Objective-C for Experienced Programmers

Venkat Subramaniam
venkats@agiledeveloper.com
twitter: venkat_s

Objective-C

- An Object-Oriented extension to C
- If you're familiar with C/C++/Java syntax, you're at home
  - Though you are closer to home if you know C++ :)
- If you're used to VB.NET/Ruby/... then you need to get used to the curly braces and the pointers
- The biggest challenge is to learn and to remember to manage memory
  - Following certain practices will ease that pain

A Traditional HelloWorld!

```
#import <Foundation/Foundation.h> // or stdio.h

int main (int argc, const char * argv[]) {
  printf("Hello World!\n");
  return 0;
}
```

```
Hello World!
```


printf Variable Types

- %i Integer
- %c Character
- %d Signed decimal Integer
- %e Scientific notation using e character
- %E .. using E character
- %f Floating-point decimal
- %g The shorter of %e or %f
- %G The shorter of %E or %f
- %s String
- %u Unsigned decimal integer

%@ to print an object

Data Types

char	A char	1 byte (8 bits)
double float	Double precision	8 bytes
float	Floating point	4 bytes
int	Integer	4 bytes
long	Double short	4 bytes
long long	Double long	8 bytes
short	Short integer	2 bytes

The id type

- id is a type that can refer to any type of object

```
id vehicle = carInstance;
```

- This provides dynamic typing capability in Objective-C
- You can specify the type if you like or you can leave it to the runtime to figure it out (you use id in the latter case)

nil is an object

- nil is a special object which simply absorbs calls
- It will return nil or 0 as appropriate, instead of failing

```
Goe* goe = nil;
printf("Lat is %g\n", [goe lat]); // will print Lat is 0
```


Behavior of nil

- Objective-C is very forgiving when you invoke methods on nil
- This is a blessing and a curse
- Good news is your App won't blow up if you invoke methods on nil
  - This can also be quite convenient if you don't care to check for nil, call if object exists, otherwise no-big-deal kind of situation
- Bad news is, if you did not expect this, your App will quietly misbehave instead of blowing up on your face

NS Objects

- In Objective-C several classes will start with letters NS
- These can be included by including Foundation/Foundation.h
- NS stands for NeXtStep, creator of Objective-C
  - NextStep was the company that made the Next computers (back in those nostalgic days)

NSString

- Regular 'C' style strings are UTF-8
- NSString is Objective-C string
- Supports unicode and several useful operations
- Use @"" to create an instance of NSString from a literal
- You can also use the class method **stringWithFormat** to form a string with embedded values

Creating a NSString

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
  NSString *helloWorld = @"Hello World!";

  printf("%s\n", [helloWorld UTF8String]);

  NSLog(@"%@", helloWorld);

  [helloWorld release];
  return 0;
}
```

```
__NSAutoreleaseNoPool(): Object 0x104110 of
class NSCFData autoreleased with no pool
in place - just leaking

Hello World!

... ObjectiveCSample[29597:a0f] Hello World!
```

Above code has a memory leak in spite of calling release! We'll learn how to fix it real soon.

NSLog is a useful tool to log messages. A tool you'll come to rely upon during development.

Calling a Method

- [receiver method] format (place your call within [])
- Use [instance method: paramList] format to call methods which take parameters
- For example see how we called UTF8String on the helloWorld instance

Creating a Class

Car.h

```
#import <Foundation/Foundation.h>

@interface Car : NSObject
{
}

@property (nonatomic) NSInteger miles;
-(void) drive: (int) distance;
+(int) recommendedTirePressure;

@end
```


Creating a Class

Car.m

```
#import "Car.h"

@implementation Car

@synthesize miles;

-(void) drive: (int) distance {
  miles += distance;
}

+(int) recommendedTirePressure {
  return 32;
}

@end
```

Creating an Instance

main.m

```
#import <Foundation/Foundation.h>
#import "Car.h"

int main(int argc, const char* argv[]) {
    Car *car = [[Car alloc] init];
    NSLog(@"Car driven %d miles\n", [car miles]);

    [car drive: 10];
    NSLog(@"Car driven %d miles\n", [car miles]);

    NSLog(@"Recommended tire pressure %i psi.\n",
          [Car recommendedTirePressure]);

    [car release];
    return 0;
}
```

```
Car driven 0 miles
Car driven 10 miles
Recommended tire pressure 32 psi.
```


Class and Instance Methods

- You define instance methods using a -
- You define class methods using a +

Class field

```
@implementation Car

//...

static int tirePressure = 32;

+(int) recommendedTirePressure {
  return tirePressure;
}

@end
```


Multiple Parameters

```
-(void) turn: (int) degreeOfRotation speed: (int) speed {
  printf("turning %i degrees at speed %i MPH\n",
    degreeOfRotation, speed);
}
```

Parameters are separated by :

```
  [car turn: 20 speed: 50];
```

```
turning 20 degrees at speed 50 MPH
```

Mystery of Method Names

- Objective-C method names can contain colons and can have multiple parts.
- So, when you write setLat: (int) lat lng: (int) lng
  the actual method name is setLat:lng: and it takes two parameters
- You call it as [instance setLat: 38.53 lng: 77.02];

Properties

- Properties are attributes that represent a characteristic of an abstraction
- They provide encapsulation
- You have getter and setter methods to access them
- Objective-C relieves you from the hard work of writing these mundane methods (and their fields)
- You can use a @property to declare properties
- To synthesize the getter and setter, use @synthesize
  - While @synthesize creates these methods at compile time, mark it @dynamic to postpone creation to runtime

Property Accessors

- The getter for a property has the form propertyName
- The setter for a property has the form setPropertyName
  - setters are not created if you mark your property as readonly
- You can create custom getters and setters by setting the getter and setter attribute

Attaching Attribute Flavors

- You can attach a certain attributes or flavors to a property
- For example, @property (nonatomic, retain) NSString* firstName;

Property Attributes

- Atomicity
  - nonatomic (default is atomic—but there's no keyword for that—will incur locking related performance overhead)
- Setter
  - assign, retain, copy (default is assign, retain will increase retain count on set, release on reassignment)
- Writability
  - readwrite or readonly (default is readwrite)

Properties and iVar

- In the legacy runtime, you need to declare a field with the same name as the property or map it using = in the @synthesize
- In the "modern" runtime (64-bit and latest iPhone), you don't need a field (iVar) to backup the property. They are generated for you internally

Properties and Attributes

```
@interface Person : NSObject {}

@property (nonatomic, retain) NSString* firstName;
@property (nonatomic, retain) NSString* lastName;

@end

@implementation Person

@synthesize firstName; creates firstName and setFirstName: methods
@synthesize lastName;  creates lastName and setLastName: methods

-(void) dealloc {
  self.firstName = nil;
  self.lastName = nil;
  [super dealloc];
}

@end
```

Setting this to nil releases the held instance

Accessing Properties

```
#import <Foundation/Foundation.h>
#import "Person.h"

int main (int argc, const char * argv[]) {
  NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
  Person* dbl07 = [[Person alloc] init];

  [dbl07 setFirstName: @"James"];
  dbl07.lastName = @"Bond";

  NSString* fName = [dbl07 firstName];
  NSString* lName = dbl07.lastName;

  printf("%s ", [fName UTF8String]);
  printf("%s\n", [lName UTF8String]);

  [dbl07 release];

  [pool drain];
  return 0;
}
```

You can use either the dot (.) notation or the method call notation []

Creating an Instance

- Two step process: First allocate memory (using alloc), then initialize it, using one of the init methods
  - If it takes no parameters, method is often called init
  - If it takes parameters, it gets to be descriptive, like initWithObjects:
- If you follow the above steps, you're responsible to release the object
  - You can either release it or put that into an auto release pool right after you create

Make it simple and easy

- Help users of your class
- Write your class so we're not forced to use alloc and init
- Please provide convenience constructors

Convenience Constructors

- Classes may short-circuit the 2-step construction process and provide a class level convenience method to initialize the instances
- These methods generally start with name className... (like stringWithFormat: or arrayWithObjects:)
- If you use a convenience constructor, don't release the instance!
  - These methods add the instance to the autorelease pool for you

Creating Instances

```
#import <Foundation/Foundation.h>
#import "Person.h"

int main (int argc, const char * argv[]) {
  NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];

  NSString* str1 = [[NSString alloc]
                initWithString: @"you release"];
  NSString* str2 = [[[NSString alloc]
                initWithString: @"auto"] autorelease];

  NSString* str3 = [NSString stringWithString: @"No worries"];

  printf("%s", [[NSString
      stringWithFormat: @"%@ %@ %@", str1, str2, str3] UTF8String]);

  [str1 release];
  [pool drain];
  return 0;
}
```

We'll learn about memory management and release pool soon.

```
you release auto No worries
```

The Magic of init

- The init method returns self after it does its initialization
- One benefit is convenience, but the other benefit is morphing
- You can cascade calls on to the call to init (like [[[Something alloc] init] doWork];)
- init may actually decide to create an instance of another specialized type (or another instance) and return that instead
  - This allows init to behave like a factory
  - Don't assume init only initializes, you may get something different from what you asked for

Don't do this

```
Something* something = [Something alloc];
[something init];
[something doWork];
```

- You are ignoring the instance returned from init
- If init decided to create or return something other than what you had asked for
  - at the best, you're working with a poorly constructed instance
  - at the worst, you're working with a object that may've been released

Do this

```
Something* something = [[Something alloc] init];
[something doWork];
[something release];
```

or

```
Something* something = [[[Something alloc] init] autorelease];
[something doWork];
```

You may check to ensure init did not return a nil

Designated Initializer

- Each class has a designated initializer
- This is the most versatile initializer
- All other initializers call this designated initializer
- The designated initializer is the one that calls the super's designated initializer
- Each class should advertise its designated initializer (solely for the benefit of the person writing a subclass)

Your Own Initializers

- Begin your initializers with the letters **init**
- Return type of init should be **id**
- Invoke your own designated initializer from your initializers
- Invoke base class's initializer from your designated initializer
- Set self to what the base initializer returns
- Initialize variables directly instead of using accessor methods
- If something failed, return a nil
- At point of failure (if you're setting nil, that is) release self

init(s) with inheritance

- If your designated init method has different signature than the designated method of the base class, you must override the base's designated method in your class and route the call to your designated init method

Writing Constructors

- Typically every instance has at least one constructor method.
- These methods start with the name init, but may be of any name following init and may take parameters

Writing Constructors

```
#import <Foundation/Foundation.h>


@interface Person : NSObject {}

@property (nonatomic, retain) NSString* firstName;
@property (nonatomic, retain) NSString* lastName;
@property NSInteger age;

-(id) initWithFirstName: (NSString*) fName
     lastName: (NSString*) lName andAge: (NSInteger) theAge;

-(id) initWithFirstName: (NSString*) fName
     lastName: (NSString*) lName;

@end
```

Writing Constructors

```
-(id) initWithFirstName: (NSString*) fName
     lastName: (NSString*) lName andAge: (NSInteger) theAge {
  if (self = [super init]) {
    self.firstName = fName;
    self.lastName = lName;
    self.age = theAge;
  }

  return self;
}

-(id) initWithFirstName: (NSString*) fName
   lastName: (NSString*) lName {
  return [self initWithFirstName: fName lastName: lName andAge: 1];
}
```


Using Constructors

```
NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];

Person* james =
[[[Person alloc] initWithFirstName: @"James"
      lastName:@"Bond" andAge: 16] autorelease];

Person* bob =
[[[Person alloc] initWithFirstName: @"Bob"
      lastName: @"Smith"] autorelease];

[pool drain];
```

Type checking

- isMemberOfClass function can help you with this. true only if instance is of specific type
- isKindOfClass will tell you if instance is of type or of a derived type

```
if ([james isMemberOfClass: [Person class]] == YES) {
  printf("Yes, James is of type Person\n");
}

if ([james isKindOfClass: [NSObject class]] == YES) {
  printf("Yes, james is of type NSObject or a derived type\n");
}
```

```
Yes, James is of type Person
Yes, james is of type NSObject or a derived type
```


Selectors

- Objective-C allows you to get a "pointer" or "handle" to a method
- This is useful to register event handlers dynamically with UIView or controls
- This is also useful to delegate method execution
  - An ability to pass functions around to other functions

SEL

- A SEL is a special type that holds a pointer to the symbolic name of a method (after the compiler has converted the method name into an entry in the symbol table)
- You can ask the compiler to give you a handle to that entry using the @selector directive
- SEL mymethod = @selector(someMethod:)
- If you don't know the method name at compile time (to make things real dynamic), you can get a SEL using NSSelectorFromString method
  - NSStringFromSelector does the reverse for you

Invoking Methods using SEL

- You can indirectly invoke a method using the selectors
- [instance performSelector: @selector(methodName:) withObject: anotherInstance]; is same as [instance.methodName: anotherInstance];

Using Selector

Let's first define some methods

```
-(void) drive: (NSNumber*) speed {
  printf("%s", [[NSString
    stringWithFormat: @"driving at speed %@\n", speed] UTF8String]);
}

-(void) swim {
  printf("swimming\n");
}

-(void) run: (NSNumber*) distance {
  printf("%s", [[NSString
    stringWithFormat:
      @"running distance %@\n", distance] UTF8String]);
}
```


Using Selector

```
NSNumber* speed = [NSNumber numberWithInt: 100];
[james drive: speed]; // direct method call

[james performSelector: @selector(drive:) withObject: speed];
[james performSelector: @selector(swim)];

[james performSelector: @selector(run:)
   withObject: [NSNumber numberWithInt: 5]];

SEL aMethod = NSSelectorFromString(@"swim");
[james performSelector: aMethod];
```

```
driving at speed 100
driving at speed 100
swimming
running distance 5
swimming
```

Responds to a message?

- You can check if an instance responds to a message

```
if([james respondsToSelector: @selector(swim)]) {
  printf("Can swim!\n");
}
```


Invoking a Method Later

- You can ask Objective-C to invoke a method, just a little, later—using the afterDelay option
- This is quite convenience for you to handle touches/ tapping on the iPhone
  - You don't know if this is a single tap or first of a two tap
  - You can ask the effect to take place in moments
  - Quickly cancel that if you see the second tap—using cancelPreviousPerformRequestWithTarget

Restricting Access

- You can restrict access to members using @private, @protected, or @public (which is the default)
- All members placed under a declaration have the same restriction until you change with another declaration

Inheritance

- Use : to separate class from its super class
- Call base method using [super ...]

Categories

- Categories allow you to extend a class (even if you don't have the source code to that class)
- In one sense they're like partial classes in C#
- However, they're more like open classes in Ruby
- You write them as
  @interface ClassName (CategoryName)

Categories

```
#include <Foundation/Foundation.h>
//StringUtil.h
@interface NSString (VenkatsStringUtil)

-(NSString*) shout;
@end
```

```
//StringUtil.m

#import "StringUtil.h"

@implementation NSString(VenkatsStringUtil)

-(NSString*) shout {
  return [self uppercaseString];
}

@end
```

in main.m

```
  NSString* caution = @"Stop";

  printf("%s\n", [[caution shout] UTF8String]);
```

STOP

Protocols

- Protocols are like interfaces
- You can make a class conform to the methods of a protocol
  - It can either "adopt" a protocol or inherits from a class that adopts a protocol
- Protocols can have required and optional methods!
- Adopting a protocol: @interface ClassName : SuperClass <Protocol1, Protocol2, ...>

Protocols

```
//Drivable.h
@protocol Drivable

-(void) drive: (int) distance;

@optional
-(void) reverse;

@required
-(int) miles;

@end
```

@required is the default

Protocols

```
#import <Foundation/Foundation.h>
#import "Drivable.h"

@interface Car : NSObject<Drivable> {}

@end

@implementation Car

-(void) drive: (int) distance {
  printf("Driving %d miles\n", distance);
}

-(void) reverse {
  printf("Reversing\n");
}

-(int) miles {
  return 0;
}
```


Protocols

```
Car* car = [[[Car alloc] init] autorelease];

[car drive: 10];

id<Drivable> drivable = car;
[drivable reverse];
```

You can get a reference to a protocol using the id<...>

```
Driving 10 miles
Reversing
```

Protocols and Categories

- You can have a category of methods adopt a protocol, like so

- @interface ClassName (CategoryName) <protocol1, protocol2, ...>

Checking for Conformance

- You can check if an instance conforms to a protocol by calling conformsToProtocol: method

```
if([car conformsToProtocol: @protocol(Drivable)]) {
  printf("Car is drivable\n");
}
```

References to Protocol

- You can store a explicit reference of type protocol like id<ProtocolName> ref
  - Useful for type checking, ref can only refer to an instance that conforms to ProtocolName
- You can also write SomeClass<SomeProtocol> ref
  - In this case ref can only refer to an instance of SomeClass or its derived class that conforms to SomeProtocol

Collections

- You often have need to work with collections of objects
- There are three common collections you would use
- Arrays, Dictionaries, Sets
- These come in mutable and immutable flavors
- If you want to add (or remove) to a collection after you create it, use mutable flavors

Using Arrays

Ordered sequence of objects

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

int ageOfFriends[2] = {40, 43};
printf("Age of First friend %i\n", ageOfFriends[0]);

NSArray* friends = [[[NSArray alloc]
    initWithObjects: @"Joe", @"Jim", nil] autorelease];
int count = [friends count];
printf("Number of friends %d\n", count);

NSArray* friends2 = [NSArray arrayWithObjects:
  @"Kate", @"Kim", nil];
printf("A friend %s\n", [[friends2 objectAtIndex: 0] UTF8String]);

[pool drain];
```

You're adding to the pool

Added to the pool for you

NSArray is immutable, once you create it,
you can no longer add or remove elements to it

Iterating Arrays

```
NSEnumerator* friendsEnumerator = [friends objectEnumerator];
id aFriend;
while ((aFriend = [friendsEnumerator nextObject])) {
  printf("%s\n", [aFriend UTF8String]);
}

int friendsCount = [friends count];
for(int i = 0; i < friendsCount; i++) {
  printf("%s\n", [[friends objectAtIndex: i] UTF8String]);
}
```

```
for(NSString* aFriend in friends) {
  printf("%s\n", [aFriend UTF8String]);
}
```

Fast enumeration!

Using Dictionary

Associative key-value pairs

```
NSDictionary* friends = [[NSDictionary
 dictionaryWithObjectsAndKeys: @"44", @"Joe", @"43", @"Jim", nil]
 autorelease];
```

Key can't be null, you specify value and then key

```
printf("Joe is %s years old\n",
       [[friends objectForKey: @"Joe"] UTF8String ]);

//Iterating
for(NSString* aFriend in friends) {
  printf("%s is %s years old\n",
      [aFriend UTF8String],
      [[friends objectForKey: aFriend] UTF8String]);
}
```

```
Joe is 44 years old
Joe is 44 years old
Jim is 43 years old
```


Mutable vs. Immutable

- NSArray is immutable, you can't add elements to it or change it once it is created
- For mutable arrays, use NSMutableArray
- Similarly for mutable dictionary, you may use NSMutableDictionary

Exception Handling

- @try, @catch, @finally directives to handle exceptions
- @throw to raise exceptions
- Very similar in construct to Java/C# exception handling
- Exception base class is NSException (but you could throw any type of exception - just like in C++)
- To re-throw an exception simply use @throw with no argument

Exception Handling

```
int madMethod(int number) {
  @throw [NSException exceptionWithName: @"Simply upset"
          reason: @"For no reason" userInfo: nil];
}

int main (int argc, const char * argv[]) {

  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

  @try {
     madMethod(1);
  }
  @catch (NSException* ex) {
     printf("Something went wrong %s\n", [[ex reason] UTF8String]);
  }
  @finally {
     printf("Finally block...\n");
  }

  [pool drain];
  return 0;
}
```

```
Something went wrong For no reason
Finally block...
```

#include vs. #import

- These help you to bring in, typically, header files
- They're similar except that import will ensure a file is never included more than once
- So, it is better to use #import instead of #include

Forward Declaration

- While #import is quite helpful, there are times when you'll have trouble with cyclic dependency between classes or simply you want to defer #import to the implementation file
- In these cases, use @class for forward declaration, like @class SomeClass;
- For forward declaring protocols, write @protocol ProtocolName;

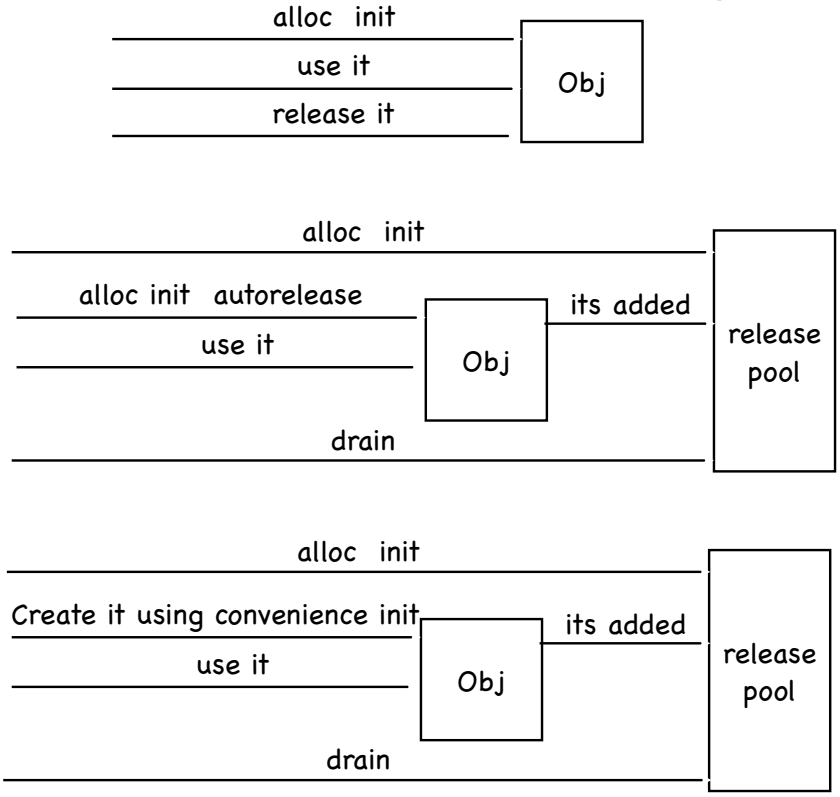
Memory Management

- On the iPhone, you're responsible for garbage collection
- It can be very intimidating if you come from a JVM or a CLR background
- It is much less painful when compared to C++
- But there is quite a bit of discipline to follow

Memory Management

- Objective-C uses retain counting to keep track of objects life—seems like COM all over again?!
- For most part you don't want to poke into retain counting, but you could!
- An object dies when its reference count goes to zero
- You have to take care of releasing objects you create using alloc or copy
- Objects you created without using alloc or copy are added to a NSAutoreleasePool—you don't release these
- Your object should clean up objects it owns—dealloc is a good place for this

Three ways to Manage Mem



Autorelease pool

- Auto release pool is a managed object that holds references to objects
- When the pool is drained, it releases objects it holds
- Use drain and not release no pool (drain is a no-op in runtimes that provide automatic GC)
- You can have nested pools
- In iPhone dev, you rarely create a pool—its given for you
  - Each invocation of event is managed by a pool
  - Create a pool if you want quicker clean up (large objects in a loop)

Memory Management Rules

- Some rules to follow
- Release objects you obtained by calling alloc, copy, etc.
- If you don't own it, don't release it
- If you store a pointer, make a copy or call retain
- Be mindful of object's life. If you obtain an object and cause its removal from a collection or remove its owner, the object may no longer be alive. To prevent this, retain while you use and release when done

Memory Management

```
@interface Engine : NSObject {
  int _power;
}

-(int) power;
-(id) initWithPower: (int) thePower;
+(id) engineWithPower: (int) thePower;

@end
```

You made power readonly

You have provided a convenience constructor and a regular constructor

Memory Management

```
@implementation Engine

-(int) power { return _power; }

-(id) initWithPower: (int) thePower {
  printf("Engine created\n");
  if (self = [super init]) {
    _power = thePower;
  }

  return self;
}

- (id)init { return [self initWithPower: 10]; }

+(id) engineWithPower: (int) thePower {
  return [[[Engine alloc] initWithPower: thePower] autorelease];
}

- (void)dealloc {
  printf("Engine deallocated\n");
  [super dealloc];
}
```

invoke your designated constructor from the init method

Convenience constructor adds instance to the pool

Memory Management

```
#import "Engine.h"

@interface Car : NSObject {
  int _year;
  Engine* _engine;
}

-(Engine*) engine;
-(void) setEngine: (Engine*) engine;
-(int) year;

-(id) initWithYear: (int) year engine: (Engine*) engine;

+(id) carWithYear: (int) year engine: (Engine*) engine;

@end
```

Memory Management

```
@implementation Car

-(Engine*) engine {
  return _engine;
}

-(void) setEngine: (Engine*) engine {
  [_engine release];
  [engine retain];
  _engine = engine; // or you could make a copy
}

-(int) year {
  return _year;
}
```

Your setEngine should take care of cleanup. It should also call retain to take ownership of the engine.

Memory Management

```
-(id) initWithYear: (int) year engine: (Engine*) engine {
  printf("Car created\n");
  if (self = [super init]) {
    _year = year;
    [engine retain];
    _engine = engine;
  }

  return self;
}

-(id) init {
    @throw [[NSException alloc] initWithName:
       @"Invalid construction" reason: @"provide year and engine"
       userInfo:nil];
}

+(id) carWithYear: (int) year engine: (Engine*) engine {
  return [[[Car alloc] initWithYear: year engine: engine]
        autorelease];
}
```

Remember to call retain.

Memory Management

```
- (void)dealloc {
  printf("Car deallocated\n");
  [_engine release];
  [super dealloc];
}

@end
```

Remember to release.

Memory Management

```
Car* createCar(int year, int enginePower) {
  Engine* engine = [[[Engine alloc] initWithPower: enginePower]
          autorelease];

  Car* car = [Car carWithYear: year engine: engine];
  return car;
}

int main (int argc, const char * argv[]) {
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  printf("\n");
  Car* car1 = createCar(2010, 20);
  Car* car2 = createCar(2010, 30);

  Engine* engine = [Engine engineWithPower: 25];
  [car2 setEngine: engine];

  printf("%d %d\n", [car1 year], [[car1 engine] power]);
  printf("%d %d\n", [car2 year], [[car2 engine] power]);

  [pool drain];
  return 0;
}
```

```
Engine created
Car created
Engine created
Car created
Engine created
2010 20
2010 25
Car deallocated
Engine deallocated
Engine deallocated
Car deallocated
Engine deallocated
```

# of objects created should be equal to # destroyed.

Easing Pain With Properties

- You have to remember to call retain and release on objects
- Your setter gets complicated because of this
- You can ease the pain using properties
- The generated setter knows when and what to release
- When you call set, it releases existing object and adds retain on the new one

Easing Pain With Properties

```
@interface Engine : NSObject {}

@property (readonly) int power;
-(id) initWithPower: (int) thePower;
+(id) engineWithPower: (int) thePower;

@end

@synthesize power;

-(id) initWithPower: (int) thePower {
  printf("Engine created\n");
  if (self = [super init]) {
    self->power = thePower;   //Way to set the readonly property
  }

  return self;
}

...
```

Property make life a bit easy here.

Easing Pain With Properties

```
@interface Car : NSObject {}

@property (nonatomic, retain) Engine* engine;
@property (readonly) int year;

-(id) initWithYear: (int) year engine: (Engine*) engine;
+(id) carWithYear: (int) year engine: (Engine*) engine;
@end

@synthesize year;
@synthesize engine;

-(id) initWithYear: (int) theYear engine: (Engine*) theEngine {
  printf("Car created\n");

  if (self = [super init]) {
    self->year = theYear;
    self.engine = theEngine;
  }

  return self;
}
```

Property make life a lot easier here.

No need to write getters and setters

```
- (void)dealloc {
  printf("Car deallocated\n");
  self.engine = nil;
  [super dealloc];
}
```


Easing Pain With Properties

```
Car* createCar(int year, int enginePower) {
  Engine* engine = [[[Engine alloc] initWithPower: enginePower]
autorelease];

  Car* car = [Car carWithYear: year engine: engine];
  return car;
}

int main (int argc, const char * argv[]) {
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  printf("\n");
  Car* car1 = createCar(2010, 20);
  Car* car2 = createCar(2010, 30);

  Engine* engine = [Engine engineWithPower: 25];
  [car2 setEngine: engine];

  printf("%d %d\n", [car1 year], [[car1 engine] power]);
  printf("%d %d\n", [car2 year], [[car2 engine] power]);

  [pool drain];
  return 0;
}
```

Usage of these classes does not change

```
Engine created
Car created
Engine created
Car created
Engine created
2010 20
2010 25
Car deallocated
Engine deallocated
Engine deallocated
Car deallocated
Engine deallocated
```

Blocks in Objective-C

- Represents a chunk of code
- They're like closures or function values in functional languages
- They respond to NSObject methods

Declaring a Block

- You use the symbol ^ to indicate you're declaring a block
- You can create an anonymous function
- You can assign it to a variable (or handle) if you like

Using A Block

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

NSArray* values = [NSArray arrayWithObjects:
          [NSNumber numberWithInt:1],
          [NSNumber numberWithInt:2],
          [NSNumber numberWithInt:3],
          [NSNumber numberWithInt:4],
          nil];

[values enumerateObjectsUsingBlock:
   ^(id obj, NSUInteger number, BOOL* breakOut) {
     printf("number at index %lu is %d\n", number,
              [obj intValue]);
}];

[pool drain];
```

```
number at index 0 is 1
number at index 1 is 2
number at index 2 is 3
number at index 3 is 4
```


Block Can Reach Out

```
int factor = 2;
[values enumerateObjectsUsingBlock:
   ^(id obj, NSUInteger number, BOOL* breakOut) {
     printf("double of number at index %lu is %d\n", number,
        [obj intValue] * factor);
}];
```

```
double of number at index 0 is 2
double of number at index 1 is 4
double of number at index 2 is 6
double of number at index 3 is 8
```

You're able to access factor from within the block.

block makes a copy of the variable it reaches out to.

You can't change the outside variable... unless you mark them with __block

__block

```
__block int total = 0;
[values enumerateObjectsUsingBlock:
   ^(id obj, NSUInteger number, BOOL* breakOut) {
     total += [obj intValue];
}];

printf("Total of values is %d\n", total);
```

```
Total of values is 10
```

If you want to modify an outside variable from within a block, you have to annotate it with a __block

Thank You!

Venkat Subramaniam
venkats@agiledeveloper.com
twitter: venkat_s