# Basic Exercises Part 9.5 Blocks

## Blocks

* Objective C Blocks are self contained structures that reduces the complexity of delegations. The asynchronous calls that return results in background can use blocks. Blocks make sure both the calling and result handling to be placed in the same place.
* An Objective-C class defines an object that combines data with related behavior. Sometimes, it makes sense just to represent a single task or unit of behavior, rather than a collection of methods.
* Blocks are a language-level feature added to C, Objective-C and C++ which allow you to create distinct segments of code that can be passed around to methods or functions as if they were values. Blocks are Objective-C objects which means they can be added to collections like NSArray or NSDictionary. They also have the ability to capture values from the enclosing scope, making them similar to closures or lambdas in other programming languages.
* Blocks are like closures. Or function pointers (or lambdas in other programming languages).
* The symbol for block is: ^
* Concurrency. Blocks are a natural gateway to Grand Central Dispatch (GCD) and being able to take advantage of parallelism, whether through multi-cores, multi-processors, or multi-threads. If it’s OK to iterate through the collection in parallel and out of order:

[myArray enumerateObjectsWithOptions:NSEnumerationConcurrent

usingBlock:^(id obj, NSUInteger idx, BOOL \*stop) {

// do something

}];

So there’s nothing about blocks themselves that speeds up your code. What they offer is a very easy way to segment bits of hard-working code and then lean on GCD to handle the dispatching.

* The practical usage and effect of putting the \_\_block keyword before the declaration of a variable (yes, with two underscores)
  + Blocks are created on the stack.
  + Blocks can be copied to the heap.
  + Blocks have their own private const copies of stack variables (and pointers).
  + Mutable stack variables and pointers must be declared with the “\_\_block” keyword.

If blocks aren’t kept around anywhere, will remain on the stack and will go away when their stack frame returns. While on the stack, a block has no effect on the storage or lifetime of anything it accesses. If blocks need to exist after the stack frame returns, they can be copied to the heap and this action is an explicit operation. This way, a block will gain reference-counting as all objects in Cocoa. When they are copied, they take their captured scope with them, retaining any objects they refer. If a block references a stack variable or pointer, then when the block is initialized it is given its own copy of that variable declared const, so assignments won't work. When a block is copied, the \_\_block stack variables it reference are copied to the heap and after the copy operation both block on the stack and brand new block on the heap refer to the variables on the heap.

* For more information visit:

https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Blocks/Articles/00\_Introduction.html

### **Block declaration syntax**

return\_type (^blockName)(argumentType);

// or

return\_type (^blockName)(parameterType1, parameterType2, ...) = ^returnType(argument1, argument2, ...) {...};

* Declare as a property:

@property (nonatomic, copy, nullability) returnType (^blockName)(parameterTypes);

* Declare as a method parameter:

- (void)someMethodThatTakesABlock:(returnType (^nullability)(parameterTypes))blockName;

* Declare as an argument to a method call:

[someObject someMethodThatTakesABlock:^returnType (parameters) {...}];

* Declare as a typedef:

typedef returnType (^TypeName)(parameterTypes);

TypeName blockName = ^returnType(parameters) {...};

* Declare a C function return a block object:

BLOCK\_RETURN\_TYPE (^function\_name(function parameters))(BLOCK\_PARAMETER\_TYPE);

### **1.2 A simple block implementation**

return\_type (^blockName)(argumentType)= ^{

};

### **1.3 Hello Block implementation**

void (^simpleBlock)(void) = ^{

NSLog(@"Hello block");

};

In this case, that block won’t actually do anything; it’s like declaring a function and not calling it. So, let’s try to calling it:

simpleBlock();

Block declarations look very much like function pointers, if you have dealt with those in C. In our example here:

* void is the return type
* ^ indicates it’s a block
* simpleBlock is the name of the variable
* the () means it doesn’t take any arguments; we could also write (void)
* note that blocks are C statements, so the closing brace is followed by a semicolon.

### **1.4 A simple block with no return parameter**

// Block declaration

void (^block)(void);

// Block definition

        block = ^void(void){

            NSLog(@"Perform task");

        };

 // Calling block

block();

### **1.5 Sorting arrays**

The most flexible ways to sort an array is with the sortedArrayUsingComparator: method. This accepts an **^NSComparisonResult(id obj1, id obj2) block**.

Return Value Description

NSOrderedAscending obj1 comes before obj2

NSOrderedSame obj1 and obj2 have no order

NSOrderedDescending obj1 comes after obj2

Example:

- (**void**)viewDidLoad {

[**super** viewDidLoad];

NSArray \*categoryArray = @[@"Apps", @"Music", @"Songs",

@"iTunes", @"Books", @"Videos"];

NSArray \*sortedArray = [categoryArray sortedArrayUsingComparator:

^NSComparisonResult(**id** obj1, **id** obj2) {

**if** ([obj1 length] < [obj2 length]) {

**return** NSOrderedAscending;

} **else** **if** ([obj1 length] > [obj2 length]) {

**return** NSOrderedDescending;

} **else** {

**return** NSOrderedSame;

}

}];

NSLog(@"%@", sortedArray);

}

### **1.6 Filter Arrays**

NSMutableArray \*array = [NSMutableArray arrayWithObjects:@"Ken", @"Tim", @"Chris", @"Steve",@"Charlie",@"Melissa", **nil**];

NSPredicate \*bPredicate = [NSPredicate predicateWithFormat:@"SELF beginswith[K] 'K'"];

NSArray \*beginWithB = [array filteredArrayUsingPredicate:bPredicate];

// beginWith "K" is { @"Ken" }.

NSLog(@"%@", beginWithB);

NSPredicate \*sPredicate = [NSPredicate predicateWithFormat:@"SELF contains[c] 's'"];

[array filterUsingPredicate:sPredicate];

// array now contains { @"Chris", @"Steve", @"Melissa" }

NSLog(@"%@", array);

### **1.7 Blocks as properties**

@interface MyObject : MySuperclass

@property (copy) void (^blockProperty)(NSString \*string);

@end

When assigning, since self retains blockProperty, block should not contain a strong reference to self. Those mutual strong references are called a “retain cycle” and will prevent the release of either object.

\_\_weak \_\_typeof(self) weakSelf = self;

self.blockProperty = ^(NSString \*string) {

// refer only to weakSelf here. self will cause a retain cycle

};

It is highly unlikely, but self might be deallocated inside the block, somewhere during the execution. In this case weakSelf becomes nil and all messages to it have no desired effect. This might leave the app in an unknown state. This can be avoided by retaining weakSelf with a \_\_strong ivar during block execution and clean up afterward.

\_\_weak \_\_typeof(self) weakSelf = self;

self.blockProperty = ^(NSString \*string) {

\_\_strong \_\_typeof(weakSelf) strongSelf = weakSelf;

// refer only to strongSelf here.

// ...

// At the end of execution, clean up the reference

strongSelf = nil;

};

### **1.8 Blocks as local variables**

returnType (^blockName)(parameterType1, parameterType2, ...) = ^returnType(argument1, argument2, ...) {...};

float (^square)(float) = ^(float x) {return x\*x;};

square(5); // resolves to 25

square(-7); // resolves to 49

### **1.7 Defining and Assigning**

A block that performs addition of two double precision numbers, assigned to variable addition:

double (^addition)(double, double) = ^double(double first, double second){

return first + second;

};

The block can be subsequently called like so:

double result = addition(1.0, 2.0); // result == 3.0

### **1.6 Exercises**

1. Write a program that looks for a name in an array of names and reports back the name's index in the array:
   * Create an NSArray with some names (use NSString objects).
   * Create a local NSString variable to hold the name you are searching for and an integer block variable to report back at what index the name was found.
   * Search the array using -enumerateObjectsUsingBlock:.
   * Make sure your block uses the stop argument to stop looking when the name is found.
   * If the name you are looking for isn't in the array, the block variable holding the index should have a value of -1.
2. Write the declaration of a function called “weird” that takes as its only parameter a block with no parameters and returns no value, and returns a block of the same type.
3. Let's write a function that takes on options integer representing different types of binary operations (like addition, subtraction, multiplication, etc.) and returns a "calculator block" that you can apply to your operands in order to get the result of that operation.

By a binary operation, we simply mean an operation that operates on two values, known as the "operands".

Suppose we're dealing with double operands. Our calculations also return a double. What's the type of our binary operation block?

1. Define a function that takes an array of integers, an integer representing the size of the array, and a block that takes an int and returns an int. The job of the function will be to apply the block (which we imagine represents a mathematical formula) on each value in the array.

We want the type of our block to be int(^)(int). Let’s typedef and then define our function:

typedef int(^iblock\_t)(int);

void func(int arr[], int size, iblock\_t formula)

{

   for ( int i = 0; i < size; i++ )

   {

      arr[i] = formula(arr[i]);

   }

}

Let's use this in a program.