Effective Objective-C by Example

Here you'll learn about Objective-C best practices using examples that can be applied in the real world. This is about helping you write faster, cleaner, more readable and overall better Objective-C code. You'll pick up some good habits, find solutions to everyday issues, and learn the things no "Beginning Objective-C" tutorial or book tells you about.

You will also learn why certain practices are better than others, which old habits (from C/C++/Java/etc) to drop, and how this helps you accomplish your goals faster with fewer issues.

Even though these pages are a service to Kobold2D users and often make references to Cocos2D, you can apply what you learn to any Objective-C project.

- Objective-C Frequently Asked Questions (FAQ)
- Objective-C Orientation Course — This section is for those who would want to pick up on Objective-C quickly while avoiding some of the common pitfalls that befall those who apply their previous programming language's habits or best practices.
- Performance Tips — This section helps you write faster Objective-C code. The performance gain may not be much, but these tips are easy to apply and sometimes even make the code easier to read (and write).

Todo list of sorts ...

Since these pages are being filled over the next couple weeks/months, following is a todo list of things that I'd like to explain in more detail. If you want to see something explained in particular, please suggest it if it's not already suggested.

- Performance
  - multiply, don't divide
  - doubles, floats and C math methods with and without f
  - fast enumeration with NSArray and CCArray
  - avoid doing the same thing twice (cache computation results)
- Storing and Accessing Objects and Data
  - accessing another node in the hierarchy
  - storing hierarchical objects in own array - pros/cons
  - Lightweight Objective-C class instead of C struct
  - using basic data types in collections (NSNumber, NSValue, NSData)
  - passing data in initializer, vs properties, vs global storage/singleton
- Guidelines
  - prefer aggregation/composition over inheritance
  - OOP: when to subclass and when not to
  - Apple Cocoa coding guidelines in a nutshell
  - don't prefix/suffix instance variables
    - it's unnecessary
    - ensures instance var and property name match
    - property differentiation: myVar = 10 --> property: self.myVar = 10
    - method parameters: use "aSender" or "theSender" to avoid conflict with instance var "sender"
  - avoid #ifdef as if they were the plague
    - code that's not compiled will break
- Common Compiler Warnings & Errors (what they mean, how to fix)
  - Advice: Treat Warnings as Errors
  - "... may not respond to ..."
  - duplicate symbol(s)
  - undefined symbol(s)
- Common Runtime Errors and how to debug them
  - unrecognized selector
  - deallocated instance
  - zombies, wuaaaah!
- Pro Tips
  - performing selectors
    - perform selector for each object in array
  - sending messages
  - dynamically creating and forwarding messages

Objective-C Frequently Asked Questions (FAQ)

- What are property and synthesize keywords for?

- What is the difference between + and - methods?
- When should I add an instance variable to my class?

What are property and synthesize keywords for?

Objective-C 2.0 introduces properties and dot notation. It's a common feature in many programming languages. Previously to allow other classes access to instance variables, you had to write two methods, usually referred to as getter and setter:

```
@interface MyClass : NSObject
{
    int value;
}
@end

@implementation MyClass
-(void) setValue:(int)aValue
{
    value = aValue;
}
-(int) value
{
    return value;
}
@end
```

You can then get and set the value like this:

```
MyClass* instance = [[MyClass alloc] init];
int val = [instance value] + 10;
[instance setValue:val];
```

Writing these methods becomes tedious and error prone quickly. Especially in pre-ARC code where you had to balance retain counts of objects as well. Therefore properties were introduced. The @property and @synthesize keywords simply automate the creation of getter and setter methods, and if you want also the instance variable itself. The above code can be rewritten as:

```
@interface MyClass : NSObject
@property int value;
@end

@implementation MyClass
@synthesize value;
@end
```

This is a lot less code! The @property keyword declares the property in the @interface section, and if necessary, adds an instance variable of the same name and type to the class behind the scenes. The @synthesize statement then causes the compiler to generate the getter and (if not set to be a readonly property) the setter methods for that property. Again this happens behind the scenes when the code is compiled.

You can still use the same code to access the value as before, and you can use dot notation as an alternative (and more natural) way to access properties of an Objective-C class:

```
MyClass* instance = [[MyClass alloc] init];
int val2 = instance.value + 10;
instance.value = val2;

// this still works and does the exact same thing:
int val = [instance value] + 10;
[instance setValue:val];
```

It's worth mentioning that whichever syntax you use to access properties, they are treated the same way by the Objective-C compiler. There is zero performance or behavioral difference between the two ways of getting and setting a property. It's a purely syntactical choice.

In fact, you can even use dot notation for well-formed but non-@property methods that weren't created by the @synthesize keyword.

What is the difference between + and - methods?

In Objective-C all methods start with either a - or + character, for example:

```
@interface MyClass : NSObject
- (void) aMethod;        // instance method
+ (void) anotherMethod;  // class method
@end
```

The + and - characters specify whether a method is a class method or an instance method respectively. The class methods are sometimes referred to as static methods, although that's not quite correct in Objective-C.

The difference is best seen when you actually call these methods. I'm assuming the class that defines these methods is named `MyClass`.

```
// instance method require an instance of the class
MyClass* instance = [[MyClass alloc] init];
[instance aMethod];

// inside MyClass other methods can call instance methods of MyClass using self
-(void) someMethod
{
    [self aMethod];
}

// class methods must be called on the class itself
[MyClass anotherMethod];

// this won't work:
[instance anotherMethod]; // ERROR
[self anotherMethod];     // ERROR
```

When to use class methods?

Class methods are generally used when you don't have or need an instance of the class. A commonly well-known class method is `+(id) alloc` for example. Or the many initializers of cocos2d and Cocoa classes, like `[NSString stringWithFormat:]` or `[CCSprite spriteWithFile:]`. And the Singleton pattern uses a class method to get access to the one instance of the class, for example `[NSFileManager defaultManager]` or `[CCDirector sharedDirector]`.

Instance methods are used whenever you want the individual instance to change its state. They are the norm. If in doubt, use an instance method. Only instance methods get access to a class' instance variables, and only instance methods are "inherited" by subclasses. If you override (write the same method again in the subclass) an instance method then you can call the base class implementation by sending the message to super:

```
// in subclass of MyClass
-(void) aMethod
{
    [super aMethod];
}
```

When should I add an instance variable to my class?

You want an instance variable for these reasons:

1) If the variable is part of the state of an object (ie on or off, alive or dead, number of remaining hitpoints, which weapon is being carried, position, rotation, and so on).

2) If other classes need access to the variable, and it is not trivial to compute the value every time it is being requested (ie the number of game objects in a specific state). You can then return the cached value and update the cached value only if a game object changes its state.

3) For performance reasons or readability. For example if you use a Singleton a lot in a class, it's both faster and leads to shorter code if you store a reference to the singleton as an instance variable.

4) If the class uses the variable a lot. For example while cocos2d provides a `getChildByTag` method it's unnecessarily complex to use that method every time you need access to a specific object that you need often. It may also be slow because the getChildByTag method potentially has to do a lot of work to find that object.

When not to use an instance variable

There are also good reasons to avoid using an instance variable. For example:

1) If it's a temporary object that may become invalid after you received it. For example physics engine collision callback information is typically discarded after the collision. If you store a collision data object (ie contact info) then that pointer to the object may be invalid in the next frame, and accessing it causes the app to crash. Usually this behavior is well documented and expressly mentioned that you should not retain references to the objects in question.

2) If you use the variable only in one method. No point in storing a value in the class if no one else is using it.

3) Exchanging objects is dangerous. Typically it's a good idea for one class (class A) to hold a reference to another class (class B) in order to access B's methods and properties easily. But it is bad practice to do the opposite at the same time, meaning that B holds a reference to A as well. This can lead to retain cycles where neither object can let go of the reference to the other object it holds, thus leaking memory and keeping both objects alive. This description only scratches the surface and is oversimplified. Learn how to avoid retain cycles.

Objective-C Orientation Course

This section is for those who would want to pick up on Objective-C quickly while avoiding some of the common pitfalls that befall those who apply their previous programming language's habits or best practices.

- Getting Started with Objective-C — Here are learning resources like books and high-quality articles for those who need to start learning Objective-C (almost) from scratch.
- Migrating to Objective-C from another language — This article aims to make it easier to migrate to Objective-C for those who already know one or more other languages. Find specific "from x to Objective-C" guides as well as cheat sheets and some soothing information why Objective-C is not as bad as you may believe it is.
- The not so obvious differences of Objective-C — This page lists only the most commonly made mistakes by programmers of other programming languages. It covers easily overlooked implementation, syntax or behavioral differences.

If you have a particular migratory issue (not: a migraine) that's not covered here, then please leave a suggestion.

Getting Started with Objective-C

Here are learning resources like books and high-quality articles for those who need to start learning Objective-C (almost) from scratch.

- Some of the best Books to learn Objective-C from
- The best free online tutorials about Objective-C
- Objective-C Video Tutorials

Some of the best Books to learn Objective-C from

These books are relatively up to date and cover Objective-C 2.0. These books explain the Objective-C language itself, rather than iOS or Mac OS programming with Objective-C. Although the one doesn't really make sense without the other, the focus of these books is on Objective-C and not Cocoa / Cocoa Touch. Most of these books also cover the basics of working with Xcode.

- Objective-C for Absolute Beginners (Apress)
- Programming in Objective-C 2.0 (Addison-Wesley / Pearson)
- Learn Objective-C on the Mac (Apress)
- Objective-C Fundamentals (Manning)
- Objective-C for Dummies (Wiley & Sons)
- Objective-C Pocket Reference (O'Reilly)

As time passes by successful books are often updated to a 2nd or newer edition. You should check that you get the most recent version of these books, refer to the publication date if in doubt.

The best free online tutorials about Objective-C

These online tutorials are mainly for the beginner. They are chosen from the hundreds of available tutorials as those that stand out in terms of quality, focus, applicability and up-to-date-ness (is that a word?).

- Introduction to Objective-C (Apple)
  - The official introduction by Apple. Recommended reading but may be overwhelming at first for the absolute beginner.
- Learn Objective-C (Cocoa Dev Central)
  - Short and concise. You get a quick overview of the Objective-C language concepts in just a few minutes.

    - Related: Objective-C with Style Part 1 and Part 2 introduce you to Objective-C coding guidelines and best practices.
- Learn Objective-C: Day 1-6 (MobileTuts+)
    - A 6-part series that starts with fundamentals of (object-oriented) programming and doesn't skip over best practices and coding guidelines.
- Objective-C Beginner's Guide
    - This guide is from 2004 and doesn't cover the Objective-C 2.0 extensions (for example: properties). But it does stand out as one of the few tutorials with lots of example code.

Objective-C Video Tutorials

- 65-part Objective-C Programming Tutorials (TheNewBoston Education)
    - Everything from the ground up. But unfortunately only covers Xcode 3.
- The video below is Lesson 1 of a 33-part Objective-C Tutorial series. While it's discussing only Xcode 3 it does contain a wealth of information that still applies.

Migrating to Objective-C from another language

This article aims to make it easier to migrate to Objective-C for those who already know one or more other languages. Find specific "from x to Objective-C" guides as well as cheat sheets and some soothing information why Objective-C is not as bad as you may believe it is.

- Side-by-Side comparison
- Objective-C Cheat Sheets
- How You Learned to Stop Complaining and Love the Objective-C Bomb
- Migrating from Planet C/C++
- Migrating from Planet C#
- Migrating from Planet Java
- Official Objective-C Documentation

Side-by-Side comparison

The C++ Style Languages: C, C++, Objective C, Java, C# website compares these languages together side-by-side in a table format. For each task (ie write to file, define class, create object, ...) the code is given for each language.

This makes it a great guide for the cases where you knew how to write the code in C, C++, C# or Java but can't figure out how it ought to be written in Objective-C. Just look up the code for the language you know, then look up the Objective-C example code in the same row.

Related to this article is the Differences and Similarities Between Objective-C, Java, and C++ section from the GNUStep Objective-C documentation. It tells you short and concise about the major differences of Objective-C in regards to Java and C++.

Objective-C Cheat Sheets

- Objective-C for Dummies Cheat Sheet (from: Neal Goldstein)
- Objective-C Cheat Sheet and Quick Reference (from: Ray Wenderlich)
- iPhone Objective-C Cheat Sheet (from: Johann Dowa)
- Objective-C Cheat Sheet (from: Ferruccio Barletta)
- Objective-C 2.0 @ Compiler Directives Cheat Sheet (from: Johann Dowa)
- iOS SDK Quick Reference (must register to download)

How You Learned to Stop Complaining and Love the Objective-C Bomb

- StackOverflow: Why do Programmers Love/Hate Objective-C?
- informIT: Top Ten Reasons Cocoa is Great Because of Objective-C (and Not In Spite of It)

Migrating from Planet C/C++

- From C++ to Objective-C (PDF) by Pierre Chatelier © 2005-2009
- informIT: Objective-C for C++ Programmers, Part 1
- informIT: Objective-C for C++ Programmers, Part 2
- informIT: Objective-C for C++ Programmers, Part 3

Migrating from Planet C#

- Presentation: Objective-C for C# Developers
- iPhone Objective-C for the C# Developer
- StackOverflow: How does Objective-C compare to C#?

Migrating from Planet Java

- informIT: Objective-C for Java Programmers, Part 1
- informIT: Objective-C for Java Programmers, Part 2

- Effective Objective-C (for Java developers)
- Book: Learn Objective-C for Java Developers (Apress)

Official Objective-C Documentation

- Introduction to Objective-C
- Object-Oriented Programming with Objective-C
- Introduction to Coding Guidelines for Cocoa
- For experts:
    - Objective-C Runtime Programming Guide
    - Objective-C Runtime Reference

The not so obvious differences of Objective-C

This page lists only the most commonly made mistakes by programmers of other programming languages. It covers easily overlooked implementation, syntax or behavioral differences.

You don't call methods, you send messages

Under the hood Objective-C isn't calling methods of a class instance, it is actually sending a message to that instance instructing it to run a method with the given name and signature.

In other words: you call a method in C/C++/Java/C#/etc but in Objective-C you're sending messages. For the most part, both will be the same for you and you can regard it as a simple difference in terminology.

You can send messages to nil objects

The following code is legal in Objective-C:

```
myObject = nil;
[myObject doSomething];
```

Since `myObject` is nil the message `doSomething` will not be sent, meaning the statement is skipped silently. Many a newcoming Objective-C programmer has tripped over this.

It also makes it needless to write statements like these:

```
MyClass* myObject = [self getSomeObjectOfMyClass];
if (myObject)
{
    [myObject doSomething];
}
```

It is customary in Objective-C to simply write:

```
MyClass* myObject = [self getSomeObjectOfMyClass];
[myObject doSomething];
```

To `nil` or to `NULL`?

In Objective-C you use `nil` instead of `NULL`, unless you're dealing with C/C++ objects. This is just a guideline however since `nil` and `NULL` are the exact same thing.

`YES` or `NO`: `true` or `false`?

Similar to the above question, the Objective-C boolean variable macros are `YES` and `NO`. If you're using true and false, you may get some raised eyebrows by Objective-C programmers but other than that nothing much else will happen.

`BOOL` and `bool` are not the same thing!

Careful: `BOOL` and `bool` are not the same type!

`BOOL` is a `signed char`
`bool` is an `int`

The standard type for Objective-C code is `BOOL`.

#import, don't #include!

If your background is in C/C++ you'll be tempted to write:

```
#include "MyClass.h"
```

You should only do so if the MyClass header file declares C methods or a C++ class. If it's an Objective-C class you should use #import instead:

```
#import "MyClass.h"
```

The benefit of #import is that it avoids cyclic redundancies.

No safeguarding Objective-C header files

In C/C++ it is customary to write something like this in your header file to prevent including the same header multiple times:

```
#ifndef __MyClass_h__
#define __MyClass_h__

// header declarations here ...

#endif
```

You should not do this for Objective-C header files, only for C/C++ header files. See #import vs. #include above.

The id is a pointer type (no asterisk!)

The `id` type already is a pointer type. To be precise is a typedef of a pointer type, hiding implementation details of the Objective-C runtime. A common mistake is to view `id` as similar to `void` and then trying to use it like this:

```
// WRONG!
id *myObject = [self getSomeObject];
```

```
// CORRECT!
id myObject = [self getSomeObject];
```

String-Format objects with %@

You can print out objects with the %@ format string:

```
NSLog(@"object: %@", myObject);
```

This will call `[myObject description]` behind the scenes, or print "nil" if myObject is nil. If you need custom logging of your own Objective-C classes, you should override the description method:

```
-(NSString*) description
{
    return [NSString stringWithFormat:@"%@ - position: (%f, %f)",
                                      self, position.x, position.y];
}
```

Don't assert, do NSAssert!

The C style `assert` function is still available but more common and more powerful are the NSAssert (NSAssert1, NSAssert2, ...) macros:

```
// assertion with format string using 1 format string parameter
NSAssert1(myObject == nil, @"%@ <-- should be nil!", myObject);
```

Performance Tips

This section helps you write faster Objective-C code. The performance gain may not be much, but these tips are easy to apply and sometimes even make the code easier to read (and write).

Caching a Singleton Instance

A Singleton is a class of which there will be only one instance for the entire lifetime of your application. Usually you access a Singleton via a method that begins with `default` or `shared`. This is called the "Singleton accessor" method (or function).

Sending the Singleton Accessor message unnecessarily

Since the Singleton pattern is so convenient to use, developers often forget that it's still a message that is sent to the Singleton class every time. For example this code is a bit wasteful:

```
// Code intended to be instructional, not functional
-(void) updateEveryFrame
{
    [[CCDirector sharedDirector] doSomething];
    [[CCDirector sharedDirector] setSomething:123];
    int x = [[CCDirector sharedDirector] getSomeValue];
    [[CCDirector sharedDirector] doSomethingElse];
    [CCDirector sharedDirector].someProperty = 10;

    for (int i = 0; i < 100; i++)
    {
        CGSize winSize = [[CCDirector sharedDirector] winSize];
        // do stuff here
    }
}
```

In this case you're sending the `sharedDirector` message 105 times to the CCDirector to get the Singleton instance of the director, where you only needed to send one.

See a real world example where the Singleton accessor message is sent multiple times in a method.

Caching the Singleton Instance in a local variable

Since the CCDirector instance never changes, this can be optimized to only get the CCDirector singleton instance once:

```
-(void) updateEveryFrame
{
    CCDirector* director = [CCDirector sharedDirector];
    [director doSomething];
    [director setSomething:123];
    int x = [director getSomeValue];
    [director doSomethingElse];
    director.someProperty = 10;

    for (int i = 0; i < 100; i++)
    {
        CGSize winSize = [director winSize];
        // do stuff here
    }
}
```

The performance gain will be far from dramatic but since this also makes the code easier to read (and shorter to write), you should use this optimization strategy in general.

The line `CGSize winSize = [director winSize];` can be optimized further by moving it before the loop. More on this in Loop Optimizations.

Caching the Singleton Instance in an instance variable of a class

If you call `updateEveryFrame` a lot (supposedly every frame) or if you have many such methods that need the same Singleton class, you can also store an instance of the Singleton class in your class instance. The tradeoff is a tiny increase of the instance size of the class (4-Bytes on iOS and 32-Bit Mac, 8-Bytes on 64-Bit Mac). The upside is that the Singleton class is readily available all the time in that class.

You will need to add the Singleton as an instance variable in your class' `@interface`:

```
@interface MyClass : NSObject
{
    CCDirector* director;
    // ...
}
// ...
@end
```

Then you assign the singleton in your class' init method:

```
@implementation MyClass

-(id) init
{
    if ((self = [super init]))
    {
        director = [CCDirector sharedDirector];
    }
}

-(void) updateEveryFrame
{
    // director is already available:
    [director doSomething];
    [director setSomething:123];

    // etc etc
}

// ...
@end
```

You do not and should not retain nor release the director instance variable. Since the Singleton's lifetime begins when the App starts and ends

only when the app is quit, you can safely assign the Singleton instance to the instance variable without retaining it.

Caching the results of a computation

...

Reducing Sent Messages in a Loop

A loop is always a critical block of code simply because loops are iterated over from 0 to n times. That means every unnecessary instruction or message sent to another object within a loop is multiplied by n.

While the compiler is often able to do some optimizations there for you, for most developers it's next to impossible to tell what the compiler is going to optimize and what not. It's safer to do these optimizations manually, and these changes often keep the code inside the loop cleaner, shorter, more readable.

Moving immutable results outside the loop

Do you see how this code could be optimized?

```
-(CCNode*) getAnyEnemyInScreen
{
    do
    {
        int i = [self getNextEnemyIndex];
        if (i < 0)
            break;

        CCNode* node = [[self children] objectAtIndex:i];
        CGSize winSize = [[CCDirector sharedDirector] winSize];
        CGRect screen = CGRectMake(0, 0, winSize.x, winSize.y);
        if (CGRectIntersectsRect([node boundingBox], screen))
        {
            return node;
        }
    } while (YES);

    return nil;
}
```

It may not be immediately obvious to the untrained eye but there's some redundancy inside the loop. Think of values that never change, at least not while this loop is performing its job.

The winSize (window size) won't change while you're going over the loop. So the `screen` CGRect does not need to be recreated for every new enemy node. You can and should optimize this loop by moving the creation of the CGRect outside the loop. As a positive side effect, the code inside the loop looks just a bit less threatening.

An optimized version of the example code

While you're having a look at the following optimized code example, see if you can spot the remaining thing that we can optimize by moving outside the loop. Not that obvious, is it?

```
-(CCNode*) getAnyEnemyInScreen
{
    CGSize winSize = [[CCDirector sharedDirector] winSize];
    CGRect screen = CGRectMake(0, 0, winSize.x, winSize.y);

    do
    {
        int i = [self getNextEnemyIndex];
        if (i < 0)
            break;

        CCNode* node = [[allEnemiesNode children] objectAtIndex:i];
        if (CGRectIntersectsRect([node boundingBox], screen))
        {
            return node;
        }
    } while (YES);

    return nil;
}
```

So what else isn't changing within the do/while loop?

You always get a new index i that's for sure. And since each node returned will be a different node, we can't move the `[node boundingBox]` message outside the loop — if that's what you were thinking.

The fully optimized example code

What remains? Of course: the `[allEnemiesNode children]` array!

Every iteration you get the children array from the allEnemiesNode, and the array's contents won't change while you're going over this loop. You can move that outside the loop as well:

```
-(CCNode*) getAnyEnemyInScreen
{
    CGSize winSize = [[CCDirector sharedDirector] winSize];
    CGRect screen = CGRectMake(0, 0, winSize.x, winSize.y);
    CCArray* enemies = [allEnemiesNode children];

    do
    {
        int i = [self getNextEnemyIndex];
        if (i < 0)
            break;

        CCNode* node = [enemies objectAtIndex:i];
        if (CGRectIntersectsRect([node boundingBox], screen))
        {
            return node;
        }
    } while (YES);

    return nil;
}
```

Tiny change, a wee bit more readable code, and one less message sent during each iteration. Not a performance optimization to gloat over and it'll be hardly noticeable in most cases, but it does have the added benefit of making the code more readable without any disadvantages. That's the kind of optimization you should welcome into your code.

Optimizing the not-so-obvious extraneous code in loops

Here's a slightly different version of the above loop. It's already fully optimized ... or is it?

```
-(CCNode*) isAnyEnemyInScreen
{
    CGSize winSize = [[CCDirector sharedDirector] winSize];
    CGRect screen = CGRectMake(0, 0, winSize.x, winSize.y);
    CCArray* enemies = [allEnemiesNode children];

    for (int i = 0; i < [enemies count]; i++)
    {
        CCNode* node = [enemies objectAtIndex:i];
        if (CGRectIntersectsRect([node boundingBox], screen))
        {
            return YES;
        }
    }

    return NO;
}
```

There's one thing you need to know about the `for` loop. It's syntax goes something like this:

```
for (init; looptest; counting) {...}
```

The `init` part is executed once before the loop begins, whereas the `looptest` and the `counting` parts are run during each iteration. That means the code that tests whether the loop should continue or end is executed every iteration. You can actually rewrite a for loop with a do/while loop:

```
int i = 0;
while (i < [enemies count])
{
    i++;
}
```

The optimization possibility will be even more obvious if we gave Objective-C the "repeatforever" pseudo-statement:

```
int i = 0;
repeatforever  // <-- pseudo-statement, not an Objective-C keyword!
{
    if (i < [enemies count])
        break;

    i++;
}
```

As you can see, the test `i < [enemies count]` is actually executed every frame, and that includes the `count` message being sent to the enemies object.

With this knowledge we can rewrite the example code with a slight optimization:

```
-(CCNode*) isAnyEnemyInScreen
{
    CGSize winSize = [[CCDirector sharedDirector] winSize];
    CGRect screen = CGRectMake(0, 0, winSize.x, winSize.y);
    CCArray* enemies = [allEnemiesNode children];
    int numberOfEnemies = (int)[enemies count];

    for (int i = 0; i < numberOfEnemies; i++)
    {
        CCNode* node = [enemies objectAtIndex:i];
        if (CGRectIntersectsRect([node boundingBox], screen))
        {
            return YES;
        }
    }

    return NO;
}
```

Many compilers are able to optimize the loop test code in the way we just did. But the compiler may not cover all cases and it obviously depends on which compiler is being used and how aggressively it optimizes the code.

It's recommended to perform this optimization in time-critical loops manually to be sure. Again, no disadvantages to doing so other than adding an extra line of code.

Notice that I cast the result from `[enemies count]` to `int`. This is because count is actually an `unsigned int` or more precisely a `NSUInteger`. Not performing the cast to `(int)` might generate a "sign mismatch" warning.