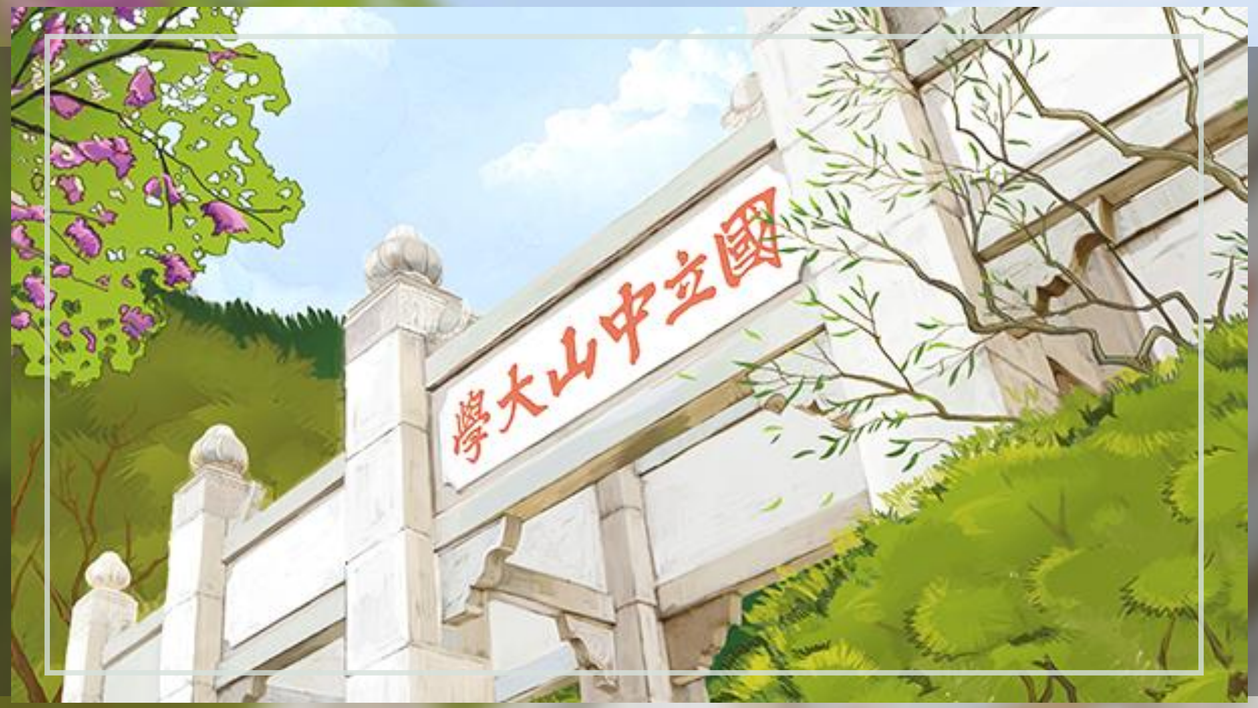
ByteDance 字节跳动



20秋

MOSAD

现代操作系统应用开发

#3 Objective-C进阶

Content

- Dynamic Binding
- Object Typing
- Introspection
- Protocols
- Categories & Extensions
- Blocks
- Foundation Framework
- Enumeration
- Property List

Dynamic Binding

- All objects are allocated in the heap, so you always use a pointer

  NSString *s = ⋯ ; // "statically" typed

  id obj = s; // not statically typed, but perfectly legal

  Never use "id *" (that would mean "a pointer to a pointer to an object").

- Decision about code to run on message send happens at runtime

  Not at compile time. None of the decision is made at compile time.

  Static typing is purely an aid to the compiler to help you find bugs.

  If neither the class of the receiving object nor its superclasses implements that method: crash!

- It is legal (and sometimes even good code) to "cast" a pointer

  But we usually do it only after we've used "introspection" to find out more about the object.

  id obj = ...;

  NSString *s = (NSString *)obj; // dangerous! best know what you are doing

Object Typing

```
@interface Vehicle
-(void)move;
@end
@interface Ship : Vehicle
-(void)shoot;
@end

Ship *s=[[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v=s;
[v shoot];
```

```
id obj = …;
[obj shoot];
[obj aMethodNoAnyObjectRespondsTo];

NSString *hello=@"hello";
[hello shoot];
Ship *helloShip=(Ship*)hello;
[helloShip shoot];
[(id)hello shoot];
```

Object Typing

```
@interface Vehicle
-(void)move;
@end
@interface Ship : Vehicle
-(void)shoot;
@end

Ship*s=[[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v=s;
[v shoot];
```

编译器无警告
s "isa" Vehicle

```
id obj = …;
[obj shoot];
[obj aMethodNoAnyObjectRespondsTo];

NSString *hello=@"hello";
[hello shoot];
Ship *helloShip=(Ship*)hello;
[helloShip shoot];
[(id)hello shoot];
```

Object Typing

```
@interface Vehicle
-(void)move;
@end
@interface Ship : Vehicle
-(void)shoot;
@end

Ship*s=[[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v=s;
[v shoot];
```

编译器无警告
s "isa" Vehicle,完全合法

```
id obj = …;
[obj shoot];
[obj aMethodNoAnyObjectRespondsTo];

NSString *hello=@"hello";
[hello shoot];
Ship *helloShip=(Ship*)hello;
[helloShip shoot];
[(id)hello shoot];
```

Object Typing

```
@interface Vehicle
-(void)move;
@end
@interface Ship : Vehicle
-(void)shoot;
@end

Ship*s=[[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v=s;
[v shoot];
```

编译器警告!
运行时不会crash，因为v "isa" Ship.
但编译器只知道v是Vehicle.

```
id obj = …;
[obj shoot];
[obj aMethodNoAnyObjectRespondsTo];

NSString *hello=@"hello";
[hello shoot];
Ship *helloShip=(Ship*)hello;
[helloShip shoot];
[(id)hello shoot];
```

Object Typing

```
@interface Vehicle
-(void)move;
@end
@interface Ship : Vehicle
-(void)shoot;
@end

Ship *s=[[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v=s;
[v shoot];
```

```
id obj = …;
[obj shoot];
[obj aMethodNoAnyObjectRespondsTo];

NSString *hello=@"hello";
[hello shoot];
Ship *helloShip=(Ship*)hello;
[helloShip shoot];
[(id)hello shoot];
```

编译器不警告！
编译器知道存在shoot函数，
obj有可能是Ship.
如果不是，运行时将crash!

Object Typing

```
@interface Vehicle
-(void)move;
@end
@interface Ship : Vehicle
-(void)shoot;
@end

Ship *s=[[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v=s;
[v shoot];
```

编译器警告!
编译器很肯定obj没有该函数!

```
id obj = …;
[obj shoot];
[obj aMethodNoAnyObjectRespondsTo];

NSString *hello=@"hello";
[hello shoot];
Ship *helloShip=(Ship*)hello;
[helloShip shoot];
[(id)hello shoot];
```

Object Typing

```
@interface Vehicle
-(void)move;
@end
@interface Ship : Vehicle
-(void)shoot;
@end

Ship *s=[[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v=s;
[v shoot];
```

编译器警告!
编译器很肯定没有该函数!

```
id obj = …;
[obj shoot];
[obj aMethodNoAnyObjectRespondsTo];

NSString *hello=@"hello";
[hello shoot];
Ship *helloShip=(Ship*)hello;
[helloShip shoot];
[(id)hello shoot];
```

Object Typing

```
@interface Vehicle
-(void)move;
@end
@interface Ship : Vehicle
-(void)shoot;
@end

Ship*s=[[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v=s;
[v shoot];
```

编译器不警告!
编译器认为程序员知道
自己在干嘛!

```
id obj = …;
[obj shoot];
[obj aMethodNoAnyObjectRespondsTo];

NSString *hello=@"hello";
[hello shoot];
Ship *helloShip=(Ship*)hello;
[helloShip shoot];
[(id)hello shoot];
```

Object Typing

```
@interface Vehicle
-(void)move;
@end
@interface Ship : Vehicle
-(void)shoot;
@end

Ship*s=[[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v=s;
[v shoot];
```

编译器不警告!
但运行时必然crash!

```
id obj = …;
[obj shoot];
[obj aMethodNoAnyObjectRespondsTo];

NSString *hello=@"hello";
[hello shoot];
Ship *helloShip=(Ship*)hello;
[helloShip shoot];
[(id)hello shoot];
```

编译器不警告!
但运行时必然crash!

Introspection 自检

- So when do we use id? Isn't it always bad?

  No, we might have a collection (e.g. an array) of objects of different classes.

  But we'd have to be sure we know which was which before we sent messages to them. How do we do that? Introspection.

- All objects that inherit from NSObject know these methods

  isKindOfClass: returns whether an object is that kind of class (inheritance included)

  isMemberOfClass: returns whether an object is that kind of class (no inheritance)

  respondsToSelector: returns whether an object responds to a given method

- Arguments to these methods are a little tricky

  Class testing methods take a Class

  You get a Class by sending the class method class to a class :)

```
if ([obj isKindOfClass:[NSString class]]) {
    NSString *s = [(NSString *)obj stringByAppendingString:@"xyzzy"];
}
```

Introspection

- Method testing methods take a selector (SEL)

  Special @selector() directive turns the name of a method into a selector

```
if ([obj respondsToSelector:@selector(shoot)]) {
    [obj shoot];
} else if ([obj respondsToSelector:@selector(shootAt:)]) {
    [obj shootAt:target];
}
```

- SEL is the Objective-C "type" for a selector

```
SEL shootSelector = @selector(shoot);
SEL shootAtSelector = @selector(shootAt:);
SEL moveToSelector = @selector(moveTo:withPenColor:);
```

Introspection

- If you have a SEL, you can ask an object to perform it

  Using the performSelector: or performSelector:withObject: methods in NSObject

  ```
  [obj performSelector:shootSelector];
  [obj performSelector:shootAtSelector withObject:coordinate];
  ```

  Using makeObjectsPerformSelector: methods in NSArray

  ```
  [array makeObjectsPerformSelector:shootSelector];
  [array makeObjectsPerformSelector:shootAtSelector withObject:target]; // target is an id
  ```

  In UIButton, - (void)addTarget:(id)anObject action:(SEL)action ...;

  ```
  [button addTarget:self action:@selector(digitPressed:) ...];
  ```

Protocols

- Protocols are defined in a header file.
- Declare methods and properties that are independent of any specific class.
- Similar to @interface, but someone else does the implementing
- NSObject协议包含了大部分NSObject类的方法
- 通过指定NSObject为子协议，使得很多协议的实现者基本上成为NSObject类

Foo.h

```
//实现Foo协议者也必须实现Other和NSObject协议
@protocol Foo<Other, NSObject>
//必须实现的方法（缺省是@required）
- (void) doSomething;
@optional //可实现、可不实现
- (int) getSomething;
- (void) doWithArgument: (NSString *)arg;
@required // 必须实现
(NSArray *)getBar:(int)bar;
//必须指定强弱（除非readonly）
@property (nonatomic, strong) NSString *barProp;
@end
```

Protocols

- Classes then say in their @interface if they implement a protocol

  导入协议头文件

  声明实现<协议>

- Must implement all non-optional methods.
- <协议>可用于声明id变量、id参数、id属性：表示该对象需要实现指定协议

MyClass.m

```
#import "Foo.h"
@interface MyClass : NSObject <Foo>
…
@end

id<Foo>obj= [[MyClass alloc] init];
id<Foo>obj= [NSArray array];//Warning!

- (void)giveMeFooObject:
        (id <Foo>) anObjectImplementingFoo;

@property (nonatomic, weak)
    id <Foo> myFooProperty;
//传参或赋值时，对象必须满足协议，否则警告
```

Protocols

- Just like static typing, this is all just compiler-helping-you stuff

  It makes no difference at runtime

- Think of it as documentation for your method interfaces

  It's another powerful way to leverage the id type

- #1 use of protocols in iOS: delegates and data sources （学完后面界面再回头理解）

A delegate or dataSource is pretty much always defined as a weak @property, by the way.

@property (nonatomic, weak) id <UISomeObjectDelegate> delegate;

This assumes that the object serving as delegate will outlive the object doing the delegating. Especially true in the case where the delegator is a View object (e.g. UIScrollView)

& the delegate is that View's Controller.

Controllers always create and clean up their View objects (because they are their "minions").

Thus the Controller will always outlive its View objects.

dataSource is just like a delegate, but, as the name implies, we're delegating provision of data.

Views commonly have a dataSource because Views cannot own their data!

Categories

- Categories are an Objective-C syntax for adding to a class . .

  Without subclassing it.

  Without even having to have access to the code of the class (e.g. its .m).

- Categories have their own .h and .m files (usually ClassName+PurposeOfExtension.[mh]).
- Categories cannot have instance variables, so no @synthesize allowed in its implementation.

Categories

PhotoAddOn.h

```
//为Photo类增加AddOn方法与属性
@interface Photo (AddOn)
- (UIImage *)image;
@property (readonly) BOOL isOld;
@end
```

Photo类中只保存了photoURL，
image函数把图片URL转换为图片UIImage.

isOld检查照片上传是否超过1天

PhotoAddOn.m

```
#import "PhotoAddOn.h"
@implementation Photo(AddOn)
- (UIImage*)image
{
  NSData *imageData =
  [NSData dataWithContentsOfURL:
    self.photoURL];
  return [UIImage imageWithData:
    imageData];
}
- (BOOL)isOld
{
  return [self.uploadDate
    timeIntervalSinceNow] < -24*60*60;
}
@end
```

Categories

- Categories常用于类型变换、数据格式处理
- Sometimes we add @propertys to an NSManagedObject subclass via categories to make accessing BOOL attributes (which are NSNumbers) cleaner.
- Or we add @propertys to convert NSDatas to whatever the bits represent.
- Any class can have a category added to it, but don't overuse/abuse this mechanism.

Categories

- Most common category on an NSManagedObject subclass?

  Creation

```
@implementation Photo (Create)
+ (Photo *)photoWithFlickrData:(NSDictionary *)flickrData
        inManagedObjectContext:(NSManagedObjectContext *)context
{
   Photo *photo = ...; // see if a Photo for that Flickr data is already in the database
   if (!photo) {
      photo = [NSEntityDescription insertNewObjectForEntityForName:@ “Photo”
                                                inManagedObjectContext:context];
      // initialize the photo from the Flickr data
      // perhaps even create other database objects (like the Photographer)
   }
   return photo;
}
@end
```

Extensions

- A class extension can only be added to a class for which you have the source code at compile time (the class is compiled at the same time as the class extension).
- The methods declared by a class extension are implemented in the @implementation block for the original class.
- Can't declare a class extension on a framework class, such as a Cocoa or Cocoa Touch class like NSString.
- Syntax (no name given, *anonymous categories*) :

```
@interface ClassName (){
… //这里定义私有实例变量
}
…//这里定义私有属性、方法
@end
```

Category vs. Extension

- Category
  - 不能增加实例变量
  - 增强类功能
  - 分离类实现
  - 可扩展框架类
- Extension
  - 增加实例变量
  - 主要用于接口分离
  - 私有化(private)属性和方法
  - 扩展原有类（源码重新编译，框架类无源码）

Blocks

- What is a block?

```
void (^simpleBlock)(void) = ^{
    NSLog(@"This is a block");
};
simpleBlock();
```

A language feature introduced to C, OC and C++.

It's a sequence of statements along with captured state.

Usually included "in-line" with the calling of method as argument.

Often used to simplify common tasks such as *collection enumeration*, *sorting* and *testing*.

They also make it easy to schedule tasks for concurrent or asynchronous execution using technologies like Grand Central Dispatch (GCD).

Blocks

- An example of calling a method that takes a block as an argument

```
[aDictionary enumerateKeysAndObjectsUsingBlock:^(id key, id value, BOOL *stop) {
    NSLog(@"value for key %@ is %@", key, value);
    if ([@"ENOUGH" isEqualToString:key]) {
        *stop = YES;
    }
}];
```

This NSLog()s every key and value in aDictionary (but stops if the key is ENOUGH).

Blocks

- Can use local variables declared before the block inside the block

```
double stopValue = 53.5;
[aDictionary enumerateKeysAndObjectsUsingBlock:^(id key, id value, BOOL *stop) {
    NSLog(@"value for key %@ is %@", key, value);
    if ([@"ENOUGH" isEqualToString:key] || ([value doubleValue] == stopValue)) {
        *stop = YES;
    }
}];
```

- But they are readonly!

Blocks

```
BOOL stoppedEarly = NO;  //Block外面定义的局部变量，Block内是只读的!
double stopValue = 53.5;
//初始化字典变量
NSDictionary *aDict = [NSDictionary dictionaryWithObjectsAndKeys:
    @"Value1", @"Key1", @"Value2", @"Key2", @53.5, @"Key3", @"Stop", @"ENOUGH", nil];
[aDict enumerateKeysAndObjectsUsingBlock:^(id key, id value, BOOL *stop) {
        NSLog(@"value for key %@ is %@", key, value);
        if ([@"ENOUGH" isEqualToString:key]) {
            *stop = YES;  //遇到ENOUGH为Key，跳出遍历
        }
        if ([obj doubleValue] == stopValue) {
            *stop = stoppedEarly = YES;  //遇到stopValue，提前跳出遍历。
                                         //但是给只读变量赋值是非法的!
        }
    }];
if (stoppedEarly) NSLog(@"I stopped logging dictionary values early!");
```

Blocks

- Unless mark the local variable as __block

```
__block BOOL stoppedEarly = NO;  //Block外面定义的局部变量，现在是可读写的!
double stopValue = 53.5;
//初始化字典变量
NSDictionary *aDict = [NSDictionary dictionaryWithObjectsAndKeys:
    @"Value1", @"Key1", @"Value2", @"Key2", @53.5, @"Key3", @"Stop", @"ENOUGH", nil];
[aDict enumerateKeysAndObjectsUsingBlock:^(id key, id value, BOOL *stop) {
        NSLog(@"value for key %@ is %@", key, value);
        if ([@"ENOUGH" isEqualToString:key]) {
            *stop = YES;  //遇到ENOUGH为Key，跳出遍历
        }
        if ([obj doubleValue] == stopValue) {
                *stop = stoppedEarly = YES;  //遇到stopValue，提前跳出遍历。现在合法了!
        }
    }];
if (stoppedEarly) NSLog(@"I stopped logging dictionary values early!");
```

- Or if the "variable" is an instance variable

But we only access instance variables (e.g. _display) in setters and getters. 用处不大:(

Blocks

- So what about objects which are messaged inside the block?

```
NSString *stopKey = [@"Enough" uppercaseString];
[aDictionary enumerateKeysAndObjectsUsingBlock:^(id key, id value, BOOL *stop) {
    NSLog(@"value for key %@ is %@", key, value);
    if ([stopKey isEqualToString:key]) {
        *stop = YES;
    }
}];
```

stopKey will essentially have a strong pointer to it until the block goes out of scope or the block itself leaves the heap (i.e. no one points strongly to the block anymore).

Blocks

- Creating a "type" for a variable that can hold a block

Blocks are kind of like "objects" with an unusual syntax for declaring variables that hold them.

```
typedef double (^unary_operation_t)(double op);
unary_operation_t  square; // Declare a variable, square, of type " unary_operation_t"
square = ^(double operand) { // the value of the square variable is a block
    return operand * operand;
}
```

Use the variable square like this:

```
double squareOfFive = square(5.0); // squareOfFive would have the value 25.0 after this
```

Another way to create square without typedef:

```
double (^square)(double op) = ^(double op) { return op * op; };
```

Blocks

- We could then use the unary_operation_t to define a method

  For example:

```
typedef double (^unary_operation_t)(double op);
- (void)addUnaryOperation:(NSString *)opKey
        whichExecutesBlock:(unary_operation_t)opBlock
{
    ……; //把block加入字典(键值表示的对象集合)
}
```

  Note that the block can be treated somewhat like an object (e.g., adding it to an array or dictionary).

Blocks

- We don't always typedef

For example:

~~typedef double (^unary_operation_t)(double op);~~

- (void)addUnaryOperation:(NSString *)opKey

whichExecutesBlock:(~~unary_operation_t~~ double (^)(double op))opBlock

{

……; //把block加入字典(键值表示的对象集合)

}

该类型没有名字

Note that the block can be treated somewhat like an object (e.g., adding it to an array or dictionary).

Blocks

- Some shorthand allowed when coding a block
  1. Do not have to declare the return type if it can be inferred from your code in the block.
  2. If no arguments to the block, you do not need to have any ().

```
NSNumber *secret = [NSNumber numberWithDouble:42.0];
[brain addUnaryOperation:@"MoLtUaE"
        whichExecutesBlock:^(double operand) {
    return operand * [secret doubleValue];
}];
```

- Another example . .

```
[UIView animateWithDuration:5.0
                 animations:^{
    view.opacity = 0.5;
}];
```

No arguments to this block.
No need to say ^(){ ... }.

Blocks

- Memory Cycles (a bad thing)

Property in a class:

```
@property (nonatomic, strong) NSArray *myBlocks; // array of blocks
```

And then tried to do the following in one of that class's methods?

```
[self.myBlocks addObject:^() {
    [self doSomething];
}];
```

Blocks keep a strong pointer to all objects referenced inside of them.

The block keep strong pointer to self.

And self keeps strong pointer to the block (through myBlocks property)

问题很严重!

Neither self nor the block can ever escape the heap now.

That's because there will always be a strong pointer to both of them (each other's pointer).

This is called a memory "cycle".

Blocks

Memory Cycles Solution with __weak

You'll recall that local variables are always strong.

That's okay because when they go out of scope, they disappear, so the strong pointer goes away.

But there's a way to declare that a local variable is weak.

```
__weak MyClass *weakSelf = self;
[self.myBlocks addObject:^() {
    [weakSelf doSomething];
}];
```

Problem solved!

The block keep ~~strong~~ weak pointer to self.

And self still keeps strong pointer to the block, but that's Okay.

Try to understand:

As long as someone in the universe has a strong pointer to this self, the block's pointer is good.

And since the block will not exist if self does not exist (since myBlocks won't exist), all is well!

Blocks

- When do we use blocks in iOS?
  - Enumeration
  - View Animations (more on that later in the course)
  - Sorting (sort this thing using a block as the comparison method)
  - Notification (when something happens, execute this block)
  - Error handlers (if an error happens while doing this, execute this block)
  - Completion handlers (when you are done doing this, execute this block)
- And a super-important use: Multithreading
  - With Grand Central Dispatch (GCD) API
  - Learn later in the course

Foundation Framework基础框架类

NSObject

Base class for pretty much every object in the iOS SDK

Implements introspection methods, etc.

```
-(NSString *)description
```

// is a useful method to override, use format "%@" to output in NSLog()

```
-(id)copy;
```

// not all objects implement mechanism (raises exception if not)

```
-(id)mutableCopy;
```

// not all objects implement mechanism (raises exception if not)

Foundation Framework

NSString

International (any language) strings using Unicode.

Used throughout iOS instead of C language's char * type.

Compiler will create an NSString for you using @"foo" notation.

An NSString instance can not be modified! They are immutable.

Usual usage pattern is to send a message to an NSString and it will return you a new one.

```
self.display.text = [self.display.text stringByAppendingString:digit];
self.display.text = [NSString stringWithFormat:@"%g", brain.operand]; // class method
```

Tons of utility functions available (case conversion, URLs, substrings, type conversions, etc.).

NSMutableString可变字符串类

Mutable version and subclass of NSString. Somewhat rarely used.

Can do some of the things NSString can do without creating a new one (i.e. in-place changes).

```
NSMutableString *ms = [[NSMutableString alloc] initWithString:@"0."];
NSMutableString *ms = [NSMutableString stringWithString:@"0."];
// stringWithString: is inherited from NSString
[ms appendString:digit];
```

Foundation Framework

NSNumber

Object wrapper around primitive types like int, float, double, BOOL, etc.

```
NSNumber *num = [NSNumber numberWithInt:36];
float f = [num floatValue]; // would return 36 as a float (i.e. will convert types)
```

Useful when you want to put these primitive types in a collection (e.g. NSArray or NSDictionary).

NSValue

Generic object wrapper for other non-object data types.

```
CGPoint point = CGPointMake(25.0, 15.0); // CGPoint is a C struct
NSValue *pointObject = [NSValue valueWithCGPoint:point];
```

Foundation Framework

- NSData

  "Bag of bits." Used to save/restore/transmit data throughout the iOS SDK.

- NSDate

  Used to find out the time right now or to store past or future times/dates.

  See also NSCalendar, NSDateFormatter, NSDateComponents.

Foundation Framework

NSArray

Ordered collection of objects.

Immutable. That's right, once you create the array, you cannot add or remove objects.

```
+ (id)arrayWithObjects:(id)firstObject, ...; // nil-terminated arguments
NSArray *primaryColors = [NSArray arrayWithObjects:@"red", @"yellow", @"blue", nil];
+ (id)arrayWithObject:(id)soleObjectInTheArray; // more useful than you might think!
- (int)count;
- (id)objectAtIndex:(int)index;
- (id)lastObject; // returns nil (doesn't crash) if there are no objects in the array
- (NSArray *)sortedArrayUsingSelector:(SEL)aSelector;
- (void)makeObjectsPerformSelector:(SEL)aSelector withObject:(id)selectorArgument;
- (NSString *)componentsJoinedByString:(NSString *)separator;
- (BOOL)containsObject:(id)anObject; // could be slow, think about NSOrderedSet
```

Foundation Framework

NSMutableArray

Mutable version of NSArray.

```
+ (id)arrayWithCapacity:(int)initialSpace; // initialSpace is a performance hint only
+ (id)array;

-(void)addObject:(id)anObject; // at the end of the array
-(void)insertObject:(id)anObject atIndex:(int)index;
-(void)removeObjectAtIndex:(int)index;
-(void)removeLastObject;

-(id)copy; // returns an NSArray (i.e. immutable copy); and NSArray implements mutableCopy
```

Don't forget that NSMutableArray inherits all of NSArray's methods

e.g. count, objectAtIndex:, etc.

Foundation Framework

NSDictionary字典类

Immutable hash table. Look up objects using a key to get a value.

```
+ (id)dictionaryWithObjects:(NSArray *)values forKeys:(NSArray *)keys;
+ (id)dictionaryWithObjectsAndKeys:(id)firstObject, ...;
e.g. NSDictionary *base = [NSDictionary dictionaryWithObjectsAndKeys:
                              [NSNumber numberWithInt:2], @"binary",
                              [NSNumber numberWithInt:16], @"hexadecimal", nil];
-(int)count;
-(id)objectForKey:(id)key;
-(NSArray *)allKeys;
-(NSArray *)allValues;
```

Keys are sent - (NSUInteger)hash & - (BOOL)isEqual:(NSObject *)obj for unique identification.

NSObject returns the object's pointer as its hash and isEqual: only if the pointers are equal.

Keys are very often NSStrings (they hash based on contents and isEqual: if characters match).

Foundation Framework

NSMutableDictionary可变字典类

Mutable version of NSDictionary.

```
+ (id)dictionary; // creates an empty dictionary
                  // (don't forget it inherits + methods from super)
- (void)setObject:(id)anObject forKey:(id)key;
- (void)removeObjectForKey:(id)key;
- (void)removeAllObjects;
- (void)addEntriesFromDictionary:(NSDictionary *)otherDictionary;
```

Foundation Framework

NSSet

Immutable, unordered collection of distinct objects.

Can't contain multiple "equal" objects at the same time (for that, use NSCountedSet).

```
+ (id)setWithObjects:(id)firstObject, ...;
+ (id)setWithArray:(NSArray *)anArray;
- (int)count;
- (BOOL)containsObject:(id)anObject;
- (id)anyObject;
- (void)makeObjectsPerformSelector:(SEL)aSelector;
```

Foundation Framework

NSMutableSet

Mutable version of NSSet.

- (void)addObject:(id)anObject;

// does nothing if object that isEqual:anObject is already in

- (void)removeObject:(id)anObject;
- (void)unionSet:(NSSet *)otherSet;
- (void)minusSet:(NSSet *)otherSet;
- (void)intersectSet:(NSSet *)otherSet;

Foundation Framework

NSOrderedSet

Immutable ordered collection of distinct objects.

Sort of a hybrid between an array and a set.

Faster to check "contains" than an array, but can't store an (isEqual:) object multiple times.

Not a subclass of NSSet! But contains most of its methods plus

- (int)indexOfObject:(id)anObject;
- (id)objectAtIndex:(int)anIndex;
- (id)firstObject; and - (id)lastObject;
- (NSArray *)array;
- (NSSet *)set;

Foundation Framework

NSMutableOrderedSet

Mutable version of NSOrderedSet.

- (void)insertObject:(id)anObject atIndex:(int)anIndex;

- (void)removeObject:(id)anObject;

- (void)setObject:(id)anObject atIndex:(int)anIndex;

Enumeration枚举（遍历）

- Looping through members of a collection in an efficient manner

  Language support using for-in.

  Example: NSArray of NSString objects

```
NSArray *myArray = ...;
for (NSString *string in myArray) { // no way for compiler to know what myArray contains
    double value = [string doubleValue]; // crash here if string is not an NSString
}
```

  Example: NSSet of id (could just as easily be an NSArray of id)

```
NSSet *mySet = ...;
for (id obj in mySet) {
    // do something with obj, but make sure you don't send it a message it does not respond to
    if ([obj isKindOfClass:[NSString class]]) {
        // send NSString messages to obj with impunity
    }
}
```

Enumeration

- Looping through the keys or values of a dictionary

  Example:

```
NSDictionary *myDictionary = ...;
for (id key in myDictionary) {
    // do something with key here
    id value = [myDictionary objectForKey:key];
    // do something with value here
}
```

Property List

- The term "Property List" just means a collection of collections

  Specifically, it is any graph of objects containing only the following classes:

  NSArray, NSDictionary, NSNumber, NSString, NSDate, NSData

- An NSArray is a Property List if all its members are too

  So an NSArray of NSString is a Property List.

  So is an NSArray of NSArray as long as those NSArray's members are Property Lists.

- An NSDictionary is one only if all keys and values are too

  An NSArray of NSDictionarys whose keys are NSStrings and values are NSNumbers is one.

- Why define this term?

  Because the SDK has a number of methods which operate on Property Lists.

  Usually to read them from somewhere or write them out to somewhere.

  `[plist writeToFile:(NSString *)path atomically:(BOOL)]; // plist is NSArray or NSDictionary`

Other Foundation

NSUserDefaults

Lightweight storage of Property Lists.

It's basically an NSDictionary that persists between launches of your application.

Not a full-on database, so only store small things like user preferences.

Read and write via a shared instance obtained via class method standardUserDefaults

```
[[NSUserDefaults standardUserDefaults] setArray:rvArray forKey:@"RecentlyViewed"];
```

Sample methods:

```
- (void)setDouble:(double)aDouble forKey:(NSString *)key;
- (NSInteger)integerForKey:(NSString *)key; // NSInteger is a typedef to 32 or 64 bit int
- (void)setObject:(id)obj forKey:(NSString *)key; // obj must be a Property List
- (NSArray *)arrayForKey:(NSString *)key; // will return nil if value for key is not NSArray
```

Always remember to write the defaults out after each batch of changes!

```
[[NSUserDefaults standardUserDefaults] synchronize];
```

Summary

- 顺利~~渡过~~学完第3~~劫~~节！
- 更多OC内容在后续学习中穿插学习
- 下周开始介绍iOS平台及应用设计模式等内容
- 继续完成第一次作业，最迟在第6周实验课完成验收