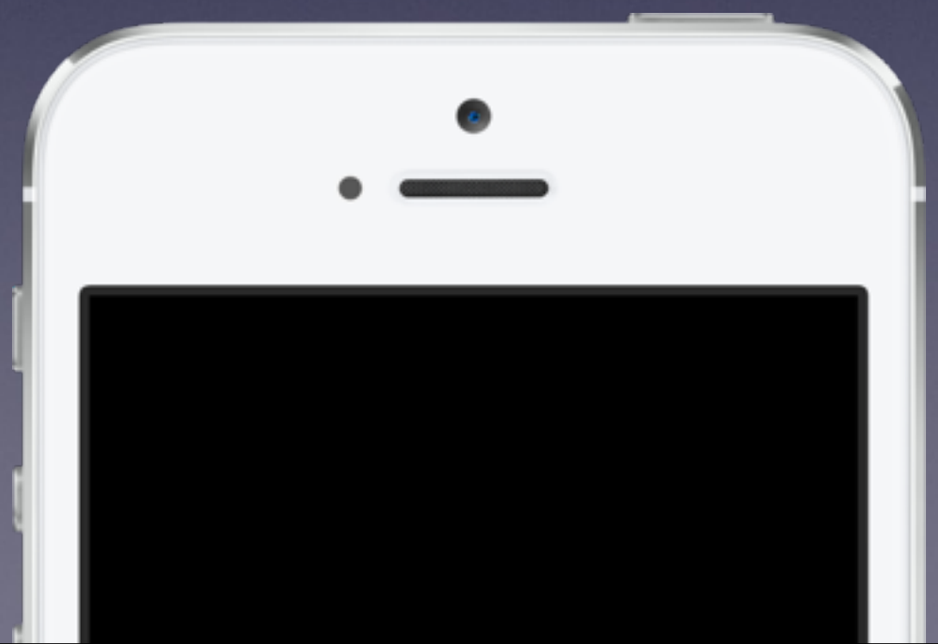
COMS W3101: Programming for iOS

Michael Vitrano



Diving into Obj-C

- Memory management and why its important
- More about classes and methods
- Types and Typing in Objective-C
- Introduction to NSObject and Foundation
- Blocks

Memory Management

- Memory is where your app stores its object graph
- iOS devices have significantly less memory than desktop computers
- If your application uses too much memory, the OS reserves the right to kill it without warning
- Understanding ObjC memory management is fundamental to iOS development
- All ObjC objects are allocated on the heap

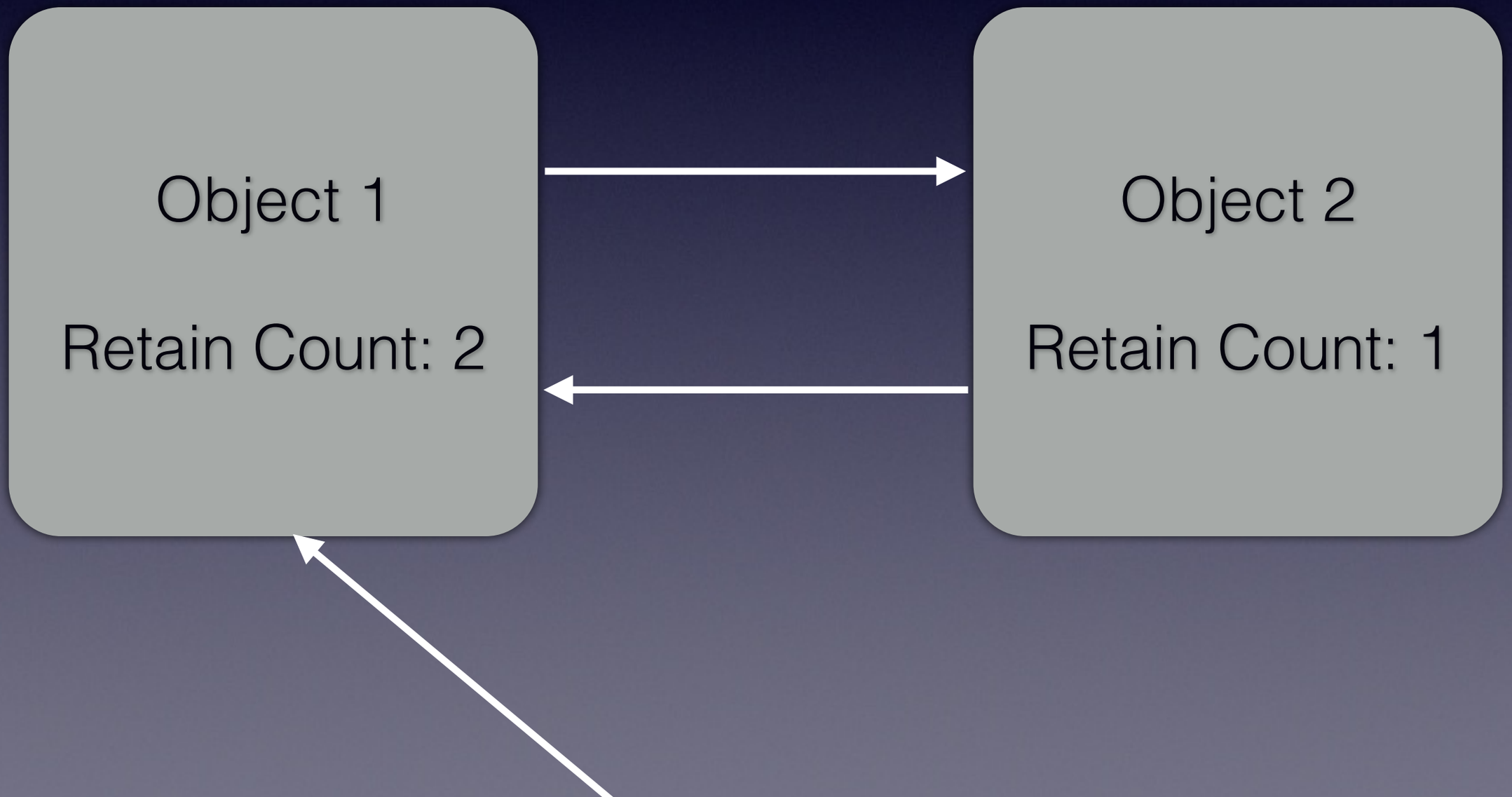
Reference Counting

- Objective C uses reference counting to manage the life-cycle of its objects
- Reference counting is fundamentally different than garbage collection
- Requires more attention in exchange for far superior performance
- Easy to leak memory if not careful!

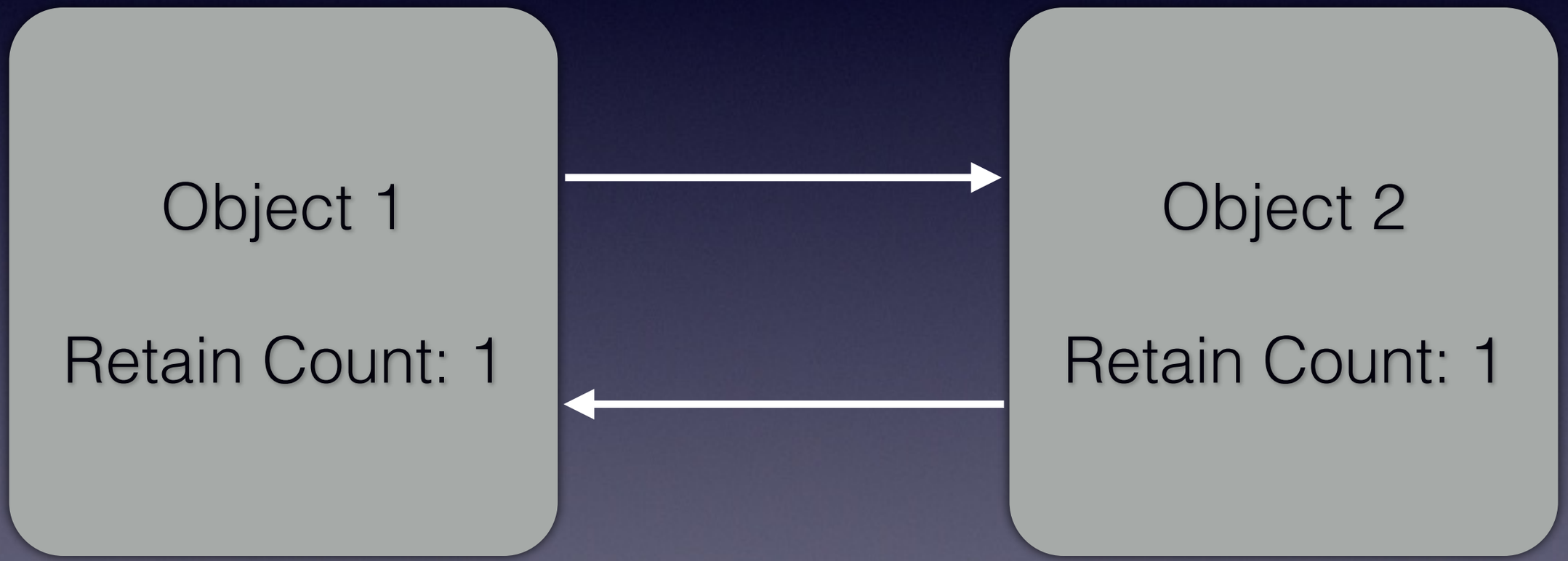
Reference Counting

- Each object maintains a count of **strong** references to it
- When that count reaches zero, the object is deallocated
- Each **weak** reference does not increment that reference count
- Local variables are of type **strong** by default

Retain Cycles

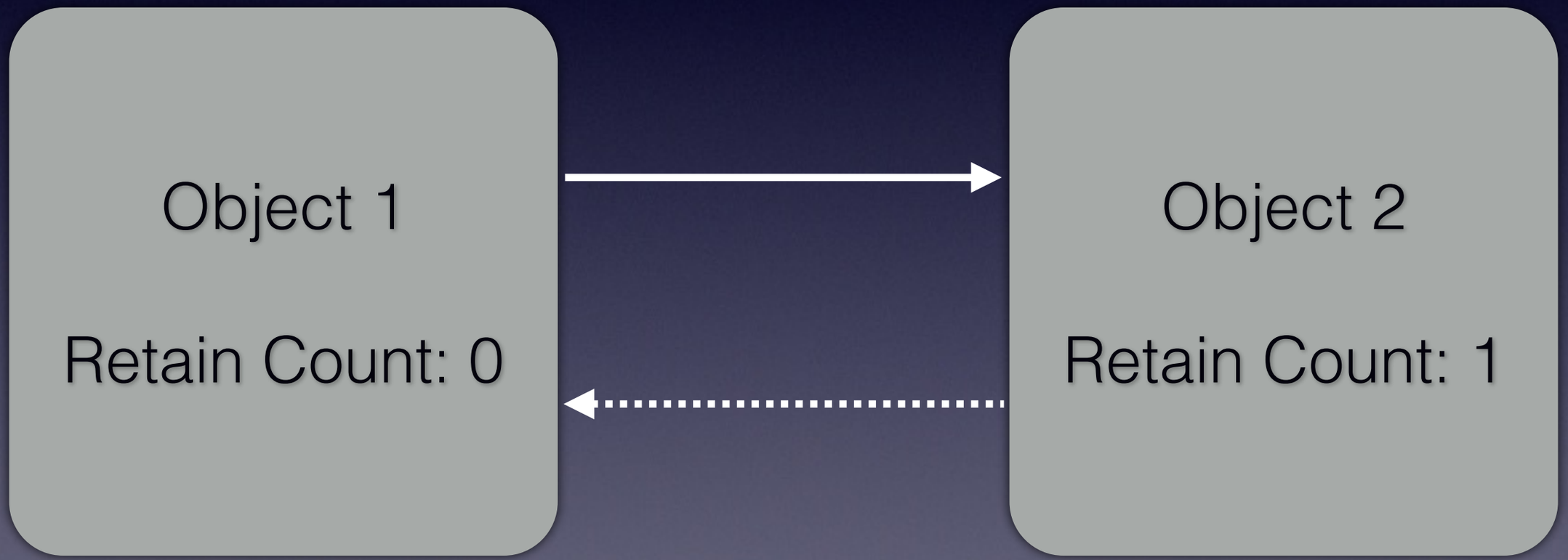


Retain Cycles



Objects 1 and 2 can never be deallocated!

Retain Cycles



Using a **weak** reference breaks the cycle

Retain Cycles

Object 2

Retain Count: 0

Object 1 gets deallocated first, then Object 2

Classes vs. Instances

- Classes are blueprints for objects
- There can be many instances of a given class in existence

- Classes are objects too!
  - There exists exactly one Class object per type
  - Classes can have methods but no storage
  - Access an instance's class object by calling the **class** method on the object

Instance Methods

- Called on individual instances of a class
- Definitions begin with a **-**
- Examples:

```
- (void)doCoolThingsWithRobot:(MSVRobot *)robot
                      options:(MSVRobotOptions *)options;


- (NSInteger)integerValue; // From NSNumber

// From UIViewController
- (void)transitionFromViewController:(UIViewController *)fromViewController
                    toViewController:(UIViewController *)toViewController
                            duration:(NSTimeInterval)duration
                             options:(UIViewAnimationOptions)options
                          animations:(void (^)(void))animations
                          completion:(void (^)(BOOL finished))completion
```

Class Methods

- Called on the class instance
- Definitions begin with a **+**
- Within the implementation, **self** refers to the class object
- Used for creation and utility methods
- Examples:

```
+ (void)isRobot:(MSVRobot *)robot compatibleWithSoftwareUpdate:(MSVRobotUpdate *)update;


+ (instancetype)stringWithFormat:(NSString *)format, ...; // From NSString

// From NSObject
+ (id)alloc;
```

Instance and Class Methods Example

```
@interface Robot : NSObject

+ (void)isRobot:(MSVRobot *)robot
compatibleWithSoftwareUpdate:(MSVRobotUpdate *)update;

@property (nonatomic) NSString *version;
@property (nonatomic) NSString *name;

- (void)doCoolThingsWithRobot:(MSVRobot *)robot
                      options:(MSVRobotOptions *)options;

@end
```

Allocation and Initialization

- Creating an object is a two step process:

```
MyObject *obj = [[MyObject alloc] init];
UIButton *button = [[UIButton alloc] init];
```

- **Alloc** creates creates enough space for the object instance on the heap.
    - The size of an instance is determined by the object's instance variables
    - All instances variables are set to zero on allocation
- **Init** the default initialization method
- An object is not ready for use until initialization has occurred

More Initialization

- Initialization provides a point at which classes can set default values for variables and allocate other necessary objects

- You can have initialization methods that take arguments:

```
- (instancetype)initWithString:(NSString *)aString;
```

  - Initializers must begin with **init**

  - They should return type **instancetype**, which translates to “an instance of the current class”

  - Each class must have a designated initializer which subclasses must use to initialize themselves in their designated initializer

Example Initialization

```
// Initializer for AwesomeView
- (instancetype)initWithFrame:(CGRect)frame
{
    // initWithFrame: is UIView’s designated initializer
    if (self = [super initWithFrame:frame]) {
        // Do customization here
    }
    return self
}
```

Example Initialization

```
// Convenience initializer for AwesomeView that will set
// the background color with the one provided
- (instancetype)initWithFrame:(CGRect)frame color:(UIColor *)color
{
    // initWithFrame: is UIView’s designated initializer
    // Note that we don't call super initWithFrame
    if (self = [self initWithFrame:frame]) {
        [self setBackgroundColor:color];
    }
    return self
}
```

Convenience Methods

- Many framework classes have convenience methods for allocating instances

- Examples:

```
+ (id)array; // NSArray
+ (NSNumber *)numberWithFloat:(float)value // NSNumber
+ (instancetype)stringWithFormat:(NSString *)format, ...; //NSString
+ (UIImage *)imageNamed:(NSString *)name // UIImage
```

id

- **id** is a keyword in the Objective C language that represents a pointer to an unknown class
  `id obj = … // obj is a variable that holds an unknown object`
- When combined with introspection and casting, we can assign a variable of type **id** to another variable with a definite type for safer use
- Useful for collection classes which can hold objects of arbitrary type
  - There is no concept of generics in ObjC
- Useful in conjunction with **protocols** to express the need for an object to be able to implement certain methods without knowing the exact type

Class

- Pointer to a class object
- You can get values in a number of ways:

```
Class objClass = [obj class] // objClass holds obj’s class object

Class stringClass = [NSString class] // string class holds a
                                     //reference to the NSString Class

Class stringClass = NSClassFromString(@“NSString”) // same as above
```

- Useful for meta-programming:

```
id obj = [[classObj alloc] init];
```

nil

- Represents the value of a object pointer that is not pointing to anything
- Defined as zero
- All variables that point to an object, whether instance variables or stack variables are initially set to **nil**
- Sending a message to **nil** is a no op
  - If that method is non-void, then zero is returned

    ```
    id obj = [array objectAtIndex:i] // obj will be nil if array is nil
    ```

  - If a method returns a struct, the return value is undefined

    ```
    CGRect frame = [view frame] // frame will be undefined if view is nil
    ```

nil, Nil, Null, NSNull

- **nil** - value of a object pointer that is not pointing to anything
- **Nil** - value of a class pointer that is not pointing to anything
- **NULL** - value of a non-object pointer that points to nothing
- **NSNull** - used to represent nothingness in collections

BOOL

- Objective C's boolean type
- **YES** represents true, **NO** represents false
- **NO** is defined as being equal to zero
- Since **nil** is also defined as zero we can test for objects existence like so:

```
if (obj) { /* do something if object exists */ }
```

Messaging

```
NSNumber *number = ...
int intValue =  [number intValue];
```

- The decision of what code to run when **intValue** is sent to **number** is made entirely at runtime
- Specifying a variable's type in code is used solely as a means of letting the compiler help you find bugs
- If a message is sent to an object that does not implement the given method an exception will be raised
- The name of each method is represented internally by a **Selector**
- Runtime resolution of messages is known as **Dynamic Binding**

SEL

- The ObjC type representing a **Selector**

- The **@selector** compiler directive takes the name of method and returns the **SEL** representing it

- Used in the target-action design pattern

```
[obj addTarget:self action:@selector(buttonPressed:)];
```

Protocols

- Protocols define a set of methods that an adopting object is expected to implement

- Define a protocol like so:

```
@protocol MyProtocol <NSObject>

@property (nonatomic, strong) NSString *name;

- (void)doSomethingInteresting;

@end
```

- Refer to an object that conforms to a protocol:

```
id<MyProtocol> obj = … // obj is an object
                       // that conforms to MyProtocol
```

Protocols

- Declare adopting a protocol:

```
@interace MyObject : NSObject <MyProtocol>

@end
```

- Protocols can have optional methods:

```
@protocol MyProtocol <NSObject>

@required
- (void)someRequiredMethod;

@optional
- (void)doSomethingInteresting;

@end
```

- Before calling an optional method, it is the responsibility of the caller to ensure that the object implements the method using introspection

Introspection

- We oftentimes need to determine the type and characteristics of an object at runtime so that we can safely use it, e.g. an object of type **id**

- Determining class membership:
  - **isKindOfClass:** returns whether an object is an instance of the given Class or any of its subclasses
  - **isMemberOfClass:** returns YES only when the object is an instance of the given class, not including subclasses

Introspection, part 2

- **respondsToSelector:** returns whether the object will respond to messages corresponding to the given selector

- **conformsToProtocol:** returns whether the object has adopted the given protocol

NSObject

- NSObject is the default root object used in Foundation and the iOS SDK
- It provides core functionality required by all objects
  - allocation: **+alloc**
  - initialization: **-init**
  - Hashing and equality testing
  - copying: **-copy** and **-mutableCopy**
  - introspection methods
  - **description**

Foundation

- Foundation is a collection of classes that provide a set of data structures and utilities that are the building blocks of your application
- **NS** class prefix dates back to NeXT
- Combined with UIKit, Foundation forms the core of the Cocoa Touch framework

NSString

- Represents an immutable array of Unicode characters
- @“” syntax for creating string constants in code:

```
NSString *str = @“Some String”;
```

- There is a mutable subclass: **NSMutableString**
  - Having Immutable mutable versions of a class is a common design pattern
  - In the case of **NSString**, its rare to use the mutable version, just create a new one when you need to make a modification

NSArray

- An immutable collection of ordered objects
- @[] syntax for creating arrays

```
NSArray *arrayOne = @[@“hello”, @“world”];
NSArray *arrayTwo = @[obj];
```

  - Crashes if obj is nil
  - Arrays can only hold objects

- Subscripting syntax for accessing members of an array:

```
id obj = array[2];
id otherObj = array[intVariable];
```

NSArray

- Fast enumeration syntax for iterating through elements:

```
for (NSString *str in array) {
     // do something interesting
}
```

- Basic methods for querying an array:

```
- (int)count; //returns the number of items in the array

- (id)objectAtIndex:(NSInteger)idx; // returns the object at index
                                    // idx will crash if idx >= count

- (id)firstObject; - (id)lastObject; // peeks at the first and
                                      // last objects in the array
                                       // respectively. wont crash
                                         // on empty array
- (NSInteger)indexOfObject:(id)obj // returns the idx of obj or
                                   // NSNotFound
```

NSMutableArray

- Mutable variant of NSArray
- Create an empty mutable array with **+array**
- Use **mutableCopy** to get a copy of an existing NSArray
- Use **copy** to create an immutable NSArray for an NSMutableArray
- Interacting with an mutable array:

```
- (void)addObject:(id)object // adds to end
- (void)removeLastObject;
- (void)insertObject:(id)obj atIndex:(NSInteger)idx;
- (void)removeObject:(id)obj atIndex:(NSInteger)idx;

mutableArray[3] = someObject;
```

NSDictionary

- Immutable key-value store
- @{} syntax for creating dictionaries:

```
NSDictionary *dict = @{@“key” : value,
                       @“otherKey” : otherVal};
```

- Accessing dictionaries with subscript syntax:

```
id value = dict[@“key”];
id otherValue = dict[otherKeyObj];
```

- Equality of keys is done through **-hash** and **-isEqual:** functions of NSObject

NSDictionary

- Fast enumeration iterates over the keys in the dictionary:

```
for (NSString *key in dict) {
    id value = dict[key];
    // do something interesting
}
```

- Values can’t be nil!

NSMutableDictionary

- Mutable variant of NSDictionary

- Create an empty dictionary with **+dictionary**

- Interacting with a NSMutableDictionary

```
dict[@“key”] = value
- (void)removeObjectForKey:(id)object
- (void)removeAllObjects;
```

Set Classes

- **NSSet** and **NSMutableSet**
  - Unordered collections of distinct objects
  - Very fast to determine whether an object exists
- **NSOrderedSet** and **NSMutableOrderedSet**
  - Ordered collection of distinct objects
- **NSCountedSet** (no separate immutable variant)
  - Keeps a count of the number of times an item is inserted

Other Objects

- **NSNumber** - Wraps C scalars into objects
  - Uses the @ syntax for creating instances in code

```
NSNumber *boolNum = @YES;
NSNumber *intNum = @4;
NSNumber *doubleNum = @(4.0 * 0.5);
```

- **NSData** - wraps a block of memory
- **NSDate** - Represents a date