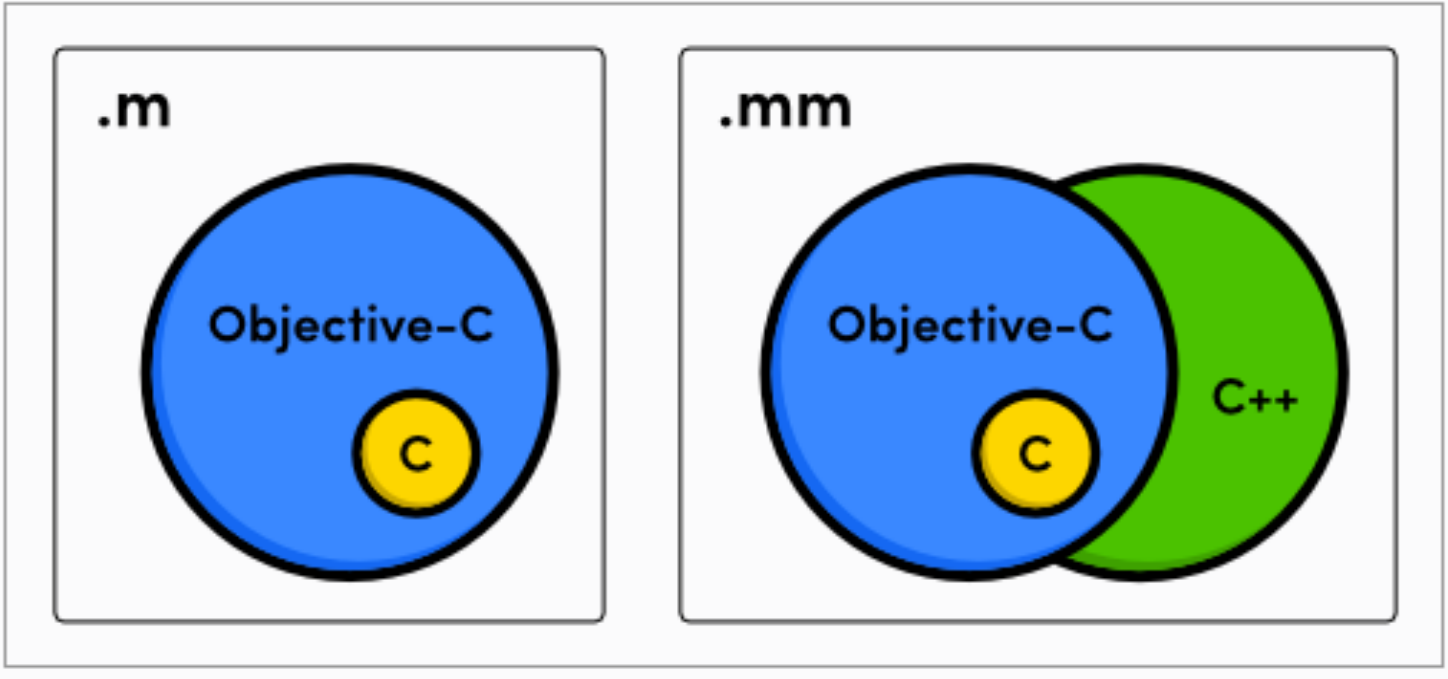
Komprehensive Learning!

Why Objective C?

- Originally developed in 1980s
  - Objective-C is the primary language used by Apple's Cocoa and Cocoa Touch APIs
  - C programming language – 1972
  - Smalltalk – object oriented language in 1980 – Is older than C ++ language
  - NeXTStep(OS in 1990s) built on Objective C – Apple bought Next in 1996
  - NeXTStep is foundation for Mac OS X
- Strict superset of C language
  - Mix C or C ++ with Objective C
  - Adds object oriented constructs to C language
  - When compiling, largely gets turned into C
  - Objective-C compiler will compile C, C++ and Objective-C source code

C,C++,Objective C, Swift



Languages available to files with .m and .mm extensions

LANGUAGE FUNDAMENTALS!

Data Types, Variables & Naming

What is a "variable"?

- A variable is a location in RAM reserved for storing data used by an application. Each variable is given a name, assigned a type, and assigned a value. The name can be used by the source code to access the value assigned to the variable, either to read it or change it.

What is a "variable"?

- A variable is a symbolic name given to a block of memory that holds information in memory that can be changed Variables are created by "declaring" them In Objective-C, variables are typically declared at the class level (available to all methods in the class), or at the method level (available only inside the method where they are declared). The availability of a variable is called its "scope".
- General Rule: a variable is available only within the code block where it is declared
- Variables declared in the attribute section of a class definition (header file) are available to all methods within that class Variables declared inside a method are available only to that method

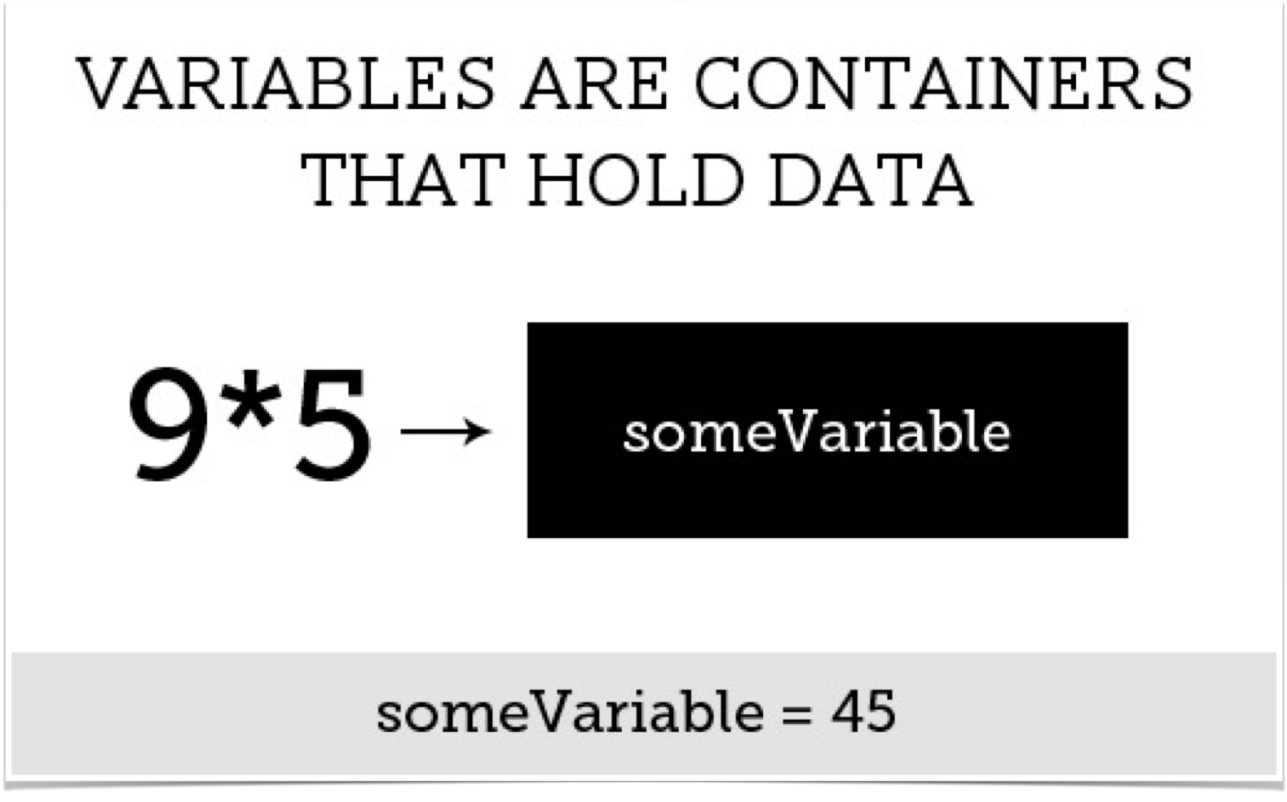
Defining Variables

```
int     x ;
type    name
```

Defining Variables

```
int      x   =  100;
type    name     value
```

Variables


VARIABLES ARE CONTAINERS
THAT HOLD DATA
9*5 →
someVariable
someVariable = 45

Variable Naming

- Rules for declaring a variable
  - It can consist of alphabets, digits and special characters i.e $ and _
  - First character can’t be a numeric value
  - Keywords can’t be used as an Identifier

Data types

- int
- float
- double
- char
- BOOL
- id

Int

- Whole Numbers
  - 1, 43, 1000000
- Declaration
  - int age = 23;
- Can be signed or unsigned
- More bytes = larger range
- short<=int<=long<=long long

Int

- 2 categories of Integers
  - Architecture dependent types
  - Explicit sized types

Architecture Dependent

Type	32-bit size	64-bit size
char	1	1
short	2	2
int	4	4
long	4	8
long long	8	8

Architecture Dependent

Type	32-bit size	64-bit size
char	1	1
short	2	2
int	4	4
long	4	8
long long	8	8

Integer data types

Type	32-bit size	64-bit size
NSInteger	4	8
NSUInteger	4	8

Explicit Sized Types

Type	32-bit size	64-bit size
int8_t	1	1
int16_t	2	2
int32_t	4	4
int64_t	8	8

NSInteger/NSUInteger

```
#if __LP64__ || TARGET_OS_EMBEDDED || TARGET_OS_IPHONE || TARGET_OS_WIN32 || NS_BUILD_32_LIKE_64
typedef long NSInteger;
typedef unsigned long NSUInteger;
#else
typedef int NSInteger;
typedef unsigned int NSUInteger;
#endif
```

Format Specifiers

Type	Format Specifier
8-bit unsigned	%c
32-bit signed integer	%d
32-bit unsigned integer	%u %x %X %o %O
Type	**modifier**
Following d,o,u,x specifier applies short	h
Following d,o,u,x specifier applies long	l
Following d,o,u,x specifier applies long long	ll

Floating Point

- Decimal point numbers
  - 3.14 , 65.544
- Very Large numbers
  - 5.7 * (2^50)
- Sign Mantissa * Base ^Exponent

Floating Point

Type	32-bit size	64-bit size
float	4	8
double	8	8
CGFloat	4	8

CHAR

- Store a single character such as digit, letters or special characters
  - char ch = ‘a’ ;
- Escape sequences
  - \n,\b,\t, etc.

CGFloat

```
typedef float CGFloat;// 32-bit
typedef double CGFloat;// 64-bit
```

BOOL

- C bool – type from C99 standard(int)
  - true/false
- Objective C BOOL is signed char
  - YES/NO
- **bool ! = BOOL**

Bool,Boolean,C Bool

Name	Type	Header	True	False
BOOL	signed char / bool	objc.h	YES	NO
bool	_Bool (int)	stdbool.h	TRUE	FALSE
Boolean	unsigned char	MacTypes.h	TRUE	FALSE
NSNumber	__NSCFBoolean	Foundation.h	@(YES)	@(NO)

C FOR OBJECTIVE C

C Storage Classes

- Auto
- Register
- Extern
  - extern makes functions/variables visible globally to all files.
- Static
  - A static variable inside a method or function retains its value between invocations.
  - A static variable declared globally can called by any function or method, so long as those functions appear in the same file as the static variable.

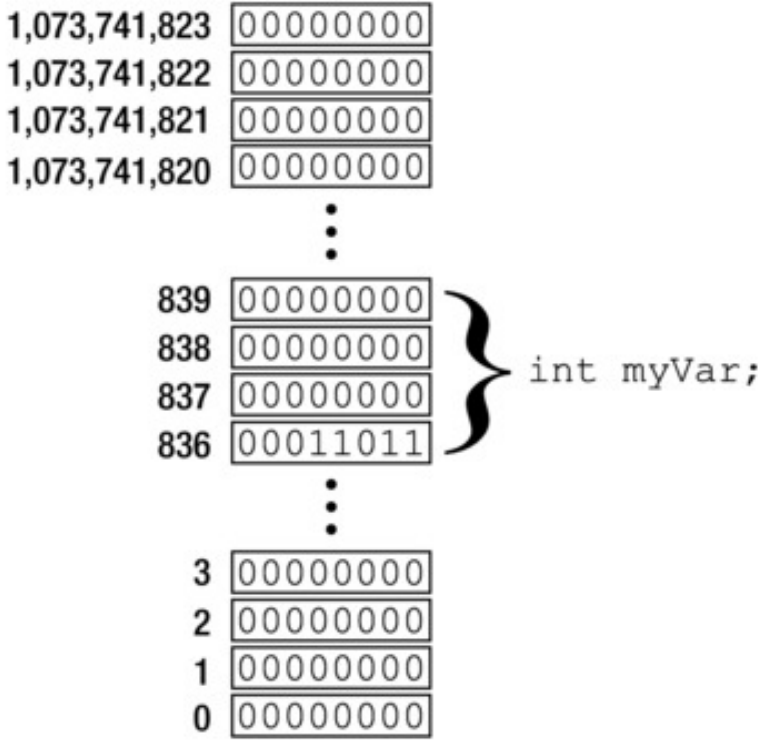
Pass by value

DEMO

Pointers

- Pointers are address variables.
- 1 GB RAM = 2^{30} bytes = 1,073,741,824 bytes of memory
- Every one of those bytes has a unique address.
- All variables in our program takes one of those addresses

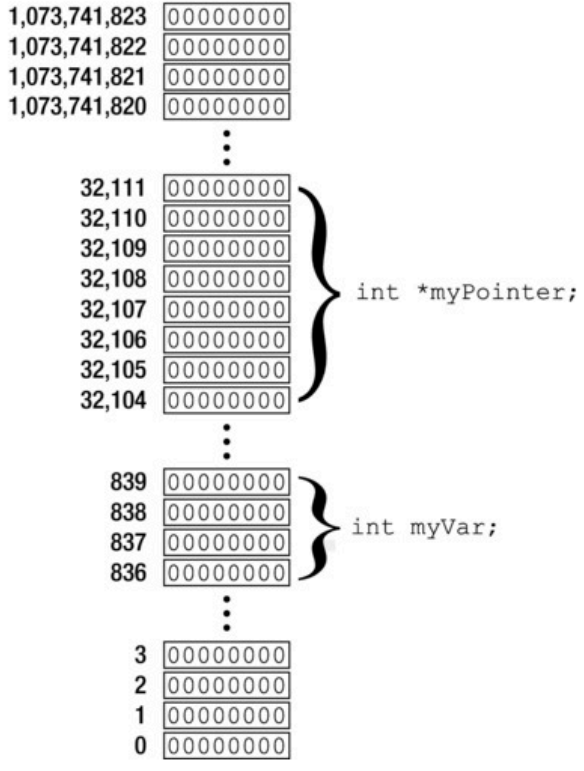
Memory allocated for myVar



The & * operator

- The & operator, called the address-of operator
- &myVar refers to myVar's address in memory.
- int *myPointer;
- myPointer = &myVar;
- The complement to the address-of (&) operator is the unary * or indirection operator. It does just the opposite of the & operator: it takes a pointer variable and turns it into the variable the pointer variable points to.

Memory allocated for integer pointer



Pass-By-Value vs. Pass-By-Reference

- In pass-by-value, a copy of the value is passed to the function and the function can do anything it wants to with its copy; it will never affect another variable in the program.

- In pass-by-reference, instead of copying the value, a reference to the value is passed to the function. The function can use this reference to access and/or modify the original value at will.

Pointers and Arrays

- A special relationship exists between pointers and arrays.
- An array name without brackets is a pointer to the array's first element.
- So, if a program declared an array,

  – `char chName[10];`

- `chName` (array's name) is the address of the first array element and is equivalent to the expression,

  – `&chName[0]`

- Which is a reference to the address of the array's first element.
- `chName` equivalent to `&chName[0]`.
- The array's name is, therefore a pointer to the array's first element and therefore to the string if any.

Structures

- A ***struct*** is also a collection of data items, except with a struct the data items can have different data types, and the individual ***fields*** within the struct are accessed ***by name*** instead of an integer index.
- Structs are very powerful for bundling together data items that collectively describe a thing, or are in some other way related to each other.

Struct example

```
struct mystruct {

char name[32];

int age;

char *addr;

};
struct mystruct m1, m2;
struct mystruct person; // uninitialized

struct mystruct person2 = { // initialization

.name = {’ f’ ,’ o’ ,’ o’ ,’ \0’ },

.age = 22,

.addr = NULL

};
```

Enums

- Basically integers
- Can use in expressions like ints
- Makes code easier to read
- Cannot get string equiv.
- caution: day++ will always add 1 even if enum
- values aren’t contiguous.

Enums

```
enum Weekdays {
    Monday,
    Tuesday,
  Wednesday,
    Thursday,
    Friday
};
enum Weekdays whichDay;
whichDay = Thursday;
enum Colors {
    red,
green = 5,
    blue,
    magenta,
    yellow = blue + 5
} myColor;
myColor = blue;
```

Enums

```
enum days day;

// Same as: int day;

for(day = mon; day <= sun; day++) {

if (day == sun) {

printf("Sun\n");

} else {

printf("day = %d\n", day);

}

}
```

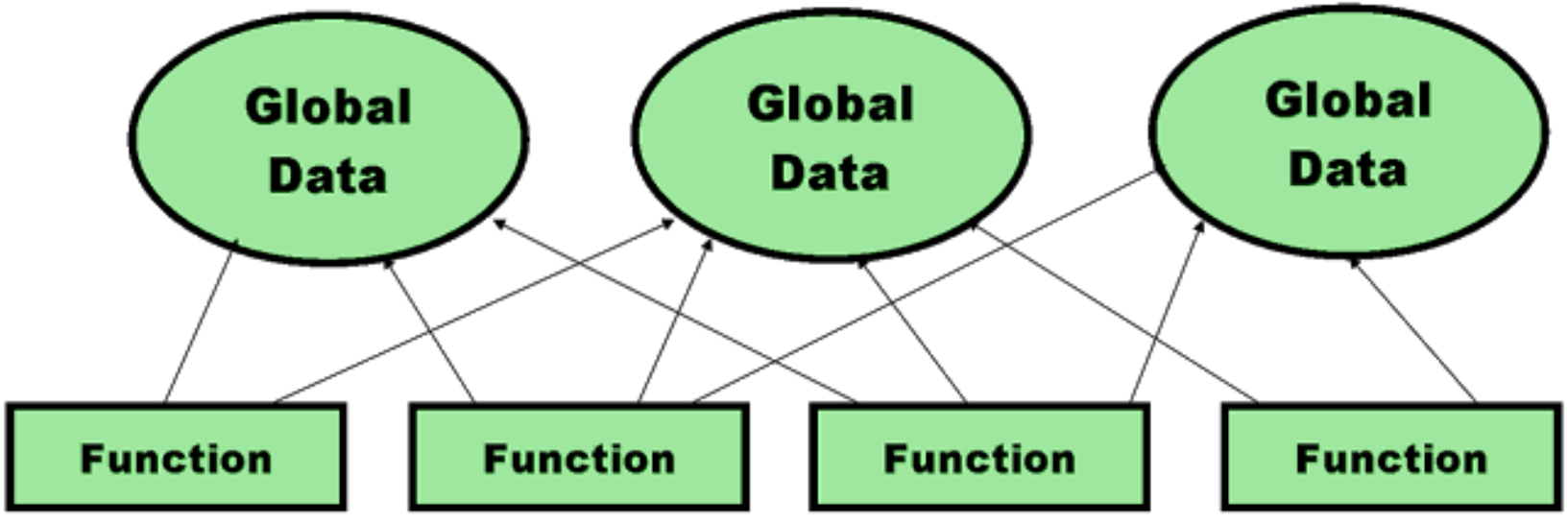
Typedef Structs

```
typedef struct DVDInfoTag {
    char rating;
    char title[ kMaxTitleLength ];
    char comment[ kMaxComLength ];
    struct DVDInfoTag *next;
} DVDInfo;
DVDInfo oneDVD;
```

Object oriented programming

Problems with Structured programming

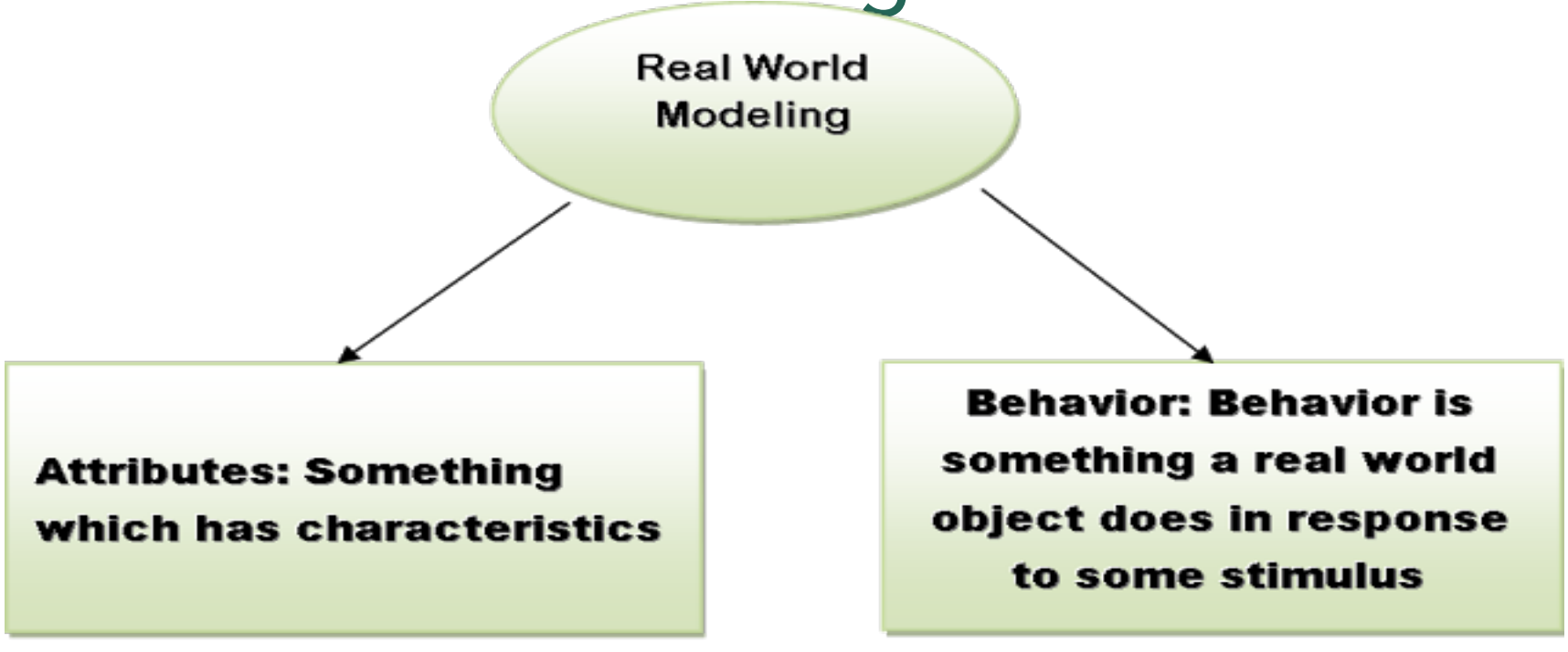
- Unrestricted Access



The Procedural Paradigm

Problems (cont.)

- Real World Modeling

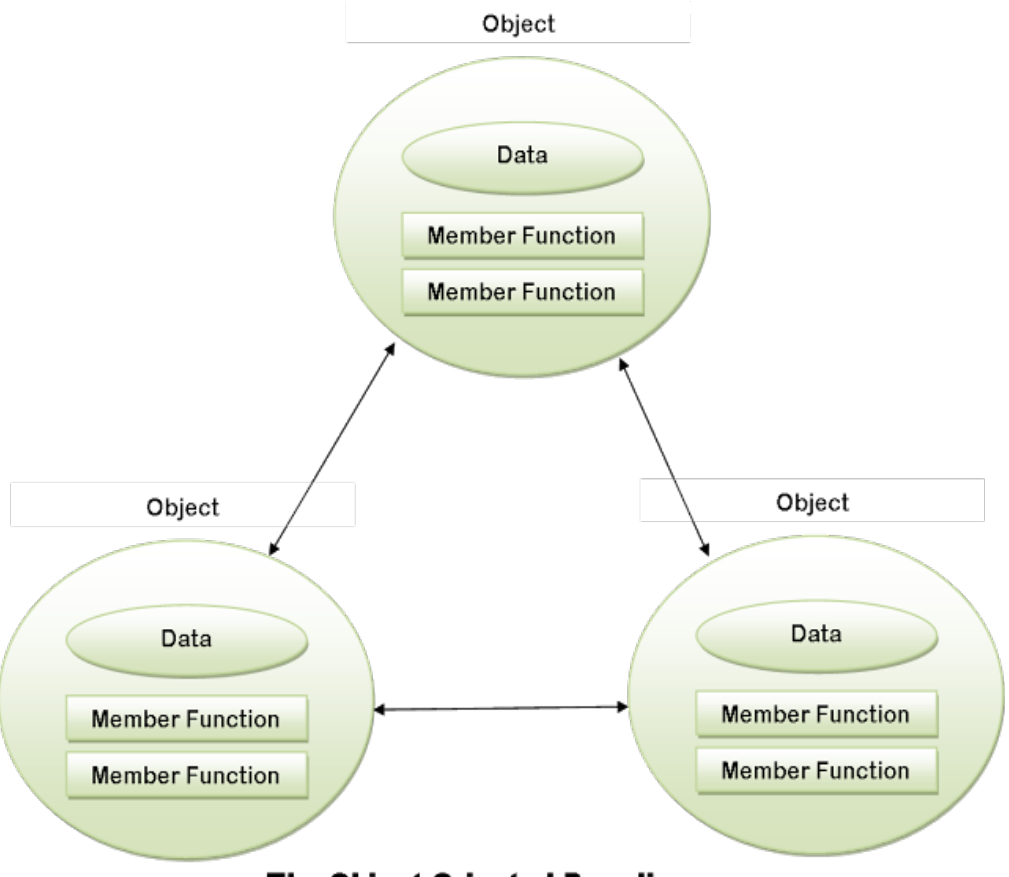


The Real World Paradigm

Procedural Programming language

- Program is build around functions
- Focus is not on Data but on functions as well as sequence of actions to do
- Adding new data and functionality is not so easy
- Data can move freely from one to another function
- Overloading, dynamic binding, abstraction, inheritance, polymorphism and encapsulation like modern programming features are supported .
- Very difficult to develop enterprise applications.
- Example
  - C, Pascal, FORTRAN

Object Oriented Programming Language



The Object-Oriented Paradigm

Object Oriented Programming Language

- Program is build around objects
- Importance given to data rather than functions or procedures
- Objects can move or communicate with each other through member functions
- Easier to add new data and functions
- Overloading, dynamic binding, abstraction, inheritance, polymorphism and encapsulation like modern programming features not supported .
- Easy to develop real world enterprise applications
- Example
  - Java, C#,C++,Objective-C

Real Difference

- If you want to change the struct or global variables, you have to modify all the functions which are using them
- Can be solved by oop-encapsulation
- Object- packages together data with the particular operations that can use or affect that data
- We can move the struct and function prototypes into a class

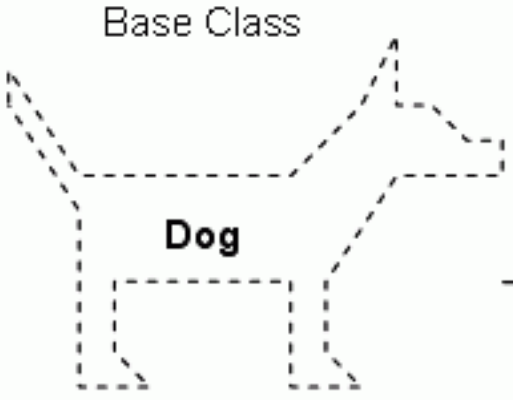
Classes are blueprints for objects

- A class is meant to define an object and how it works
- Class is a template having both data and behavior(methods) affecting or using the data
- It consist of instance variables and methods
- Instance variables(iVars)/Attributes → data elements in a class
- Methods(functions, procedures) → implements the behaviors of the class

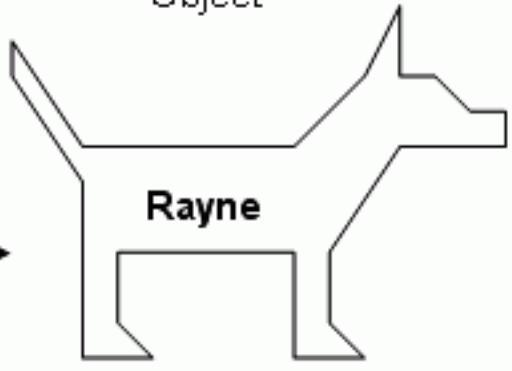
Object

- Class definition is a template for an object because
  - It declares instance variables that becomes part of each object
- Class represents an object type
- Analogy –Class is the blueprint and object is the house
- An application is made up of lots of objects and they communicate messages among themselves
- Each object will have its own set of instance variables/attributes but shares methods

Example



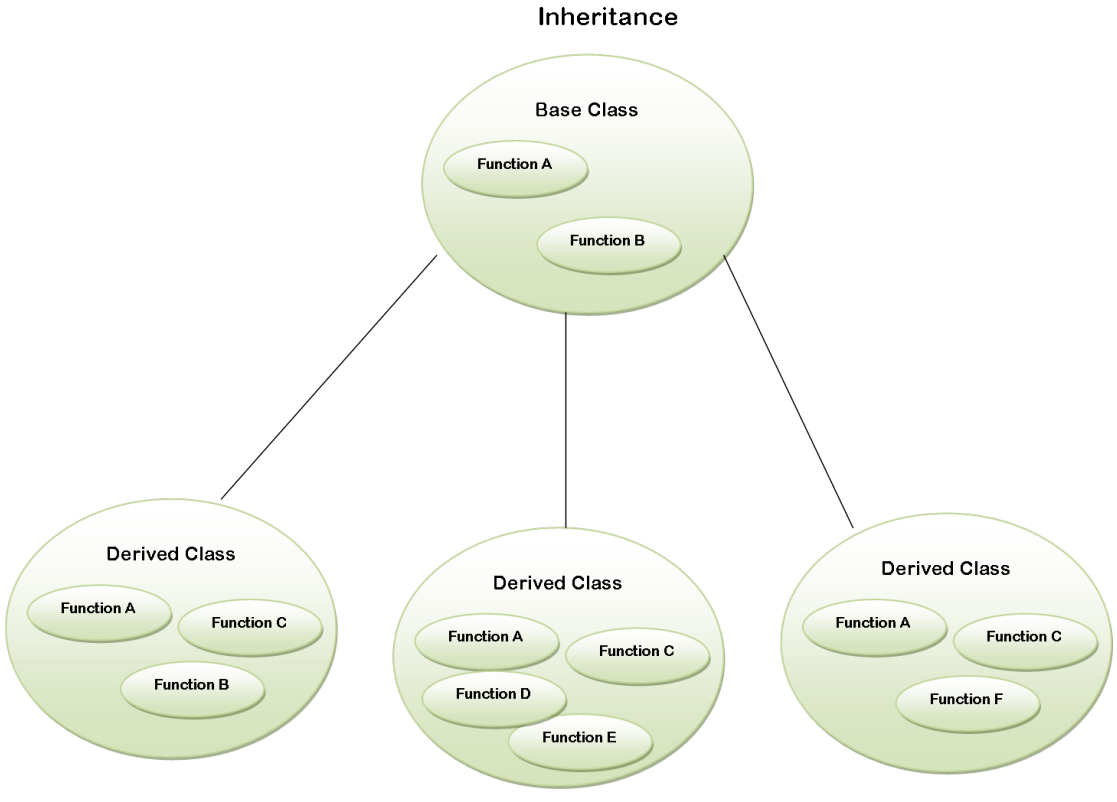
Create Instance



Properties	**Methods**
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property values	**Methods**
Color: Gray, White, and Black	Sit
Eye Color: Blue and Brown	Lay Down
Height: 18 Inches	Shake
Length: 36 Inches	Come
Weight: 30 Pounds	

Inheritance



Inheritance

- Inheritance is a fundamental Object Oriented concept
- A class can be defined as a "subclass" of another class.
  - The subclass inherits all data attributes of its superclass
  - The subclass inherits all methods of its superclass
  - The subclass inherits all associations of its superclass
- The subclass can:
  - Add new functionality
  - Use inherited functionality
  - Override inherited functionality

Defining a Class

- In Objective-C, classes are defined in two parts:
  - An *interface* (.h) that declares the methods and instance variables of the class and names its super class
  - An *implementation* (.m) that actually defines the class (contains the code that implements its methods)

@INTERFACE

```
@interface ClassName : ParentClassName
{
declare member variable here;
declare another member variable here;
}
declare method functions here;
@end
```

- Goes in header (.h) file
- Functions outside curly braces
- Don’t forget the @end tag

Declaration

Instance Variables:

```
float width;
float height;
BOOL filled;
NSColor *fillColor;
```

Methods:

- names of methods that can be used by class objects, class methods, are preceded by a plus sign

```
+ alloc
```

- methods that instances of a class can use, instance methods, are marked with a minus sign:

```
- (void) display;
```

Declaration (cont.)

- **Importing the Interface:** The interface is usually included with the **#import** directive

```
#import "Rectangle.h"
```

- To reflect the fact that a class definition builds on the definitions of inherited classes, an interface file begins by importing the interface for its super class
- **Referring to Other Classes:** If the interface mentions classes not in this hierarchy, it must declare them with the @class directive:

```
@class Rectangle, Circle;
```

@IMPLEMENTATION

```
#import <Foundation/Foundation.h>
#include “Example.h“

@implementation ClassName
define method function here;
define another method function here;
define yet another method function here;
@end
```

Goes in source (.m) file
Don't forget the @end tag

Object Creation

- Objects are created in a two-step process: allocation (alloc) and initialization (init)
- Allocation sets aside memory for the object; initialization sets base values for the object variables

Methods

Instance Method (-) or Class method (+)	**(return type)** Can be Class objects or primitive types	Method **name** part	**: (param type)** Can be Class objects or primitive types	Param name

- (type)someMethod;
- (type)someMethodWithAParam: (type)param;
- (type)someMethodWithAParam: (type)param withBParam:(type)param;

Class Method vs. Instance Method

- Class Method
  - A class method is static and associated with a class
  - Denoted by a "+" sign in method declaration
  - e.g., in the class ClassA, we can add a method **`printClassName`** that prints the class name on the screen
  - Invoke class method by `[ClassA printClassName];` `(use the class name directly)`
  - Since this method does not require association with a particular object, so **`printClassName`** should be a class method

Class Method vs. Instance Method

- Instance method
  - An instance method associates with an object
  - Denoted by a "-" sign in method declaration
  - e.g., we call the **move** method on a particular ***object*** by
  - **//create an object called myObject of type ClassA**
  - `[myObject move: 10.0: 5.0]; (myObject is an OBJECT, not the class name)`
  - Since every object can move differently, so **move** should be an instance method

Id data type

- Generic type for all Objective-C objects
- It can store a reference to any type of object

```
id mysteryObject = @"An NSString object";
id mysteryObject = @"An NSString object";
NSLog(@"%@", [mysteryObject description]);
mysteryObject = @{@"model": @"Ford",
@"year": @1967};
NSLog(@"%@", [mysteryObject description]);
```

Properties

- Generating accessor methods
- Synthesizing accessor methods
- No more synthesizing required

Initializers

- Default Initializer
- NSObject init method
- Self and nil
- Designated iniitializer

Init Method Inheritance

- Almost all Objective-C classes inherit ultimately from NSObject
- As a result, all Objective-C classes have some common initialization methods
- Initialization methods can be customized for your declared class by overriding inherited methods, or writing new initialization methods

Return Type of Init

- **Initialization methods should be typed to return an id rather than a pointer to a specific instance of the class**
  - an init method may release its receiver and substitute an object belonging to a subclass (very common with class clusters)
  - typing the return value as id allows the init method to be inherited and used by child classes of the original class

The Form of init

```
(id) init
{
    if (self = [super init])
    {
        //perform class specific init tasks
    }
    return self;
}
```

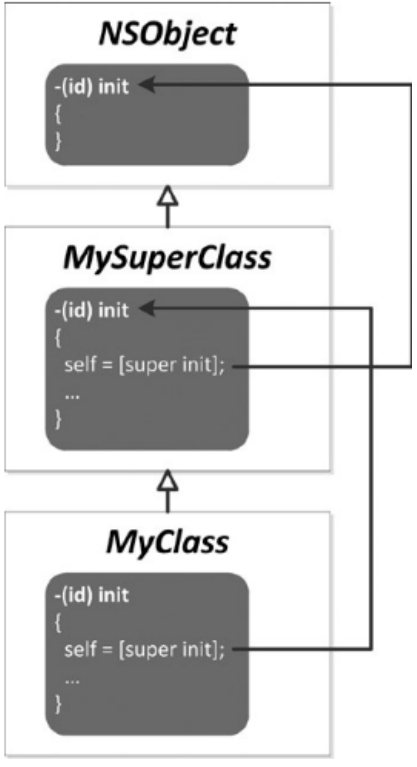
Naming Init Methods

- **When declaring custom init methods for your class, the universal naming convention is to begin the method name with "init", i.e. initWithString, initWithFloat, etc.**
- Or Else complier will consider it as a normal instance method

Initializers with Arguments

- **Initializers that take one or more arguments can be used to initialize your class's instance variables to a specific value**
  - If not set to a specific value, all instance variables are set to zero or nil (depending on their type) by the alloc method when the object is created.

Object Initialization By init methods



Designated Initializer

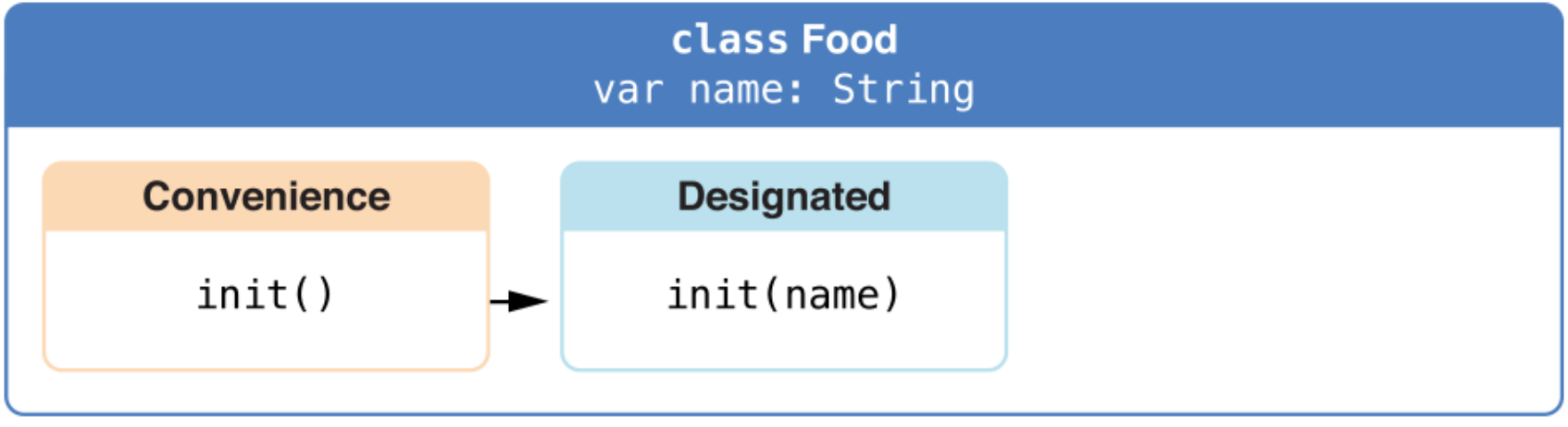
- Classes can have multiple initialization methods that will initialize the instance variables in different ways
- A class can, however, have only one "designated initializer"

- The designated initializer is the initialization method that completely initializes an instance of the class
- The designated initializer is usually the initializer that:
  - has the most arguments, or
  - does the most work in setting up the object
  - For classes that don't require initializers with arguments, the designated initializer is init

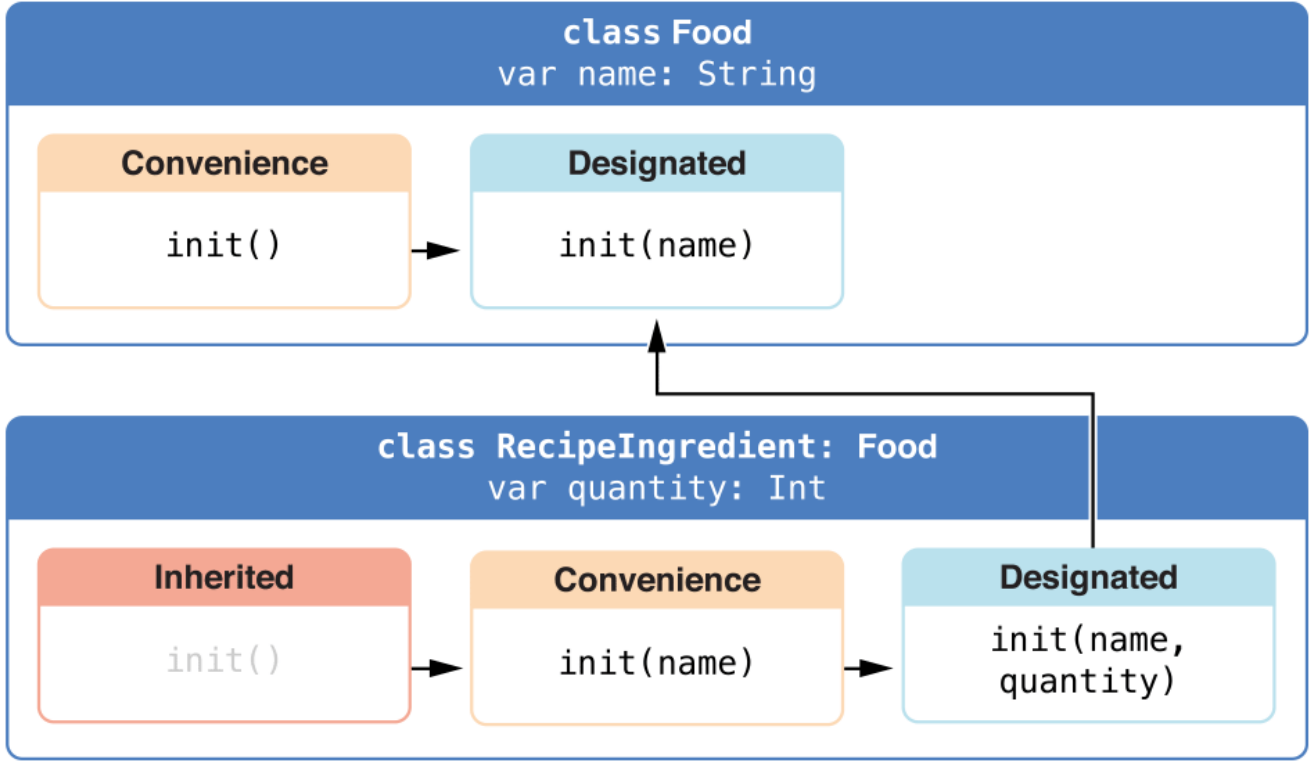
Rules for Designing Initializers

- A class's designated initializer must invoke the designated initializer of its superclass
- All other class initializers must eventually invoke the class's designated initializer

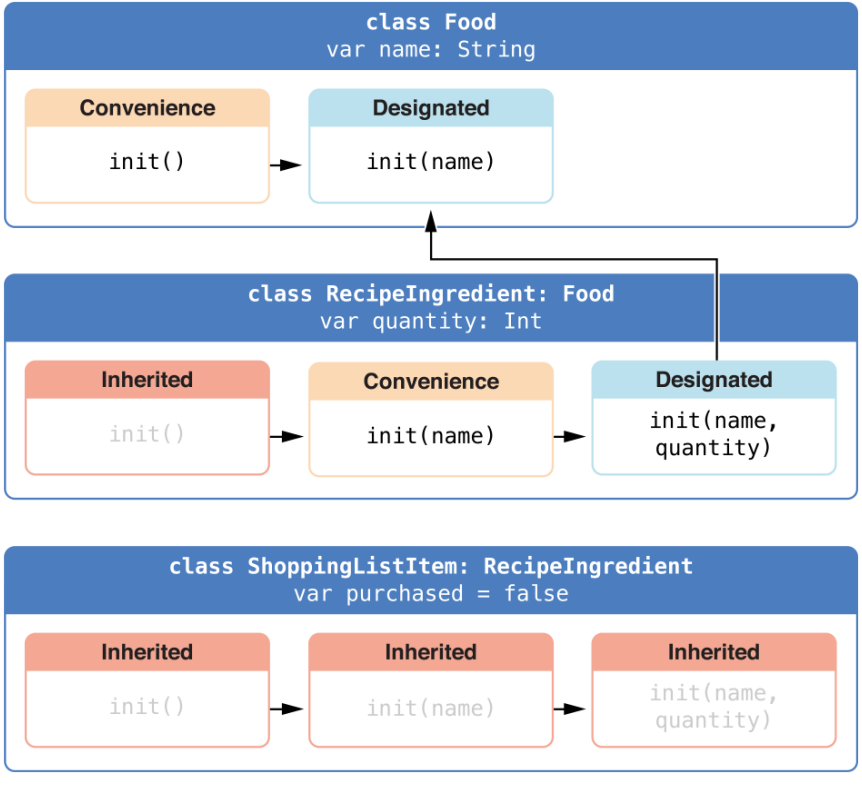
Initializers Figure 1



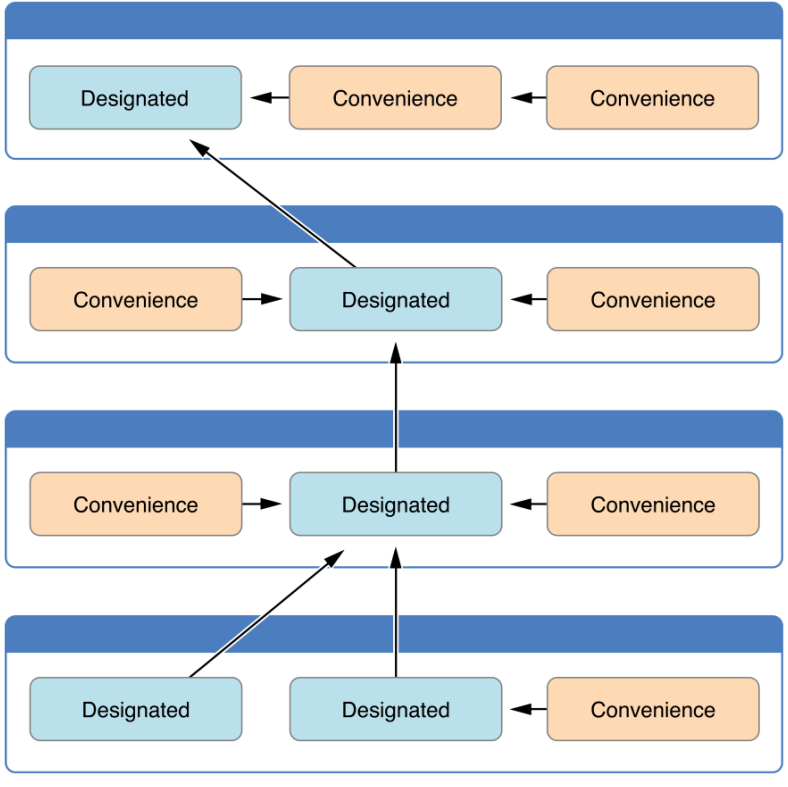
Initializers Figure 2



Initializers Figure 3



Initializers Figure 4



Composition

- Has-A relationship
- Can't have a is-a relationship between Customer and BankAccount class or Car and Driver class
- Use it to combine objects so that they can work together
- Composition is included by including pointers to objects as reference variables
- Combination of both IS-A and HAS relationship is used for designing any class

Polymorphism

- *Polymorphism* enables programs to be developed so that objects from different classes can define methods that share the same name
- Same Method Name but Different class
- Superclass type reference can be used to declare any subclass objects
- But methods declared in superclass only can be accessed
- *Dynamic typing* defers the determination of the class that an object belongs to until the program is executing. Id is used
- *Dynamic binding* defers the determination of the actual method to invoke on an object until program execution time.
- Static typing is also important. Compiler checking ensures less bug in the code.

Polymorphism

- Many forms

For example, suppose Window is a subclass of View, and they both implement a method called flush.

```
id  window = [[Window alloc] init];
id  view = [[View alloc] init];
id  myObject;

[view flush];      // the flush method in View
[window flush];    // the flush method in Window

myObject = view;
[myObject flush];  // the flush method in View

myObject = window;
[myObject flush];  // the flush method in Window
```

Dynamic Typing & Dynamic Binding

- Dynamic typing → id data type is used to represent a variable but the actual data type is unknown until runtime.
  - For example
    id myObject = [anArray objectAtIndex:index];
  - Static type checking doesn't give the flexibility
- Dynamic Binding → determining the method to invoke at runtime and not at compile time.
  - [myObject run];
  - Also known as late binding.
  - In Objective C , all methods are resolved dynamically at runtime.

id and NSObject

- id means ”an object”
- NSObject * means “an instance of NSObject or one of its subclasses ”
- Use id for declaring a method which can take any class instance as an argument.
- Theoretically, we can have non-NSObject root objects.
- id could also apply to any non-NSObject derived objects

Nothingness

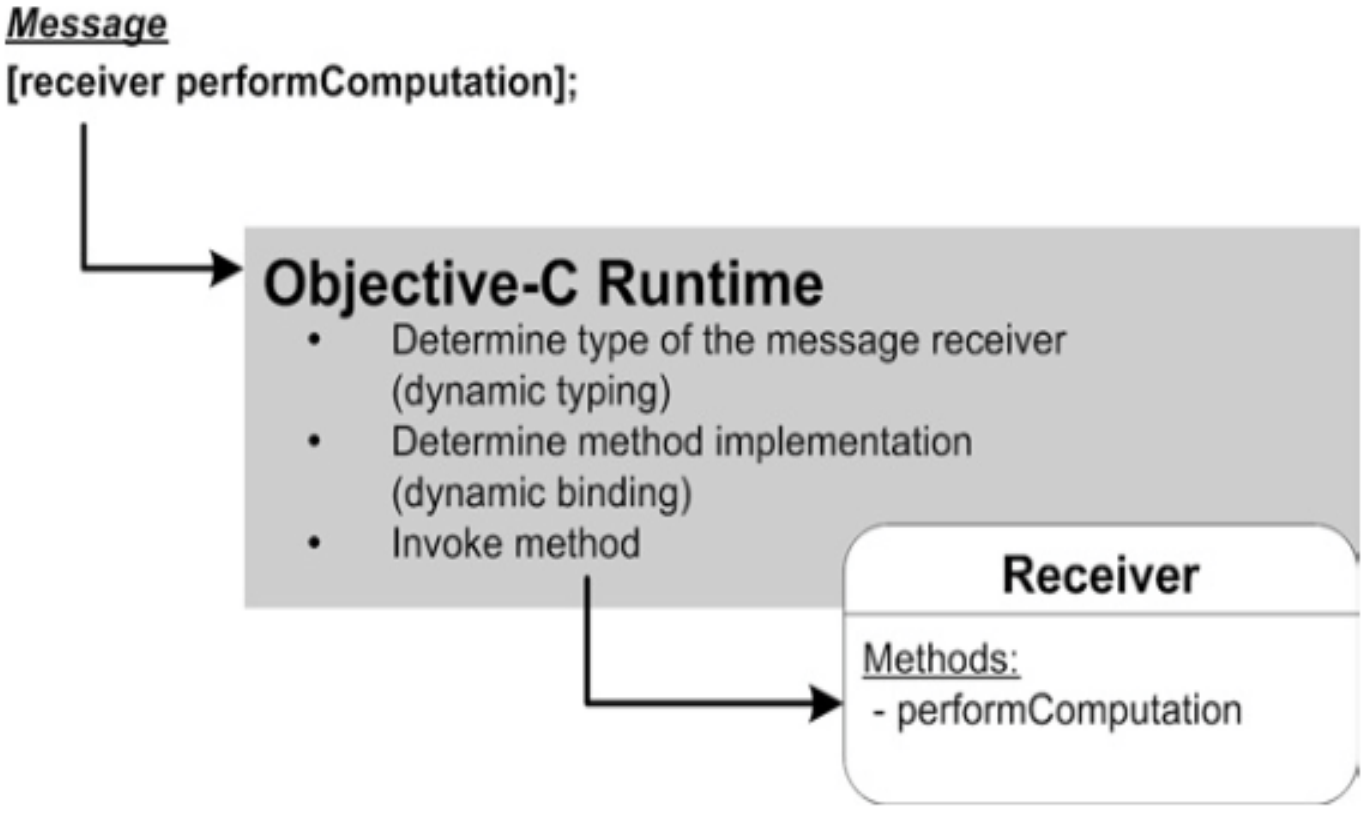
Symbol	Value	Meaning
NULL	(void *)0	literal null value for C pointers
nil	(id)0	literal null value for Objective-C objects
Nil	(Class)0	literal null value for Objective-C classes
NSNull	[NSNull null]	singleton object used to represent null

Message passing

- Every object has methods. In Objective-C, the common concept to interact with an object is sending an object a message:
  - (void) launchPlane:(NSString*)planeName;

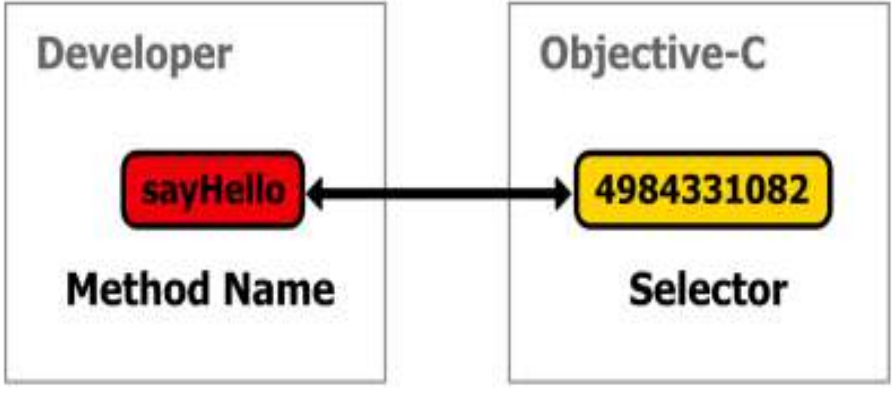
```
[planeLauncher launchPlane];

- (void) launchPlane:(NSString*)planeName
     fuelCapacity:(int)litresOfFuel;

[planeLauncher launchPlane:@"Boeing 747-300"
fuelCapacity:183380];
```

Message passing

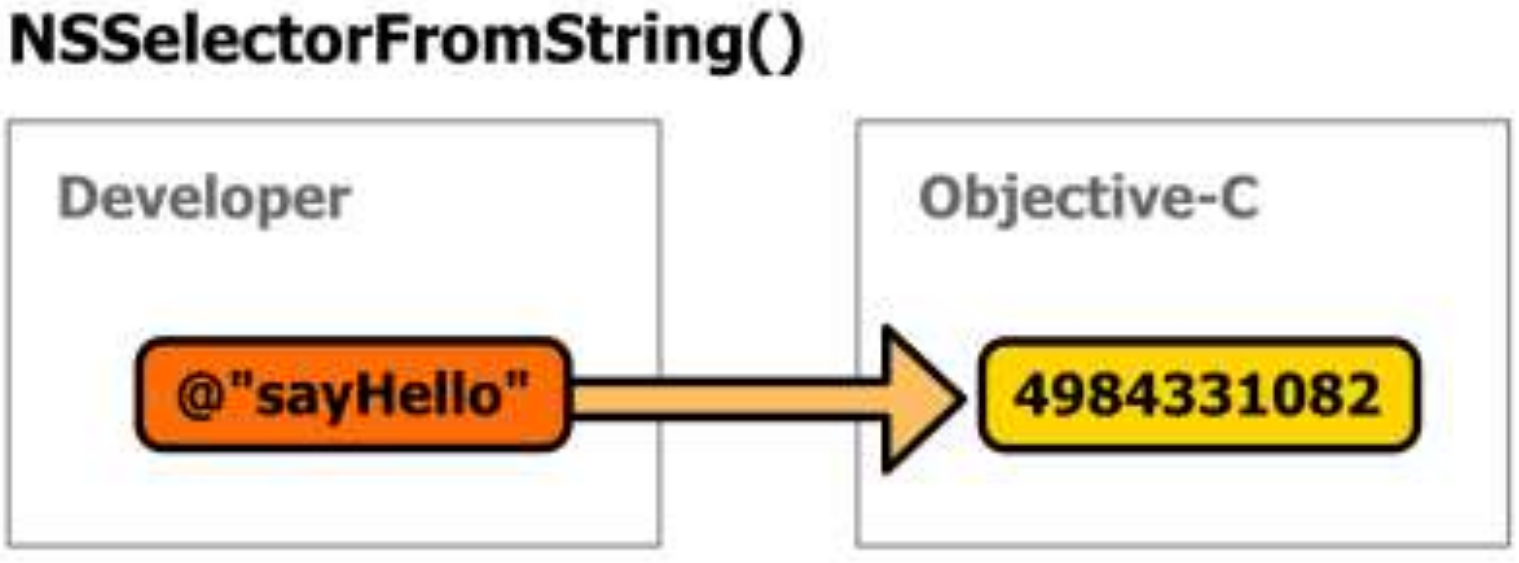


Selectors

- Selectors are Objective-C's way of representing methods.



Selectors


NSSelectorFromString()
Developer
@"sayHello"
Objective-C
4984331082

Equality

- Two objects may be equal or equivalent to one another, if they share a common set of properties. Yet, those two objects may still be thought to be distinct, each with their own identity.
- Approximate NSObject isEqual method

```
@implementation NSObject
- (BOOL)isEqual:(id)object
{
    return self == object;
}
@end
```

isEqual

- Subclasses of NSObject implementing their own isEqual: method are expected to do the following:
  - Implement a new isEqualTo*ClassName*: method, which performs the meaningful value comparison.
  - Override isEqual: to make class and object identity checks, falling back on the aforementioned class comparison method.
  - Override hash, which will be discuss later

IsEqualTo___

NSDictionary	-isEqualToDictionary:
NSHashTable	-isEqualToHashTable:
NSIndexSet	-isEqualToIndexSet:
NSNumber	-isEqualToNumber:
NSOrderedSet	-isEqualToOrderedSet:
NSSet	-isEqualToSet:
NSString	-isEqualToString:

NSString Equality

```
NSString *a = @"Hello";
NSString *b = @"Hello";
BOOL compare= (a == b); // YES
```

- Optimization technique known as string interning, whereby one copy of immutable string values.
- Objective-C selector names are also stored as interned strings in a shared string pool.

NOTE: *The correct way to compare two NSString objects is -isEqualToString:. Under no circumstances should NSString objects be compared with the == operator.*

Hashing Fundamentals

- A hash table is a fundamental data structure in programming, and it's what enables NSSet & NSDictionary to have fast (O(1)) lookup of elements.
- Rather than storing elements sequentially (0, 1, ..., n-1) like an Array, a hash table allocates n positions in memory, and uses a function to calculate a position within that range.
- NSDate Hash method (Approximate)

```
@implementation NSDate
- (NSUInteger)hash
{
 return abs([self timeIntervalSinceReferenceDate]);
}
```

Implementing -isEqual: and hash in a Subclass

```
@interface Person
@property NSString *name;
@property NSDate *birthday;
!
- (BOOL)isEqualToPerson:(Person
*)person;
@end
```

Implementing -isEqual: and hash in a Subclass

```
@implementation Person
-       (BOOL)isEqualToPerson:(Person *)person
{
 if (!person) { return NO; }
  BOOL haveEqualNames = (!self.name && !person.name) ||
  [self.name isEqualToString:person.name];
 BOOL haveEqualBirthdays = (!self.birthday && !    person.birthday) || [self.birthday
isEqualToDate:person.birthday]; return haveEqualNames &&    haveEqualBirthdays;
}
-(BOOL)isEqual:(id)object
  { if (self == object) { return YES; }
  if (![object isKindOfClass:[Person class]]) { return NO; }

  return [self isEqualToPerson:(Person *)object];
}
-       (NSUInteger)hash
 {
          return [self.name hash] ^ [self.birthday hash];
 }
@end
```

PROPERTY ATTRIBUTES

Default behavior

```
// Car.h
#import <Foundation/Foundation.h>
@interface Car : NSObject
@property BOOL running;
@end
// Car.m
#import "Car.h"
@implementation Car
@end
//Setter and getter method generated by compiler
- (BOOL)running
 {
        return _running;
}
-      (void)setRunning:(BOOL)newValue
{
        _running = newValue;
}
```

Default behavior

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"
int main(int argc, const char * argv[]) {
      @autoreleasepool
      {
      Car *honda = [[Car alloc] init];
      honda.running = YES;                  // [honda    setRunning:YES]

      NSLog(@"%d", honda.running);       // [honda running]
}
return 0; }
```

GETTER=<NAME>

- @property (getter=isRunning) BOOL running;

Changing the default getter name to something else

```
Car *honda = [[Car alloc] init];
honda.running = YES;            // [honda setRunning:YES]
NSLog(@"%d", honda.running);      // [honda isRunning]
NSLog(@"%d", [honda running]); // Error: method no
                                                longer exists
```

Setter=<name>

- @property (setter=putRunning) BOOL running;
- Changing the default setter name to something else

READONLY

- Declaring your property as readonly you tell compiler to not generate setter method automatically.
- Indicates that the property is read-only.
- If you specify readonly, only a getter Moreover, if you attempt to assign a value using the dot syntax, you get a compiler error.

Example:

@property (nonatomic, readonly)NSString*name;

ATOMIC

- Atomicity has to do with how properties behave in a threaded environment.
- Atomic means only one thread access the variable(static type).
- Atomic properties lock the underlying object to prevent this from happening, guaranteeing that the get or set operation is working with a complete value.
- Atomic is thread safe but it is slow in performance
- It is default behavior
- It is not actually a keyword.
- Accessors for atomic properties must both be either generated or user-defined.Example :

@property (atomic) NSString *name;

NONATOMIC

- Nonatomic means multiple thread access the variable(dynamic type).
- Nonatomic is thread unsafe but it is fast in performance
- Nonatomic is NOT default behavior,we need to add nonatomic keyword in property attribute.
- it may result in unexpected behavior, when two different process (threads) access the same variable at the same time.

Example:

@property (nonatomic) NSString *name;

STRONG

- It says "keep this in the heap until I don't point to it anymore"
- In other words " I'am the owner, you cannot dealloc this before "
- You use strong only if you need to retain the object.
- By default all instance variables and local variables are strong pointers.
- We generally use strong for UIViewControllers (UI item's parents)
- strong is used with ARC and it basically helps you , by not having to worry about the retain count of an object. ARC automatically releases it for you when you are done with it.Using the keyword strong means that you own the object.

Example:

@property (strong, nonatomic) ViewController *viewController;

WEAK

- It says "keep this as long as someone else points to it strongly"
- A "weak" reference is a reference that you do not retain.
- We generally use weak for IBOutlets (UIViewController's Childs).This works because the child object only needs to exist as long as the parent object does.
- A weak reference is a reference that does not protect the referenced object from collection by a garbage collector.
- Weak is essentially assign, a unretained property. Except the when the object is deallocated the weak pointer is automatically set to nil

Example :

@property (weak, nonatomic) IBOutlet UIButton *myButton;

ASSIGN

- Assign is the default and simply performs a variable assignment
- Assign is a property attribute that tells the compiler how to synthesize the property's setter implementation
- Use assign for C primitive properties and weak for weak references to Objective-C objects.
- This is the default behavior for primitive data types

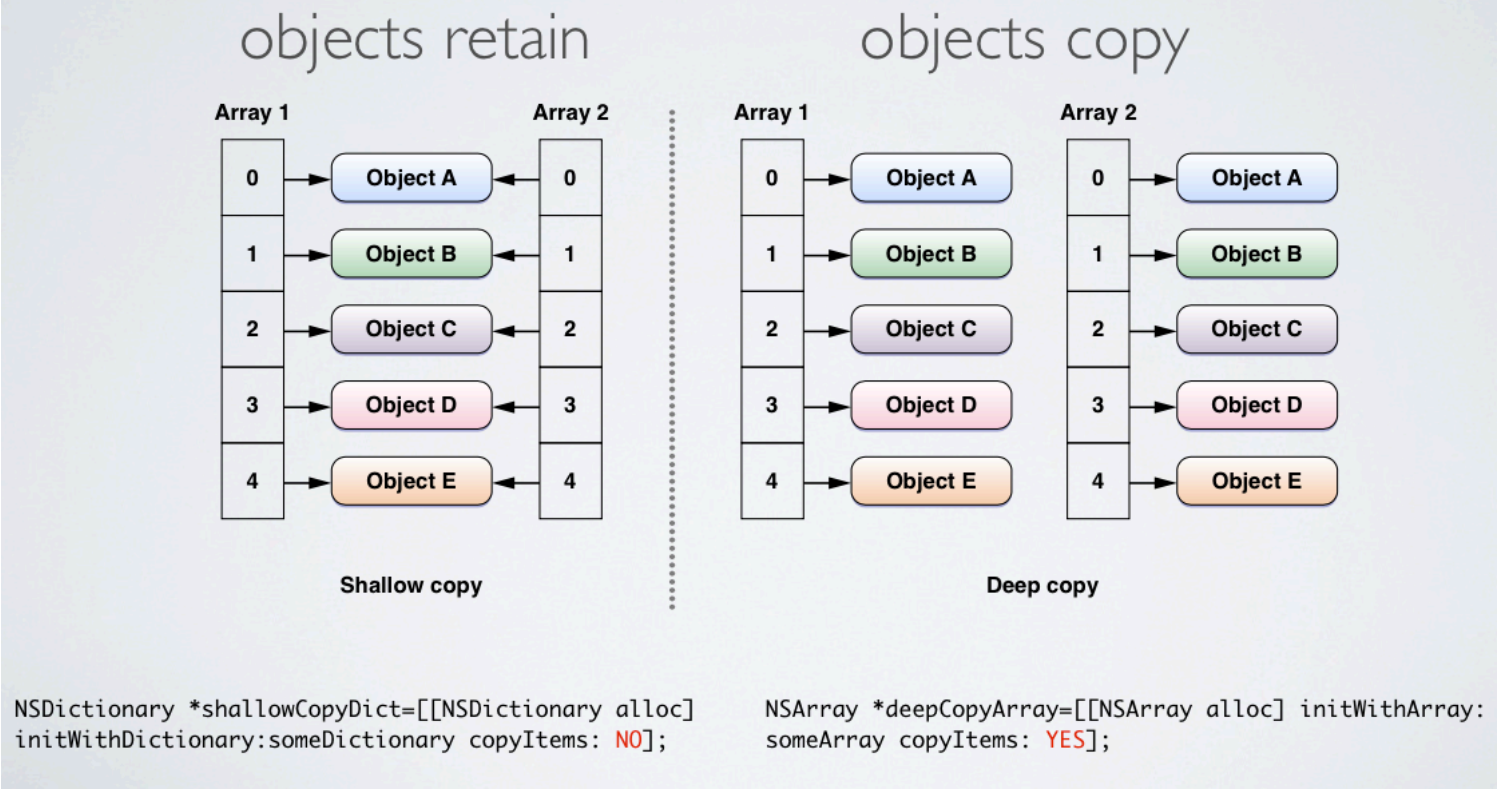
Example:

@property (assign) int score;

COPY

- The copy attribute is an alternative to strong.
- Instead of taking ownership of the existing object, it creates a copy of whatever you assign to the property, then takes ownership of that.
- Only objects that conform to the NSCopying protocol can use this attribute.

Deep copy vs. Shallow copy

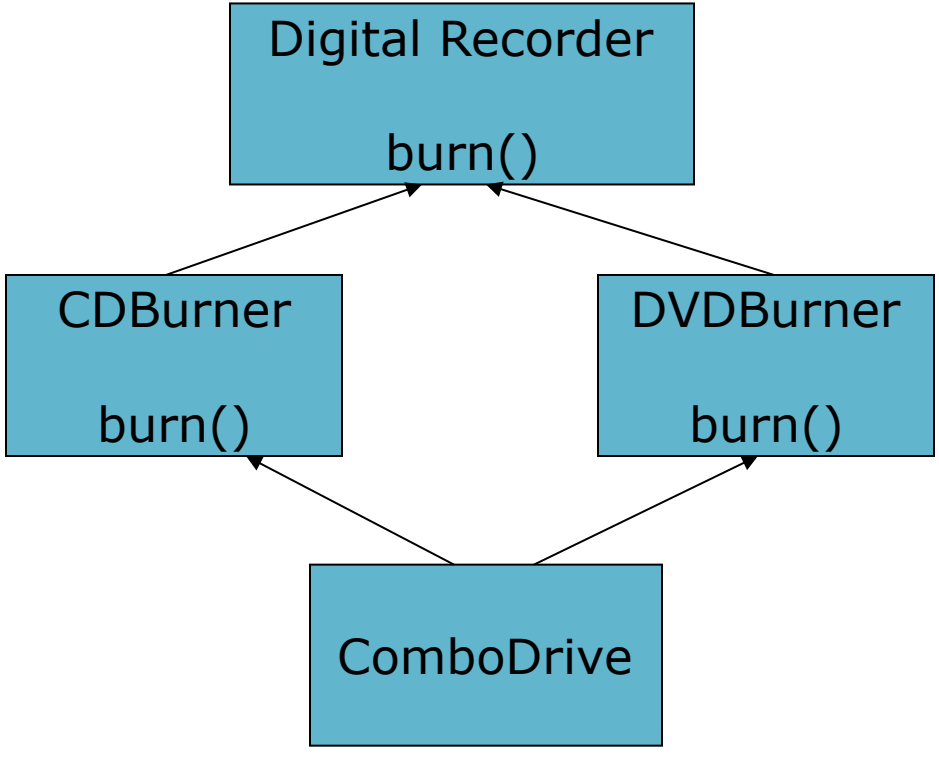



```
NSDictionary *shallowCopyDict=[[NSDictionary alloc]
initWithDictionary:someDictionary copyItems: NO];
```

```
NSArray *deepCopyArray=[[NSArray alloc] initWithArray:
someArray copyItems: YES];
```

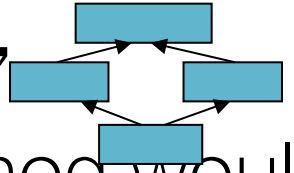
Multiple Inheritance

Why can multiple inheritance be a bad thing?



Multiple Inheritance

- Leads to ambiguity
  - "The Deadly Diamond of Death"
  - In our example: which burn method would ComboDrive inherit?



- Don't want the language to have to support/encode special rules to deal with ambiguity
- Ambiguity is bad in programming!

PROTOCOLS

Set of rules to follow

What is a Protocol?

Formal Protocols:

- A formal protocol is a collection of method declarations defined independently of any class, example:

```
@protocol Prot1 <NSObject>
    -(void) print;
  -(void) show;
  -(float) compute;
@end
```

Informal Protocols:

- These are actually categories.

Adopting a Protocol

The protocol Prot1 can be adopted by any class:

```
@interface Student <Prot1>
   . . . .
@end
```

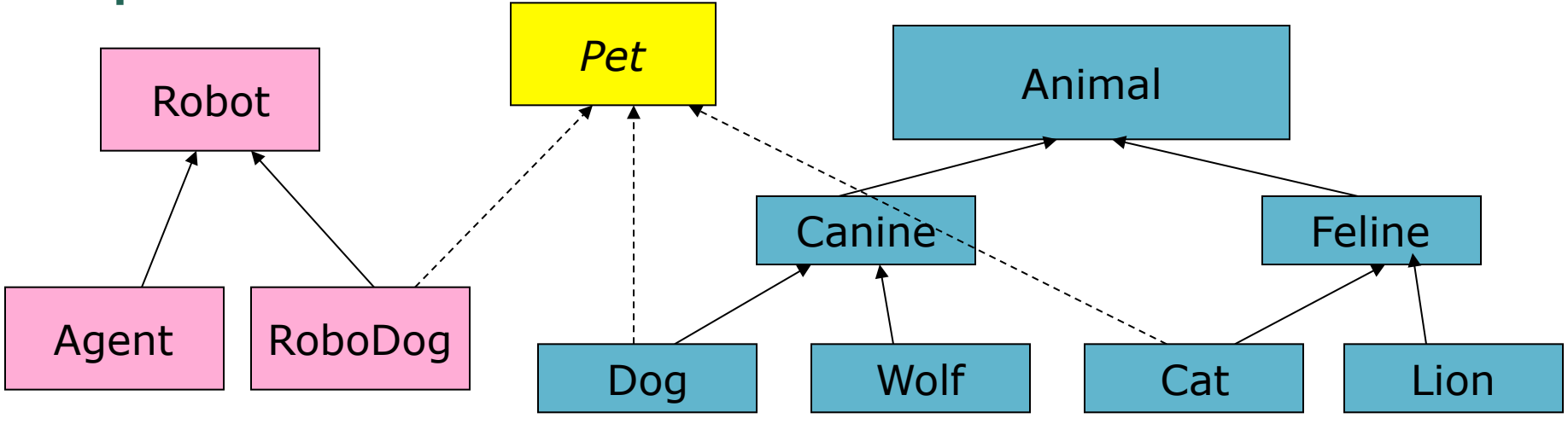
The class Student must implement the methods in the protocol Prot1 in its implementation.

Failure to implement

- If the class that adopts a protocol does not implement the methods in the protocol, the compiler will issue a warning for each missing method.
- As long as the adopted methods are not called, there will be no consequences for the running code.

Protocol benefit

- You can now have classes from different inheritance trees implement the same protocol



Checking for conformity

During running of the program, checking can be performed whether or not a given class (or a given object) conforms to a given protocol. The checking is done like below:

pt is a pointer to an object,

prot1 is a pointer to a Protocol

```
[pt conformsToProtocol:prot1]
```

This method returns *true* if the class of the object *pt* has adopted the protocol and *false* otherwise.

It does not check whether the methods in the protocol are implemented or not.

Grouping of classes

Protocols allow the grouping of classes and objects according to which protocols they have implemented.

This grouping overrides hierarchical relationships.

MEMORY MANAGEMENT

MEMORY MANAGEMENT

- Allocation and Recycling

When a program requests a block of memory, memory manager assigns that block to program.

When the program no longer need that data, those blocks become available for reassignment.

This task can be done manually (by the programmer) or automatically (by the memory manager)

Memory Management

- ios 4 and below – Manual memory management
- ios 5 and above – Automatic Reference Counting

- OS X 10.4 and below – Manual memory management
- OS X 10.5 – 10.7 – Garbage Collector
- OS X 10.7 – Automatic Reference Counting

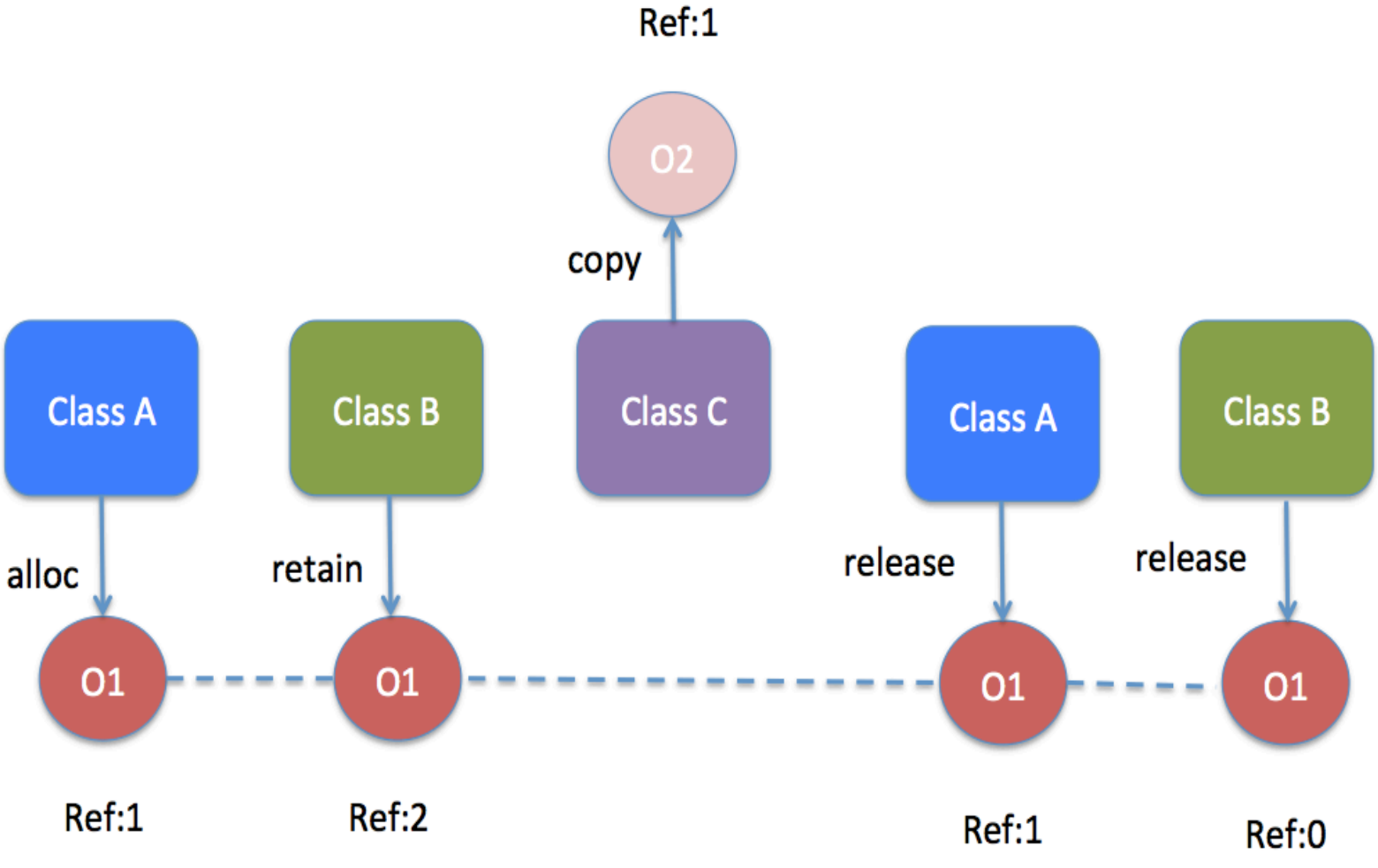
Reference Counting

- When an object is created, it has a reference count of 1. The creator is the owner.
- Every time, a new owner is added, the reference count increases by 1.
- Every time, a owner relinquishes control , the reference count decreases by 1.
- When reference count is 0, runtime destroys the object by calling dealloc() on it. – You never call dealloc() directly. release does it automatically

Reference Counting

- Every object has a **retain count**
  - Defined on NSObject
  - As long as retain count is > 0, object is alive and valid
- +alloc and -copy create objects with retain count == 1
- -retain increments retain count
- -release decrements retain count
- When retain count reaches 0, **object is destroyed**
  - -dealloc method invoked automatically
  - One-way street, once you're in -dealloc there's no turning back

A Graphical Explanation...



Reference counting in action

```
Person *person = [[Person alloc] init];
```

Retain count begins at 1 with +alloc

```
[person retain];
```

Retain count increases to 2 with –retain

```
[person release];
```

Retain count decreases to 1 with –release

```
[person release];
```

Retain count decreases to 0, -dealloc automatically called

Taking ownership of an object

alloc - Create an object and claim ownership of it.

copy - Copy object and claim ownership of it.

retain - Claim ownership of an existing object.

You own a object if:

```
SomeClass* obj = [[SomeClass alloc]init];
SomeClass* obj = [SomeClass copy];
SomeClass* obj = [SomeClass new];
```

Relinquishing ownership of an Object

- Explicitly using **-release**
  - Object deallocated as soon as reference count is 0

Put it in an **-autorelease** pool.

- System takes care of deallocating unreferenced objects periodically from autorelease pool at some later time

When you forget to balance any alloc, retain & copy call with a release or autorelease on same object :

- If you forget to release an object, its underlying memory is never freed, resulting in **memory leak**.
- If you release object too many times, it will result into **dangling pointer**.

Messaging deallocated objects

```
Person *person = [[Person alloc] init];
// ...
[person release]; // Object is deallocated
[person doSomething]; // Crash!

person = nil;
[person doSomething]; // No effect
```

Implementing a -dealloc method

```
#import "Person.h“

@implementation Person

- (void)dealloc {
        // Do any cleanup that’s necessary
        // ...

        // when we’re done, call super to clean us up
        [super dealloc];
}

@end
```

Practical Memory Management (Method Names & Autorelease)

- Methods whose names includes **alloc, copy, or new** return a retained object that the **caller needs to release**

```
NSMutableString *string = [[NSMutableString alloc] init];
// We are responsible for calling -release or -autorelease
[string autorelease];
```

- All other methods return autoreleased objects

```
NSMutableString *string = [NSMutableString string];
// The method name doesn't indicate that we need to release it
```

- This is a convention- **follow it in methods you define**!

Examples

```
- (void)printHello {
    NSString *string;
    string = [[NSString alloc] initWithString:@"Hello"];
    NSLog(string);
    [string release];
}
- (void)printHello {
    NSString *string;
    string = [NSString stringWithFormat:@"Hello"];
    NSLog(string);
}
- (void)printWindowTitle {
    NSString *string;
    string = [myWindow title];
    NSLog(string);
}
```

Retain

- Create an object
  - alloc, new, copy!
- Using “retain”
  - Retain the object to ensure it is not deallocated elsewhere
- Strong References
  - Object not freed as long as there is a strong reference to it. Use “retain”

```
@property (retain) id myProperty; // iOS <= 4.3
@property (strong) id myProperty; // iOS 5+
```

The Four Basic Rules

- There are only 4 basic rules when it comes to memory management in Objective-C:

1. If you own it, release it.
2. If you don't own it, don't release it.
3. Override dealloc in your classes to release the fields that you own.
4. Never call dealloc directly.

- That's it. The first two are the most important and we'll get into the basics of object ownership next.

Rule #1. When You Own It.

You own an object in Objective-C when you **alloc** it, **copy** it or **new** it. For example:

```
1  -(void)someMethod
2  {
3    // I own this!
4    SomeObject *iOwnThis = [[SomeObject alloc] init];
5
6    [iOwnThis doYourThing];
7
8    // I release this!
9    [iOwnThis release];
10 }
11
12 -(void)someOtherMethod:(SomeObject *)someThing
13 {
14   // I own this too!
15   SomeObject *aCopyOfSomeThing = [someThing copy];
16
17   [aCopyOfSomeThing doSomething];
18
19   // I release this!
20   [aCopyOfSomeThing release];
21 }
22
23 -(void)yetAnotherMethod
24 {
25   // I own this too!
26   SomeObject *anotherThingIOwn = [SomeObject new];
27
28   [anotherThingIOwn doSomething];
29
30   // I release this!
31   [anotherThingIOwn release];
32 }
```

How easy is that? I can't even think of anything more to write about this, it's that simple. But to reiterate:

You own it if you alloc it.
You own it if you copy it.
You own it if you new it. (New is simply a shortcut for alloc/init).

Rule #2. When You Don't Own It.

This is the tricky part. You don't own it when you don't own it. So if you did not **alloc** it, or you did not **copy** it, or you did not **new** it – you don't own it.

What about the following?

```
-(void)sayWhat
{
   NSString *doIOwnThisIWonder = [NSString stringWithFormat:@"%@",@"Nope"];
   NSImage *iOwnThisImage = [[NSImage alloc] initWithContentsOfFile:@"/tmp/youownthis.jpg"];
   NSData *perhapsThisData=[iOwnThisImage TIFFRepresentation];
}
```


Do you own that NSString? Nope, you didn't **alloc/copy/new** it.

The NSImage? Yes you own this, you **alloc**'d that bad boy.

The NSData? Nope, again, you didn't **alloc/copy/new** it.

Here is how that method should really look, btw:

```
-(void)sayWhat
{
   NSString *doIOwnThisIWonder = [NSString stringWithFormat:@"%@",@"Nope"];
   NSImage *iOwnThisImage = [[NSImage alloc] initWithContentsOfFile:@"/tmp/youownthis.jpg"];
   NSData *perhapsThisData=[iOwnThisImage TIFFRepresentation];

   ... do my thing ...

  [iOwnThisImage release];
}
```

Rules #3 and #4. Dealloc.

Ok, so here's the hardest part of the whole thing. If you have any class with object properties that you retain, you'll have to release them when your object has been deallocated via the [dealloc] message. An example to demonstrate:

```
//
//  SomeObject.h
//

#import <Cocoa/Cocoa.h>

@interface SomeObject : NSObject {
    NSMutableArray *things;
    NSMutableDictionary *someOtherThings;
}

@end
```

```
#import "SomeObject.h"

@implementation SomeObject

- (id)init {
        if (self=[super init]) {
                things=[[NSMutableArray arrayWithObjects:@"one",@"two",@"three",nil] retain];
                someOtherThings=[[NSDictionary alloc] init];
        }

        return self;
}

-(void)dealloc {
        [things release];
        [someOtherThings release];
        [super dealloc];
}

@end
```

In the above example, our SomeObject has two fields: things and someOtherThings. You can see in our init method that we create the objects and assign them to our properties. For things, because we are using [NSMutableArray arrayWithObjects:], we have to call retain. Remember, we didn't alloc, copy or new, instead we called a convenience method that returns an autoreleased object that we must explicitly retain. What is autorelease you might be asking? I'll explain in the next part, but for now all you have to remember is the basics of object ownership described above.
In the [dealloc] method you'll see that we are releasing the objects we created. Nothing more, nothing less.

The retain method

```
@interface  MyClass : NSObject
{
      NSMutableArray* _values;
 }
-   (NSMutableArray*)values;
-   (void)setValues : (NSMutableArray*)newValues;
```

Will create weak reference if implemented as below:

```
- (NSMutableArray*)values
{
    return _values;
}
- (void)setValues : (NSMutableArray*)newValues
{
     _values = newValues;
}
```

Now in main,

```
NSMutableArray*  addValues = [[NSMutableArray
   alloc]init];
[addValues addObject: @”abcd”];

MyClass* class1 = [[MyClass alloc]init];
[class1 setValues: addValues];
[addValues release];

NSLog(@”%@”,[class1 values]); //error, dangling pointer
```

Because the objet is already released, class1 object has weak reference to array.

Correct way....

MyClass needs to claim ownership of the array as :

```
- (void)setValues : (NSMutableArray*)newValues
{
    _values = [newValues retain] ;  //refcount = 2
}
```

This ensures the object wont be released while our class object is using it.

The retain call isn't balanced with a release, so we have another memory leak.

Here as soon as we pass another value to setValues: , we cant access the old value, which means we can never free it.

To fix this setValues: needs to call release on the old value

```
- (void)setValues : (NSMutableArray*)newValues
{
    if(_values  ==  newValues)
     {
            return;
     }
      NSMutableArray* oldval = _values;
    _values = [newValues retain] ;  //refcount = 2
       [oldval release];
}
```

The copy method..

- Creates a brand new instance of the object and increments the reference count on that leaving the original unaffected.

Autorelease method

- Relinquishes ownership of an object
- Doesn't destroy the object immediately
- Defer the freeing of memory until later on in the program

Ex:-

```
+(MyClass*) classObj{
        MyClass* obj = [[MyClass alloc]init];
         return [obj autorelease];
}
```

Methods like stringWithFormat: and stringWithString: works the same way…

It waits until the nearest @autoreleasepool{} block, after which

AutoRelease Pools

- Maintains a list of objects that will be sent release messages when those pools are destroyed.
- Can be nested
- You should always create a autorelease pool whenever you create a new thread.

```
1  -(NSImage *)getAnImage {
2
3    return [[NSImage alloc] initWithContentsOfFile:@"/tmp/youownthis.jpg"];
4
5  }
```

This Gist brought to you by GitHub. gistfile1.m view raw

Who owns that NSImage? Well, in this example, the caller of this method is the owner and it is, therefore, the caller's responsibility to release it. However, this method is **bad form**. The correct form:

```
1  -(NSImage *)getAnImage {
2
3    return [[[NSImage alloc] initWithContentsOfFile:@"/tmp/youownthis.jpg"] aut
   orelease];
4
5  }
```

This Gist brought to you by GitHub. gistfile1.m view raw

- Notice the autorelease message at the end. What happens when autorelease is called, the object in question is added to an autorelease pool so that it will receive a release message when the pool is destroyed. The caller needn't have to worry about it, nor does the callee.
- Autorelease pools can be nested, too. This is great for situations where you are creating thousands upon thousands of temporary objects and want to keep your maximum memory footprint slim and tight. Also, if you are doing threaded work, anytime you create a new thread, you'll have to create an autorelease pool for that thread.

The dealloc method

- Called right before the object is destroyed, giving you a chance to clean up any internal objects.
- Called automatically by runtime.
- You should never try to call dealloc yourself.
- Always call superclass dealloc to make sure that all of the instance variables in parent classes are properly released.

Automatic Reference Counting

- ARC works exact same way as MRR, but it automatically inserts the appropriate memory management methods for you.
- Takes the human error out of memory management.
- To enable ARC – Build Settings -> Automatic Reference Counting -> Yes
- Not allowed to manually call retain, release or autorelease
- New @property attributes assign -> weak

- Fully featured version of ARC was delivered by apple in 2011 on MAC OS X Lion & ios5

- Before that a limited version of ARC (ARCLite) was supported in Xcode4.2 or later, MAC OS X 10.6, ios 4.0… which does not included **weak reference support**

- In case of ARCLite we can use unsafe_unretained in place of weak.

Zeroing weak references

- Difference between ARC & ARCLite is of zeroing weak references
- It is a feature in ARC to set weak reference pointers to nil when the object it is pointing to is deallocated.
- Prior to this weak not set to nil, leading to dangling pointers (unsafe_unretained). The programmer has to manually set the reference to nil.
- This needs additional support from objC runtime, hence only available after OSXLion or ios5

Garbage Collector

- Starting in MAC OS X 10.5 we got automatic memory management on MAC OS X (not in ios) with Autozone(libAuto)

- Works at runtime
- Trade a bit of CPU time to let libauto collect objects that are out of scope and no longer referenced.

- Libauto is opensource.. http://opensource.apple.com/source/libauto/libauto-141.2/

- Xcode, Rapidweaver etc use garbage collector

Properties

- There is really only two rules:

- If you retain or copy your property, you need to set it to nil in your dealloc.
- If you initialize the property during init, autorelease it.

```objc
//
//  SomeObject.h
//

#import <Cocoa/Cocoa.h>

@interface SomeObject {
    NSString *title;
    NSString *subtitle;
}

@property (retain) NSString *title;
@property (retain) NSString *subtitle;

@end
```



```objc
#import "SomeObject.h"

@implementation SomeObject

@synthesize title;
@synthesize subtitle;

-(id)init {
    if (self=[super init])
    {
        self.title=[NSString stringWithFormat:@"allyouneed"];
        self.subtitle=[[[NSString alloc] init] autorelease];
    }

    return self;
}

-(void)dealloc {
    self.title=nil;
    self.subtitle=nil;

    [super dealloc];
}

@end
```

The title property we are calling a convenience method on NSString that returns an autoreleased object, so we don't have to do a thing. For subtitle, we are allocating a new NSString ourselves, therefore we need to autorelease it. The reason being is that when the value is assigned to the property, the retain count is incremented by 1, so without the autorelease, that object will maintain a retain count of 2 (one for the alloc/init and another for the assignment to the property). With a retain count of 2, the object will never be released and you'll leak memory.

Strong,Weak,Copy

- Within your code, you can now use the “strong” and “weak” keywords in your object instantiation. With these, the compiler understands how to handle the memory for the object
- A strong property is claiming “ownership,” and its object is automatically released when the object is no longer referenced. A weak property does not extend the lifetime of the object. In other words, it does not assume ownership, so it is possible that the object is released while the property has a reference to it. However, weak properties are automatically set to nil when the object is released.

CATEGORIES

An extension to an existing class without subclassing it

Why categories?

- Existing classes are good to use but maybe not enough to give all the functionalities we need.
- Probably, you don’t have access to the source code for the class
- Inheriting an existing class is a good idea but may lead to another problems.
- Categories let you add functionality to an existing class without having access to source code and without subclassing.
- It helps us in splitting a large class’s implementation file across multiple files or categories.

Category Format

@interface ClassToAddMethodsTo(categoryName)

...methods go here

@end

For example, below is a category that adds a method to the NSString class. The method *reverseString* adds the capability to all NSString objects to reverse the characters in the string.

@interface NSString (reverse)

-(NSString *) reverseString;

@end

Limitations

- You can’t add instance variables or properties to a category
- You can only add methods to a category
- The second limitation is if the original class and category has the same method, then original class method will be hidden or not used. You can avoid this by adding a prefix to a category.

EXTENSIONS

Adding private methods and variables

Extensions

- An extension cannot be declared for any class, only for the classes that we have original implementation of source code.
- An extension is adding private methods and private variables that are only specific to the class.
- Any method or variable declared inside the extensions is not accessible even to the inherited classes.
- An extension's API must be implemented in the main implementation file—it cannot be implemented in a category

Extensions

What's the difference between using a class extension to declare a private method, and not declare it at all (it seems to compile in run in both cases)? (example 1 vs 2)

Example 1

```
@interface Class()
-(void) bar;
@end

@implementation Class
-(void) foo {
    [self bar];
}

-(void) bar {
    NSLog(@"bar");
}
```

Extensions

Example 2

```
@implementation Class
-(void) foo {
    [self bar];
}
-(void) bar {
    NSLog(@"bar");
}
@end
```

Extensions

Second question: What's the difference between declaring ivars inside the extension and declaring it directly inside the implementation? (Exemple 3 vs 4)

Example 3

```
@interface Class() {
    NSArray *mySortedArray;
}
@end

@implementation Class
@end
```

Example 4

```
@implementation Class
NSArray *mySortedArray;
@end
```

Two Root Classes

NSObject

NSProxy

NSObject Protocol

- Specifies the basic API required by any Objective C root class
- Methods can be grouped into following categories
  - Describing objects
  - Identifying and Comparing objects
  - Identifying classes and proxies
  - Object Introspection
  - Sending messages
  - Memory Management

NSObject Class

- Initializing a Class
- Creating, Copying, and Deallocating Objects
- Identifying Classes
- Testing Class Functionality
- Testing Protocol Conformance
- Obtaining Information About Methods
- Describing Objects
- Sending Messages
- Forwarding Messages
- Dynamically Resolving Methods
- Error Handling
- Archiving
- Working with Class Descriptions
- Deprecated Methods

FOUNDATION CLASSES

Most important classes!

Mutable vs Immutable

- Almost all the foundation classes has both mutable and immutable version
- **Immutable objects are static, and as such take less space in memory and are faster to access, but cannot be changed once they are created**
- **Mutable objects are dynamically allocated and can be changed (arrays, dictionaries and sets can grow and shrink, and strings can be changed), but take more space in memory and are a bit slower to access**

NSString Class

- An array of Unicode characters:

```
NSString *englishString = @"English";
```

E	n	g	l	i	s	h
0x0045	0x006E	0x0067	0x006C	0x0069	0x0073	0x0068

Creating and Converting String

Source	Creation method	Extraction method
In code	`@"..."` compiler construct	N/A
UTF8 encoding	`stringWithUTF8String:`	`UTF8String`
Unicode encoding	`stringWithCharacters: length:`	`getCharacters:` `getCharacters:range:`
Arbitrary encoding	`initWithData: encoding:`	`dataUsingEncoding:`
Existing strings	`stringByAppendingString:` `stringByAppendingFormat:`	N/A
Format string	`localizedStringWithFormat:` `initWithFormat: locale:`	Use `NSScanner`

Search and Compare

earch methods	Comparison methods
angeOfString:	compare:
angeOfString: options:	compare:options:
angeOfString: options:range:	compare:options: range:
angeOfString: options:range: locale:	compare:options: range:locale:
angeOfCharacterFromSet:	
angeOfCharacterFromSet: options:	
angeOfCharacterFromSet: options:range:	

Search options

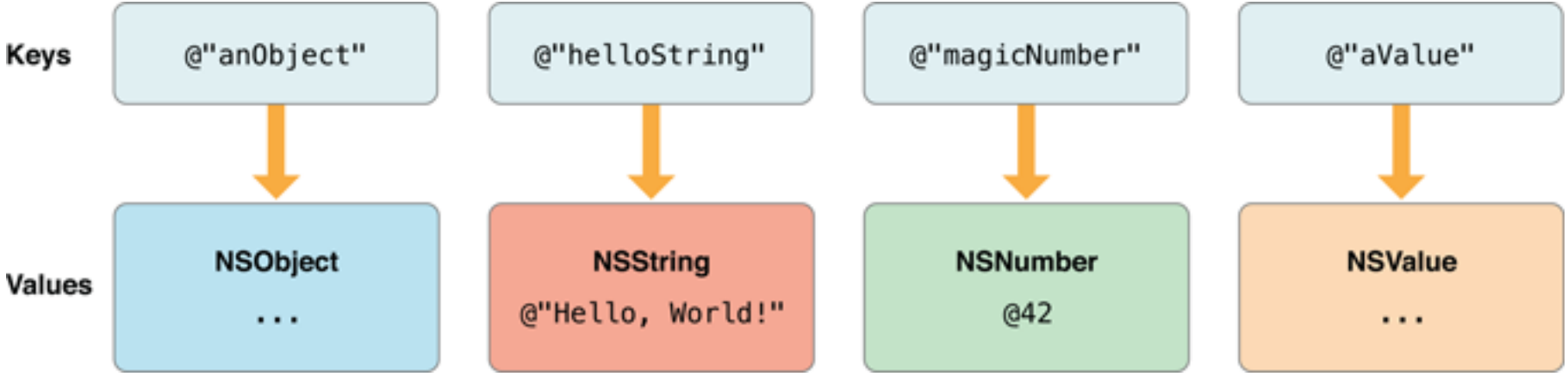
`NSCaseInsensitive-Search`	Ignores case distinctions among characters.
`NSLiteralSearch`	Performs a byte-for-byte comparison. Differing literal sequences (such as composed character sequences) that would otherwise be considered equivalent are considered not to match. Using this option can speed some operations dramatically.
`NSBackwardsSearch`	Performs searching from the end of the range toward the beginning.
`NSAnchoredSearch`	Performs searching only on characters at the beginning or, if `NSBackwardsSearch` is also specified, the end of the range. No match at the beginning or end means nothing is found, even if a matching sequence of characters occurs elsewhere in the string.
`NSNumericSearch`	When used with the `compare:options:` methods, groups of numbers are treated as a numeric value for the purpose of comparison. For example

NSArray

- NSArray is used to manage an ordered collection of objects as an array
- While it is common practice to store objects of the same type in an array, you can mix the type of objects stored in an NSArray object
- NSArray has built-in behaviors to facilitate enumerating through the objects stored in them

NSDictionary

- NSDictionary objects handle key/value pairs, or associative arrays



NSSet

- NSSet holds a collection of objects as a mathematical set
- The set is not ordered
- A given object appears only once

NSData

- NSData objects are used to hold a block of bytes and treat it as an Objective-C object
- NSData uses any bytes assigned to it as a buffer

NSNumber

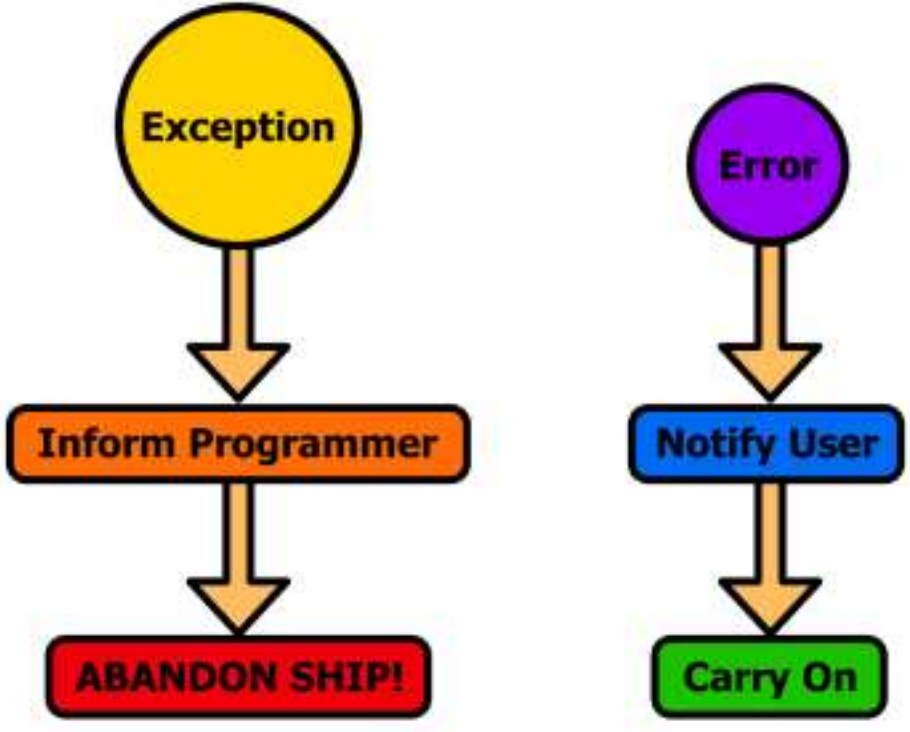
- **NSNumber objects are used to hold numbers in Objective-C**
  - Objective-C collection classes (NSArray, NSDictionary, NSSet) cannot hold C numeric types or structures, only Objective-C objects
  - NSNumber has methods to convert C types to Objective-C objects, and visa versa

NSURL

- NSURL objects are used to hold both file URLs (paths to file names preceded with “file://”) and network URLs (such as website addresses)

ERRORS AND EXCEPTIONS

Exception vs. Error



The NSException Class

- name –
  - An instance of NSString that uniquely identifies the exception.
- reason –
  - An instance of NSString containing a human-readable description of the exception.
- userInfo –
  - An instance of NSDictionary that contains application-specific information related to the exception.
    The Foundation framework defines several constants that define the "standard" exception names. These strings can be used to check what type of exception was caught.
    You can also use the initWithName:reason:userInfo: initialization method to create new exception objects with your own values. Custom exception objects can be caught and thrown using the same methods covered in the upcoming sections.

Compiler Directives

@try

—Defines a block of code that is an exception handling domain: code that can potentially throw an exception.

@catch()

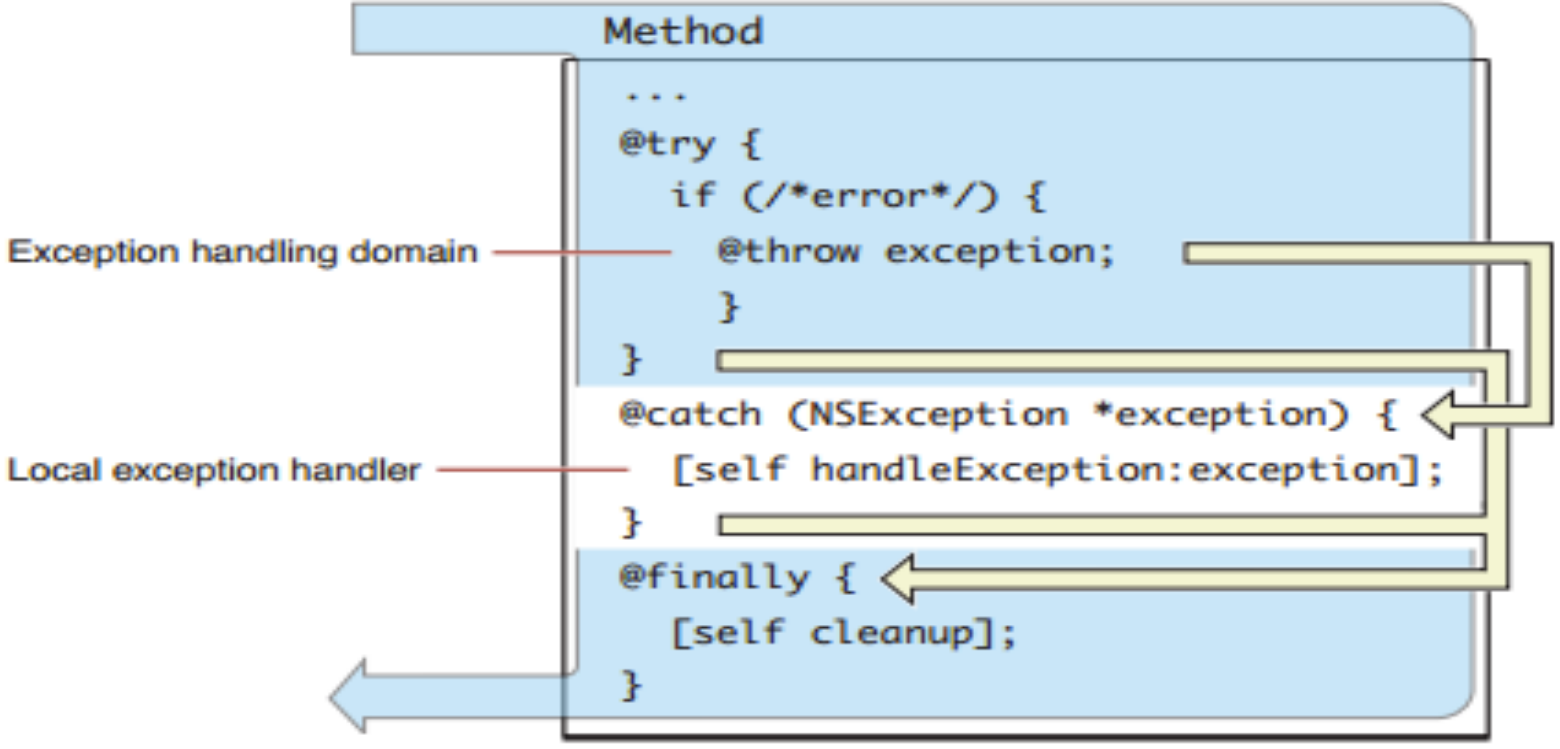
—Defines a block containing code for handling the exception thrown in the @try block. The parameter of @catch is the exception object thrown locally; this is usually an NSException object, but can be other types of objects, such as NSString objects.

@finally — Defines a block of related code that is subsequently executed whether an exception is thrown or not.

@throw — Throws an exception; this directive is almost identical in behavior to the raise method of NSException. You usually throw NSException objects, but are not limited to them.

Compiler Directives

Flow of exception handling using compiler directives



Throwing Exceptions

You throw (or raise) an exception by instantiating an NSException object and then doing one of two things with it:

- Using it as the argument of a @throw compiler directive
- Sending it a raise message

```
NSException* myException = [NSException

exceptionWithName:@"FileNotFoundException"

reason:@"File Not Found on System"

userInfo:nil];

@throw myException;

// [myException raise]; /* equivalent to above directive */
```

Predefined Exceptions

- NSRangeException
- NSInvalidArgumentException
- NSInternalInconsistencyException
- NSObjectInaccessibleException
- NSObjectNotAvailableException

NSError

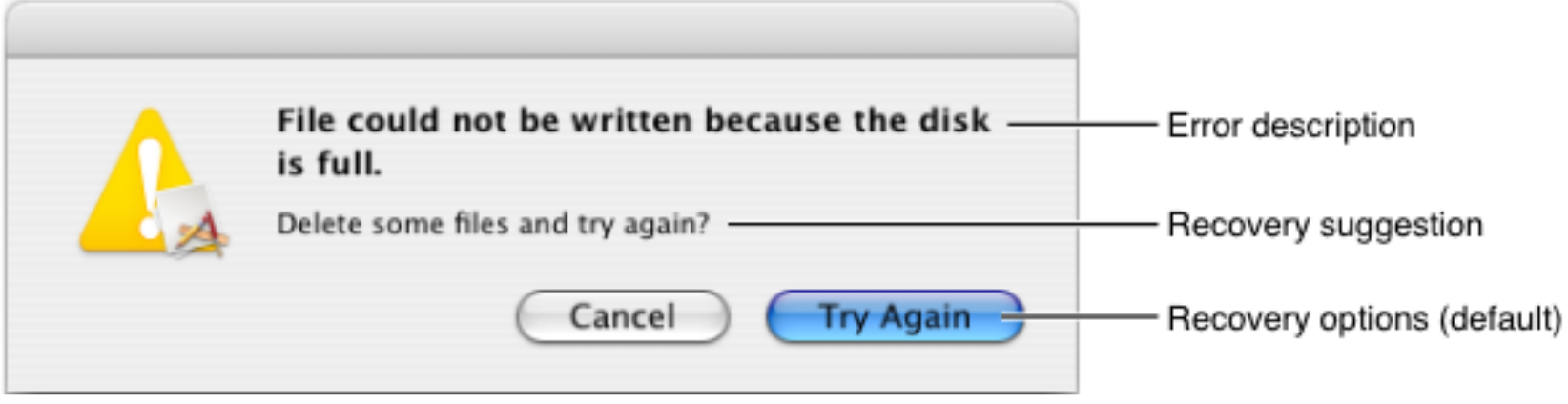
- **code**
  - An NSInteger that represents the error's unique identifier.
- **domain**
  - An instance of NSString defining the domain for the error
- **userInfo**
  - An instance of NSDictionary that contains application-specific information related to the error. This is typically used much more than the userInfo dictionary of NSException.

NSError

- **localizedDescription**
  - An NSString containing the full description of the error, which typically includes the reason for the failure. This value is typically displayed to the user in an alert panel.
- **localizedFailureReason**
  - An NSString containing a stand-alone description of the reason for the error. This is only used by clients that want to isolate the reason for the error from its full description.
- **recoverySuggestion**
  - An NSString instructing the user how to recover from the error.
- **localizedRecoveryOptions**
  - An NSArray of titles used for the buttons of the error dialog. If this array is empty, a single OK button is displayed to dismiss the alert.
- **helpAnchor**
  - An NSString to display when the user presses the Help anchor button in an alert panel.

NSError

The localized strings of an NSError object



Abstract class Implementation

- There is no compile-time enforcement that prevents instantiation of an abstract class.
- Raising an exception in abstract class

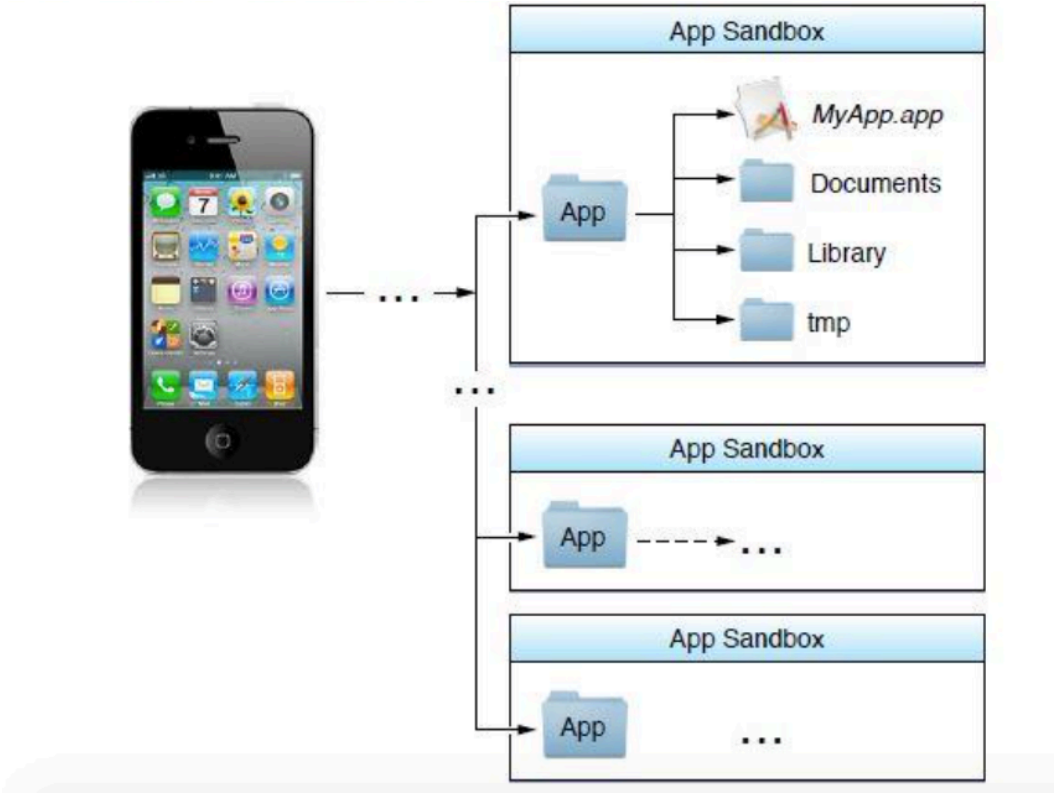
```
[NSException raise:NSInternalInconsistencyException
            format:@"You must override %@ in a subclass",
NSStringFromSelector(_cmd)];

//if method returns a value
@throw [NSException
exceptionWithName:NSInternalInconsistencyException
                                reason:[NSString stringWithFormat:@"You
must override %@ in a subclass", NSStringFromSelector(_cmd)]
                                userInfo:nil];
```

FILES IOS

File Handling

- Every iOS app is its own island
- Each app contains its own directory structure.
- Apps are prohibited from accessing or creating files in directories outside of its home directory (with exceptions)

Sandbox


App Sandbox
MyApp.app
App
Documents
Library
tmp
App Sandbox
App
App Sandbox
App

Understanding SandBox

- The purpose of a sandbox is to limit the damage that a compromised app can cause to the system.
- Sandboxes do not prevent attacks from happening to a particular app and it is still your responsibility to code defensively to prevent attacks.
- For example, if your app does not validate user input and there is an exploitable buffer overflow in your input-handling code, an attacker could still hijack your app or cause it to crash.
- The sandbox only prevents the hijacked app from affecting other apps and other parts of the system.

Exceptions

- Public system interfaces
  - Contacts
  - Calendars
  - Photo library
- An App can use such public system interfaces provided that the user approves its access.

iOS Standard Directories

- **<Application_Home>/AppName.app**
  This is the bundle directory containing the app itself.
- **<Application_Home>/Documents/**
  This directory is used to store critical user documents and app data files.
- **<Application_Home>/Documents/Inbox**
  Your app can read and delete files in this directory but cannot create new files or write to existing files

iOS Standard Directories

- <Application_Home>/Library/

  This directory is the top-level directory for files that are not user data files.

  You can create custom subdirectories for files you want backed up but not exposed to the user. Do not use this directory for user data files.

- <Application_Home>/tmp/

  Use this directory to write temporary files that do not need to persist between launches of your app.

  Your app should remove files from this directory when it determines they are no longer needed

Where You Should Put Your App's Files

- To prevent the syncing and backup processes on iOS devices from taking a long time, be selective about where you place files inside your app's home directory.
- Apps that store large files can slow down the process of backing up to iTunes or iCloud.
- Put user data in the <Application_Home> / Documents/. User data is any data that cannot be recreated by your app, such as user documents and other user-generated content

Accessing file paths

- Use a Foundation convenience function to generate path

```
NSArray * paths =
NSSearchPathForDirectoriesInDomains(NSDocumentDire
ctory, NSUserDomainMask, YES);
```

- NSDocumentDirectory, NSLibraryDirectory, NSCachedDirectory, NSTemporary Directory

```
NSString * dirPath = [paths firstObject];
NSString * path = [dirPath
stringByAppendingPathComponent:@"my_file.txt"];
```

KVC – KEY VALUE CODING

KVC

- Access properties by name
- Every class essentially becomes a dictionary

How it works

- **NSKeyValueCoding** (informal) protocol
- **Implemented by NSObject**
- NSKeyValueCoding.h
- KVC uses the runtime information generated by the compiler

SET AND ARRAY OPERATORS

- Arrays and sets support 'aggregate' keys:
- **@avg, @count, @max, @min, @sum, @distinctUnionOfArrays, @distinctUnionOfObjects, @distinctUnionOfSets, @unionOfArrays, @unionOfObjects, @unionOfSets**

ARCHIVE AND SERIALIZATION

Archives VS Serialization

- An archive can store an arbitrarily complex object graph.
- The archive preserves the identity of every object in the graph and all the relationships it has with all the other objects in the graph.
- When unarchived, the rebuilt object graph should, with few exceptions, be an exact copy of the original object graph.
- Serializations store a simple hierarchy of value objects, such as dictionaries, arrays, strings, and binary data.
- The serialization only preserves the values of the objects and their position in the hierarchy.
- Multiple references to the same value object might result in multiple objects when deserialized.
- The mutability of the objects is not maintained.

Coders

- Coder objects read and write objects by sending one of two messages to the objects to be encoded or decoded.
- A coder sends encodeWithCoder: to objects when creating an archive and initWithCoder: when reading an archive.
- These messages are defined by the NSCoding protocol.
- Only objects whose class conforms to the NSCoding protocol can be written to an archive.

NSCoding

- Formal protocol for en- /decoding objects
  - (void)encodeWithCoder:(NSCoder *)encoder
  - (id)initWithCoder:(NSCoder *)decoder
- NSCoder provides methods for (re)storing all basic data structures
- NSKeyed(Un)Archiver to (un)archive objects into data streams (NSData)

Encoding an Object

- When an object receives an encodeWithCoder: message, it should encode all of its vital state (often represented by its properties or instance variables), after forwarding the message to its superclass if its superclass also conforms to the NSCoding protocol.
- An object does not have to encode all of its state.
- For example, suppose you created a Person class that has properties for first name, last name, and height; you might implement encodeWithCoder: as follows

```
- (void)encodeWithCoder:(NSCoder *)coder{
[coder encodeObject:self.firstName forKey:ASCPersonFirstName];
[coder encodeObject:self.lastName forKey:ASCPersonLastName];
[coder encodeFloat:self.height forKey:ASCPersonHeight];
}
```

Decoding an Object

- In the implementation of an initWithCoder: method, the object should first invoke its superclass's designated initializer to initialize inherited state, and then it should decode and initialize its state.
- The keys can be decoded in any order.
- Person's implementation of initWithCoder: might look like this:

```
-(id)initWithCoder:(NSCoder *)coder {
 self = [super init];
if (self) {
  _firstName = [coder decodeObjectForKey:ASCPersonFirstName];
_lastName = [coder decodeObjectForKey:ASCPersonLastName];
_height = [coder decodeFloatForKey:ASCPersonHeight]; }
return self;
}
```

Saving Object

- Objects must implement a protocol.
- Archives save objects and associated references to disk which can be read back.
- Very similar to how Java handles its serialization.
- Can be slow and take a lot of memory when dealing with large object graphs.

NSKeyed(Un)Archiver

- Identity and relationships are preserved between objects and their values.
- Converts objects to machine independent stream of bytes.
- Archives support Objective-C Objects, scalars, arrays, and structures.
- Union, void *, function pointers, and long chains of pointers are not supported.

Creating an Archive

- The following code fragment, for example, archives a custom object called aPerson directly to a file.

```
Person *aPerson = <#G2et a Person#>;
NSString *archivePath = <Path for the archive#>;
BOOL success = [NSKeyedArchiver archiveRootObject:aPerson toFile:archivePath];
```

Decoding an Archive

- The following code fragment, for example, unarchives a custom object called aPerson directly from a file.

```
NSString *archivePath = <Path for the archive#>;
Person *aPerson = [NSKeyedUnarchiver unarchiveObjectWithFile:archivePath];
```

KEY VALUE OBSERVING

KVO

- Change notifications to model classes
- Observer pattern
- Based on KVC

How it works?

Subscribe:

addObserver:forKeyPath:options:context: opitons:
NSKeyValueObservingOptionOld
NSKeyValueObservingOptionNew

Unsubscribe:

removeObserver:forKeyPath:

Receive notification:

observeValue:forKeyPath: ofObject:change:context:

NSData

- Writing NSData

  BOOL success = [myData writeToFile:path options:NSDataWritingAtomic error:&error];

  - NSDataWritingAtomic
    - Writes data in a temporary file, then exchanges the file when complete
  - NSDataWritingWithoutOverwriting
    - Preserves the existing file

NSData

- Reading NSData

```
[NSData alloc]
initWithContentsOfFile:path
options:NSDataReadingMappedIfSafe
error:&error];
```

Property Lists

- Property lists are serialized collections
  NSArray or NSDictionary
  XML or binary
- Can be read/written directly
- Plists can be included as resources in a project

Types Of Property List

- XML Property List
  - Standard property list
  - Editable by hand
  - Device Independent
- Binary Property List
  - Fast loading
- ASCII Legacy Property List
  - Read only

Supported types

type	XML Element	Objective C Object	Core Foundation
array	<array>	NSArray	CFArray
dictionary	<dict>	NSDictionary	CFDictionary
string	<string>	NSString	CGString
data	<data>	NSData	CFData
date	<date>	NSDate	CFDate
integer	<integer>	NSNumber	CFNumber
float	<real>	NSNumber	CFNumber
boolean	<true/> or <false/>	NSNumber	CFBoolean

Writing Property List in Code

- Property Lists included with bundles are read only

```
[NSPropertyListSerialization propertyListWithData: data
                                          options: NSPropertyListImmutable
                                           format: &format
                                            error: &error];
```

- Options
  - NSPropertyListMutableContainers
  - NSPropertyListMutableContainersAndLeaves

Reading Property List in Code

```
[NSPropertyListSerialization dataWithPropertyList:plist
                                           format:NSPropertyListXMLFormat_v1_0
                                          options:0
                                            error:&error];
```

- Your property list must have one object as the root object.

Working with ASCII Property List

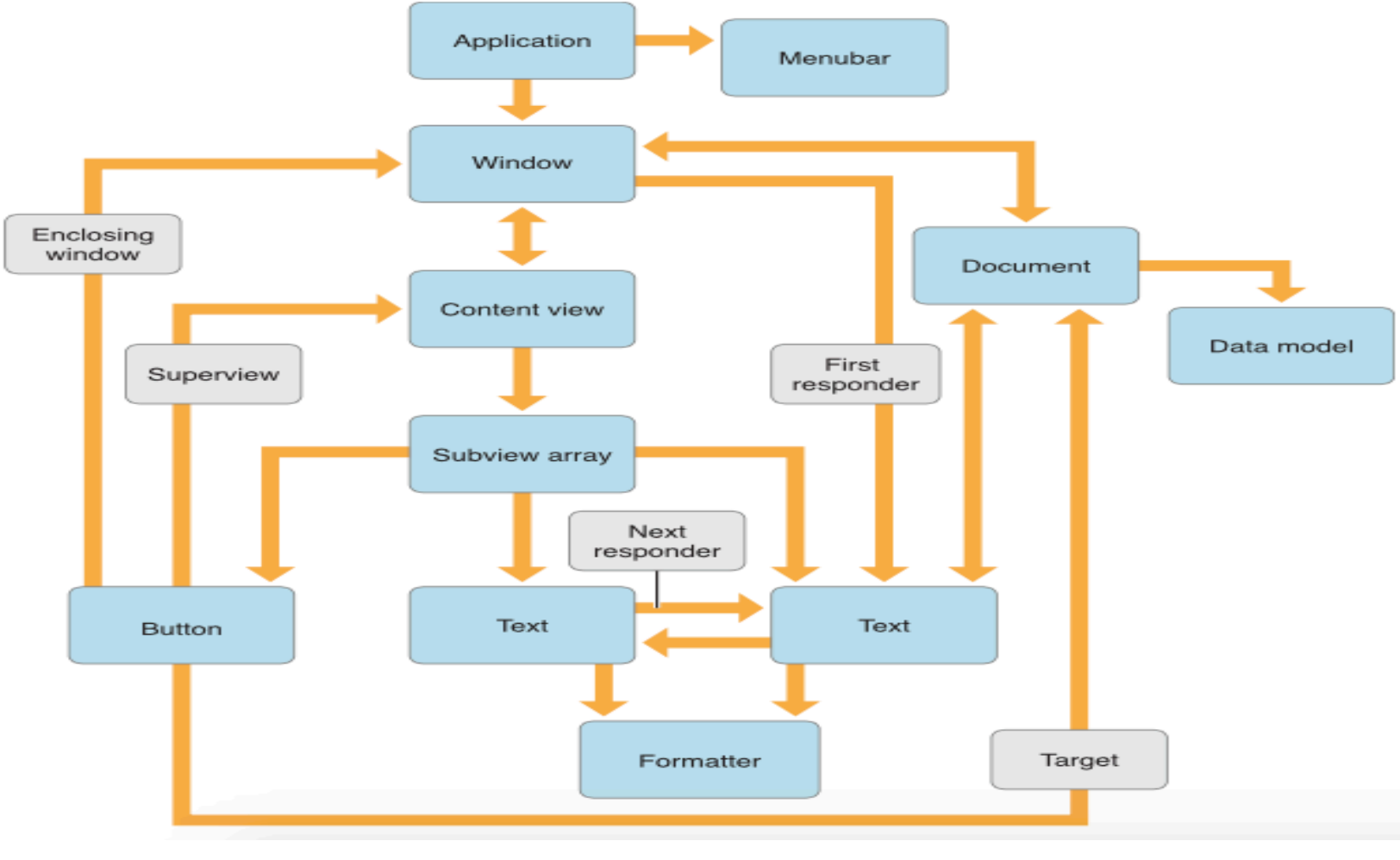
```
NSPropertyListFormat format = NSPropertyListOpenStepFormat
[NSPropertyListSerialization propertyListWithData:data
                                          options:NSPropertyListMutableContainers
                                           format:&format
                                            error:&error];
```

- Read only
- Supports only four types: NSData, NSArray, NSDictionary, NSString

Best Practices

- Try to use binary property lists
- Store in XML if you need often need to manually edit the property list
- Avoid using Property Lists:
  - Avoid large amount data, especially binary data
  - Saving complex object graphs

Object Graph



Object Graph

- Saving an section of object graph in a file and sending it over the network and then reconstructing
- Nib/Storyboard/Property list are examples where object graphs are saved to a file
- Nib/Storyboard files are archives to represent the complex UI relationships
- Property lists are serializations that store the simple hierarchical relationships of basic value objects

Plist vs. NSCoding

	plist	NSCoding
NSNumber	Y	Y
NSString	Y	Y
NSDictionary	Y	Y
NSArray	Y	Y
NSData	Y	Y
NSDate	Y	Y
NSSet	N	Y
Blocks	N	N
Anything Else	N	Y

Functional Programming

Blocks are more than functions

Blocks

- Apple extension to the C language
- Similar to closures, lambdas, etc.
- Encapsulate code like a function, but with extra “magic”

Your First Block

```
void (^helloWorldBlock)(void) = ^ {
    printf ("Hello, World!\n");
};
```

Calling Your First Block

```
void (^helloWorldBlock)(void) = ^ {
    printf ("Hello, World!\n");
};

helloWorldBlock();
```

Blocks That Return

- Blocks return just like functions.

```
int (^fortyTwo)(void) = ^{
    return 42;
}

int a = fortyTwo();
```

Blocks That Take Arguments

```
^int (int addend) {
    return addend + 1;
}
int (^incBlock) (int) = ^(int addend)
{
 return addend + 1;
};
```

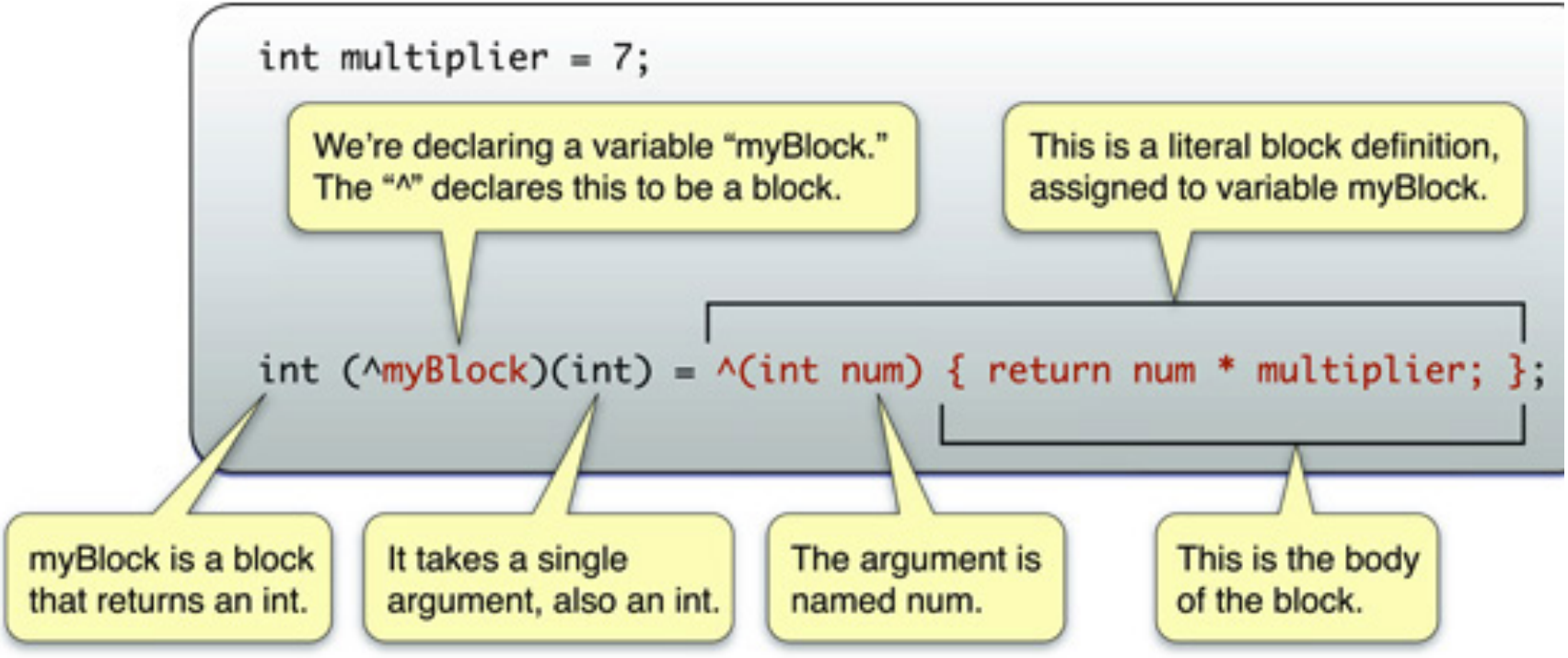
Blocks Capture Scope

```
int a = 42;

void (^myBlock)(void) = ^{
    printf("a = %d\n", a);
};

myBlock();
```

```
int a = 42;

void (^myBlock)(void) = ^{
    a++;
    printf("a = %d\n", a);
};

myBlock();
```

```
__block int a = 42;

void (^myBlock)(void) = ^{
    a++;
    printf("a = %d\n", a);
};

myBlock();
```

Block Syntax



How Do I Declare A Block in Objective-C?

- As a **local variable**:

```
returnType (^blockName)(parameterTypes) =
                        ^returnType(parameters) {...};
```

- As a **property**:

```
@property (nonatomic,
copy) returnType (^blockName)(parameterTypes);
```

- As a **method parameter**:

```
- (void)someMethodThatTakesABlock:
        (returnType (^)(parameterTypes))blockName;
```

How Do I Declare A Block in Objective-C?

- As an **argument to a method call**:

```
[someObject
someMethodThatTakesBlock:^returnType (para
meters)          {...}];
```

- As a **typedef**:

```
typedef returnType (^TypeName)
(parameterTypes);
TypeName blockName =
^returnType(parameters) {...};
```

Block Typedefs

- Simple block that returns an int and takes a float:

```
typedef int (^floatToIntBlock)(float);
Now initialize it:
floatToIntBlock foo = ^(float a) {
    return (int)a;
};
```

Using Blocks as Arguments

- Method Signature:

```
- (void)doThisBlock:(void (^)(id obj))block
```

returns void, takes id argument

Block Typedefs As Arguments

- First typedef the block, then use it as an argument:

```
typedef int (^floatToIntBlock)(float);

void useMyBlock(floatToIntBlock
foo);
```

Block Scope

```
void (^myBlock)(void);
if (someValue == YES) {
    myBlock = ^{ printf("YES!\n") };
} else {
    myBlock = ^ { printf("NO!\n") };
}
```

Blocks Retain Objects

```
NSString *helloWorld = [[NSString alloc]
initWithString:@"Hello, World!"];
void (^helloBlock)(void) = ^{
   NSLog(@"%@", helloWorld);
};
[helloWorld release];
helloBlock();
```

What Are Blocks Good For?

- Replacing Callbacks
- Notification Handlers
- Enumeration

What Are Blocks Good For?

- Replacing Callbacks
- Notification Handlers
- Enumeration

Replacing Callbacks

- Old UIView animations:

```
[UIView beginAnimations:@"foo" context:nil];
[UIView setAnimationDelegate:self];
[UIView
setAnimationDidStopSelector:@selector(animationDidStop:finished:context:)
];
```

- Then you implement the callback

Replacing Callbacks

```
- (void)animationDidStop:(NSString *)animationID finished:(NSNumber
*)finished context:(void *)context
{
    –if ([animationID isEqualToString:kIDOne]) {
        •…
    –} else if ([animationID isEqualToString:kIDTwo]) {
        •…
    –}
•}
```

Replacing Callbacks

- New UIView Animations:

```
[UIView animateWithDuration:0.3
                 animations:^{
                     [myView setAlpha:0.0f];
                 }
                  completion:^(BOOL finished) {
                     [self doSomething];

                 }];
```

Replacing Callbacks

- New UIView Animations:

```
[UIView animateWithDuration:0.3
                 animations:^{
                     [myView setAlpha:0.0f];
                 }
                  completion:^(BOOL finished) {
                     [self doSomething];

                 }];
```

Replacing Callbacks

- New UIView Animations:

```
[UIView animateWithDuration:0.3
                 animations:^{
                     [myView setAlpha:0.0f];
                 }
                  completion:^(BOOL finished) {
                     [self doSomething];

                 }];
```

Replacing Callbacks

- New UIView Animations:

```
[UIView animateWithDuration:0.3
                 animations:^{
                     [myView setAlpha:0.0f];
                 }
                  completion:^(BOOL finished) {
                     [self doSomething];
                 }];
```

What Are Blocks Good For?

- Replacing Callbacks
- Notification Handlers
- Enumeration

Notification Handlers

- Old Notifications:

```
[notificationCenter addObserver:self
                       selector:@selector(foo:)
                           name:kMyNotification
                         object:myObject];
```

Notification Handlers

- Old Notifications: Userinfo Dictionaries

```
- (void)foo:(NSNotification *)aNotification
{
   NSDictionary *userInfo = [aNotification userInfo];
   NSString *UID = [userInfo objectForKey:kIDKey];
   NSNumber *value = [userInfo objectForKey:kValueKey];
}
```

Notification Handlers

- New Notification Handlers:

```
[notificationCenter addObserverForName:kName
                                object:myObject
                                 queue:nil
                            usingBlock:^(NSNotification *aNotification) {
    [myObject doSomething];
}];
[myObject startLongTask];
```

What Are Blocks Good For?

- Replacing Callbacks
- Notification Handlers
- Enumeration

Enumeration

- Basic Enumeration

```
for (int i = 0; i < [myArray count]; i++)
{
    id obj = [myArray objectAtIndex:i];
    [obj doSomething];
}
```

Enumeration

- But...

```
for (int i = 0; i < [myArray count]; i++) {
    id obj = [myArray objectAtIndex:i];
    [obj doSomething];

    for (int j = 0; j < [obj count]; j++) {
        // Very interesting code here.
    }
}
```

Enumeration

- NSEnumerator

```
NSArray *anArray = // ... ;
NSEnumerator *enumerator = [anArray objectEnumerator];
id object;
while ((object = [enumerator nextObject])) {
    // do something with object...
}
```

Enumeration

- Fast Enumeration (10.5+)

```
for (id obj in myArray) {
    [obj doSomething];
}
```

Enumeration

- Block-based enumeration

```
[myArray enumerateObjectsUsingBlock:
    ^(id obj, NSUInteger idx, BOOL *stop) {
        [obj doSomething];
    }];
```

Enumeration

- So why use block-based enumeration?
  - -enumerateObjectsWithOptions:usingBlock:
    - NSEnumerationReverse
    - NSEnumerationConcurrent
  - Concurrent processing!

What Are Blocks Good For?

- Replacing Callbacks
- Notification Handlers
- Enumeration

What Are Blocks Good For?

- Replacing Callbacks
- Notification Handlers
- Enumeration
  - Sorting

Sorting

- NSArray
  - -sortedArrayWithOptions:usingComparator:
  - Concurrent sorting for free!

CONCURRENCY

The main thread

- The main thread is the app's initial thread and from there all the other threads are spawned.
- The main thread is responsible for handling user interface events and also for drawing the UI.
- Most of your app's activities take place on the main thread.
- Whenever the user taps a button in your app, it is the main thread that performs your action method.
- **Never block the main thread.**
-

Process

- The runtime instance of an application or program. A process has its own virtual memory space and system resources (including port rights) that are independent of those assigned to other programs.
- A process always contains at least one thread (the main thread) and may contain any number of additional threads.

- Multitasking is something that happens between different apps.
- Each app is said to have its own **process** and each process is given a small portion of each second of CPU time to perform its jobs.
- Then it is *pre-empted* and control is given to the next process.

Threads

- Each process contains one or more **threads**. Each process in turn is given a bit of CPU time to do its work.
- The process splits up that time among its threads.
- Each thread typically performs its own work and is as independent as possible from the other threads within that process.
- Each thread has its own stack space but otherwise shares memory with other threads in the same process.

Golden Rules

- The main thread is for the user
- UI can only be manipulated on main thread
- Networking must be asynchronous
- Don’t directly create threads
- Don’t block the user

Concurrency Options in iOS

- Use classes with built in concurrency
  - Many classes have completion, failure blocks
- Grand Central Dispatch
  - Simple lightweight C function calls
- NSOperation and NSQueue
- Manual multithreading code (NSThread,etc)

URL CONNECTION

- Make a url string
  - NSString *urlString = @"http://images.apple.com/v/iphone-6/a/images/overview/hero_large.jpg";
- Make a URL object
  - NSURL *url = [NSURL URLWithString:urlString];
- Make a request object
  - NSURLRequest * requestURL = [NSURLRequest requestWithURL:url];
- Get the response object
  - 3 approaches

1ST APPROACH(DELEGATE)

- Conform to URLConnectionDelegate
- Make the URL connection
  - self.urlConnection = [NSURLConnection connectionWithRequest:requestURL delegate:self];
- Implement connection delegate methods

  -(void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error

  - (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)response

  - (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data

  - (void)connectionDidFinishLoading:(NSURLConnection *)connection

2ND APPROACH (SYNCHRONOUS CONN)

```
NSURLResponse *response;
NSError *error;
self.urlData = [[NSURLConnection
sendSynchronousRequest:requestURL
returningResponse:&response
error:&error] mutableCopy];
```

3RD APPROACH(ASYNC CONN)

```
[NSURLConnection
sendAsynchronousRequest:requestURL queue:
[NSOperationQueue new]
completionHandler:^(NSURLResponse *response, NSData
*data, NSError *connectionError) {
      if(connectionError == nil)
          dispatch_async(dispatch_get_main_queue(), ^{
            self.imageview.image = [UIImage
imageWithData:data];
          });

   }];
```


GCD

Definition

- Grand Central Dispatch, or GCD for short, is a low-level C API that works with block objects. The real use for GCD is to dispatch tasks to multiple cores without making you, the programmer, worry about which core is executing which task.
- Blocks, together with Grand Central Dispatch (GCD), create a harmonious environment in which you can deliver high-performance multithreaded apps in iOS

Grand Central Dispatch

- GCD makes writing concurrent code easy
  - So you can have more responsive UI
- Working with queues
  - Main queues
- Can create custom queue

Using GCD

1. Create a queue
   - dispatch_queue_create
2. Add tasks(blocks) to the queue
   - dispatch_async
3. Use main queue for UI updations

GCD

- Efficient scheduling of multiple independent chunks of work
- Abstract threading details away from the programmer
  - OS manages and determines optimal number of threads , etc.
- GCD provides FIFO queues to which you submit task
- C API and runtime engine
  - Open sourced as libdispatch

Dispatch Queues

- Dispatch queues are pools of threads managed by GCD on the host operating system
- Heart of GCD are dispatch queues
- All methods and data types available in GCD start with a dispatch_ keyword.

Dispatch Queues

- Dispatch queues are extremely lightweight
  - GCD queue overhead = 256 bytes
  - Blocks submitted to dispatch queues are executed on pool of threads
  - Threading pool avoids cost of creating / detaching new threads

Dispatch Queues

- Maintains list of tasks to execute – submitted as blocks
- Concurrent – tasks start in FIFO order but can run concurrently
- Serial – task execute and complete one at a time in FIFO order
- Main – tasks execute serially on your application's main UI
- Suspend, resume, cancel, prioritize

Debugging

- If your app crashes with EXC_BAD_INSTRUCTION or SIGABRT, the Xcode debugger will often show you an error message and where in the code the crash happens.
- If Xcode thinks the crash happened on **AppDelegate** (not very useful!), enable the Exception Breakpoint to get more info.
- If the app crashes with a SIGABRT but there is no error message, then disable the Exception Breakpoint and make the app crash again. (Alternatively, click the **Continue program execution** button from the debugger toolbar a few times. That will also show the error message.)

Debugging

- An EXC_BAD_ACCESS error usually means something went wrong with your memory management. An object may have been "released" one time too many or not "retained" enough. With Swift these problems are mostly a thing of the past because the compiler will usually make sure to do the right thing. However, it's still possible to mess up if you're talking to Objective-C code or low-level APIs.
- EXC_BREAKPOINT is not an error. The app has stopped on a breakpoint, the blue arrow pointing at the line where the app is paused. You set breakpoints to pause your app at specific places in the code, so you can examine the state of the app inside the debugger. The "Continue program execution" button resumes the app.