What are Objective-C Blocks?

=> Note: Most of the code examples can be found in the CustomURL Project in the test target. Please play around with this code yourself.
=> Blocks are chunks of code, but unlike functions (in Objc) they’re first class citizens.
=> That means blocks can be assigned to variables, added to collections (NSArray, NSDictionary, NSSet, etc.), passed to methods, and returned from methods.
=> They are sometimes called *closures* or *lambdas* in other languages.
=> Added to C, Objc and C++ by Apple
=> Blocks can capture values within their surrounding scope (no need to explicitly pass values into them if the value you need is in the surrounding scope).
=> They can used in place of delegate callbacks in many places. When should you use delegates instead?
=> They’re everywhere in iOS! So, you have to know them.
=> Just like Swift, Objc blocks are essentially unnamed blocks of code that can be passed around, assigned to variables, returned from functions and treated as objects.

Declaring Blocks:

Unnamed literal with no return type, no params.

```
1 ^{
2    NSLog(@"This is a block in Objc");
3 };
```

```
_ = {
    print("This is a closures in Swift")
}
```

Declare a variable to keep track of a block

```
// separating declaration from assignment
void (^simpleBlock)(void);

simpleBlock = ^{
    NSLog(@"This is a block on Objc");
};
```

```
let simpleBlock:()->()!
simpleBlock = {
    print("Simple Closure in Swift")
}

```

Combining Declaration & Assignment:

```
void(^simpleBlock)(void) = ^{
   NSLog(@"This is a block");
};

```

```
let simpleBlock:()->() = {
    print("Simple Closure in Swift")
}

```

Invoke block just like a C function:

```
simpleBlock();
```

Adding Blocks To Array & Invoking in For Loop:

```

NSArray *arr = @[^{NSLog(@"Yo");}
,^{NSLog(@"Mo");}];

for (void(^item)(void) in arr) {
    item();
}

```

```

let arr = [{print("Yo")}, {print("Light")},
{print("House")}]
for a in arr {
    a()
}

```

Type Declaration with 2 parameters & a return value:

```
double (^multiplyTwoValues)(double, double);
```

```
var multiplyTwoValues:(Double, Double)->Double
```

Same block as an unnamed literal:

```

multiplyTwoValues = ^(double num1, double num2) {
    return num1 * num2;
};

```

```

```

```
multiplyTwoValues = {
    (d1:Double, d2:Double)->Double in
    return d1 * d2
}

```

Compiler can infer the return type of the literal block expression in Objc:

```

multiplyTwoValues = ^double (double n1, double n2) {
    return n1 * n2;
};


multiplyTwoValues = ^ (double n1, double n2) {
    return n1 * n2;
};

```

Compiler can infer the types of the parameters, return type and return expression in Swift:

```
multiplyTwoValues = {
    (d1, d2) in
    d1 * d2
}
```

```

multiplyTwoValues = {
    $0 * $1
}

```

Invoking block with 2 params in Objc:

```

multiplyTwoValues(2, 4);

// same in Swift
multiplyTwoValues(2, 4)

// if the definition of the Swift closure includes parameter names then you must use them in the call

var multiplyTwoValues:(d1:Double, d2:Double)->Double

multiplyTwoValues = //...

multiplyTwoValues(d1:10, d2:10)


```

Capturing Values:

```

void testMethod(){
    int i = 24;
    void (^doStuff)(void) = ^ {
        NSLog(@"%s %d", __PRETTY_FUNCTION__, i);
    };
    doStuff();
}

testMethod(); //=> 24

```

```

//Swift
func testMethod() {
```

```
    let i = 20
    let doStuff = {
        print(#line, i)
    }
    doStuff()
}

testMethod()

```

Same Function With Annoymous Block

```
void testMethod(){
    int i = 24;
    ^ {
        NSLog(@"%s %d", __PRETTY_FUNCTION__, i);
    }();
}

testMethod(); //=> 24
```

```

//Swift
func testMethod() {
    let i = 20
    {
        print(#line, i)
    }()
}

testMethod()

```

Objc Blocks capture by value not reference:

```
1
2 - (void)testObjcBlocksCaptureByValue {
3     int anInt = 42;
4     int (^testBlock)(void) = ^{
5         return anInt;
6     };
7
8     anInt = 84;
9
10    int result = testBlock();
11    int expected = 42;
12    XCTAssertEqual(result, expected, @"result should
  be 42");
13 }
14
```

Swift In Contrast Captures By Reference

```
1
2 func testSwiftBlocksCaptureByReference() {
3     var anInt = 42
4     let testBlock = {
5         return anInt;
6     }
7
8     anInt = 84;
9
10    let result = testBlock()
11    let expected = 84
12    XCTAssertEqual(result, expected)
13 }
14
```

Capture By Reference using __block:

```
// shared storage using __block

- (void)testObjcBlockCaptureByReference {
    __block int anInt = 42;

    int (^testBlock)(void) = ^ {
        return anInt;
    };

    anInt = 84;

    int result = testBlock();
    int expected = 84;
    XCTAssertEqual(result, expected);
}

```

Mutating captured value using __block:

```

- (void)testObjcBlockMutateCapturedValue {
    __block int anInt = 42;

    void (^testBlock)(void) = ^{
        NSLog(@"integer is: %i", anInt); //=> 42
        anInt = 100;
    };

    XCTAssertEqual(anInt, 42);
    testBlock();
    XCTAssertEqual(anInt, 100);

```

```
}

```

Since Swift Captures By Reference By Default We Need To Show Capture By Value

```

func testSwiftBlocksCaptureByValue() {

    var anInt = 42;

    let testBlock = {[anInt]()->Int in
        return anInt
    }
    anInt = 200
    XCTAssertEqual(anInt, 200);
    let result = testBlock();
    XCTAssertEqual(result, 42);
}

```

Passing blocks as arguments to methods:

=> Common to pass a block to be invoked later (e.g. invoking a block after a network request returns).
=> Common to use blocks when a task is completed. (e.g. a user fills in a form and you want to handle passing data back to another object on completion)
=> You could use delegation.
=> Question: When should you use delegation over blocks and vice versa? (Common interview question BTW).

```
1  // XYZWebTask.h
2  @interface XYZWebTask : NSObject
3  - (void)beginTaskWithCallbackBlock:(void(^)(NSString
   *))block;
4  @end
5
6  // XYZWebTask.m
7  @implementation XYZWebTask
8  - (void)beginTaskWithCallbackBlock:(void (^)
   (NSString *))block {
9      NSString *result = @"some result gotten from
   network";
10     block(result);
11 }
12 @end
13
14 - (void)testBlockCallback {
15     [self showProgressIndicator:YES];
16     XYZWebTask *task = [XYZWebTask new];
17     __block NSString *result;
18     [task beginTaskWithCallbackBlock:^(NSString
   *str) {
19         result = str;
20         [self showProgressIndicator:NO];
21     }];
22     XCTAssertTrue([result isEqualToString:@"some
   result gotten from network"]);
23 }
24
25 - (void)showProgressIndicator:(BOOL)indicator {}
26
27
```

```swift

// Same thing in Swift

 private class XYZWebTask {
    func beginTaskWithCallbackBlock(block:
(str:String)->()) {
        let result = "some result gotten from
network"
        block(str: result)
    }
}

func testBlockCallback() {
    showProgressIndicator(true)
    let task = XYZWebTask()
    var result:String!
    task.beginTaskWithCallbackBlock { (str) in
        result = str
        self.showProgressIndicator(false)
    }
    XCTAssertTrue(result == "some result gotten from
network")
}

private func showProgressIndicator(indicator:Bool)
{}


```

Note: Blocks should be last argument in a method:

Create **typedef's** to make blocks easier to read:

```objc

typedef void (^XYZSimpleBlock)(void); // void return void paramater is given type XYZSimpleBlock

XYZSimpleBlock anotherBlock = ^{
    NSLog(@"not much to see here");
};

- (void)beginFetchWithCallbackBlock:(XYZSimpleBlock)callbackBlock {
    // do long running operation
    callbackBlock();
}

```

```swift

// Similar Swift construct

typealias XYZSimpleBlock = ()->(String)

let anotherBlock: XYZSimpleBlock = {
    return "Not much to see here"
}

func testTypeAliasBlock() {
    var result:String!
    func beginFetchWithCallbackBlock(callback: XYZSimpleBlock) {
        result = callback()
    }
    beginFetchWithCallbackBlock(anotherBlock)
    XCTAssertTrue(result == "Not much to see here")

```

```
}
```

Using properties that are blocks (making them copy is best practice):

```
@property (copy) void (^blockProperty)(void);

// assigning to block property

self.blockProperty = ^{
    NSLog(@"not much here!");
};

self.blockProperty();

// using a typedef with a block property

typedef void (^XYZSimpleBlock)(void);

@interface XYZObject : NSObject
@property (copy) XYZSimpleBlock blockProperty;
@end

```

```

// In Swift

var blockProperty:(()->(String))!
func testBlockProperty() {
    blockProperty = {
```

```
7           return "not much here!"
8       }
9       let result = blockProperty()
10      XCTAssertTrue(result == "not much here!")
11  }
12
13  typealias XYZSimpleBlock2 = ()->(String)
14  var blockProperty2:XYZSimpleBlock!
15
16  func testBlockPropertyWithTypeAlias() {
17      blockProperty2 = {
18          return "not much here!"
19      }
20      let result = blockProperty2()
21      XCTAssertTrue(result == "not much here!")
22  }
23
```

Avoiding strong reference cycles when capturing self:

=> Objc and Swift closure both are susceptible to retain cycles
=> I demonstrated this problem twice now in Swift in the debugging lecture and during the closures lecture.
=> If an object owns a block/closure, and that block/closure owns self, then we have a retain cycle.
=> Doesn’t matter whether it’s Swift or Objc.

```
1
2  // The compiler warns us here
3  @interface XYZBlockKeeper : NSObject
```

```
@property (copy) void (^block)(void);
@end

@implementation XYZBlockKeeper
- (void)configureBlock {
    self.block = ^{
        [self doSomething];    // capturing a strong reference to self
        // creates a strong reference cycle
    };
    self.block();
}

- (void)doSomething {}

```

Capturing Weak Self in Objc

```
- (void)configureBlockWithWeakSelf {
    __weak XYZBlockKeeper *welf = self;
    self.block = ^{
        [welf doSomething];   // capture the weak reference
        // to avoid the reference cycle
    };
    welf.block();
}
```

```

// Swift syntax with retain cycle

var block:(()->())!
```

```
5
6  private func configureBlock() {
7      self.block = {
8          self.doSomething()
9      }
10     self.block()
11 }
12
13 func testConfigureBlock() {
14     configureBlock()
15 }
16
17
```

```
1
2  // Swift weak self solution
3
4  private func configureBlockWithWeakSelf() {
5      self.block = {
6          [weak self] in
7          guard let welf = self else {
8              return
9          }
10         welf.doSomething()
11     }
12 }
13
14 func testConfigureBlockWithWeakSelf() {
15     configureBlockWithWeakSelf()
16 }
17
18
```

enumerateObjectUsingBlock:

```
1
2  - (void)testEnumerateObjectUsingBlock {
3      NSArray *array = @[@11, @33, @99, @34, @11,
   @88];
4      __block NSNumber *result;
5      [array enumerateObjectsUsingBlock:^(id  _Nonnull
   obj, NSUInteger idx, BOOL * _Nonnull stop) {
6          if ([(NSNumber *)obj isEqualToNumber:@11]) {
7              result = obj;
8              *stop = YES;
9          }
10     }];
11     XCTAssert([result isEqualToNumber:@11]);
12 }
13
14 // concurrent enumeration
15
16 enumerateObjectsWithOptions:usingBlock: can also do
   concurrent enumeration with
17
18 [array
   enumerateObjectsWithOptions:NSEnumerationConcurrent
19                         usingBlock:^ (id obj,
   NSUInteger idx, BOOL *stop) {
20     ...
21 }];
22
23 // this could improve performance but the order of
   enumeration is not defined
24
```

```
// You can loop through a dict using

- (void)testEnumerateDictionaryUsingBlock {
    NSDictionary *dictionary = @{@1: @"one", @2: @"two", @3: @"three"};
    __block NSString *result;
    [dictionary enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL *stop) {
        if ([key isEqualToNumber:@3]) {
            result = dictionary[key];
        }
    }];

    XCTAssert([result isEqualToString:@"three"]);

}

```

Helpful Block Snippets:

```
// http://cocoawithlove.com/2009/10/ugly-side-of-blocks-explicit.html has a nice breakdown of the syntax--it helps to think of the ^ as similar to a pointer dereference symbol *

// block typedef:

    typedef void(^Block)(void);
    typedef void(^ConditionalBlock)(BOOL);
    typedef NSString*(^BlockThatReturnsString)(void);
    typedef NSString*(^ConditionalBlockThatReturnsString)(BOOL);
```

```
 9
10 // block property with typedef:
11
12     @property(copy)Block block;
13     @property(copy)ConditionalBlock
   conditionalBlock;
14     @property(copy)BlockThatReturnsString
   blockThatReturnsString;
15     @property(copy)ConditionalBlockThatReturnsString
   conditionalBlockThatReturnsString;
16
17 // block property without typedef:
18
19     @property(copy)void(^block)(void);
20     @property(copy)void(^conditionalBlock)(BOOL);
21     @property(copy)NSString*
   (^blockThatReturnsString)(void);
22     @property(copy)NSString*
   (^conditionalBlockThatReturnsString)(BOOL);
23
24
25 // block definition inline:
26
27     ReturnType(^block_name)(parmeter, types, here) =
   ^ReturnType(parameter, types, here) {};
28
29     void(^block)(void) = ^{
30       NSLog(@"Yo");
31     };
32
33     void(^conditionalBlock)(BOOL shouldWork) =
   ^(BOOL shouldWork){
```

```
34          NSLog(@"%@", shouldWork? @"Works":
    @"Doesn't");
35      };
36
37      NSString*(^blockThatReturnsString)(void) = ^
    NSString* {
38          return @"that string"; // compiler can infer
    return type
39      };
40
41      NSString*(^conditionalBlockThatReturnsString)
    (BOOL shouldWork) = ^ NSString* (BOOL shouldWork){
42              if (shouldWork) {
43                  return @"Works"
44              }
45              return @"Doesn't Work";
46      };
47
48  // calling blocks:
49
50      block(); //=> "Yo"
51      conditionalBlock(NO); //=> "Doesn't"
52      NSString *someString = blockThatReturnsString();
    //=> "that string"
53      NSString *conditionalString =
    conditionalBlockThatReturnsString(NO); //=> "Doesn't
    Work"
54
55
56  // blocks as return values:
57
58      -(void (^)(void))doSomething;
```

```
59      -(void (^)(BOOL))doSomethingConditionally;
60      -(NSString* (^)(void))returnString;
61      -(NSString* (^)(BOOL))returnStringConditionally;
62
63      -(NSString* (^)(BOOL))returnStringConditionally
    {
64          return ^(BOOL shouldReturn) {
65              if (shouldReturn){
66              return @"Should return";
67              }
68              return @"Shouldn't return";
69          }
70      }
71
72  // blocks as arguments:
73
74      -(void)doSomethingWithBlock:(void (^)
    (void))block;
75      -(void)doSomethingWithBlock:(void (^)
    (BOOL))conditionalBlock;
76      -(void)doSomethingWithBlock:(NSString* (^)
    (void))blockThatReturnsString;
77      -(void)doSomethingWithBlock:(NSString* (^)
    (BOOL))conditionalBlockThatReturnsString;
78
79
```

https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithBlocks/WorkingwithBlocks.html