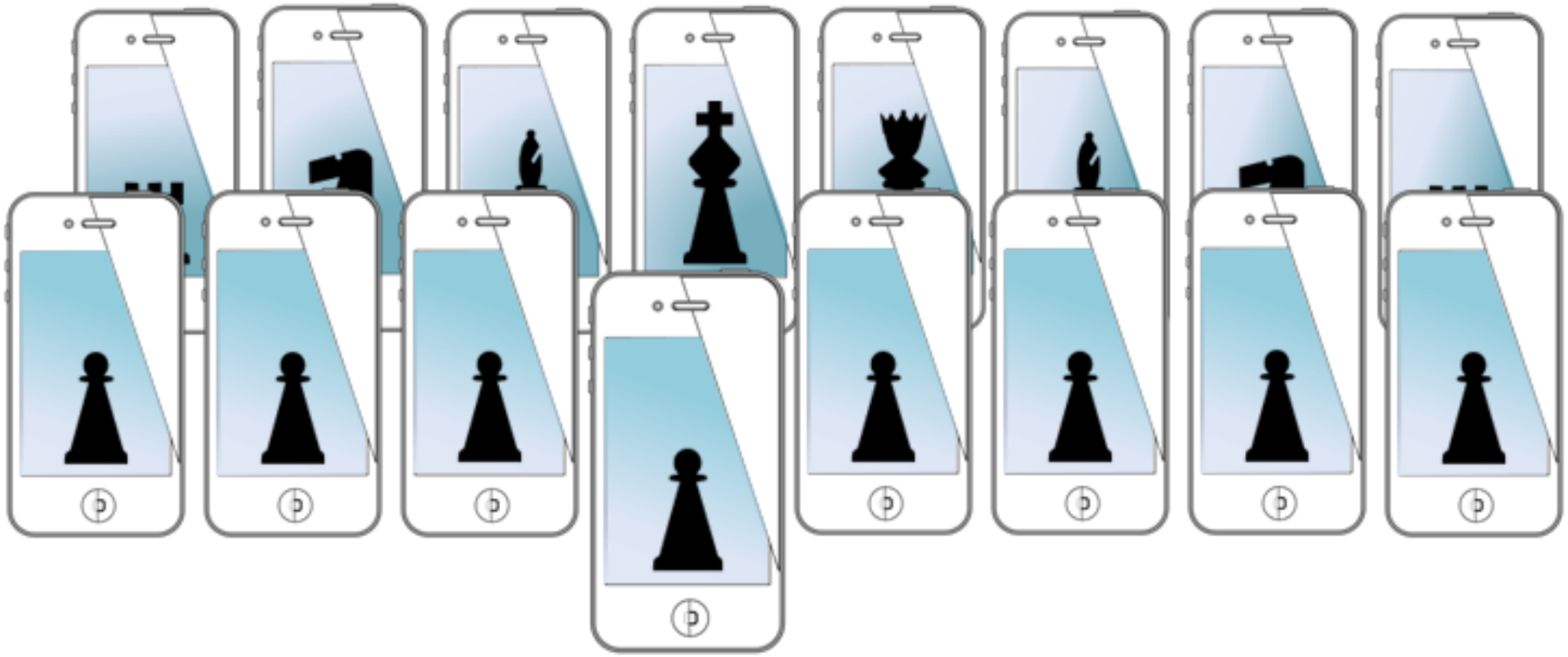
MOBILE SENSING & LEARNING



CS5323 & 7323

Mobile Sensing and Learning

objective-C, swift, and MVC

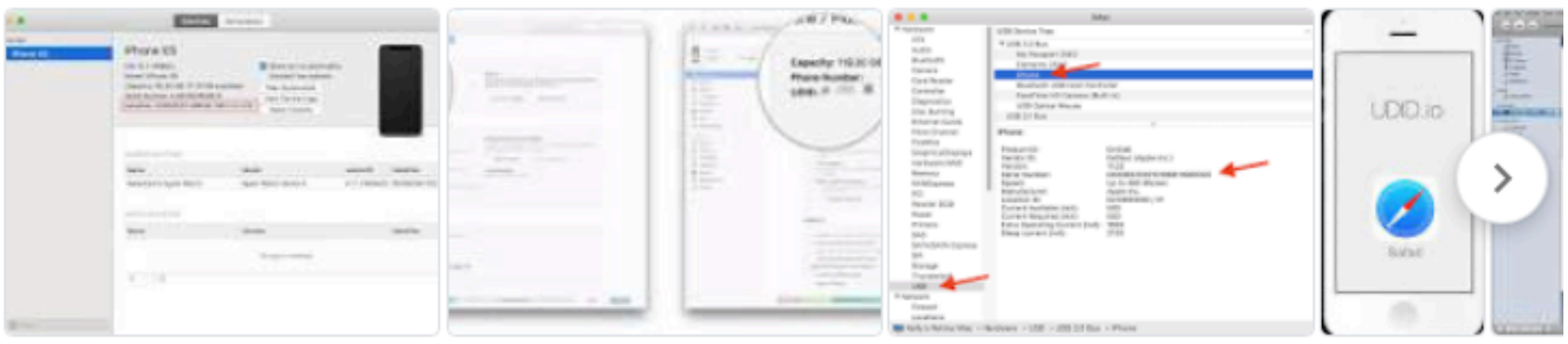
Eric C. Larson, Lyle School of Engineering,
Department of Computer Science, Southern Methodist University

course logistics

- reminder: no lab this semester
- teams: **should be on a team now**!
- **equipment checkout**: Phones
- enrollment in 5000 versus 7000 (ugrad/grad)
- Reminder: Zoom versus in-person and other classes
- Panopto videos

Apple Developer Program

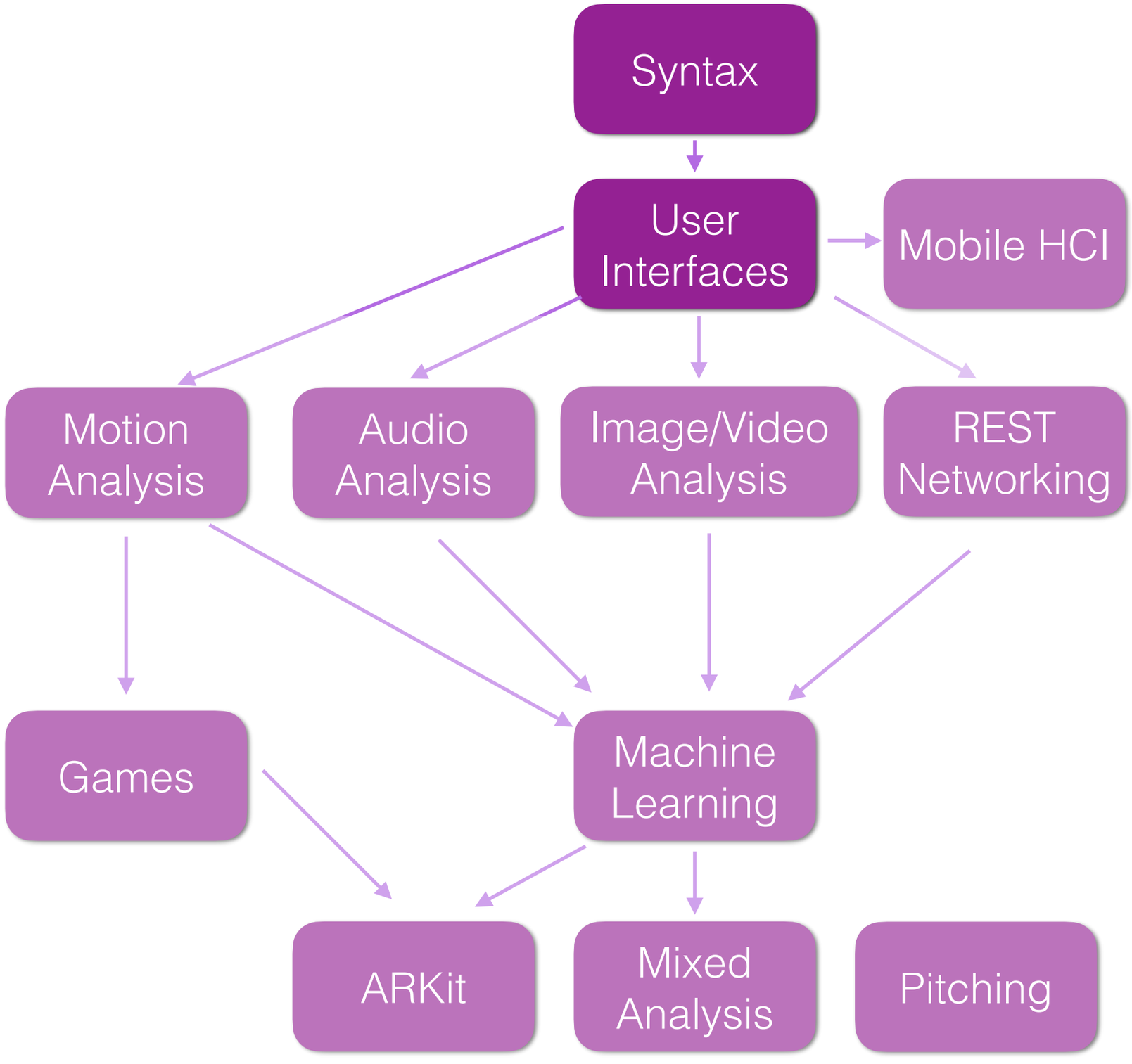
- university developer program: send me an email and I will add you to the program:
  - email that you want invite sent to
  - phone UDID: In Itunes, can also use the Xcode “simulator and devices” window



How To Find Your UDID?

1. Launch iTunes & connect your **iPhone**, iPad or iPod (device). Under Devices, click on your device. Next click on the 'Serial Number' ...
2. Choose 'Edit' and then 'Copy' from the iTunes menu.
3. Paste into your Email, and you should see the **UDID** in your email message.

class progression



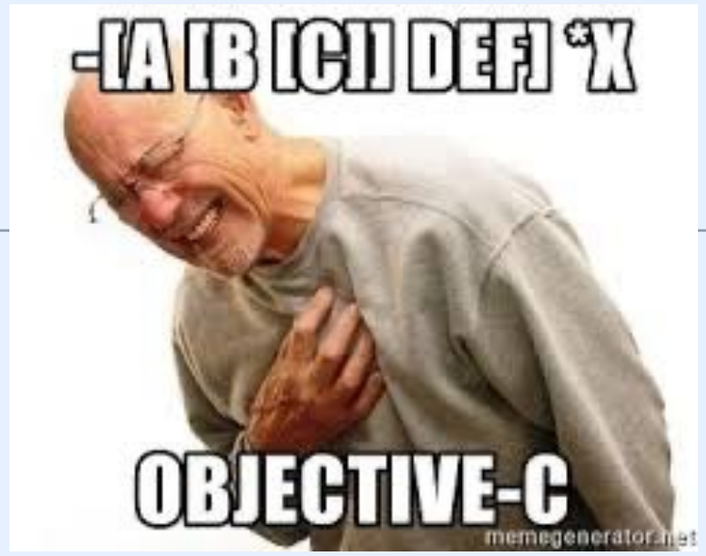
agenda

a big syntax demo…

- **objective-c and swift together**
  - class declaration
  - complex objects
  - common functions
  - encapsulation and primitives
  - memory management

and model view controllers, if time
…also available on flipped module video…

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

an example class

```
@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);


}
```

```
class SomeViewController: UIViewController {

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

    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)
        }

    }
}
```

let’s work our way up to understanding both of these examples

variables, pointers, and optionals

```
aString = nil
```

```
aString = nil
```

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

```
mutable? name:Type = Value
```

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

Primitives
Direct Access via Stack
CANNOT be nil

```
var aDouble:Double = 0.0
var aFloat:Float = 0.0
var aChar:Character = "c"
var aInt:Int = 0
var 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
```

classes

class name

inherits from

```
@interface SomeClass  : NSObject

@property (strong, nonatomic) NSString *aPublicStr;

@end
```

if in the **.h** file, it is public

obj-c property:
NOT variables, but they provide *access* to backing variables

```
@interface SomeClass ()

@property (strong, nonatomic) NSString *aPrivateStr;

@end

@implementation SomeClass

   //… implementation stuff…
@end
```

if in the **.m** file, it is private

Declared in the **.swift** file

class name

inherits from

```
class SomeClass : NSObject{

    var aPublicString = "..."
    private var aPrivateString = "..."

    // implementation stuff
}
```

swift defaults to **public properties**
must use **private** keyword

swift property:
- special variables
- can add functionality through observers and overrides

objective c

class property: access a variable in class

```
@interface SomeClass ()
@property (strong, nonatomic) NSString *aString;
@end
@implementation SomeClass
@synthesize aString = _aString;
```

property declared

backing variable: usually implicit to compiler

setter, **auto** created `self.aString=val;`

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

getter, **auto** created `val=self.aString;`

```
-(NSString *)aString{
  return _aString;
}
```

property `self.aString`

variable `_aString`

getter, **custom overwrites** auto creation

```
-(NSString *)aString{
    if(!_aString)
        _aString = @"This string was not set";
    return _aString;
}
```

```
@end
```

lazy instantiation

objective c

class properties

```
@interface SomeClass ()

  @property (strong, nonatomic) NSString *aString;


@end

@implementation SomeClass
   -(NSString *)aString{
       if(!_aString)
           _aString = @"This string was not set",
       self.aString = @“Getter Called to set”;
       return _aString;
   }
   -(void)someFunction{

       _aString = @“Direct variable Access, No getter Called”;

       self.aString = @“Getter Called to set”;

   }

@end
```

What does this do?

swift

class properties

```
class SomeClass : NSObject{

    var aPublicString = "..."
    private var aPrivateString = "..."

    var noDefaultVal:Int
    override init() {
        self.noDefaultVal = 0
    }

    lazy var aString = "Default val if not set"
    lazy var aStringAlso = {
        // could do other things here
        return "Value"
    }()


    var watchedVariable:Float = 0.0 {
        willSet(newValue){
            print("setting value to \(newValue)")
        }
        didSet{
            print("\(oldValue) set to \(watchedVariable)")
        }
    }

}
```

property declared in class directly

if no default value, must be setup in `init()`

lazy instantiation, set to values if accessed

property observers: willSet and didSet

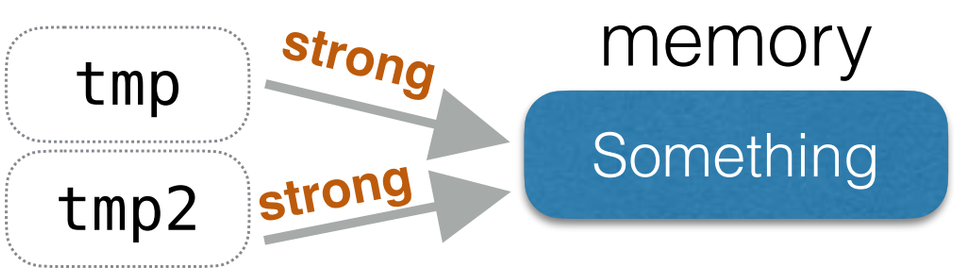
can also override "set" and "get" methods, but this is rare to need

automatic reference counting

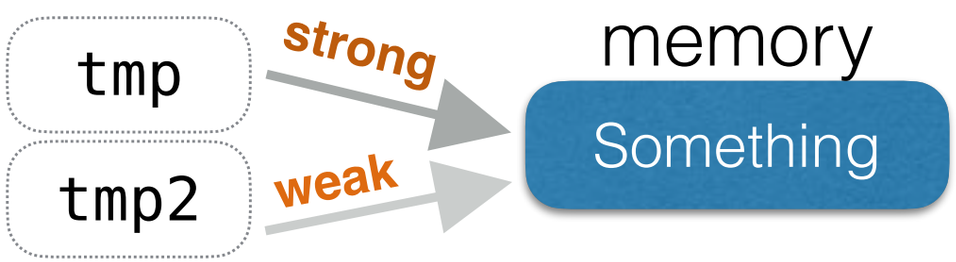
- not garbage collection
- when reference count for variable == 0, trigger event to free memory
  - **strong** pointer adds to reference count
  - **weak** pointer does not add to reference count
  - **unowned** special case of weak, always assumes there is a strong reference with longer lifetime

```
var tmp:String? = "Something"
var tmp2 = tmp
tmp = nil
tmp2 = nil
```

```
NSString* tmp = @"Something";
NSString* tmp2 = tmp;
tmp = nil;
tmp2 = nil;
```

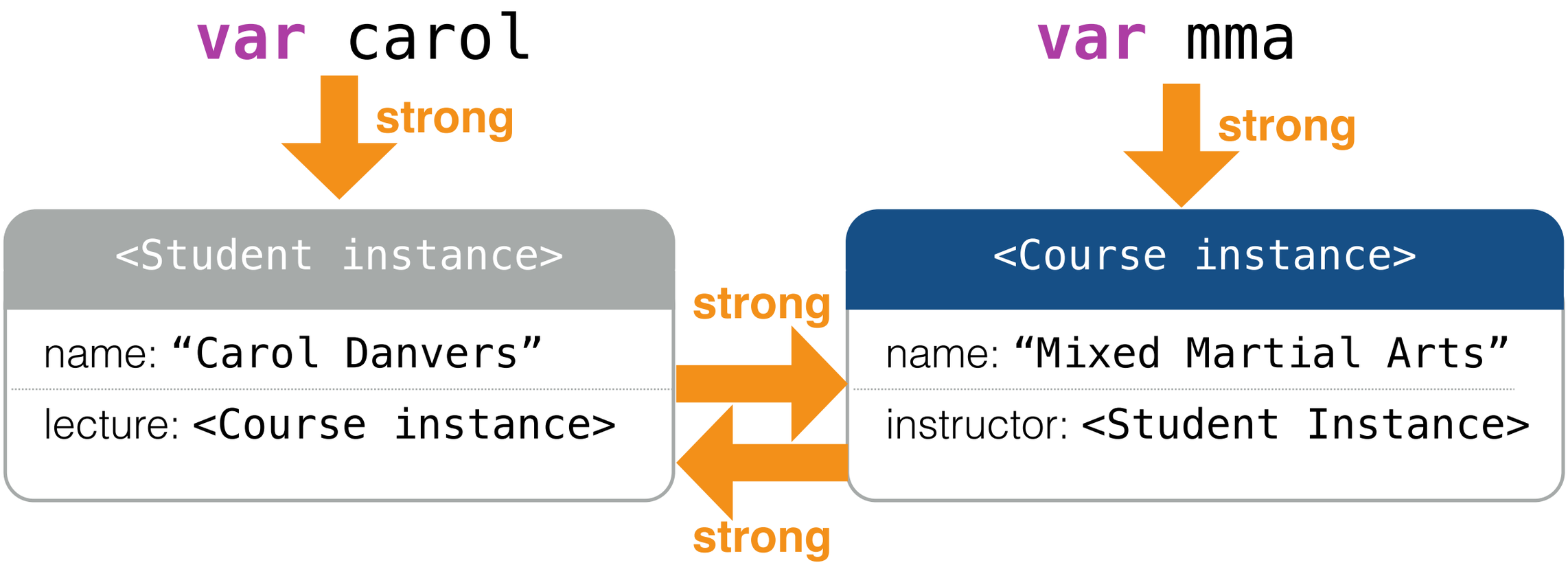


- deallocated after **both references** are `nil`



- deallocated after **strong reference** is `nil`

automatic reference counting

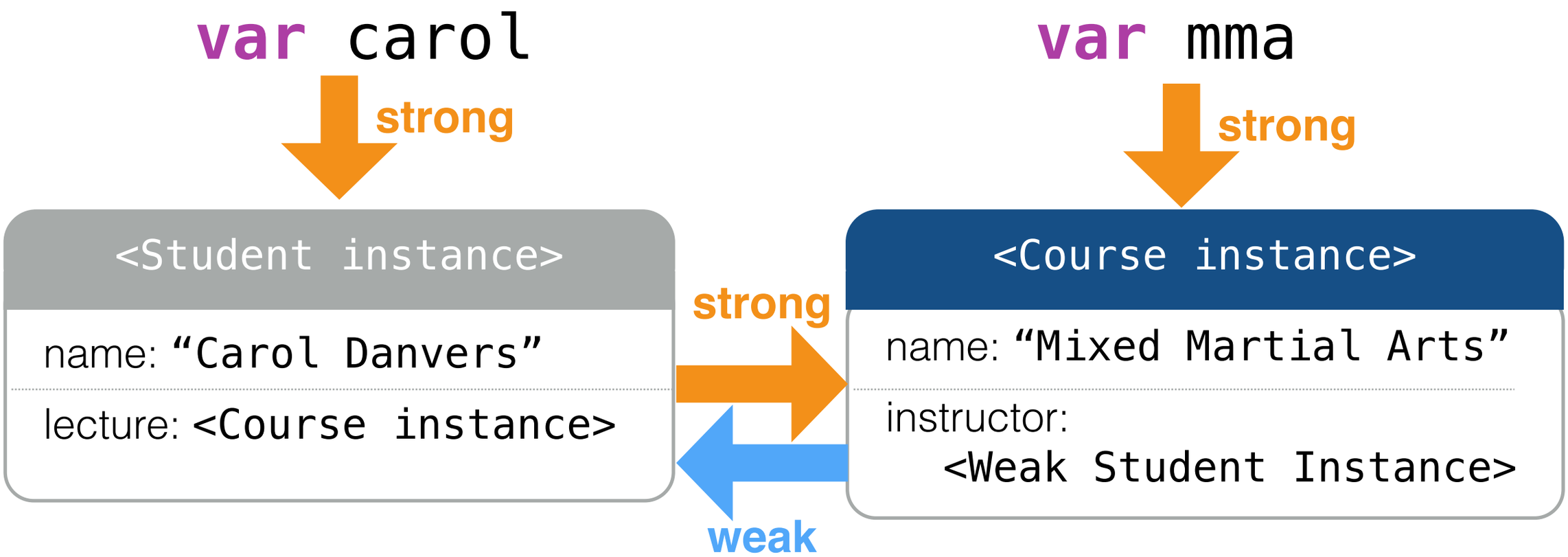



```
carol.lecture = mma
mma.instructor = carol

mma = nil
carol = nil
```

- memory never deallocated because reference cycle
- results in a memory leak if done repeatedly
- solution: weak pointers

automatic reference counting

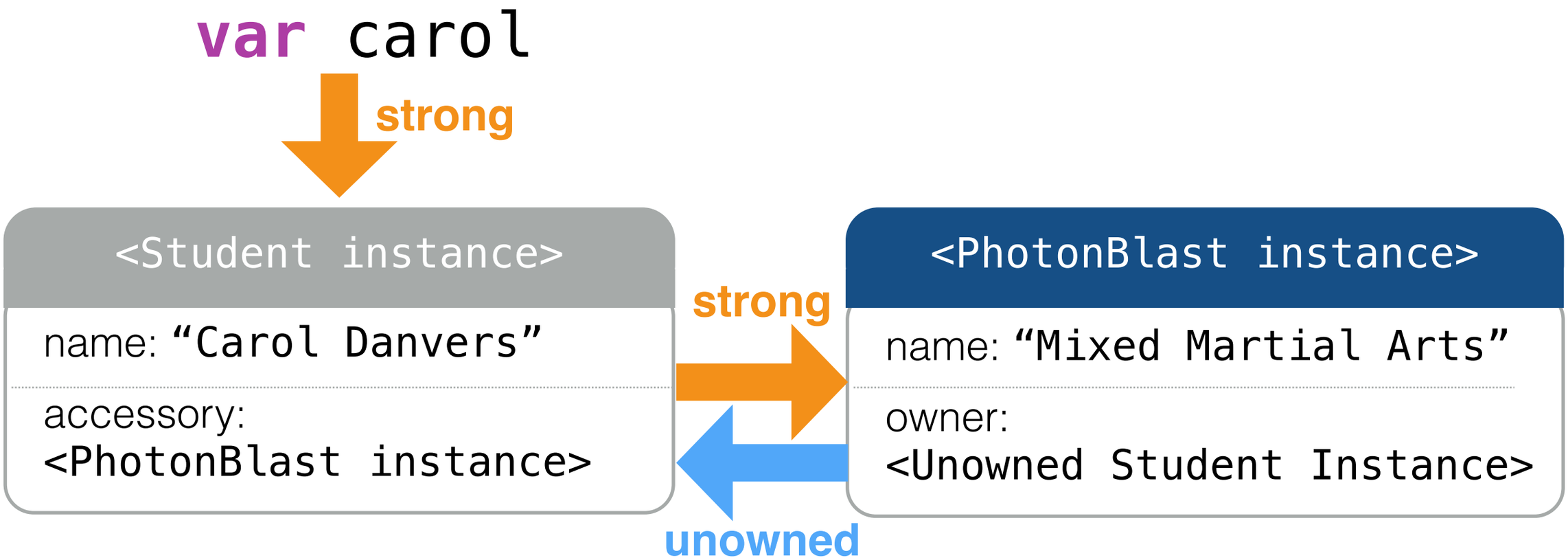



```
carol.lecture = mma
mma.instructor = carol

carol = nil
mma = nil
```

- references to parent instance cascade into properties
- all memory released immediately for use in app

unowned usage



- used primarily when there is no need for referencing a class instance without the parent instance
- typically one-to-one class instances

using strong, weak, unowned

atomic ~ thread safe property access
nonatomic ~ faster access

```
@property (strong, nonatomic) Student *aStudent;
```

strong ~ keep a reference
weak ~ no reference

```
weak var aStudent: Student?
unowned var aStudent: Student?
```

strong by default in swift
weak used when needed

```
self.aStudent = [[Student alloc] init];
```

most common initialization
syntax for obj-c and swift

```
self.aStudent = Student()
```

properties are accessed
through `self` (like c++)

iteration on objects

```
NSArray *myArray = @[@32,@"a string”, [@3.1,@3,@5,@9,@42,@32]];
for(id obj in myArray)
    NSLog(@“Obj=%@",obj);
```

can store any object

loop over an NSArray

```
@interface SomeClass ()
@property (strong, nonatomic) NSDictionary *aDictionary;
@end
```

Dictionary as a class property

Access self

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

```
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)
}
```

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

Dictionary loops through as tuple `(key, varName)`

mutable and immutable

```
NSArray *myArray = @[@32,@"a string”, [[UILabel alloc]init] ];
```

arrays are **nil** terminated

```
NSMutableArray *anArrayYouCanAddTo = [NSMutableArray arrayWithObjects:aNum,@32, nil];
[anArrayYouCanAddTo addObject:someComplexObject];
```

possible to add objects now

```
NSMutableArray *anotherArray = [@[@32,@"string me"] mutableCopy];
```

```
let myConstArray = [34, 22, 1]
var myArray = [22, 34, 12]
```

more explicit in swift regarding mutability

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

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

for next time…

- **next time**: more dual language programming
- **one week**: flipped assignment
- **then**: mobile HCI