# Effective Objective-C 2.0

# Accustoming yourself to Objective-C

## Item 1: Familiarize yourself with Objective-C’s roots

* Messaging and function calling

// Messaging (Objective-C): Runtime decides which code gets executed

Object \*obj = [Object new]; [obj performWith:parameter1 and:parameter2];

// Function calling (C++): Compiler decides which code will be executed

Object \*obj = new Object; obj->perform(parameter1, parameter2);

* The memory for Objective-C objects is always allocated in heap space and never on the stack

NSString \*str = @”Some string”; // Always a pointer

* NSString str; // Error: interface type cannot be statically allocated

NSString \*anotherStr = str; // Point to the same object

Can still use C struct which contains no Objective-C objects

## Item 2: Minimize importing headers in headers

* Always import headers at the very deepest point possible. This usually means forward declaring classes in a header and importing their corresponding headers in an implementation. Doing so avoids coupling classes together as much as possible

Forward declaration: @class EOCEmployer;

* Sometimes, forward declaration is not possible, as when declaring protocol conformance, like inheritance or implementing.

## Item 3: Prefer literal syntax over the equivalent methods

NSNumber \*someNumber = [NSNumber numberWithInt:1];

NSNumber \*someNumber = @1;

NSNumber \*floatNumber = @2.5f;

NSNumber \*doubleNumber = 3.1415926;

NSNumber \*boolNumber = @YES;

NSNumber \*charNumber = @’a’;

int x=5; float y = 6.32f; NSNumber \*expressionNumber = @(x \* y);

* Literal Arrays:

NSArray \*animals = [NSArray arrayWithObjects:@”cat”, @”dog”, nil];

NSArray \*animals = @[@”cat”, @”dog”]; // Literal array not allow nil

id obj1; id obj2; id obj3;

NSArray\* arrayA = [NSArray arrayWithObjects: obj1, obj2, obj3, nil];

NSArray\* arrayB = @[obj1, obj2, obj3];

// if obj2 is null, arrayA is created with only 1 element obj1.

// ArrayB throws exception

NSString \*dog = [animals objectAtIndex:1];

NSString \*dog = animals[1];

* Literal Dictionaries

NSDictionary \*personData = [NSDictionary dictionaryWithObjectsAndKeys:

@”Matt”, @”firstName”, @”Galloway”, @”lastName”,

[NSNumber numberWithInt:28], @”age”, nil];

// <object>, <key>, <object>, <key>, …

NSDictionary \*personData = @{@”firstName” : @”Matt”, @”lastName” : @”Galloway”, @”age” : @28 }; // Not allow nil

NSString \*lastName=[personData objectForKey:@”lastName”];

NSString \*lastName=personData[@”lastName”];

* Mutable arrays and dictionaries

[mutableArray replaceObjectAtIndex:1 withObject:@”dog”];

[mutableDictionary setObject:@”Galloway” forKey:@”lastName”];

mutableArray[1] = @”dog”;

mutableDictionary[@”lastName”] = @”Galloway”;

* Limitations
  + The class of the created object must be the one from the Foundation framework
  + Only immutable variants can be created with the literal syntax. Use mutable copy to create a mutable variant.

NSMutableArray \*mutable = [@[@1, @2] mutableCopy];

## Item 4: Prefer typed constants to pre-processor #define

#define ANIMATION\_DURATION 0.3

static const NSTimeInterval kAnimationDuration = 0.3

extern NSString \*const EOCStringConstant; // header file

NSString \*const EOCStringConstant = @”Value”; // Implementation file

## Item 5: Use enumerations for states, options and status code

* Unscoped:

enum Color { red, green };

typedef enum Color Color; // As a type, don’t have to always add enum

Color color = red;

enum Color : NSInteger; // C++ 11 allows type

Color color = red | green; // Actually the bit ‘Or’ of red(0) and green(1) value

* Scoped:

enum class Color { red, green };

Color color = Color::red;

* Objective-C macro for enum define

NS\_ENUM

typedef NS\_ENUM(NSUInteger, Color) { Red, Green };

NS\_OPTION: Allows Or

typedef NS\_OPTION(NSUInteger, Direction) { Left, Right, Up, Down };

Direction direction = Left | Up;

* Switch: Better not to have a default branch, because if new value added to enum, compiler can give warning on not handled case

# Objects, Messaging, and the Runtime

## Item 6: Understanding properties

* Auto generate variable and method by compiler

@interface Person : NSObject

@**property** NSString\* firstName

@end

@implementation Person

@**synthesize** firstName = \_firstName; // Can use @synchesize to change the

auto synthesized variable \_firstName name

//@**dynamic** firstName; // can prevent generating set/get methods

@end

Equivalent to class like:

@interface Person : NSObject {

NSString\* \_lastName; // auto-synthesized instance variable

}

-(NSString\*) firstName;

-(void)setFistName: (NSString\*)firstName;

@end

[person.firstName=@”Bob](mailto:person.firstName=@)” // The same as [person setFistName:@”Bob”];

person.firstName; // The same as [person firstName];

* Property attributes
  + Atomicity:
    - By default, synthesized accessors include locking to make atomic.
    - Use nonatomic to make no locking.
  + Read/Write
    - readwrite: Both getter and setter are available
    - readonly: Only getter is available
  + Memory-management semantics
    - assign: Simple assign operation
    - strong: Owning relationship. When a new value is set, it is first retained, the old value is released, and then the value is set.
    - weak: Non-owning relationship. When a new value is set, it is not retained; nor is the old value released. Similar to assign but the value is nilled out when the object destroyed.
    - unsafe\_unretained: Same semantics as assign but used where the type is an object type to indicate a nonowning relationship (unretained) that is not nilled out (unsafe) when the target is destroyed, unlike weak.
    - Copy: Owning relationship similar to strong, instead of retaining the value, it is copied.
  + Method names
    - getter=<name>: Name for getter. Usually for Boolean property which want the getter to be prefixed with is
    - setter=<name>: Name for setter.
* Ensure that anywhere a property’s backing instance variable is set, the declared semantics are adhered to
* Use nonatomic on iOS for performance

## Item 7: Access instance variables primarily directly when accessing them internally

* Prefer to read through instance variable internally and write data through properties
* With initializer and dealloc, always read and write data directly through instance variables. Subclass may override the function
* Need to read through property if the data is lazy initialized

## Item 8: Understand object equality

* Provide isEqual: and hash methods for objects need to check for equality
* Determine what is necessary to test for equality (like primary key) rather than testing all properties.
* Write hash method that will be quick but provide a reasonably low level of collisions.

## Item 9: Use the class cluster pattern to hide implementation detail

* Class cluster pattern: An abstract class that groups a set of private concrete subclasses, providing a simplified interface to the user through the abstract class. Like Abstract Factory pattern.

## Item 10: Use associated objects to attach custom data to existing classes

* Association management methods:

void objc\_setAssociatedObject(id object, void\* key, id value, objc\_AssocationPolicy policy)

id objc\_getAssociatedObject(id object, void\* key)

void objc\_removeAssociatedObjects(id object)

Key is treated purely as an opaque pointer

* Memory-management semantics of associated objects can be defined to mimic owning or non-owning relationship.
* Associated objects should be used only when other approach is not possible, since they can easily introduce hard-to-find bugs.

## Item 11: Understand the role of objc\_msgSend

* void objc\_msgSend(id self, SEL cmd, …);
* A message consists of a receiver, a selector, and parameters. Invoking a message is synonymous with calling a method on an object
* When invoked, all messages go through the dynamic message dispatch system whereby the implementation is looked up and then run

## Item 12: Understand message forwarding

* Message forwarding is the process that an object goes through when it is found to not respond to a selector
* Dynamic method resolution: Add methods to a class at runtime as and when they are used

+(BOOL)resolveInstanceMethod: (SEL)selector

BOOL class\_addMethod(Class cls, SEL name, IMP imp, const char \*types);

* Replacement receiver: Return another object to handle the selector

-(id)forwardingTargetForSelector: (SEL)selector

* Full forwarding is invoked only when no previous way of handling a selector is found

-(void)forwardInvocation: (NSInvocation\*)invocation;

## Item 13: Consider method swizzling to debug opaque methods

* Method implementation for a given selector of a class can be added or replaced at runtime
* Swizzling is the process of swapping one method implementation for another, usually to add functionality to the original implementation

#include <objc/runtime.h>

Method class\_getInstanceMethod(Class cls, SEL selector);

void method\_exchangeImplementation(Method m1, Method m2);

// Add logging to NSString lowercaseString

@interface NSString(MyAddition)

-(NSString\*)myLowercaseString;

@end

@implementation NSString(MyAddition)

-(NSString\*)myLowercaseString {

NSString\* lowercase = [self myLowercaseString]; // Swapped, actually calling lowercaseString

// Do logging

return lowercase;

}

@end

Method orgMethod=class\_getInstanceMethod(

[NSString class], @selector(lowercaseString));

Method newMethod=class\_getInstanceMethod(

[NSString class], @selector(myLowercaseString));

method\_exchangeImplementation(orgMethod, newMethod);

* Meddling with methods through the runtime is usually good only for debugging and should not be used just because it can

## Item 14: Understand what a class object is

* Introspection methods
  + isMemberOfClass: Check whether an object is an instance of a certain class
  + isKindOfClass: Check whether an object is an instance of a certain class or any class that inherits from it

# Interface and API design

## Item 15: Use prefix names to avoid namespace clashes

* Choose a class prefix and stick with the prefix throughout also in C functions. The prefix should be at least 3 characters because Apple reserved 2 characters
* Consider prefixing third-party library also if used in the own library

## Item 16: Have a designated initializer

* Implement a designated initializer in the class, and document which one it is. All other initializers should call through to this one
* If the designated initializer is different from the superclass, ensure that its designated initializer is overridden
* Throw an exception in initializers overridden from superclasses that should not be used in the subclass.

## Item 17: Implement the description method

* Implement the description method to provide a meaningful string description of instances

-(NSString\*) description;

NSLog(@(“object=%@”, obj);

* If the object description could do with more detail for use during debugging, implement debugDescription

## Item 18: Prefer immutable objects

* When possible, create objects that are immutable
* Extend read-only properties in a class-continuation category to read-write if the property will be set internally

@interface Person : NSObject

@property(readonly)name

@end

@implementation Person

@property(readwrite)name

@end

// But still can be set using Key-Value Coding (KVC)

[person setValue:@”bob” forKey:@”name”];

* Provide methods to mutable collections held by objects rather than exposing a mutable collection as a property

## Item 19: Use clear and consistent naming

* Some rules for method naming
  + If the method returns a newly created value, the first word in the method name should be its type (e.g stringWithString), unless a qualifier needs to go in front (e.g localizedString).
  + A parameter should be immediately preceded by a noun describing its type
  + Methods that cause an action to happen on an object should contain a verb followed by a noun (or multiple nouns) for parameters
  + Avoid abbreviations
  + Prefix Boolean property with is or has
  + Reserve the prefix get for use with methods that return values via an out-parameter, such as those that fill a C-style array

- (NSString\*)stringByReplacingOccurrencesOfString:(NSString \*)target withString:(NSString \*)replacement;

## Item 20: Prefix private method names

* Prefix private method names so that they are easily distinguished from public methods. And class inheritance will have no collision
* Avoid using a single underscore as the method prefix, since it is reserved by Apple.

## Item 21: Understand the Objective-C error model

* Use exceptions only for fatal errors that should bring down the entire application
  + Automatic Reference Counting (ARC) is not exception safe by default. Compiler flag

–fobjc-arc-exceptions allows creating exception-safe code with extra code cost.

* For nonfatal errors, either provide a delegate method to handle errors or offer an out-parameter NSError object

## Item 22: Understand the NSCopying protocol

* Implement the NSCopying protocol if the object needs to be copied
* If the object has mutable and immutable variants, implement bot NSCopying and NSMutableCopying protocols
* Decide whether a copy will be shallow or deep, and prefer shallow for a normal copy
* Consider adding a deep-copy method if it is useful

# Protocols and categories

## Item 23: Use delegate and data source protocols for inter-object communication

* Use the delegate pattern to provide an interface to the object that needs to tell other objects about pertinent events
* Define a protocol with potentially optional methods to define the interface that the delegate should support

@protocol NetworkFetcherDelete

@optional //

-(void)networkFetcher:(NetworkFetcher\*)fetcher

didReceiveData:(NSData\*)data;

-(void)networkFetcher:(NetworkFetcher\*)fetcher

didFailWithError:(NSError\*)error;

@end

@interface NetworkFetcher : NSObject

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

// Property must be defined as weak to be a nonowning relationship.

// Otherwise there will retain cycle

@end

// Check whether a delegate implements the method

if ([\_delegate

respondsToSelector:@selector(networkFetcher:didReceiveData:)] {

[\_delegate networkFetcher:self didReceiveData:data];

}

## Item 24: Use categories to break class implementations into manageable segments

@interface Person : NSObject

@property(nonatomic, copy, readonly) NSString\* name;

@end

@interface Person(Friendship) // Category

-(void)addFriend:(Person\*)person;

@end

@interface Person(Private) // Can be used to indicate private methods

@end

Files: Person+Friendship(.h/.m) Person+Private(.h/.m)

## Item 25: Always prefix category names on third-party classes

* Always prepend your naming prefix to the names of categories you add to classes that are not your own to avoid collision
* Always prepend your naming prefix to the method names within categories you add to classes that are not your own

## Item 26: Avoid properties in categories

* Keep all property declarations for encapsulated data in the main interface definition
* Prefer accessor methods to property declarations in categories, unless it is a class-continuation category

## Item 27: Use the class-continuation category to hide implementation detail

// Person.h

@interface Person : NSObject

@property(nonatomic, copy, readonly) NSString\* name;

@end

// Person.m

@interface Person**()** **{**

@private

NSString \*\_address; // Add instance variables to a class

@property(nonatomic, copy, readwrite) NSString\*name; // Redeclare as readwrite internally

std::string\* str; // Add C++ class internally so user doesn’t have to use Objective-C++ file (.mm)

**}**

@end

@implementation Person

@end

@interface Person()<SomeDelegate> // Declare protocols which the class confirms to privately

@end

## Item 28: Use a protocol to provide anonymous objects

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

-(void)setObject:(id)object forKey:(id<NSCopying>)key;

-(id<Connection>)connectionWithId:(NSString\*)id;

# Memory management

## Item 29: Understand reference counting

* Reference-counting memory management is based on a counter that is incremented and decremented. An object is created with a count of at least 1. An object with a positive retain count is alive. When the retain count drops to 0, the object is destroyed.
* autorelease: Add autorelease to the object allocated in a method and is returned. Put the object in the autorelease pool and release it when the pool is drained.
* As it goes through its life cycle, an object is retained and released by other object holding references to it.
* Retain cycle: When multiple objects reference one another cyclically. Use weak references.

## Item 30: Use ARC to make reference counting easier

* Method-naming rules applied by ARC
  + A method returning an object returns it owned by the caller (who responsible for releasing the returned object) if its method name begins with one of the following:

alloc / new / copy / mutableCopy

* + Any other method name indicates the object will be returned autoreleased, so that the value is alive across the method call boundary. If wants to stay alive longer, the calling code must retain it.
* Qualifiers to alter semantics of local and instance variables
  + \_\_strong: The default; value is retained
  + \_\_unsafe\_unretained: Value not retained and potentially unsafe, as the object may be deallocated already by the time the variable is used again
  + \_\_weak: Value not retained but safe because automatically set to nil if deallocated
  + \_\_autoreleasing: Used when an object is passed by reference to a method. Autoreleased on return.

## Item 31: Release references and clean up observation state only in dealloc

* dealloc method should be used only to release references to other objects and to unregister anything that needs to be
* Use other method to release system resources instead of dealloc
* Avoid method calls in dealloc in case those methods try to perform asynchronous work and assume the object is in a normal state.

## Item 32: Beware of memory management with exception-safe code

* When exceptions are caught, care should be taken to ensure that any required cleanup is done for objects created within the try block
* By default, ARC does not emit code that handles cleanup when exceptions are thrown. This can be enabled with a compiler flag –fobjc-arc-exceptions but produces code that is larger and comes with a runtime cost

## Item 33: Use weak references to avoid retain cycles

* Retain cycles can be avoided by making certain references weak
* General rule is that if don’t own an object, should not retain it
* Weak references may or may not be auto-nilling. Autonilling is a new feature with ARC and implemented in the runtime.

## Item 34: Use autorelease pool blocks to reduce high-memory waterline

* Autorelease pools are arranged in a stock, with an object being added to the topmost pool when it is sent the autorelease message
* Correct application of autorelease pools can help reduce the high-memory waterline
* Modern autorelease pools using the new @autoreleasepool syntax are cheap

## Item 35: Use zombies to help debug memory-management problems

* When an object is deallocated, it can optionally be turned into a zombie instead of being deallocated. Use NSZombieEnabled environment flag to turn on this feature
* Manipulating isa pointer to change the object into a zombie class. A zombie class responds to all selectors by aborting the app after printing an error message

## Item 36: Avoid using retainCount

* The retain count of an object is not accurate and not useful
* retainCount will cause a compilation error in ARC

# Blocks and Grand Central Dispatch

## Item 37: Understand blocks

* Blocks are lexical closures for C, C++ and Objective-C:
  + ^ { … }
  + Block type: return\_type (^block\_name)(parameters)

void (^someBlock)() = ^{…}

int (^addBlock)(int a, int b) = ^(int a, int b) { return a+b; }

int sum = addBlock(2 + 5);

* Variables available to the scope in which the block is declared also available inside the block but not modifiable. Add \_\_block qualifier to be modifiable.

int additional = 0;

int (^addBlock)(int a) = ^(int a) { return a + additional; }

NSArray\* array = @[@0, @1, @2];

\_\_block NSInteger sum = 0;

[array enumerateObjectsUsingBlock:

^(NSNumber\* number, NSUInteger idx, BOOL\* stop) {

sum += number;

}]; // inline block

* Blocks can be considered as an object
* Blocks can be stack allocated, heap allocated, or global.

void (^block)(); // Variable to store the block

if () { block = ^{…} }

else { block = ^{…} } // Wrong because block is stack allocated

if () { block = [^{…} copy]; }

else { block = [^{…} copy]; } // Copy into heap allocated

## Item 38: Create typedefs for common block types

* typedef return\_type (^block\_name)(parameters);
* Follow naming conventions when definng new types so won’t clash with others

## Item 39: Use handler blocks to reduce code separation

* Use a handler block when it is useful to have the business logic of the handler be declared inline with the creation of the object.
* Handlers block is associated with the object directly rather than delegation, which often requires switching based on the object if multiple instances are being observed
* Consider passing a queue as parameter to designate on which queue the block should be enqueued

## Item 40: Avoid retain cycles introduced by block referencing object owning them

* Be aware of the retain cycles by blocks that capture objects that directly or indirectly retain the block
* Ensure that retain cycles are broken at an opportune moment, but never leave responsibility to the consumer of the API.

typedef void (^CompletionHandler)(NSData\* data);

@interface NetworkFetcher

-(void)startWithCompletionHandler:(CompletionHandler)handler {

self.completionHandler = handler;

completionHandler = null; // To break the cycle

}

@end

@implementation SomeClass {

NetworkFetcher\* \_networkFetcher;

NSData\* \_fetchedData;

}

-(void)downloadData {

[\_networkFetcher startWithCompletionHandler:^(NSData\* data) {

self.\_fetchedData = data;

// SomeClass is retained by block through self

}

Q}

@end

Q// Retain cycle: SomeClass.\_networkFetcher ->

NetworkFetcher.completionHandler -> Block ->

SomeClass.\_fetchedData

## Item 41: Prefer dispatch queues to locks for synchronization

* Dispatch block in a serial queue can provide synchronization semantics and offer a simpler alternative to @synchronized blocks or NSLock objects

dispatch\_sync(queue, ^{…});

* Mixing synchronous and asynchronous dispatches can provide the same synchronized behaviour as with normal locking but without blocking the calling thread in the asynchronous dispatches. There is extra cost of copying block in asynchronous mode.
* Concurrent queues and barrier blocks can be used to make synchronized behaviour more efficient.

A barrier is executed exclusively with respect to all other blocks on that queue. When a queue is processed and the next block is a barrier block, the queue waits for all current block to finish and then executes the barrier block. When the barrier block finishes executing, processing of the queue continues as normal.

void dispatch\_barrier\_async(dispatch\_queue\_t queue, dispatch\_block\_t block);

void dispatch\_barrier\_sync(dispatch\_queue\_t queue, dispatch\_block\_t block);

## Item 42: Prefer GCD to performSelector and Friends

* performSelector family of methods is potentially dangerous with respect to memory management. If it has no way of determining what selector is going to be performed, ARC compiler cannot insert the appropriate memory-management calls.
* performSelector methods family is very limited with respect to the return type and the number of parameters that can be sent to the method
* The methods that allow performing a selector on a different thread are better replaced with certain GCD calls using blocks

[self performSelector:@selector(doSomething)

withObject:nil afterDelay:5.0];

dispatch\_time\_t time = dispatch\_time(DISPATCH\_TIME\_NOW,

(int64\_t)(5.0 \* NSEC\_PER\_SEC));

dispatch\_after(time, dispatch\_get\_main\_queue(),

^(void){[self doSomething]; });

[self performSelectorOnMainThread:@selector(doSomething)

withObject:nil waitUntilDone:NO];

dispatch\_async(dispatch\_get\_main\_queue(),

^{ [self doSomething]; }); Q// Use dispatch\_sync if waitUntilDone=YES

## Item 43: Know when to use GCD and when to use operation

* GCD is a lightweight pure C API, whereas operation queues are Objective-C objects
* Benefits of NSOperation and NSOperationQueue:
  + Cancelling operations
  + Operation dependencies: Certain operations can execute only after another operation has completed successfully.
  + Key-Value Observing of operation properties: isCanceled, isFinished, …
  + Operation priorities
  + Reuse of operations

## Item 44: Use dispatch groups to take advantage of platform scaling

* Dispatch groups are used to group a set of tasks. You can optionally be notified when the group finishes executing
* Dispatch groups can be used to execute multiple tasks concurrently through a concurrent dispatch queue. In this case, GCD handles the scheduling of multiple tasks at the same time, based on system resources.

## Item 45: Use dispatch\_once for thread-safe single-time code execution

* Thread-safe single-code execution

+(id)sharedInstance {

static SomeClass\* sharedInstance = nil;

static dispatch\_once\_t onceToken; // token needs to be the same object

dispatch\_once(&onceToken, ^{

sharedInstance = [[self alloc] init]; });

return sharedInstance;

}

## Item 46: Avoid dispatch\_get\_current\_queue

* dispatch\_get\_current\_queue function does not in general perform as expected. It has been deprectated and should only be used for debugging

dispatch\_sync(queueA, ^{

dispatch\_sync(queueB, ^{

dispatch\_block\_t block = ^{ … };

if (dispatch\_get\_current\_queue() == queueA) block;

else { dispatch\_sync(queueA, block); }) });

* dispatch queues are organized into a hierarchy (queue started by other queue); the current queue cannot be described simply by a single queue object.
* Queue-specific data can be used to avoid deadlocks owing to nonreentrant code

void dispatch\_queue\_set\_specific(dispatch\_queue\_t queue, const void\* key, void\* context, dispatch\_function\_t destructor);

# The System frameworks

## Item 47: Familiar yourself with the system frameworks

## Item 48: Prefer block enumeration to for loops

* NSEnumerator

-(NSArray\*)allObjects

-(id)nextObject

* Fast enumeration

NSArray\* array; for (id object in array) {…}

for (id object in [array reverseObjectEnumerator]) {…}

NSDictionary\* dict; for (id key in dict) { id value = dict[key]; }

* Block-based enumeration

-(void)enumerateObjectsUsingBlock:

(void(^)(id object, NSUInteger idx, BOOL \*stop))block;

-(void)enumerateKeysAndObjectsUsingBlock:

(void(^)(id key, id object, BOOL\* stop))block;

## Item 49: Use toll-free bridging for collections with custom memory-management semantics

* CoreFoundation: C API that mirrors much of the functionality of the Foundation framework
* \_\_bridge cast:

NSArray\* array; CFArrayRef aCFArray = (\_\_bridge CFArrayRef)array;

* Toll-free bridging allows casting between Foundation’s Objective-C objects and CoreFoundation’s C data structures
* Use CoreFoundation to create a collection allows specifying various callbacks that are used when the collection handles the contents to have custom memory-management semantics.

## Item 50: Use NSCache instead of NSDictionary for caches

## Item 51: Keep initialize and load implementation lean

* Classes go though a load phase in which they have the load method called on them if it has been implemented. This method may also be present in categories whereby the class load always happens before the category load. Load method does not participate in overriding.

+(void) load;

* Before a class is used for the first time, it is sent the initialize method. This method does participate in overriding, so it is usually best to check which class is being initialized.

+(void) initialize;

* Both implementations of load and initialize should be kept lean, which helps to keep applications responsive and reduces the likelihood that interdependency cycles will be introduces.

## Item 52: Remember that NSTimer retains its target

* An NSTimer object retains its target until the timer is invalidated either because it fires or though an explicit call to invalidate