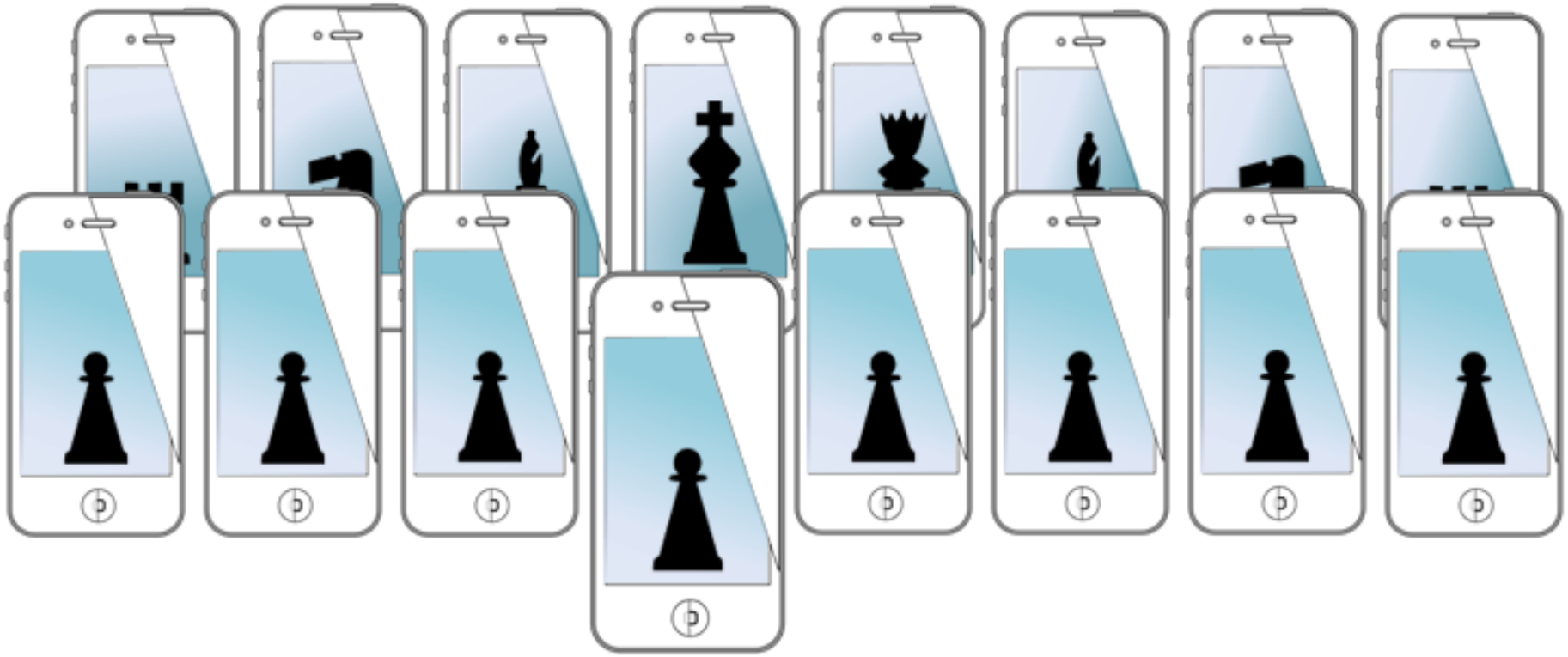
MOBILE SENSING & LEARNING



CS5323 & 7323

Mobile Sensing and Learning

objective-C, swift, and MVC

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

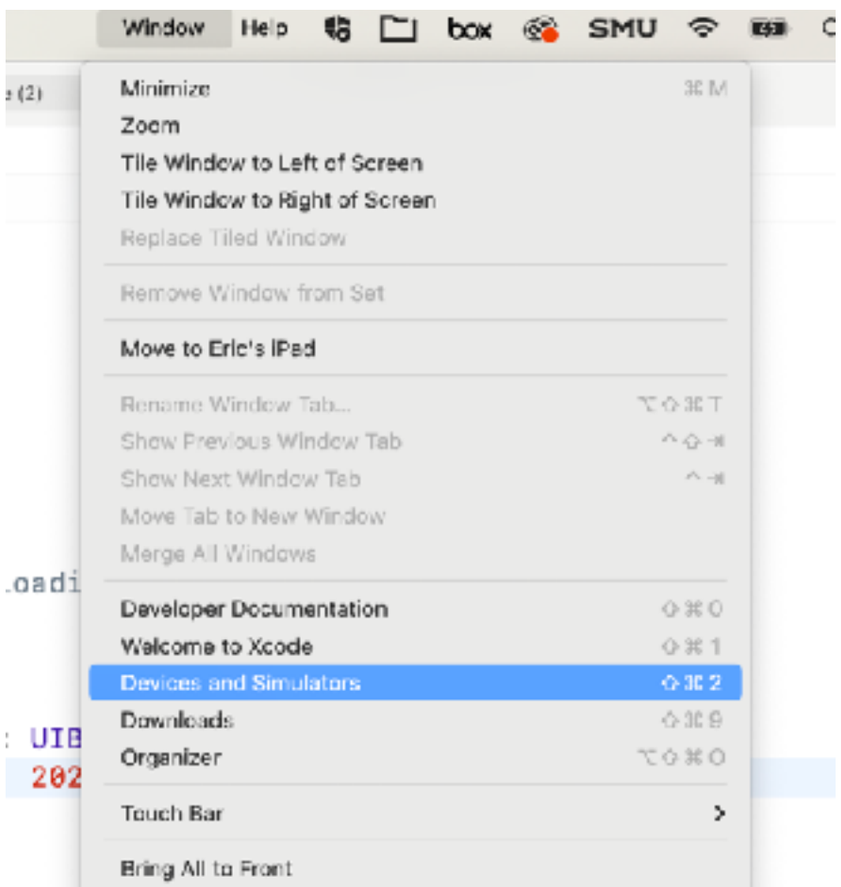
course logistics

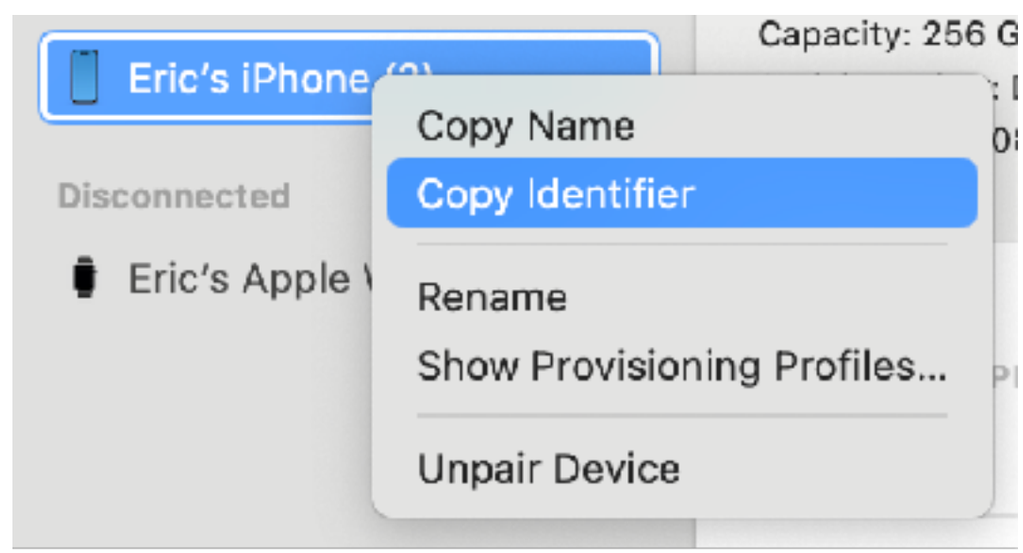
- reminder: no lab this semester
- teams: **get in a team now**! Teams can change throughout semester,
- **equipment checkout**: Phones, macs (must be on a team)
- enrollment in 5000 versus 7000 (ugrad/grad)
- Reminder: Zoom versus in-person and other classes
- Panopto videos access

Office Hours Zoom Room:	• **Instructor Office Hours**: Mondays 3:30-5:00PM, Caruth 451, CS Offices
	  ◦ **Instructor Office Hours Zoom Room:** https://smu.zoom.us/j/97583463382
	• **Teaching Assistant:** Manan Shukla (please contact via canvas)
	  ◦ **TA Office Hours**:
	  ▪ Tuesdays from 5:00 PM to 7:00 PM
	  ▪ Fridays from 1:00 PM to 3:00 PM
	  ◦ **TA Office Hours Zoom Room**: https://us04web.zoom.us/j/75504346341?pwd=8z8s5PFAArqPU3fbXtPTZpdK9hWdqP.1

Apple Developer Program

- if you update iOS, **you must update Xcode** (maybe MacOS)
- I will use my personal license, so send me:
  - **add user**: email that you want invite sent to (requires sign up)
  - **add device**: identifier, Xcode "Window>Devices and Simulators"





variables, pointers, and optionals

```
aString = nil
```

```
aString = nil
```

nil
similar to NULL_POINTER, points to nothing, can evaluate to “false” in expression

```
double aDouble;
float aFloat;
char aChar;
int aInt;
unsigned int anUnsignedInt;
...
```

Primitives
Direct Access via Stack
CANNOT be nil

```
mutable? name:Type = Value
var aDouble:Double = 0.0
var aFloat:Float = 0.0
let aChar:Character = "c"
var aInt:Int = 0
let unsignedInt:UInt = 0
...
```

Next Step **Encapsulated**
Pointers to the Heap

```
                                shorthand
NSString *myString;             @“ ”
NSNumber *myNum;                @( )
NSArray *myArray;               @[ ]
NSDictionary *myDictionary;     @{ }
NSMutableArray *arrayYouCanMutate;
```

Swift **Optionals**
Pointers to the Heap

```
let myString:String? = "Const"
var myNum:Double? = nil
let myArray:[Any]? = nil
var arrayYouCanMutate:[Any]? = nil
var myDictionary:[String:Any]? = nil
```

functions examples

return type | method name | parameter type | parameter name

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

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

second

receiver class | parameter name/value

```
MyClass *tmp = [[MyClass alloc] init];

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

throwback to **c**

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

float val = addOneToNumber(3.0);
```

(+ —) instance versus class method

```
NSNumber *obj = [NSNumber allocValue:@4];
[obj addOneToNumber:@4];
```

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

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

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

(varName:Type) -> (Return Type)

similar named second parameter syntax in swift

more functions examples

- **instance** methods:
  - require that you have [illegible] ll use the object to call the met[illegible]
- **class** methods:
  - allow you to call the m[illegible]e, mostly used for initializing or, more rarely, for utility functions

never separate `alloc` and `init`
this is illustration ONLY!

```
UILabel *tmp = [[UILabel alloc] init];
```

↓

```
UILabel *tmp = [UILabel alloc];
[tmp init];
```

class method for `UILabel`

instance method for `UILabel`

In `UILabel` Class these are declared as follows:

```
+(UILabel *)alloc{ … };
-(UILabel *)init{ … };
```

iteration on array/dictionary

```
NSArray *myArray = @[@32,@"a string”,@3.2, @3,@5,@9,@42,@32];

for(id obj in myArray)
    NSLog(@“Obj=%@",obj);
```

use **id** to signify that type of object is not known

```
NSDictionary *aDictionary = @{@"key1":@3,@"key2":@"a string”};

for(NSString key in self.aDictionary)
    NSLog(@"key=%@, value=%@“, key, aDictionary[key]);
```

use **NSString** type is known

declaration requires specifying **any** if the data is not consistent

```
let myArray: [Any] = [32,"a string”, self.aString]
for val in myArray{
      print(val)
}
self.aDictionary = ["key1":3, “key2":"String value"] as [String : Any]

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

Dictionary loops through as tuple **(key, value)**

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

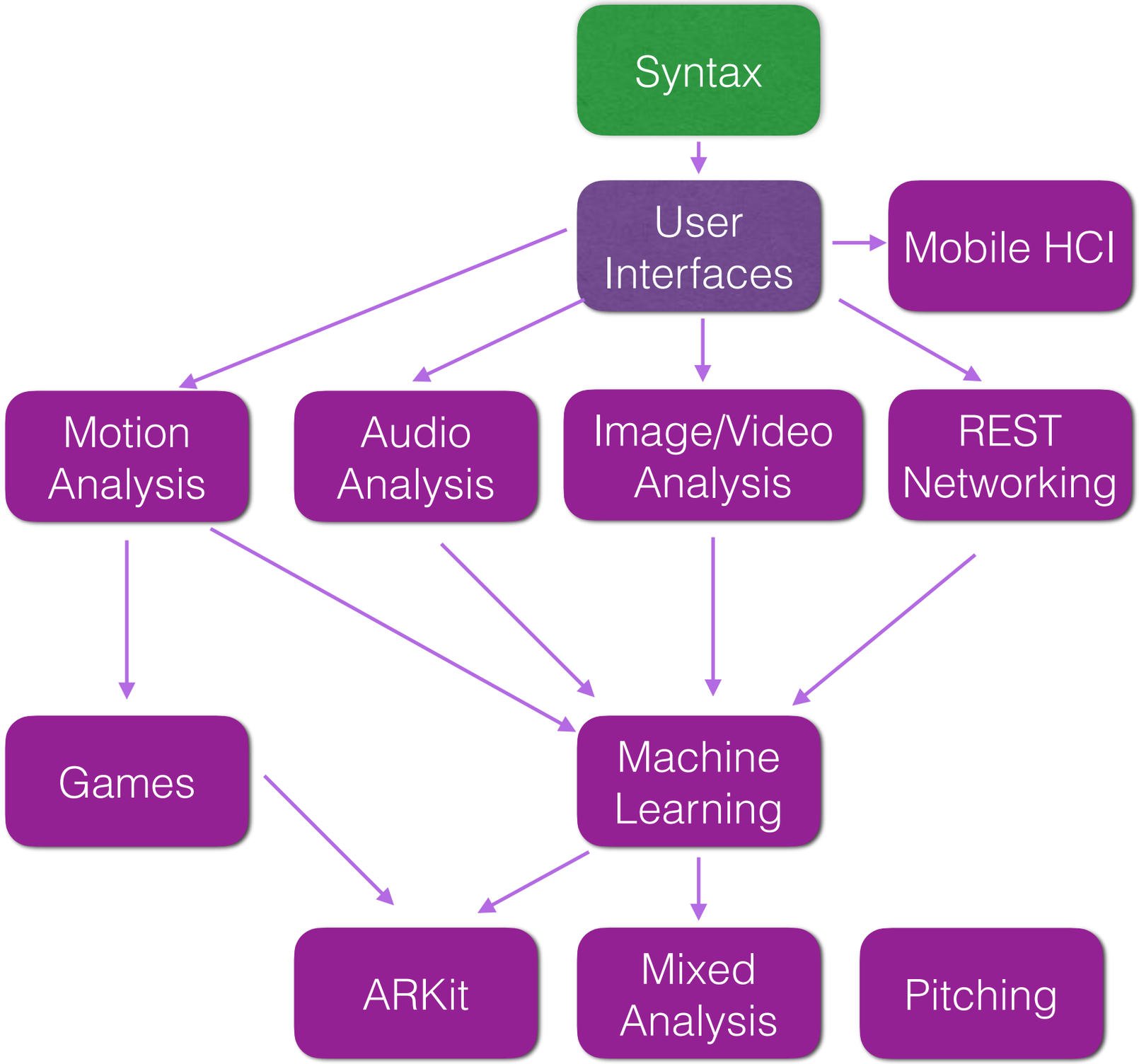
dictionary iteration

array iteration

adding to our project

- let's add to our project
  - an objective-c class and swift class
  - and practice using lazy instantiation

class progression



agenda

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

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

optional versus implicit

- optional values can be nil or a specific datatype
  - many properties of a swift class will therefore be optional
- lazily instantiated properties must be optional (no init needed), but `if let` syntax is cumbersome if used for all properties every time
- enter the world of implicitly unwrapped optionals!
  - do not check if nil, go to the data … runtime error if nil…

```
@IBOutlet weak var implictUnwrapLabel: UILabel!
@IBOutlet weak var optionalLabel: UILabel?
@IBOutlet weak var label: UILabel

implicitUnwrapLabel.text = “No need to unwrap with ?”
optionalLabel?.text = “must unwrap”

if let label = optionalLabel {
        label.text = “safely unwrapped in if let”
}
```

why can we not do this?

adding to our project

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

blocks and closures

- a block of code that you want to run at another time and perhaps pass to other classes to run
  - created at runtime
  - acts like an object that can be passed as an argument or created on the fly
  - once created, can be called repeatedly
  - can access variables from scope where defined
  - syntax is slightly different in swift and objective-c
  - common to define when calling a method that uses block
- swift calls these **closures**, objective-c says **blocks**

block/closure syntax

most common usage is as input into a function

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

this variable is in scope of block!

```
NSNumber *objInScope = @(32)
// 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]);
    NSLog(@“Scope Variable = %.2f",[objInScope floatValue]);
}];
```

swift syntax

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

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

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

Also valid if closure is last input

some semantics

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

```
NSNumber * objInScope = @5.0;
```

- Unless you use keyword (now mutable):

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

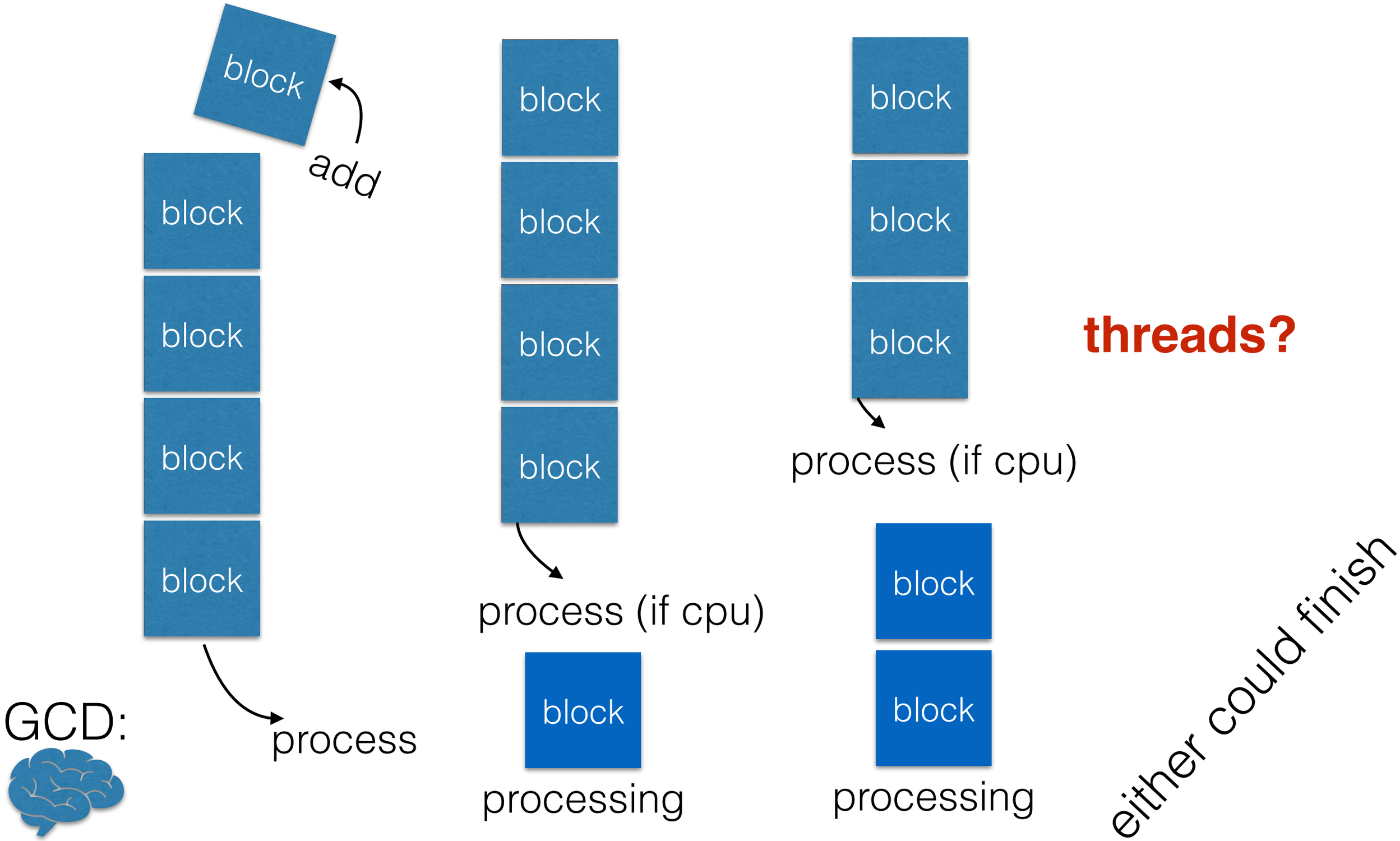
- classes hold a **strong** pointer to blocks they use
- blocks hold a **strong** pointer to `__block` variables
  - so using “self” would create a retain cycle

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

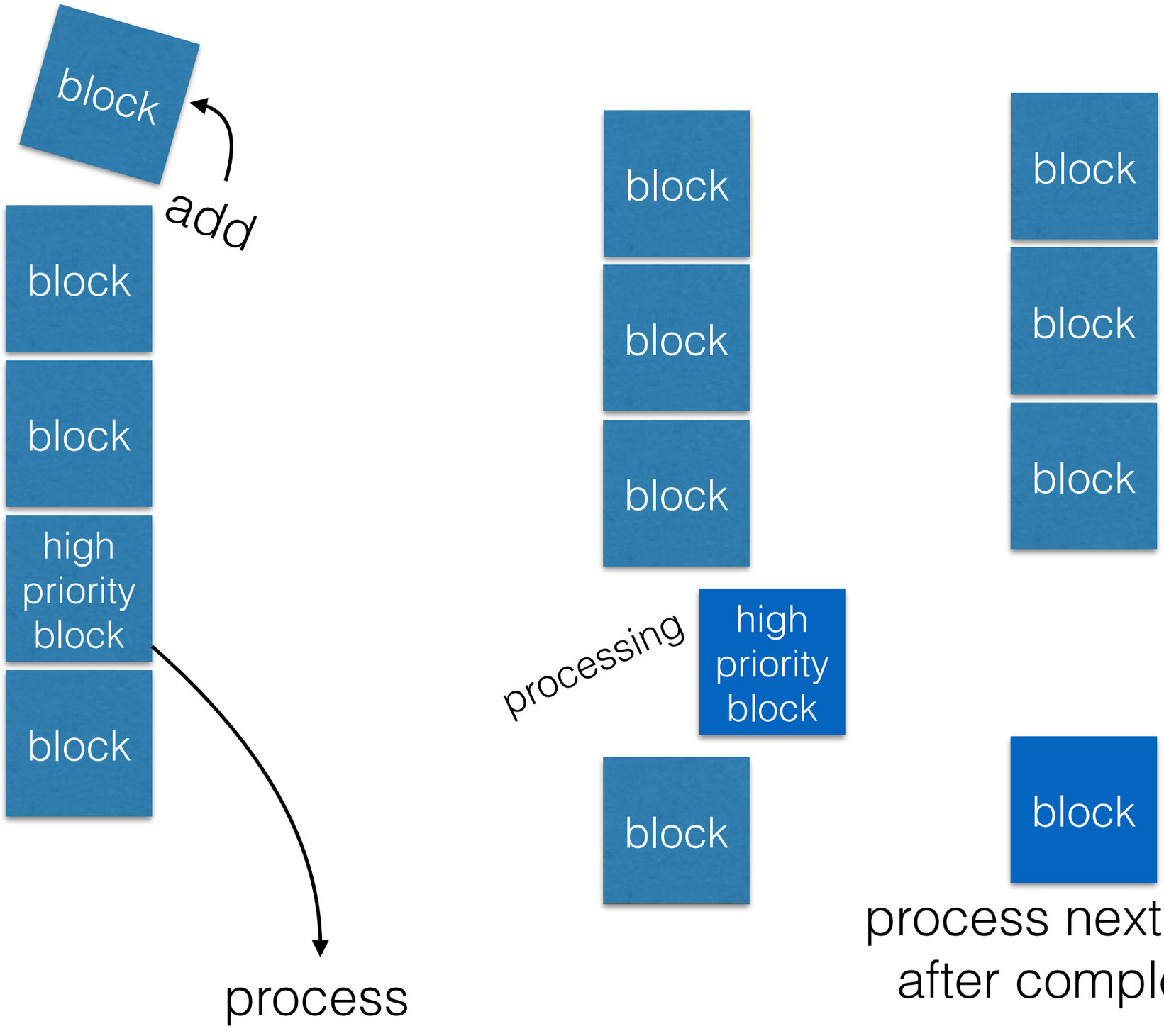
concurrency in iOS

- 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



the main queue



one thread
UI is in sync

GCD: