`** is a subclass of **`Car`** just like **`BLCViewController`** is a subclass of **`UIViewController`**:

BLCViewController.h

```
// Imports
@interface BLCViewController : UIViewController
// Properties
// Methods
@end
```

Additional Help with Inheritance

You'll need to know these inheritance terms when talking to your mentor or other developers about subclassing:

Term	Definition	Example
Inherits From	A class inherits from any of its super classes.	**`Car`** can be said to inherit from **`Vehicle`** or **`Machine`** or **`Tool`**.
Subclass (noun)	A subclass is a class that inherits from another class.	**`Car`** is a subclass of **`Vehicle`**.
Subclass (verb)	Creating a subclass.	Instead of making **`MiniCooper`** from scratch, you should subclass **`Car`**.
Superclass (noun)	A superclass is a class that a subclass inherits from.	**`Vehicle`** is a superclass of **`Car`**.
Override (verb)	Implementing a method or variable that's different from the super class.	**`MiniCooper`** has better gas mileage than an average **`Car`**, so you need to override **`engineMileagePerGallon`**.

It's really important to have a grasp of what subclassing is. Newer developers, or developers coming from languages without inheritance can really struggle with it, so make sure to talk to your mentor if you have more questions. In the mean time, continue with this lesson - the process of subclassing a **`UIViewController`** may elucidate this matter.

Converting to a Universal Project

Let's convert Alcolator to a **Universal** project, one which supports both the iPhone and the iPad.

Before we start coding though, we'll introduce a better development flow for making changes and adding features to our app. We'll employ G branches to keep our changes separated and safe, until we're confident in merging them into the master branch.

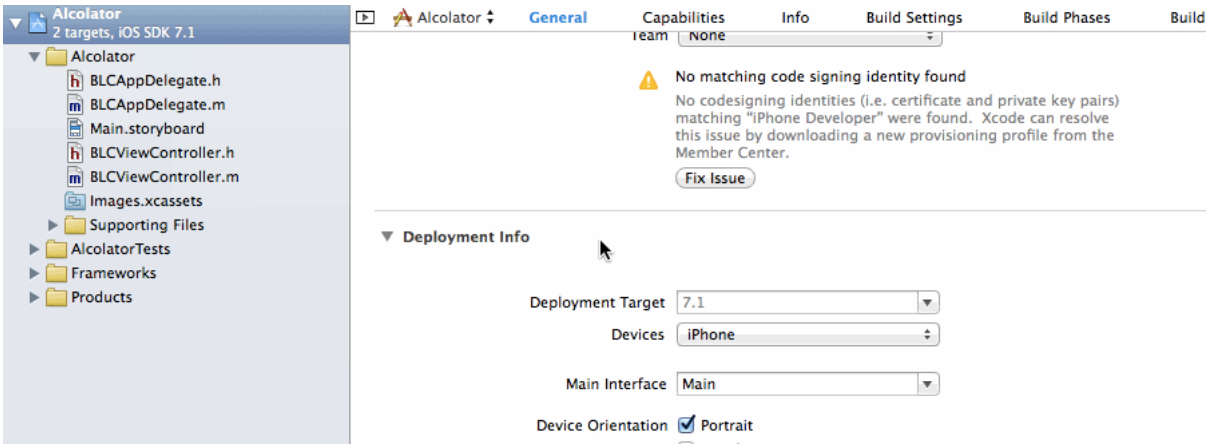
This is quite easy with Git. Open your terminal and **`cd`** into your **`Alcolator`** project folder:

Terminal

```
$ cd alcolator
$ git checkout -b custom-view-controller
```

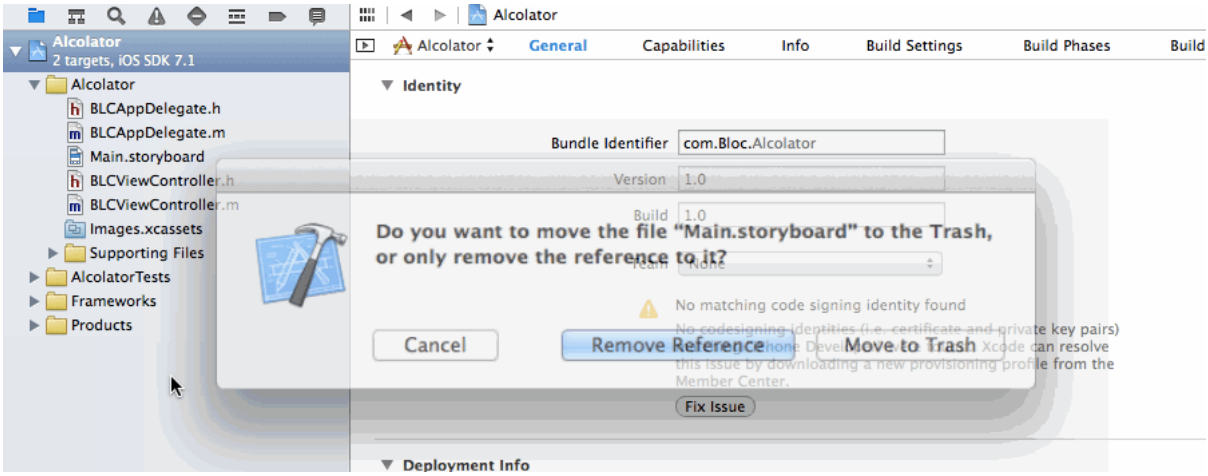
That's it. Now we're working in a separate branch, so that we can't affect our master branch, which should always be kept clean. We'll merge this branch to master at the end of the checkpoint, after we've tested our app's behavior.

Now we'll continue with converting our project to a Universal type.



In the Project Navigator, click on the **Alcolator** project. Under the **General** tab, scroll down to **Deployment Info** and change the **Devices** drop-down from *iPhone* to *Universal*. When prompted to copy `Main`, choose **"Don't Copy"**.

Next, we'll remove our Storyboard since we'll create views in code from now on:



In the Project Navigator, find the `Main.storyboard` file under `Alcolator` and delete `Main.storyboard`. Find the `Alcolator-Info.plist` file under `Alcolator` > `Supporting Files` and delete the `Main storyboard file base name` entry. Press **save** after the deletion is complete.

Writing Alcolator in Code

Time to expand on `BLCViewController` so that it no longer relies on the `Main.storyboard` file by generating all of its views programmatically.

The `BLCViewController.h` file contains your class's public interface. Since there's nothing our class does that we want other classes to know about, we'll leave it as is. The `BLCViewController.m` file contains your class's private interface, and your class's implementation.

> *A private interface is similar to a public one except only **your** class knows about it. This is where you can add additional methods and properties not accessible from outside of the class.*

A private interface is declared in the `.m` file:

SomeClass.m

```
@interface SomeClass ()
@property(weak, nonatomic) BOOL secretBoolProperty;
- (void)secretMethod;
- (void)secretAgentMethod:(MI5Agent *)agent;
@end
```

Let's setup our properties. These will be the same as before with the addition of properties for the button and tap gesture recognizer.

> *When generating views in code, it's a good practice to make a property for each view. You'll need to refer to these views in multiple methods.*

Two objects typically communicate using a "delegate protocol", which defines the communication structure. See **iOS Design Patterns: Delegation & Protocols** to read more about this, or continue on to learn by example.

We need the text field to communicate with the view controller. To accomplish this, we'll need to declare that the view controller subclass conforms to the **`UITextFieldDelegate`** protocol, since **`UIViewController`** doesn't conform out of the box.

Here's how the code looks:

BLCViewController.m

```
-   @interface BLCViewController ()
+   @interface BLCViewController () <UITextFieldDelegate>

-   @property (weak, nonatomic) IBOutlet UITextField *beerPercentTextField;
-   @property (weak, nonatomic) IBOutlet UISlider *beerCountSlider;
-   @property (weak, nonatomic) IBOutlet UILabel *resultLabel;
+   @property (weak, nonatomic) UITextField *beerPercentTextField;
+   @property (weak, nonatomic) UISlider *beerCountSlider;
+   @property (weak, nonatomic) UILabel *resultLabel;
+   @property (weak, nonatomic) UIButton *calculateButton;
+   @property (weak, nonatomic) UITapGestureRecognizer *hideKeyboardTapGestureRecognizer;
@end
```

When we created our interface in Xcode's Interface Builder, we needed the **`IBOutlet`** keyword to connect the code to the visual interface. Since we won't be creating an Interface Builder Outlet any more, we remove the **`IBOutlet`** from the properties. We can also keep our logic code from before by changing **`IBAction`** to **`void`**:

BLCViewController.m

```
-   - (IBAction)textFieldDidChange:(UITextField *)sender {
+   - (void)textFieldDidChange:(UITextField *)sender {
    // SAME CODE AS BEFORE
}

-   - (IBAction)sliderValueDidChange:(UISlider *)sender {
+   - (void)sliderValueDidChange:(UISlider *)sender {
    // SAME CODE AS BEFORE
}

-   - (IBAction)buttonPressed:(UIButton *)sender {
+   - (void)buttonPressed:(UIButton *)sender {
    // SAME CODE AS BEFORE
}

-   - (IBAction)tapGestureDidFire:(UITapGestureRecognizer *)sender {
+   - (void)tapGestureDidFire:(UITapGestureRecognizer *)sender {
    // SAME CODE AS BEFORE
}
```

`void` means that no values are returned at the end of these methods.

We need to create and configure the views. To accomplish this, you'll need to know how to override **`UIViewController`** methods.

Commonly overridden methods

familiar with them. Then we'll override the necessary methods in order to get the calculator working again.

Here are the common ones:

Method	Purpose
`loadView`	Can be overridden if you want to create the view controller's primary view `self.view` programmatically.
`viewDidLoad`	Called once, after the view is done loading, but before it appears. It's used for additional customization.

`viewWillAppear:`
`viewDidAppear:`
`viewWillDisappear:`
`viewDidDisappear:`
	Called when the view is hidden or shown, for example if the user switches apps or navigates to another area of your app. It is commonly used to update data.

`viewWillLayoutSubviews`	Called whenever `self.view` is about to resize its subviews. The most common case of this happening is on device rotation.

Common bug warning: Your implementation of everything after `viewDidLoad` should be **idempotent**. For example, write your code so that you can call `viewWillAppear:` multiple times in a row without the number of times affecting the result.

Example Code	Idempotent?	Why?
`self.calculateButton.alpha = 0.5`	Yes! :-)	`alpha` will always be set to `0.5` no matter how many times this is run.
`self.calculateButton.alpha += 0.1`	No. :-(	The result of adding `0.1` to a value changes depending on how many times you do it.

For now, we'll only override three of these methods: `loadView`, `viewDidLoad`, and `viewWillLayoutSubviews`.

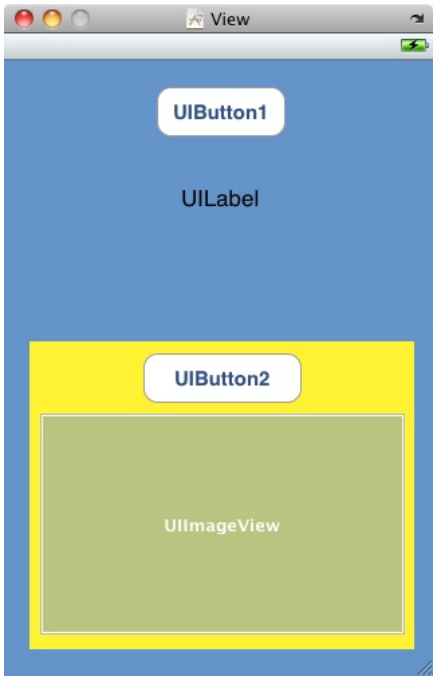
Overriding `loadView`

`loadView` is where you create the main view objects, subviews, and define how they're connected:

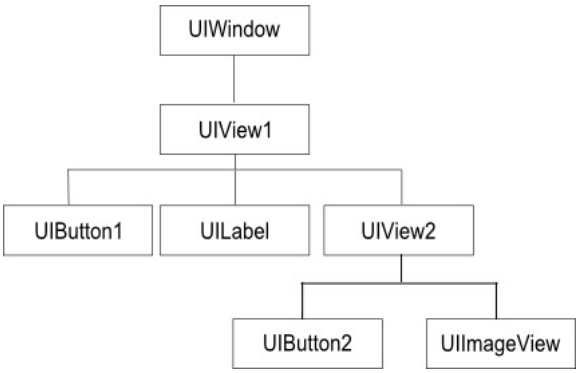
BLCViewController.m

```
    // Allocate and initialize the all-encompassing view
+       self.view = [[UIView alloc] init];

    // Allocate and initialize each of our views and the gesture recognizer
+       UITextField *textField = [[UITextField alloc] init];
+       UISlider *slider = [[UISlider alloc] init];
+       UILabel *label = [[UILabel alloc] init];
+       UIButton *button = [UIButton buttonWithType:UIButtonTypeSystem];
+       UITapGestureRecognizer *tap = [[UITapGestureRecognizer alloc] init];

    // Add each view and the gesture recognizer as the view's subviews
+       [self.view addSubview:textField];
+       [self.view addSubview:slider];
+       [self.view addSubview:label];
+       [self.view addSubview:button];
+       [self.view addGestureRecognizer:tap];

    // Assign the views and gesture recognizer to our properties
+       self.beerPercentTextField = textField;
+       self.beerCountSlider = slider;
+       self.resultLabel = label;
+       self.calculateButton = button;
+       self.hideKeyboardTapGestureRecognizer = tap;
+   }
```

`addSubview:` is a method found in the **`UIView`** class. It allows us to nest views inside one another. Here's a simple screenshot of some elements:



And here's the view hierarchy describing its layout:



`UIView1`. However, it doesn't stop there. **`UIView2`** (the view with a yellow background) has its own nested views: **`UIImageView`** and **`UIButton2`**.

We don't do any configuration in **`loadView`**- we don't choose colors, fonts, text, etc. **`viewDidLoad`** is the more stylistically appropriate place for customizing your views.

Overriding **`viewDidLoad`**

Here, we'll configure the view elements that we created in **`loadView`**:

BLCViewController.m

```
+   - (void)viewDidLoad
+   {
    // Calls the superclass's implementation
+       [super viewDidLoad];

   // Set our primary view's background color to lightGrayColor
+       self.view.backgroundColor = [UIColor lightGrayColor];

    // Tells the text field that `self`, this instance of `BLCViewController` should be treated as the text field's delegate
+       self.beerPercentTextField.delegate = self;

    // Set the placeholder text
+       self.beerPercentTextField.placeholder = NSLocalizedString(@"% Alcohol Content Per Beer", @"Beer percent placeholder text");

    // Tells `self.beerCountSlider` that when its value changes, it should call `[self -sliderValueDidChange:]`.
    // This is equivalent to connecting the IBAction in our previous checkpoint
+       [self.beerCountSlider addTarget:self action:@selector(sliderValueDidChange:) forControlEvents:UIControlEventValueChanged];

    // Set the minimum and maximum number of beers
+       self.beerCountSlider.minimumValue = 1;
+       self.beerCountSlider.maximumValue = 10;

    // Tells `self.calculateButton` that when a finger is lifted from the button while still inside its bounds, to call `[self -button
+       [self.calculateButton addTarget:self action:@selector(buttonPressed:) forControlEvents:UIControlEventTouchUpInside];

    // Set the title of the button
+       [self.calculateButton setTitle:NSLocalizedString(@"Calculate!", @"Calculate command") forState:UIControlStateNormal];

    // Tells the tap gesture recognizer to call `[self -tapGestureDidFire:]` when it detects a tap.
+       [self.hideKeyboardTapGestureRecognizer addTarget:self action:@selector(tapGestureDidFire:)];

    // Gets rid of the maximum number of lines on the label
+       self.resultLabel.numberOfLines = 0;
+   }
```

What's a Delegate? Delegates are covered more extensively in **iOS Design Patterns: Delegation & Protocols**. Here's an excerpt that explains it generally:

> *Generally speaking, classes all have some default behavior. If you choose to implement their delegate methods, your app can alter or respond to this behavior.*

In our case, we're responding to the text field's default behavior of updating the display as letters are typed.

Lastly, we need to place the views on screen.

Overriding **`viewWillLayoutSubviews`**

One quick note before we begin: **`viewWillLayoutSubviews`** is not the only place where you can layout your views. It's a good default, though: setting your view locations here supports handling rotations, as well as any events that might change the size of your view. For example, if the user is interrupted by a phone call, the status bar will become taller and therefore your available space will shrink.

BLCViewController.m

```
+  - (void) viewWillLayoutSubviews {
+      [super viewWillLayoutSubviews];

+      CGFloat viewWidth = 320;
+      CGFloat padding = 20;
+      CGFloat itemWidth = viewWidth - padding - padding;
+      CGFloat itemHeight = 44;

+      self.beerPercentTextField.frame = CGRectMake(padding, padding, itemWidth, itemHeight);

+      CGFloat bottomOfTextField = CGRectGetMaxY(self.beerPercentTextField.frame);
+      self.beerCountSlider.frame = CGRectMake(padding, bottomOfTextField + padding, itemWidth, itemHeight);

+      CGFloat bottomOfSlider = CGRectGetMaxY(self.beerCountSlider.frame);
+      self.resultLabel.frame = CGRectMake(padding, bottomOfSlider + padding, itemWidth, itemHeight * 4);

+      CGFloat bottomOfLabel = CGRectGetMaxY(self.resultLabel.frame);
+      self.calculateButton.frame = CGRectMake(padding, bottomOfLabel + padding, itemWidth, itemHeight);
+  }
```

What's a Frame?

Notice the use of **`frame`** in the code above. A **`frame`** is a view's location with respect to its parent view or *superview.* A **`frame`** is of type **`CGRect`** which is a structure that stores a **`CGPoint`** and a **`CGSize`**. A **`CGPoint`** consists of an **`x`** and **`y`** which specify a coordinate on screen. A **`CGSize`** object consists of a **`width`** and a **`height`**. These two elements combine to form a **`CGRect`**. The **`CGPoint`** specifies the top-left coordinate where the view begins and the **`CGSize`** represents the view's width and height.

`CGRectMake` is a convenient shortcut for creating **`CGRect`**s, here's its signature:

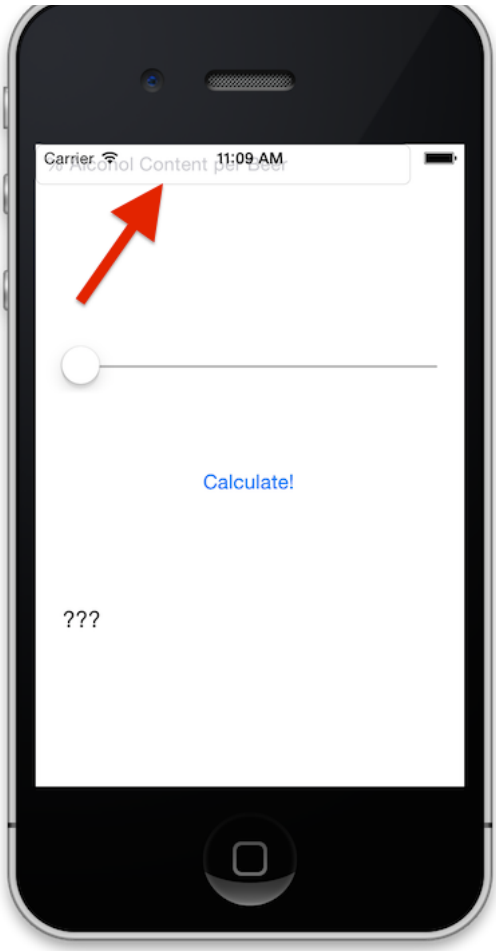
CGGeometry.h

```
CGRectMake(CGFloat x, CGFloat y, CGFloat width, CGFloat height)
```

> *You can read this as* **`CGRectMake(howFarFromTheLeft, howFarFromTheTop, howWide, howTall)`**

Remember, a **`frame`** is with respect to a view's superview. In our case, every subview (**`beerPercentTextField`**, **`beerCountSlider`**, **`resultLabe`** etc.) are subviews of **`self.view`** which happens to comprise the entire screen. Therefore, a subview of **`self.view`** with a top-left coordinate o **`(0, 0)`** will be placed at the top of the screen and all the way to the left.

Here's what our **`beerPercentTextField`** would look like if we set its **`frame`**'s **`CGPoint`** to **`(0, 0)`**:



Here's how we set **`calculateButton`**'s frame:

BLCViewController.m

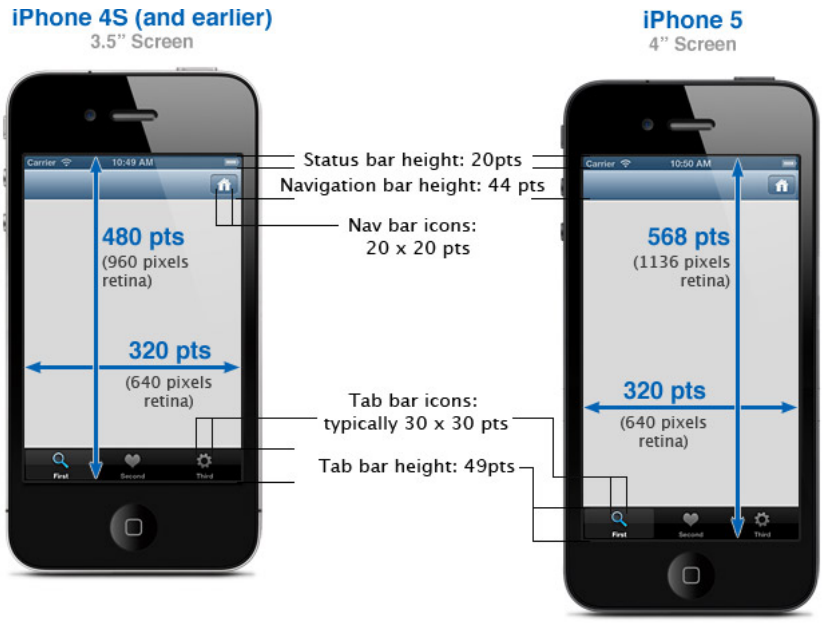
```
    self.calculateButton.frame = CGRectMake(padding, bottomOfLabel + padding, itemWidth, itemHeight);
```

This line of code affects the appearance immediately. Likely in the next microsecond, the view will be re-drawn at the location we specified with the precise height and width we've specified. This portion of the code may appear confusing, so let's break it down piece by piece startin at the top:

BLCViewController.m

```
    CGFloat viewWidth = 320;
    CGFloat padding = 20;
    CGFloat itemWidth = viewWidth - padding - padding;
    CGFloat itemHeight = 44;
```

`viewWidth` is set to **`320`**, which is the number of horizontal points available to all iPhones and iPods predating iPhone 6. Here's a convenient diagram which illustrates a few important Cartesian values: (Cartesian values pinpoint where you are on a map or graph.)



You may have noticed that iPads are missing from this diagram, as are iPhone 6 and iPhone 6 Plus. In the assignment for this section, you'll update this code to support these other devices.

As you can see, **320** is a significant number in iOS development. **20**, our choice for **padding** is another important number, it's the size of the status bar. We've chosen **20** because it will appeal to the eye. **20** will be a comfortable distance to place between views as well as between views and the edges of the screen.

itemWidth is the width each of our items will fill and thereby allow us to keep them **20** points away from the left and right sides of the screen, **280**. **itemHeight** is chosen arbitrarily but as you may have noticed, it happens to be the size of a standard navigation bar.

We're ready with a plan of attack: every view is going to be **280** points wide, at least **44** points tall and **20** points away from anything else on screen. We begin with the top-most view - **beerPercentTextField** - because once we've set its **frame** the remaining views will fall in line with respect to it:

BLCViewController.m

```
    self.beerPercentTextField.frame = CGRectMake(padding, padding, itemWidth, itemHeight);
```

Which translates to the following values:

BLCViewController.m

```
    self.beerPercentTextField.frame = CGRectMake(20, 20, 280, 44);
```

We tell **beerPercentTextField** to start at **(20, 20)**, **20** points from the left and **20** points from the top. We also tell it to be **280** points wide and **44** points tall; simple as that. Then we proceed to the view we wish to place immediately below **beerPercentTextField**, which is **beerCountSlider**.

BLCViewController.m

```
    CGFloat bottomOfTextField = CGRectGetMaxY(self.beerPercentTextField.frame);
    self.beerCountSlider.frame = CGRectMake(padding, bottomOfTextField + padding, itemWidth, itemHeight);
```

CGRectGetMaxY(CGRect) is a shortcut which performs the following calculation: **y** coordinate of **frame** + **height** of **frame**. As the variable nam suggests, we've calculated the **y** value at the very bottom edge of **beerPercentTextField**. Since we know where its bottom is, we know where to place the next view: **20** *points further down.* Therefore, the **frame** assignment translates to:

BLCViewController.m

```
    self.beerCountSlider.frame = CGRectMake(20, 64 + 20, 280, 44);
```

We repeat this process two more times for the remaining elements. We recover the previous element's bottom-most **y** coordinate, add **20** points to it and then set the **frame** based on that location.

Displaying your View Controller

Every app has one "window" (**UIWindow**) which takes up the whole screen. For our app to run properly, we'll need to set the root view controller to the view controller we made.

Start by navigating to **BLCAppDelegate.m**. Add this at the top in the import section:

BLCAppDelegate.m

```
+   #import "BLCViewController.h"
```

Every app has (at least) one "window" which takes up the full screen. This window has a root view controller, which it displays. When we used storyboards, this was done for us.

Here, we create a window (**UIWindow**), store it in **self.window**, set **BLCViewController** to the root view controller, and then set the window as the application's key window.

BLCAppDelegate.m

```
 - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
+     self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]] ;
    // Override point for customization after application launch.
+     BLCViewController *viewController = [[BLCViewController alloc] init];
+     self.window.rootViewController = viewController;
+     [self.window makeKeyAndVisible];
    return YES;
}
```

Run your app on the iPhone 5s simulator by pressing the play button in the upper-left corner.

Subclassing your subclass

Wine is great and all but let's be honest, it's not the *right* beverage for all occasions. Let's first modify **BLCViewController** so that it becomes easier for us to subclass it. First we'll need to move some of our interface information to the **.h** file:

BLCViewController.h

```
@interface BLCViewController : UIViewController

+ @property (weak, nonatomic) UITextField *beerPercentTextField;
+ @property (weak, nonatomic) UILabel *resultLabel;
+ @property (weak, nonatomic) UISlider *beerCountSlider;

+ - (void)buttonPressed:(UIButton *)sender;
@end
```

Then, delete those properties from the *secret* **interface** inside the **.m** file:

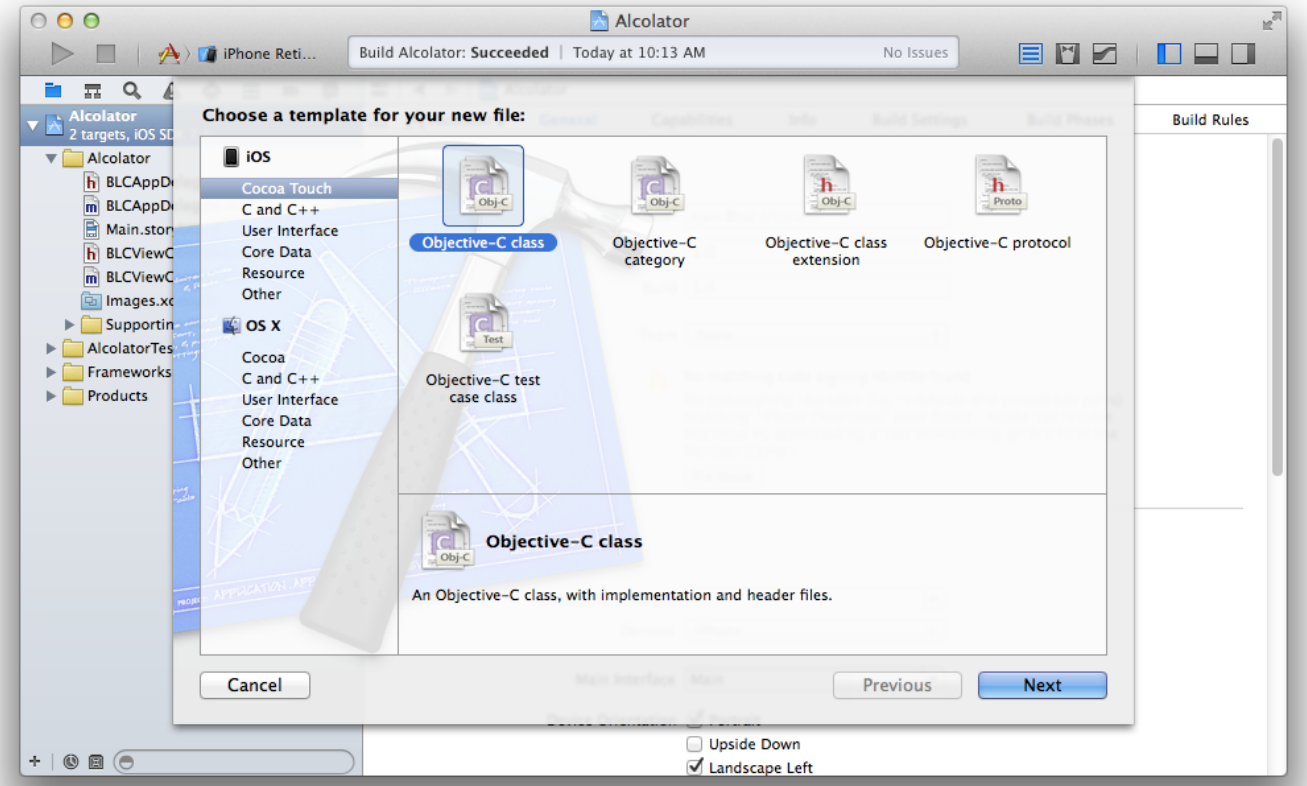
BLCViewController.m

```
 - @property (weak, nonatomic) UITextField *beerPercentTextField;
 - @property (weak, nonatomic) UILabel *resultLabel;
 - @property (weak, nonatomic) UISlider *beerCountSlider;
@property (weak, nonatomic) UIButton *calculateButton;
@property (weak, nonatomic) UITapGestureRecognizer *hideKeyboardTapGestureRecognizer;

@end
```

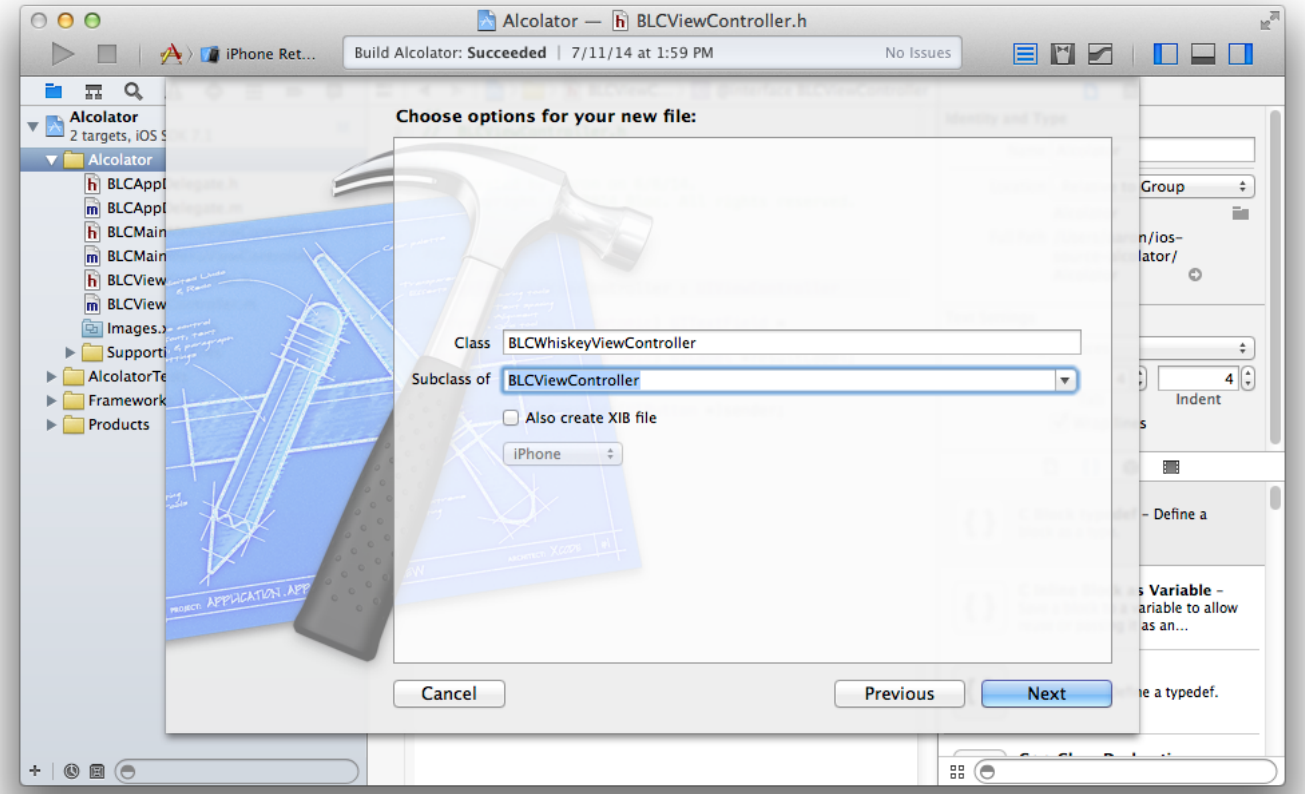
Why is this done? To reiterate what was stated earlier, only the original class is capable of accessing the properties and methods which are declared in its `.m` file inside the *secret* `interface`. In order to let outsiders (including subclasses) make use of these secret methods and/or properties, a class must make them *public* by announcing them in the `.h` file.

Create a new `BLCWhiskeyViewController` class by selecting the `Alcolator` folder in the Project Navigator window and pressing ⌘+N. Follow the on-screen guide.

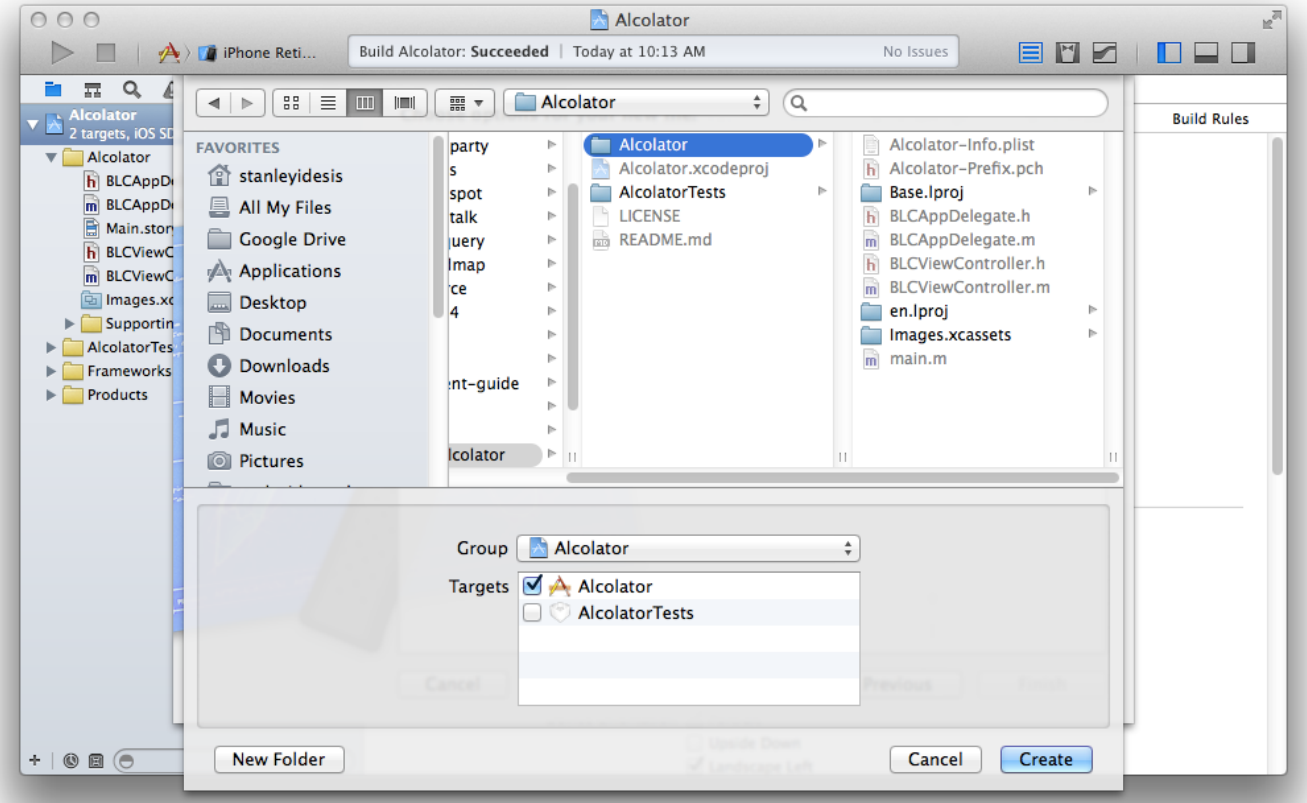
Highlight the `Objective-C Class` option under the `Cocoa Touch` section:



Press **Next** and then fill in the fields to match the screenshot below:



Press **Next** one last time and highlight the appropriate folder as shown below:



`@implementation BLCWhiskeyViewController` and **`@end`** and include the following:

BLCWhiskeyViewController.m

```
@implementation BLCWhiskeyViewController

// DELETE PRE-WRITTEN METHODS: initiWithNibName:bundle:, viewDidLoad, didReceiveMemoryWarning…

+ - (void)buttonPressed:(UIButton *)sender;
+ {
+     [self.beerPercentTextField resignFirstResponder];

+     int numberOfBeers = self.beerCountSlider.value;
+     int ouncesInOneBeerGlass = 12;  //assume they are 12oz beer bottles

+     float alcoholPercentageOfBeer = [self.beerPercentTextField.text floatValue] / 100;
+     float ouncesOfAlcoholPerBeer = ouncesInOneBeerGlass * alcoholPercentageOfBeer;
+     float ouncesOfAlcoholTotal = ouncesOfAlcoholPerBeer * numberOfBeers;

+     float ouncesInOneWhiskeyGlass = 1;  // a 1oz shot
+     float alcoholPercentageOfWhiskey = 0.4;  // 40% is average

+     float ouncesOfAlcoholPerWhiskeyGlass = ouncesInOneWhiskeyGlass * alcoholPercentageOfWhiskey;
+     float numberOfWhiskeyGlassesForEquivalentAlcoholAmount = ouncesOfAlcoholTotal / ouncesOfAlcoholPerWhiskeyGlass;

+     NSString *beerText;

+     if (numberOfBeers == 1) {
+         beerText = NSLocalizedString(@"beer", @"singular beer");
+     } else {
+         beerText = NSLocalizedString(@"beers", @"plural of beer");
+     }

+     NSString *whiskeyText;

+     if (numberOfWhiskeyGlassesForEquivalentAlcoholAmount == 1) {
+         whiskeyText = NSLocalizedString(@"shot", @"singular shot");
+     } else {
+         whiskeyText = NSLocalizedString(@"shots", @"plural of shot");
+     }

+     NSString *resultText = [NSString stringWithFormat:NSLocalizedString(@"%d %@ contains as much alcohol as %.1f %@ of whiskey.", n
+     self.resultLabel.text = resultText;
+ }
@end
```

This class exemplifies the purpose of subclassing; it reused 90% of **`BLCViewController`** and only modified it slightly to achieve an entirely nev result. As you can see from the method above, it references **`beerPercentTextField`**, **`resultLabel`** and **`beerCountSlider`** which is why those three properties had to be made public by declaring them in **`BLCViewController.h`**. Furthermore, the only method from **`BLCViewController`** that we needed to override was **`buttonPressed:`** and therefore it too had to be made public.

Roadmap · **Previous Checkpoint** **Jump to Submission** · **Next Checkpoir**

Your assignment | **Ask a question** | **Submit your work**

Our current app has some issues:

1. Rotating the iPhone into landscape doesn't resize the controls properly.
2. Running the app in the iPad, iPhone 6 or iPhone 6 Plus simulators don't size the controls properly.
3. You can't see the text field (it's transparent).
4. The UI is generally pretty ugly (it needs better fonts and colors).

your mentor for review when you finish or get stuck.

Some hints:

- Solutions to assignments #1 and #2 only require minor changes within your implementation of `viewWillLayoutSubviews`, which is always called when a view is rotated.
- You'll need to review the documentation to learn how to change the fonts and colors.

Additionally, the code in this checkpoint uses `NSLocalizedString`. Look up what that does, and send a message to your mentor describing in your own words what it does and when you should use it.

Finally, discuss the process of subclassing, and `UIViewController`, with your mentor. `UIViewController` is one of the most important classes in iOS development, so make sure you have a firm grasp of how it's used.

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'Wrote a custom view controller'
$ git checkout master
$ git merge custom-view-controller
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

assignment completed

COURSES

Full Stack Web Development
Frontend Web Development
UX Design
Android Development
iOS Development

ABOUT BLOC

Our Team | Jobs
Bloc Veterans Program
Employer Sponsored
FAQ
Blog
Engineering Blog
Refer-a-Friend
Privacy Policy
Terms of Service

Tech Talks & Resources

Programming Bootcamp Comparison

Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css

Swiftris: Build Your First iOS Game with Swift

Webflow Tutorial: Design Responsive Sites with Webflow

Ruby Warrior

Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC