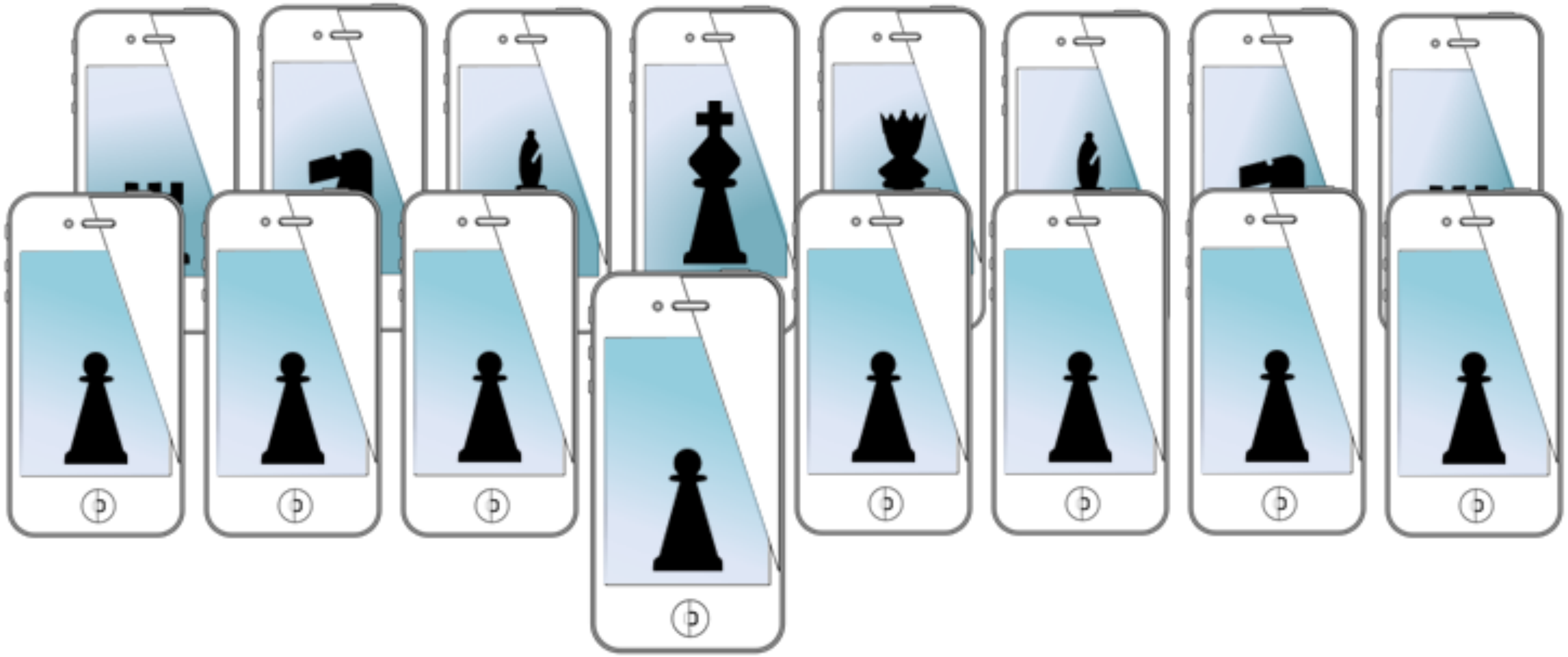
MOBILE SENSING & LEARNING



CS5323 & 7323

Mobile Sensing & Learning

UI elements

Eric C. Larson, Lyle School of Engineering,
Department of Computer Science, Southern Methodist University

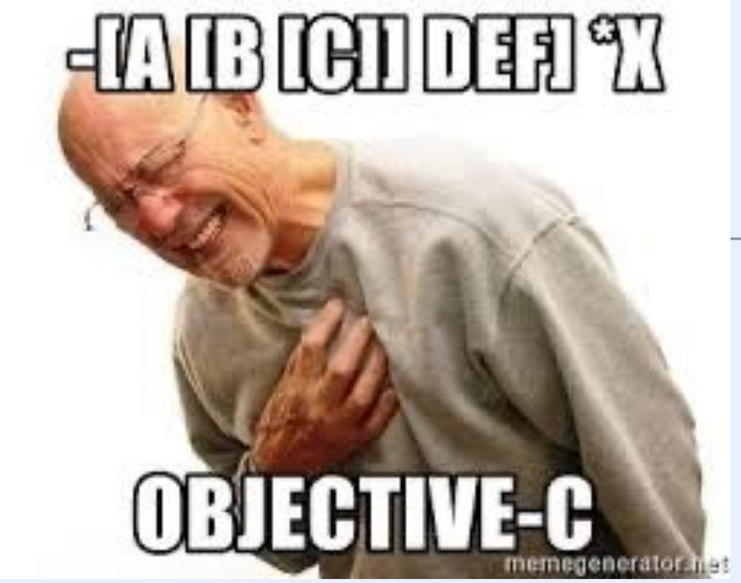
course logistics

- reminder: university developer program!
- next Time: flipped assignment, in person
- a1 due at the **end of next week**
    - make a video of the app and submit it (YouTube, dropbox, direct upload to canvas, etc.)
    - use quicktime for video (if you don't know what to use)

agenda

- syntax review
- blocks and concurrency
- target action behavior
  - and constraints
- text fields
- gesture recognizers
- timers / segmented control
- **remainder of time**: demo!

objective c



- strict superset of c
  - but with "messages"
    - so "functions" look very different (i.e., the braces in the logo)

swift

- syntax is nothing like objective-c
- but uses the same libraries…
- similarities with python syntax
  - weakly typed, no need for semicolons

functions examples

return type

method name

parameter type

parameter name

```
-(NSNumber*) addOneToNumber:(NSNumber *)myNumber {}
```

```
-(NSNumber*)     addToNumber:(NSNumber *)myNumber
            withOtherNumber:(NSNumber *)anotherNumber
```

second parameter

receiver class

parameter name/va

```
NSNumber *obj = [self addOneToNumber:@4];
NSNumber *obj = [self addToNumber:@4 withOtherNumber:@67];
```

(+ —) instance versus class method

```
[[NSNumber alloc] init]
```

throwback to **c**

```
float addOneToNumber(float myNum){
    return myNum++;
}

float val = addOneToNumber(3.0);
```

```
func addOneToNumber(myNumber:Float) -> (Float){
    return myNumber+1
}

func addOneToNumber(myNum:Float, withOtherNumber myNum2:Float) -> (Float){
    return myNum+myNum2+1
}

var obj = self.addOneToNumber(myNumber: 3.0)
var obj = self.addOneToNumber(myNum: 3.0, withOtherNumber: 67)
```

```
(varName:Type) -> (Return Type)
```

similar named second parameter syntax in swift

common logging functions

function

NSString to format

object to print

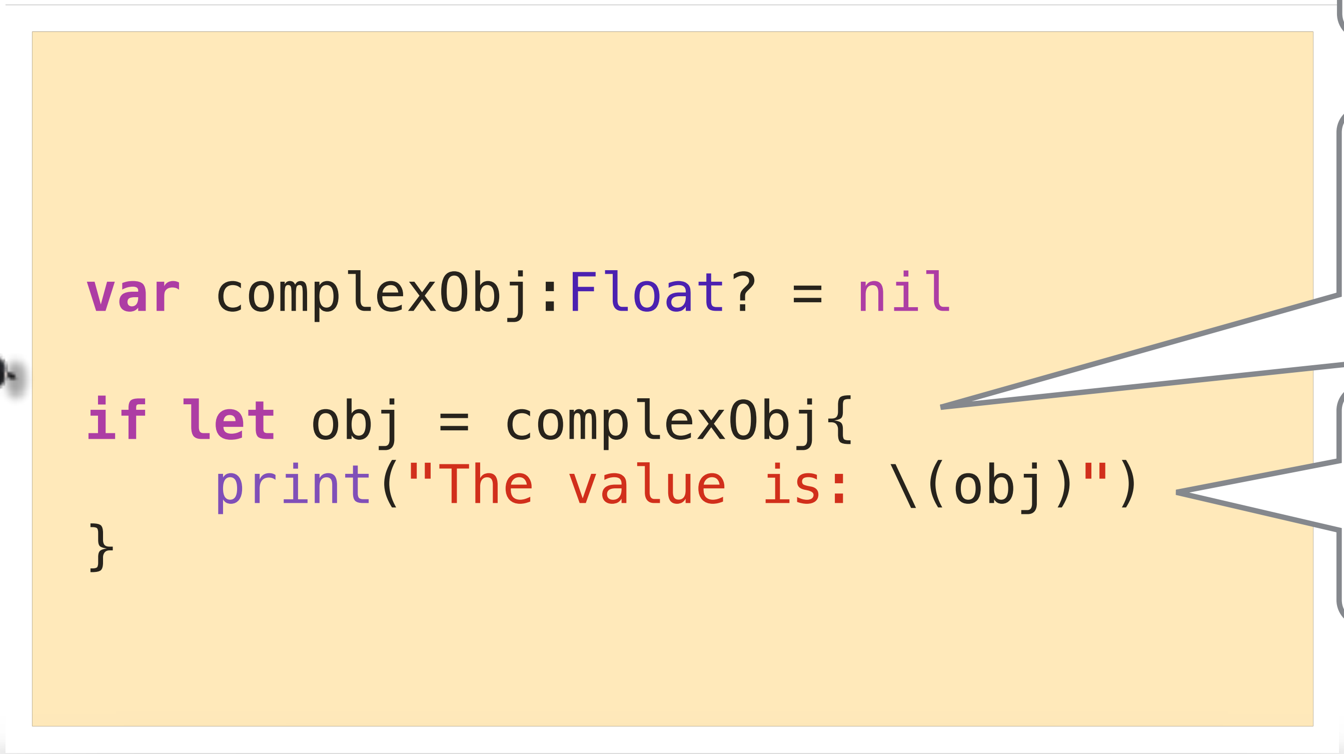
```
NSLog(@"The value is: %@",someComplexObject);
NSLog(@"The value is: %d",someInt);
NSLog(@"The value is: %.2f”,someFloatOrDouble);
```

%@ is print for serializable objects

```
 someComplexObject = nil;

 if(!someComplexObject)
     printf("Wow, printf works!");
```

set to nothing,
subtract from reference count

nil only works for objects!
no primitives, structs, or enums

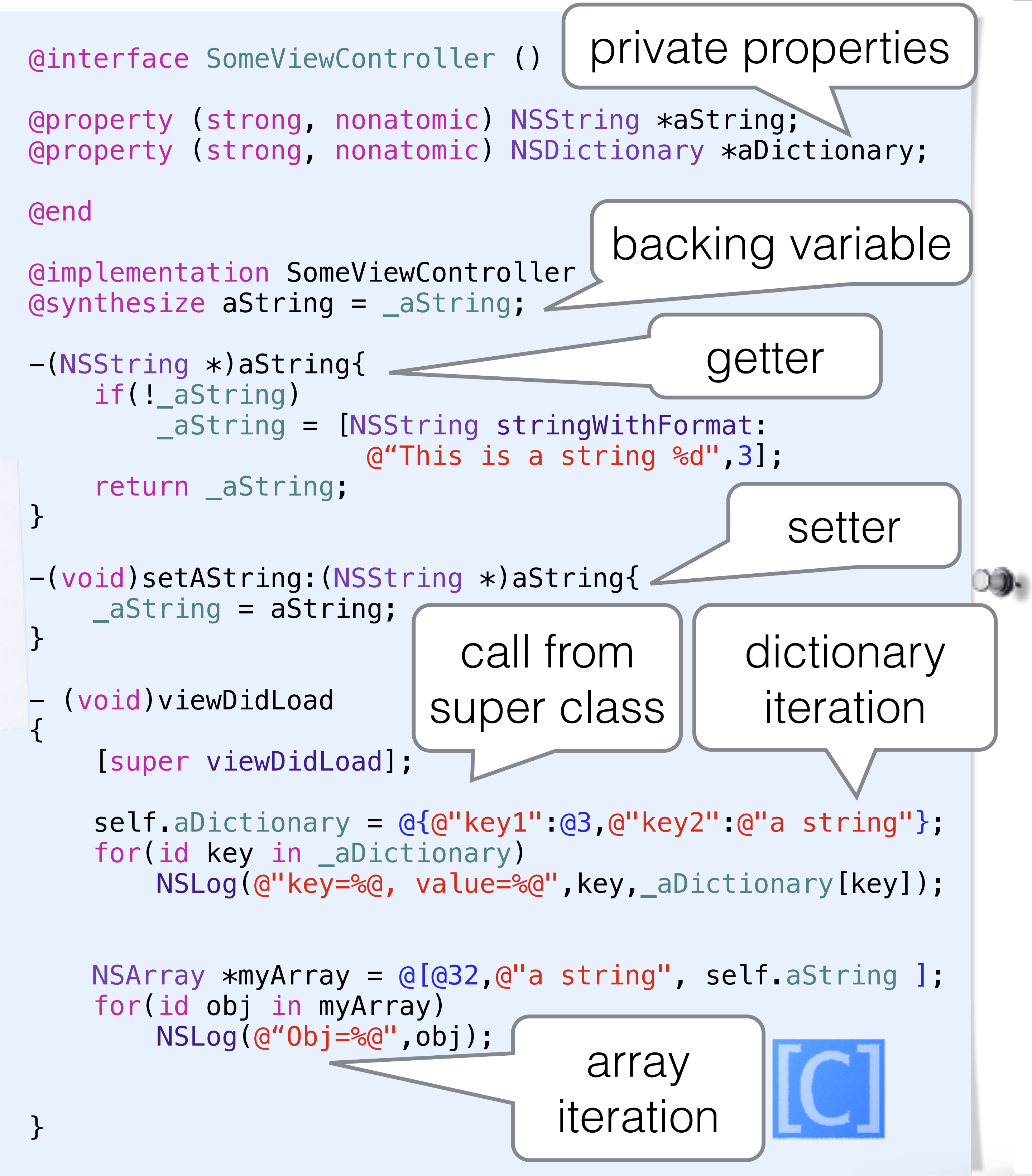


```
var complexObj:Float? = nil

if let obj = complexObj{
    print("The value is: \(obj)")
}
```

`if let` syntax, **safely unwraps** optional

print variable within string using
`\( varName )`

review



```
@interface SomeViewController ()

@property (strong, nonatomic) NSString *aString;
@property (strong, nonatomic) NSDictionary *aDictionary;

@end

@implementation SomeViewController
@synthesize aString = _aString;

-(NSString *)aString{
    if(!_aString)
        _aString = [NSString stringWithFormat:
                        @“This is a string %d",3];
    return _aString;
}

-(void)setAString:(NSString *)aString{
    _aString = aString;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.aDictionary = @{@"key1":@3,@"key2":@"a string"};
    for(id key in _aDictionary)
        NSLog(@"key=%@, value=%@",key,_aDictionary[key]);


    NSArray *myArray = @[@32,@"a string", self.aString ];
    for(id obj in myArray)
        NSLog(@“Obj=%@",obj);


}
```

private properties

backing variable

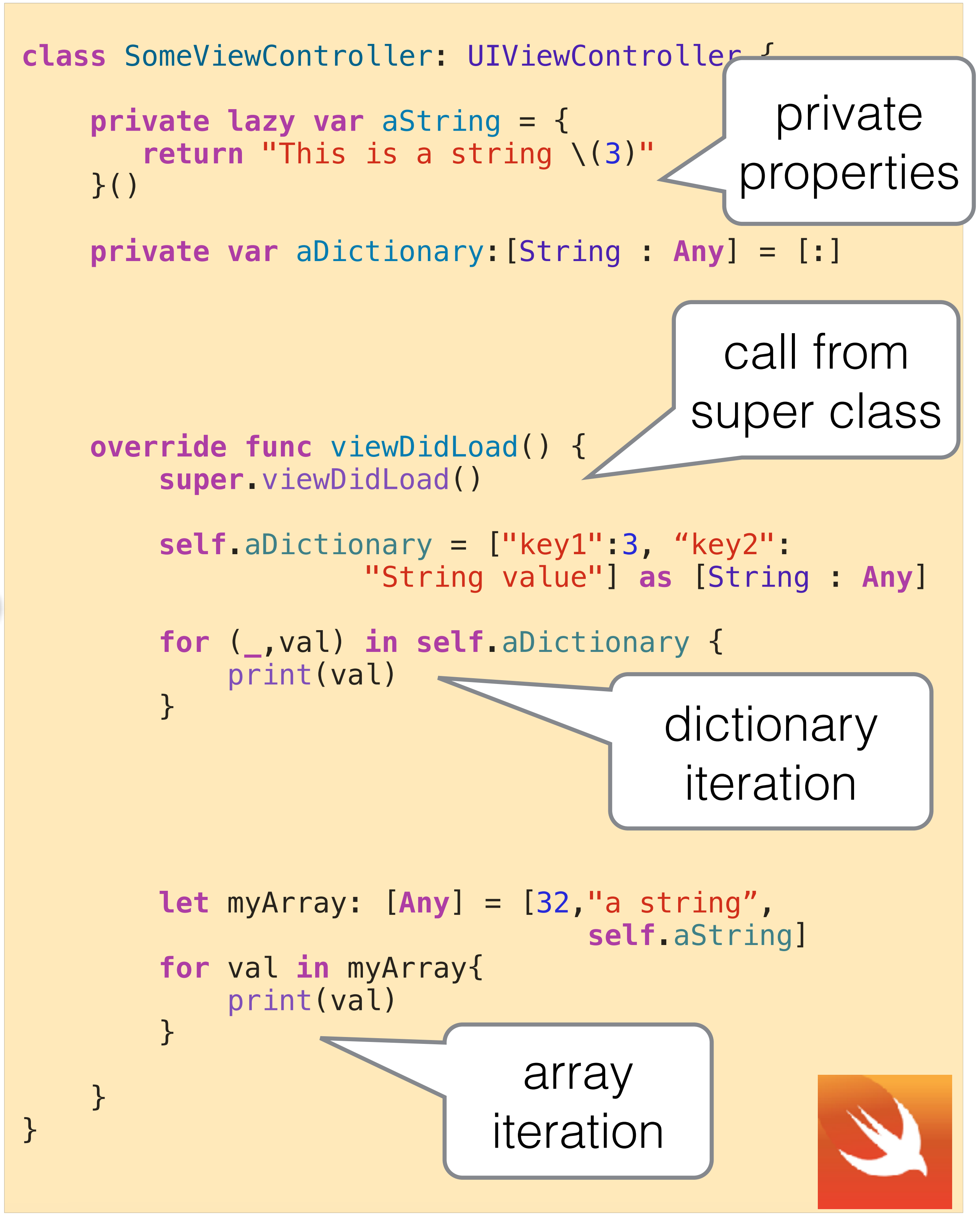
getter

setter

call from super class

dictionary iteration

array iteration



```
class SomeViewController: UIViewController {

    private lazy var aString = {
        return "This is a string \(3)"
    }()

    private var aDictionary:[String : Any] = [:]



    override func viewDidLoad() {
        super.viewDidLoad()

        self.aDictionary = ["key1":3, “key2":
                      "String value"] as [String : Any]

        for (_,val) in self.aDictionary {
            print(val)
        }


        let myArray: [Any] = [32,"a string”,
                               self.aString]
        for val in myArray{
            print(val)
        }

    }
}
```

private properties

call from super class

dictionary iteration

array iteration

adding to our project

- let’s add to our project
  - an objective-c class
  - that uses lazy instantiation

and now its time for a demo

blocks and closures

- not callback functions (but similar)
  - created at runtime
  - once created, can be called multiple times
  - can access data from scope when defined
  - syntax is different in swift and objective-c (also slightly different behavior)
- not exactly a lambda (*but similar*)
  - but it acts like an object that can be passed as an argument or created on the fly
- swift uses closures, objective-c uses blocks

block/closure syntax

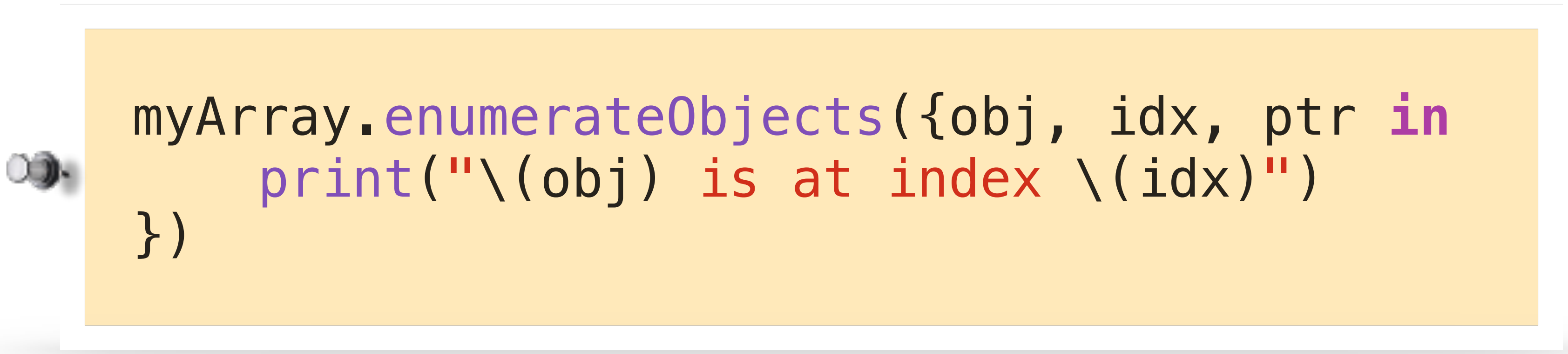
most common usage is as input into a function

```
^(Parameters) {
   // code
}
```

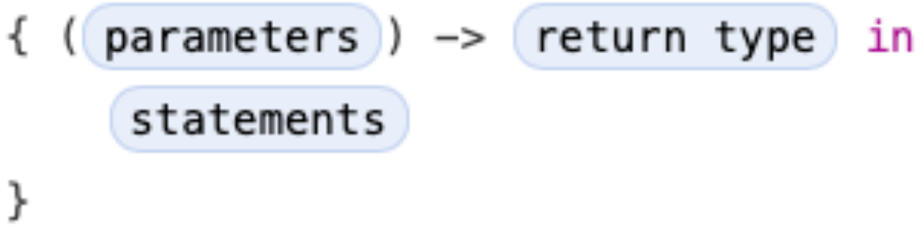
enumerate with block

```
// here the block is created on the fly for the enumeration
[myArray enumerateObjectsUsingBlock:^(NSNumber *obj, NSUInteger idx, BOOL *stop) {
    // print the value of the NSNumber in a variety of ways
    NSLog(@"Float Value = %.2f, Int Value = %d",[obj floatValue],[obj integerValue]);
}];
```

swift syntax



```
myArray.enumerateObjects({obj, idx, ptr in
    print("\(obj) is at index \(idx)")
})
```



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

some semantics

- variables from same scope where block is defined are **read only**

```
NSNumber * valForBlock = @5.0;
```

- Unless you use keyword:

```
__block NSNumber * valForBlock = @5.0;
```

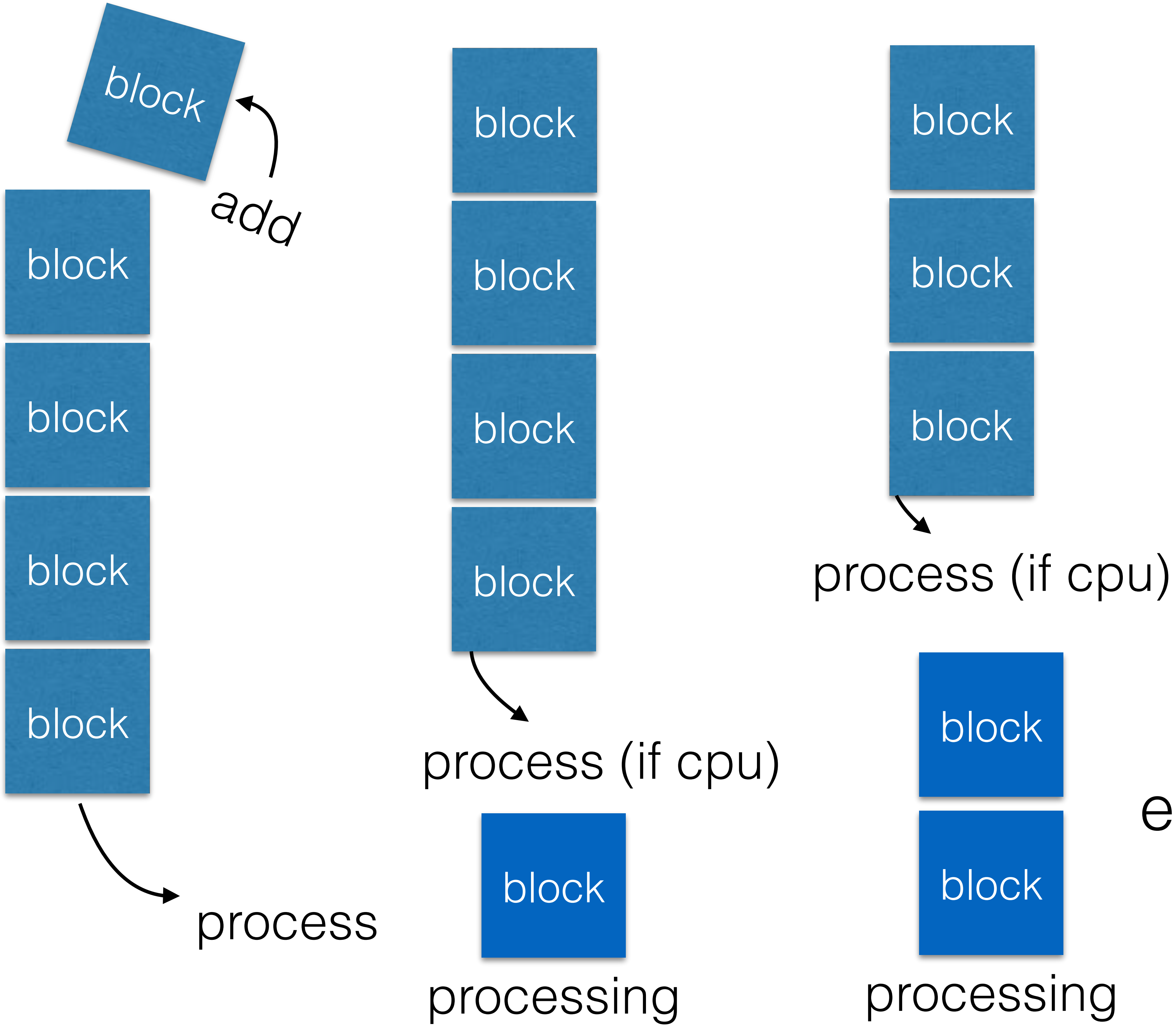
- classes hold a **strong** pointer to blocks they use
- blocks hold a **strong** pointer to `__block` variables
    - so using “self” would create a retain cycle

```
 self.value = (some function in block)
 __block ViewController * __weak  weakSelf = self;
weakSelf.value = (some function in block)
```

concurrency in iOS

- grand central dispatch (GCD) handles all operations
  - GCD looks at “queues” of **blocks** that need to be run
  - GCD and the Xcode compiler work deep inside the OS, actually in the kernel — they are optimized
  - for a **serial queue** each block is run sequentially
  - for **concurrent queues** the first block is dequeued
    - if CPU is available, then the next block is also dequeued, but could finish any time
- the **main queue handles all UI operations** (and no other queue should generate UI changes!!)
  - so, **no updating of** the views, labels, buttons, (image views*) **except from the main queue**

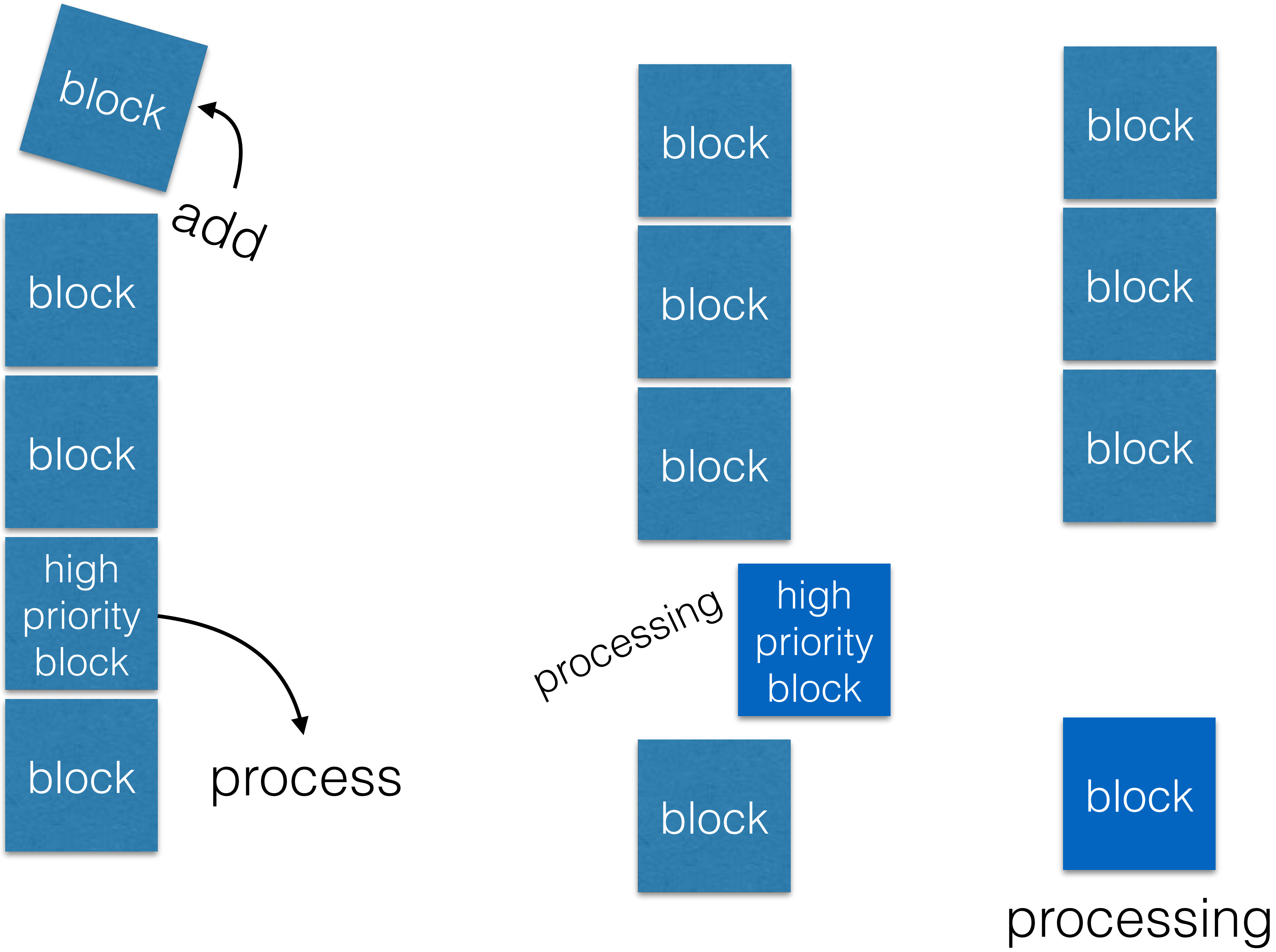
concurrent queues



threads?

either could finish

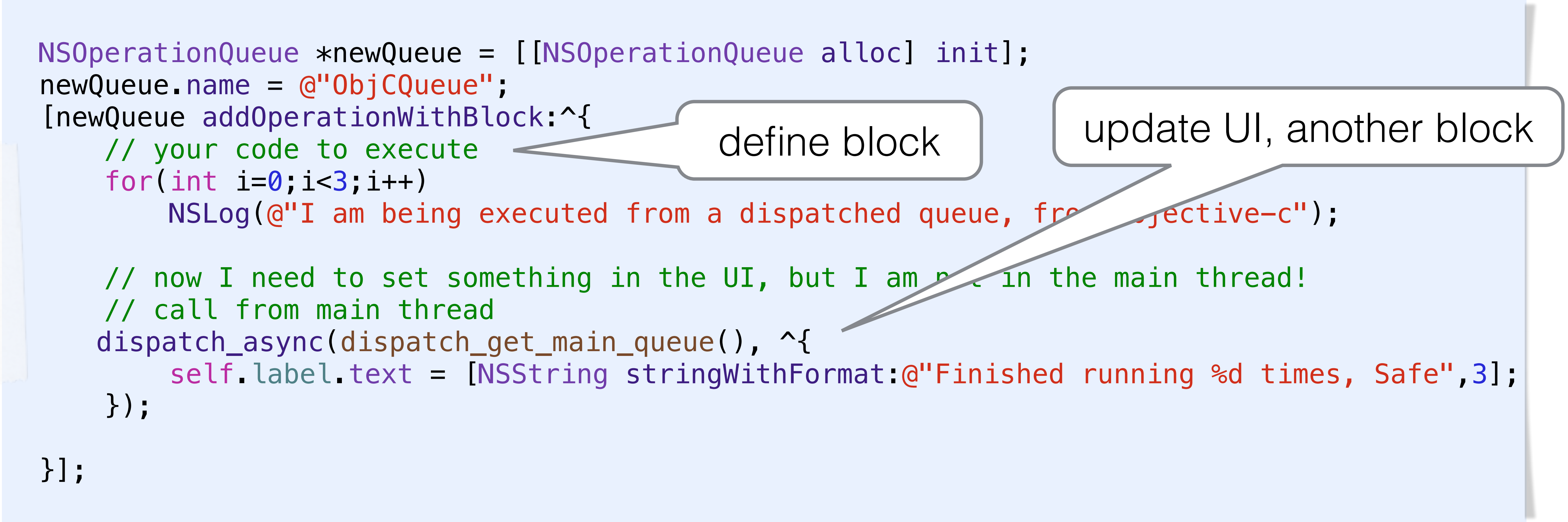
the main queue



one thread
UI is in sync

queue syntax

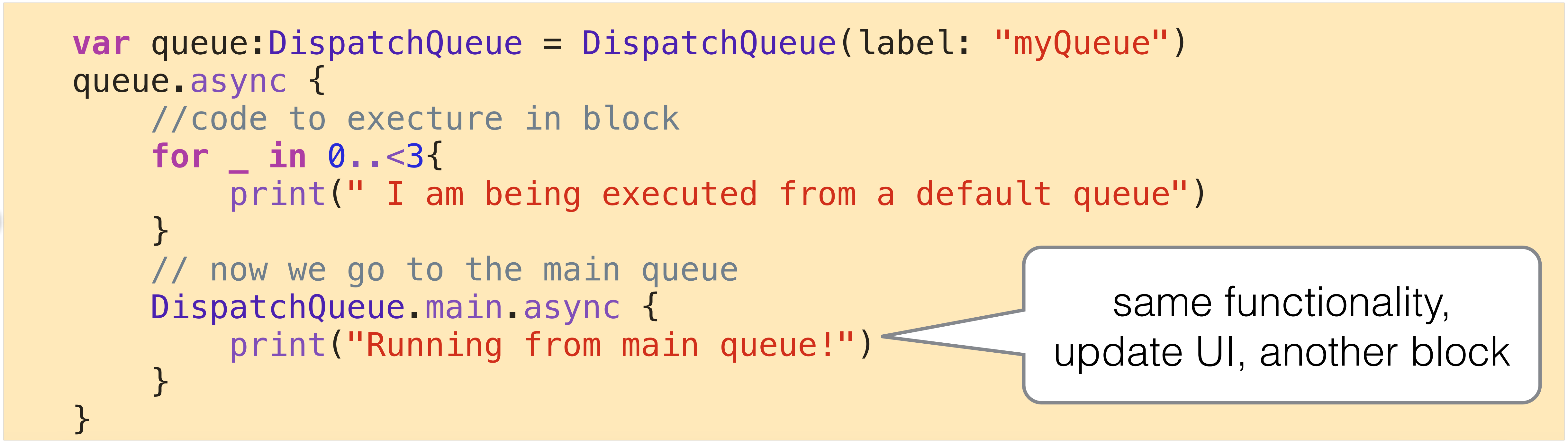
create new queue



```
NSOperationQueue *newQueue = [[NSOperationQueue alloc] init];
newQueue.name = @"ObjCQueue";
[newQueue addOperationWithBlock:^{
    // your code to execute
    for(int i=0;i<3;i++)
        NSLog(@"I am being executed from a dispatched queue, from objective-c");

    // now I need to set something in the UI, but I am not in the main thread!
    // call from main thread
   dispatch_async(dispatch_get_main_queue(), ^{
        self.label.text = [NSString stringWithFormat:@"Finished running %d times, Safe",3];
    });

}];
```

define block

update UI, another block

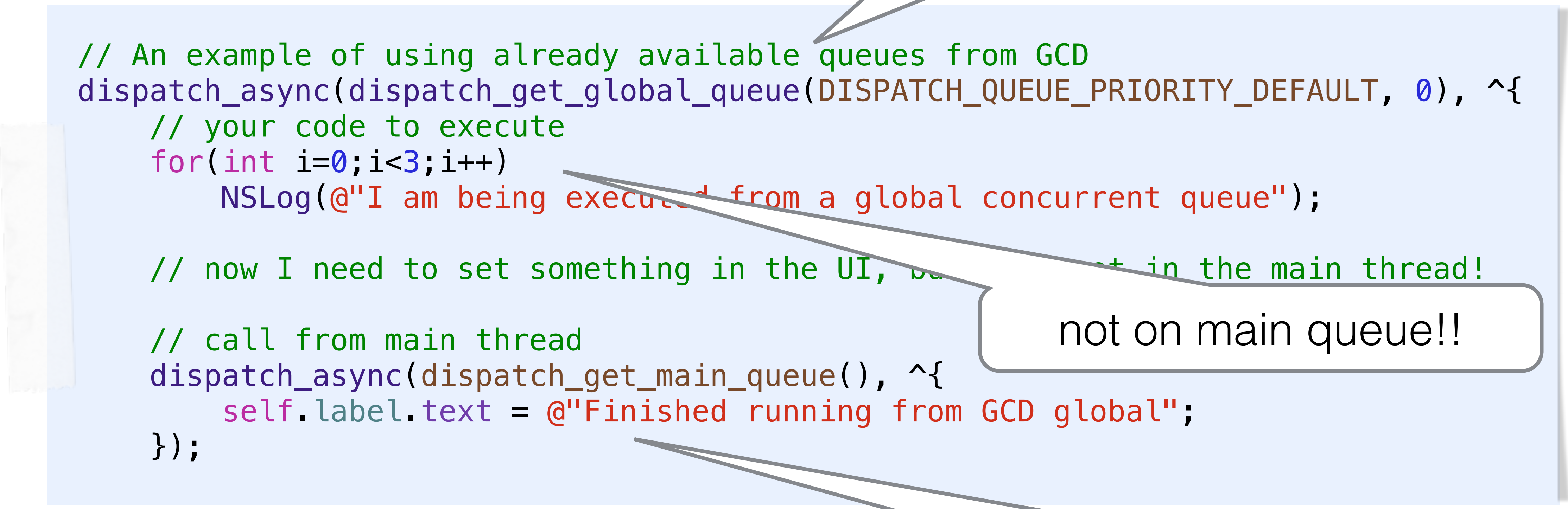


```
var queue:DispatchQueue = DispatchQueue(label: "myQueue")
queue.async {
    //code to execture in block
    for _ in 0..<3{
        print(" I am being executed from a default queue")
    }
    // now we go to the main queue
    DispatchQueue.main.async {
        print("Running from main queue!")
    }
}
```

same functionality,
update UI, another block

queue syntax

- using global queues



```
// An example of using already available queues from GCD
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    // your code to execute
    for(int i=0;i<3;i++)
        NSLog(@"I am being executed from a global concurrent queue");

    // now I need to set something in the UI, bu[illegible]t in the main thread!

    // call from main thread
    dispatch_async(dispatch_get_main_queue(), ^{
        self.label.text = @"Finished running from GCD global";
    });
```

access a global queue

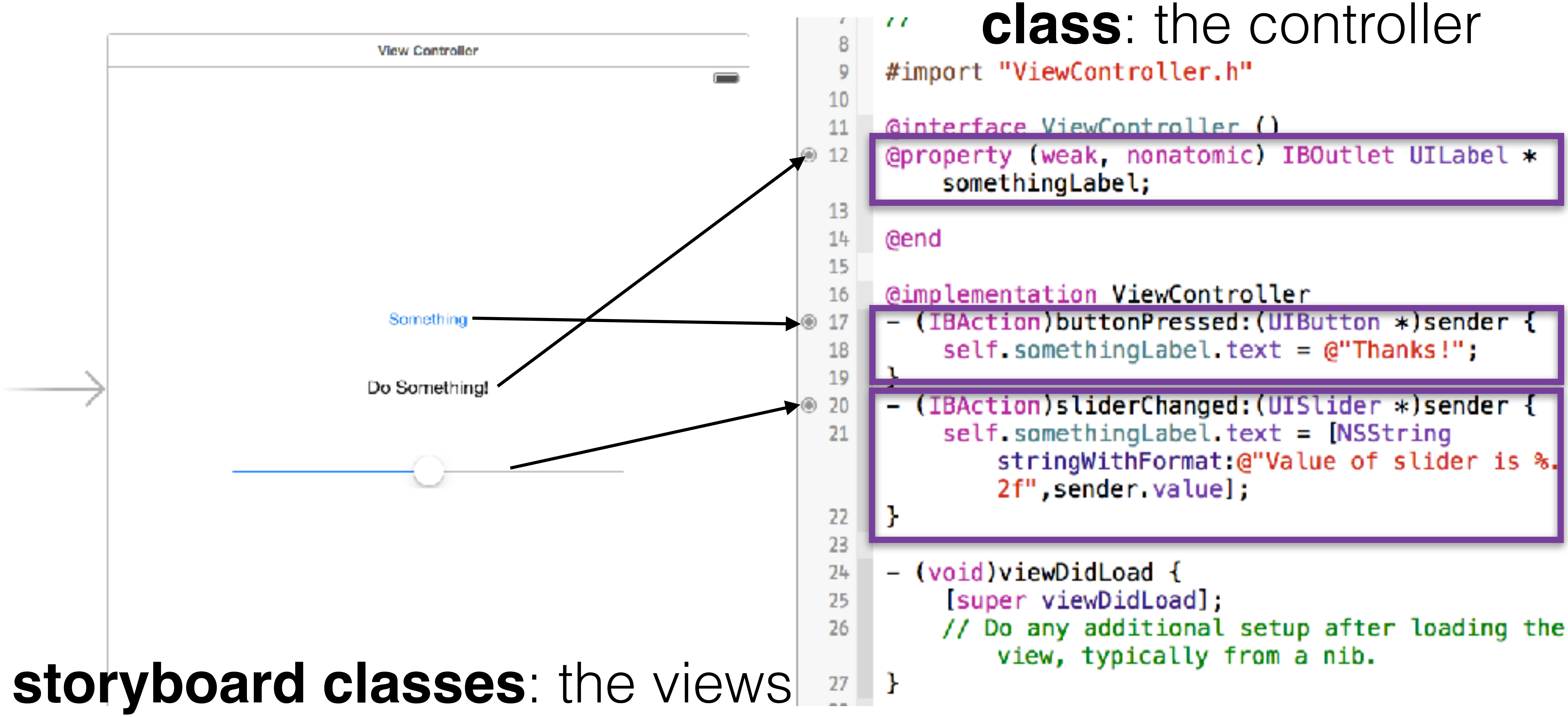
not on main queue!!

main queue!

```
DISPATCH_QUEUE_PRIORITY_LOW
DISPATCH_QUEUE_PRIORITY_DEFAULT
DISPATCH_QUEUE_PRIORITY_HIGH
DISPATCH_QUEUE_PRIORITY_BACKGROUND
```

target and action review

- UI elements communicate back to their controllers with **actions**, UI elements are called from the **Main Queue**



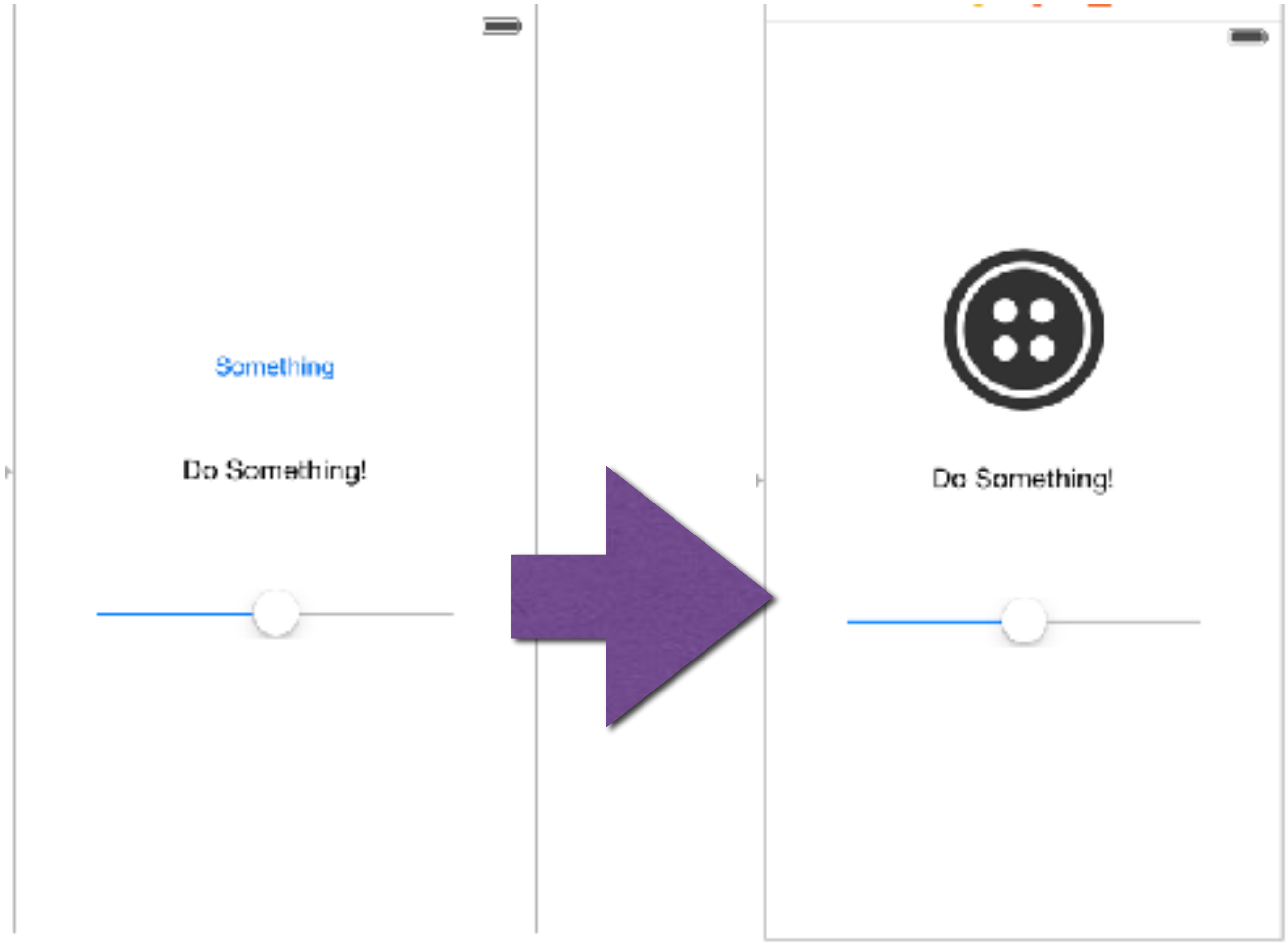
class: the controller

```
//
#import "ViewController.h"

@interface ViewController ()
@property (weak, nonatomic) IBOutlet UILabel *
    somethingLabel;

@end

@implementation ViewController
- (IBAction)buttonPressed:(UIButton *)sender {
    self.somethingLabel.text = @"Thanks!";
}
- (IBAction)sliderChanged:(UISlider *)sender {
    self.somethingLabel.text = [NSString
        stringWithFormat:@"Value of slider is %.
        2f",sender.value];
}

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

storyboard classes: the views

bring your buttons to life

- in many settings you are **given criteria** from a graphic designer
  - but right now, **you** are the graphic designer
- use **images** for more **descriptive** buttons and labels
- **good tip**: make them the right size from the start!



UI basics demo

Guess
the
Number…