iOS Application Development

Objective C

Ashiq Uz Zoha (Ayon),
ashiq.ayon@gmail.com,
Dhrubok Infotech Services Ltd.

Introduction

- Objective-C is a general-purpose, object-oriented programming language that adds Smalltalk-style messaging to the C programming language.
- It is the main programming language used by Apple for the OS X and iOS operating systems and their respective APIs, Cocoa and Cocoa Touch.
- Originally developed in the early 1980s, it was selected as the main language used by NeXT for its NeXTSTEP operating system, from which OS X and iOS are derived.

Introduction

- Objective-C was created primarily by Brad Cox and Tom Love in the early 1980s at their company Stepstone.
- Uses power of C and features of SmallTalk. We'll see later.

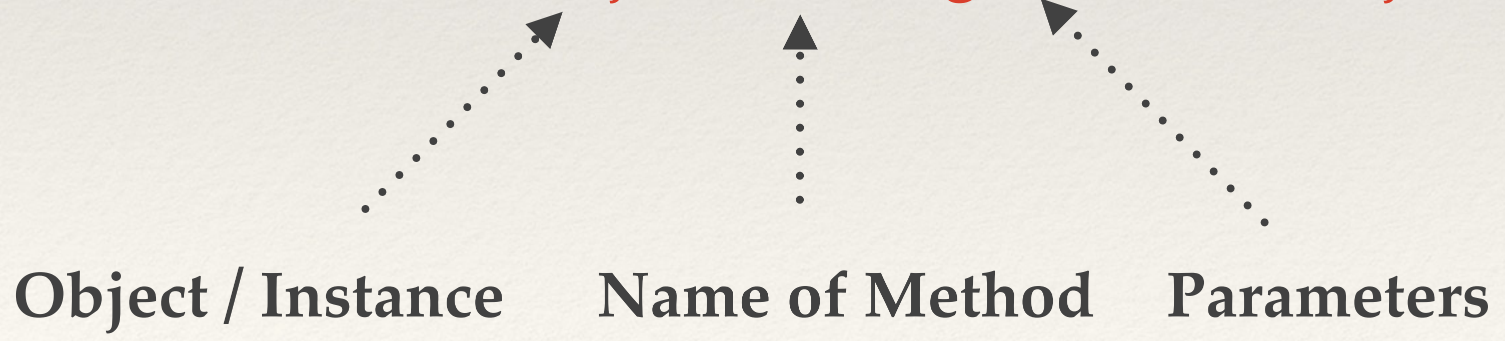
Syntax

- Objective C syntax is completely different that we have used so far in the programming languages we know. Let’s have a look…

Message

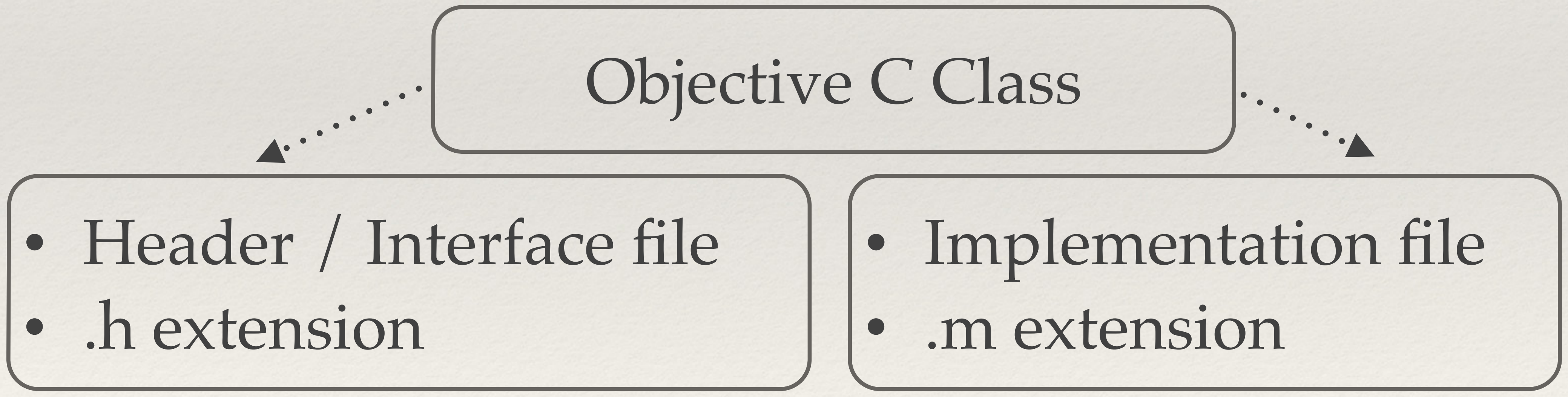
- Objective C objects sends messages to another object. It's similar to calling a method in our known language like C++ and Java .

```
obj->method(argument);  // C++
obj.method(argument); // java

[obj method:argument] ; // Objective C
```



Interfaces and implementations

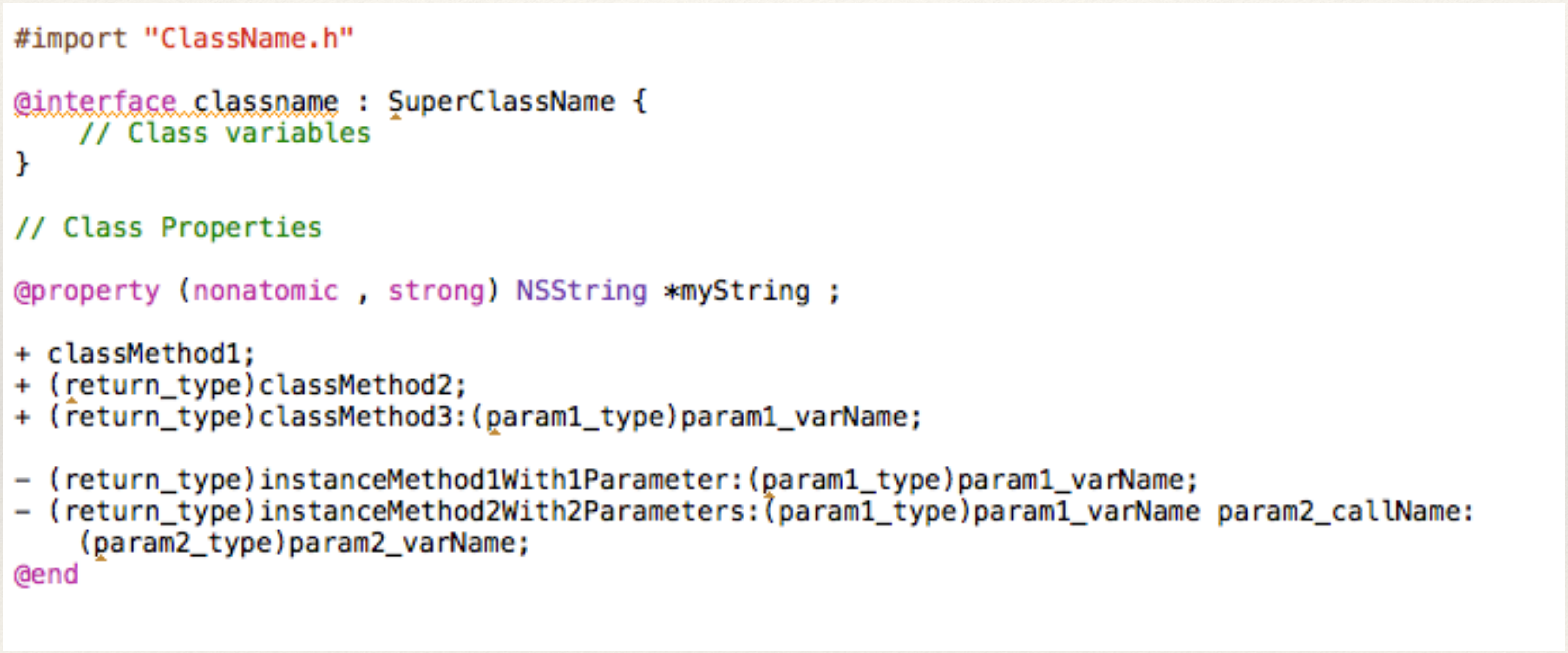
- Objective-C requires that the interface and implementation of a class be in separately declared code blocks.
- By convention, developers place the interface in a header file and the implementation in a code file. The header files, normally suffixed .h, are similar to C header files while the implementation (method) files, normally suffixed .m, can be very similar to C code files.



Interface

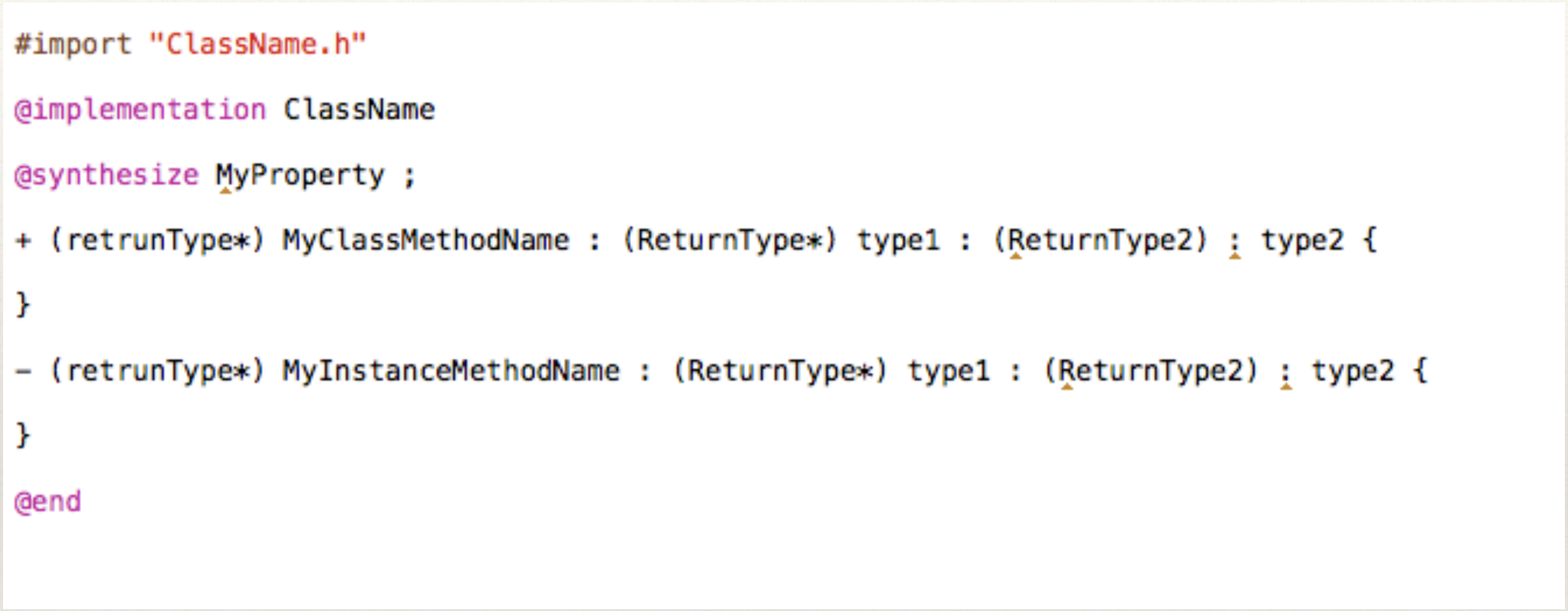
- NOT “User Interface” or “Interface of OOP in Java” :)
- In other programming languages, this is called a "class definition”.
- The interface of a class is usually defined in a header file.

Interface



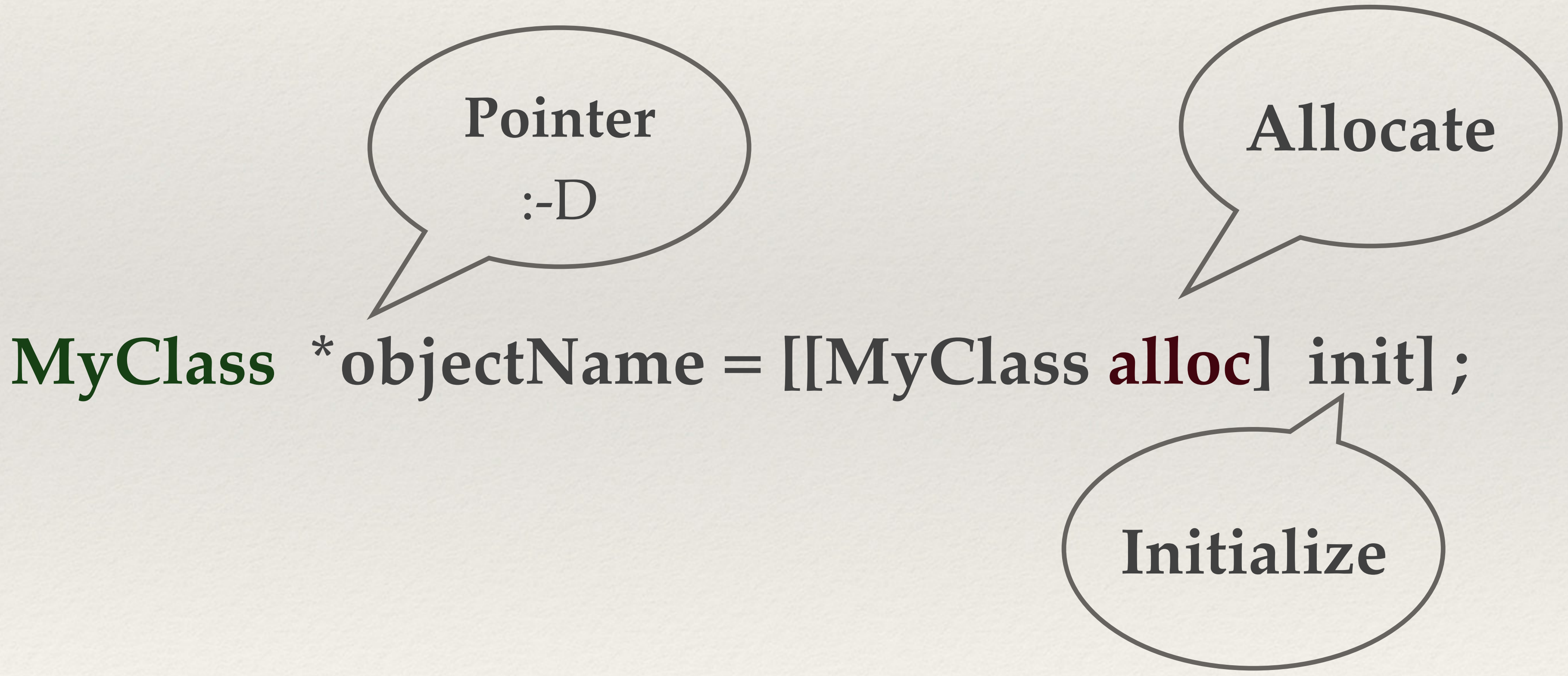
```
#import "ClassName.h"

@interface classname : SuperClassName {
    // Class variables
}

// Class Properties

@property (nonatomic , strong) NSString *myString ;

+ classMethod1;
+ (return_type)classMethod2;
+ (return_type)classMethod3:(param1_type)param1_varName;

- (return_type)instanceMethod1With1Parameter:(param1_type)param1_varName;
- (return_type)instanceMethod2With2Parameters:(param1_type)param1_varName param2_callName:
    (param2_type)param2_varName;
@end
```

Implementation



```
#import "ClassName.h"

@implementation ClassName

@synthesize MyProperty ;

+ (retrunType*) MyClassMethodName : (ReturnType*) type1 : (ReturnType2) : type2 {

}

- (retrunType*) MyInstanceMethodName : (ReturnType*) type1 : (ReturnType2) : type2 {

}

@end
```

Object Creation



```
MyClass  *objectName = [[MyClass alloc]  init] ;
```

Methods

- Method is declared in objective C as follows
- Declaration :

```
- (returnType) methodName:(typeName) variable1 :
(typeName)variable2;
```

- Example :

```
-(void) calculateAreaForRectangleWithLength:(CGfloat) length ;
```

- Calling the method:

```
[self calculateAreaForRectangleWithLength:30];
```

Class Method

- Class methods can be accessed directly without creating objects for the class. They don't have any variables and objects associated with it.
- Example :
- +(void)simpleClassMethod;
- It can be accessed by using the class name (let's assume the class name as MyClass) as follows.
- [MyClass simpleClassMethod];

Class Method

Sounds Familiar ?

- **Can we remember something like “Static Method” ?**

- public static void MethodName(){}
- Classname.MethodName() ; / / static method
- [ClassName MethodName]; / / class method

Instance methods

- Instance methods can be accessed only after creating an object for the class. Memory is allocated to the instance variables.
- Example :
- -(void)simpleInstanceMethod;
- Accessing the method :

```
MyClass  *objectName = [[MyClass alloc]init] ;
[objectName simpleInstanceMethod];
```

Property

- For an external class to access class variables properties are used.
- Example: @property(nonatomic , strong) NSString *myString;
- You can use dot operator to access properties. To access the above property we will do the following.
- self.myString = @“Test"; or [self setMyString:@"Test"];

Memory Management

- Memory management is the programming discipline of managing the life cycles of objects and freeing them when they are no longer needed.
- Memory management in a Cocoa application that doesn’t use garbage collection is based on a reference counting model.
- When you create or copy an object, its retain count is 1. Thereafter other objects may express an ownership interest in your object, which increments its retain count.

Memory Management Rules

- ***You own any object you create by allocating memory for it or copying it.***

  Related methods: alloc, allocWithZone:, copy, copyWithZone:, mutableCopy, mutableCopyWithZone:

- ***If you are not the creator of an object, but want to ensure it stays in memory for you to use, you can express an ownership interest in it.***

  Related method: retain

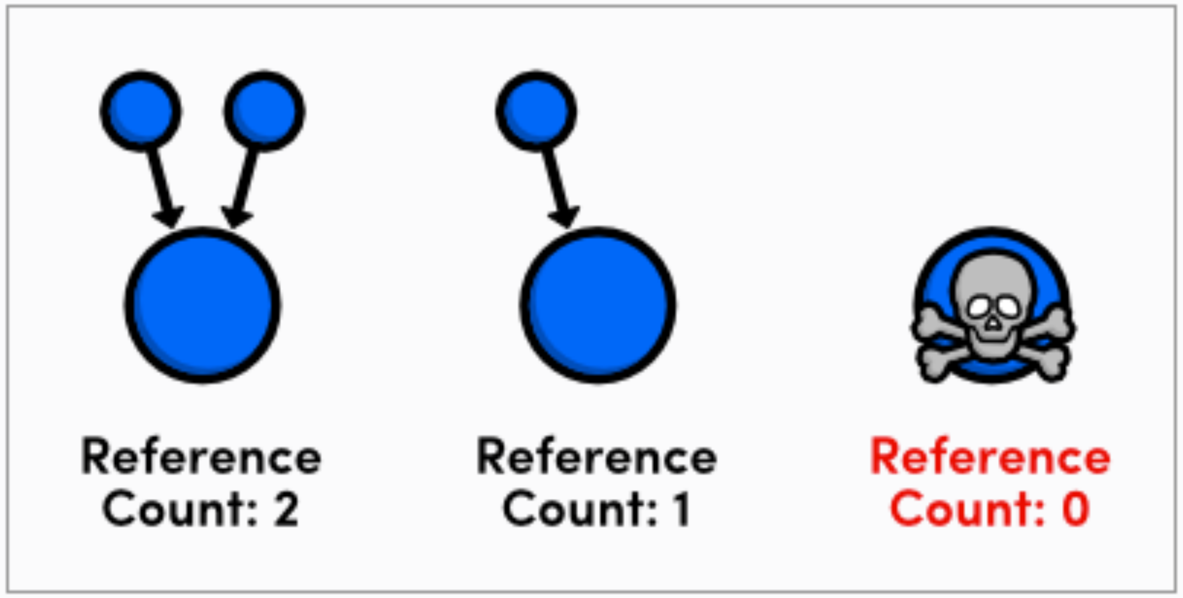
- ***If you own an object, either by creating it or expressing an ownership interest, you are responsible for releasing it when you no longer need it.***

  Related methods: release, autorelease

- ***Conversely, if you are not the creator of an object and have not expressed an ownership interest, you must not release it.***

Reference Counting System

- object-ownership scheme is implemented through a reference-counting system that internally tracks how many owners each object has. When you claim ownership of an object, you increase it’s reference count, and when you’re done with the object, you decrease its reference count.
- While its reference count is greater than zero, an object is guaranteed to exist, but as soon as the count reaches zero, the operating system is allowed to destroy it.



Destroying an object with zero references

- In the past, developers manually controlled an object's reference count by calling special memory-management methods defined by the NSObject protocol. This is called Manual Retain Release (MRR).

- In a Manual Retain Release environment, it's your job to claim and relinquish ownership of every object in your program. You do this by calling special memory-related methods,

MRR

alloc	Create an object and claim ownership of it.
retain	Claim ownership of an existing object.
copy	Copy an object and claim ownership of it.
release	Relinquish ownership of an object and destroy it immediately.
autorelease	Relinquish ownership of an object but defer its destruction.

We don’t need them Now :)

Automatic Reference Counting

- Automatic Reference Counting works the exact same way as MRR, but it automatically inserts the appropriate memory-management methods for you.
- This is a big deal for Objective-C developers, as it lets them focus entirely on what their application needs to do rather than how it does it.

Constructors , Destructors

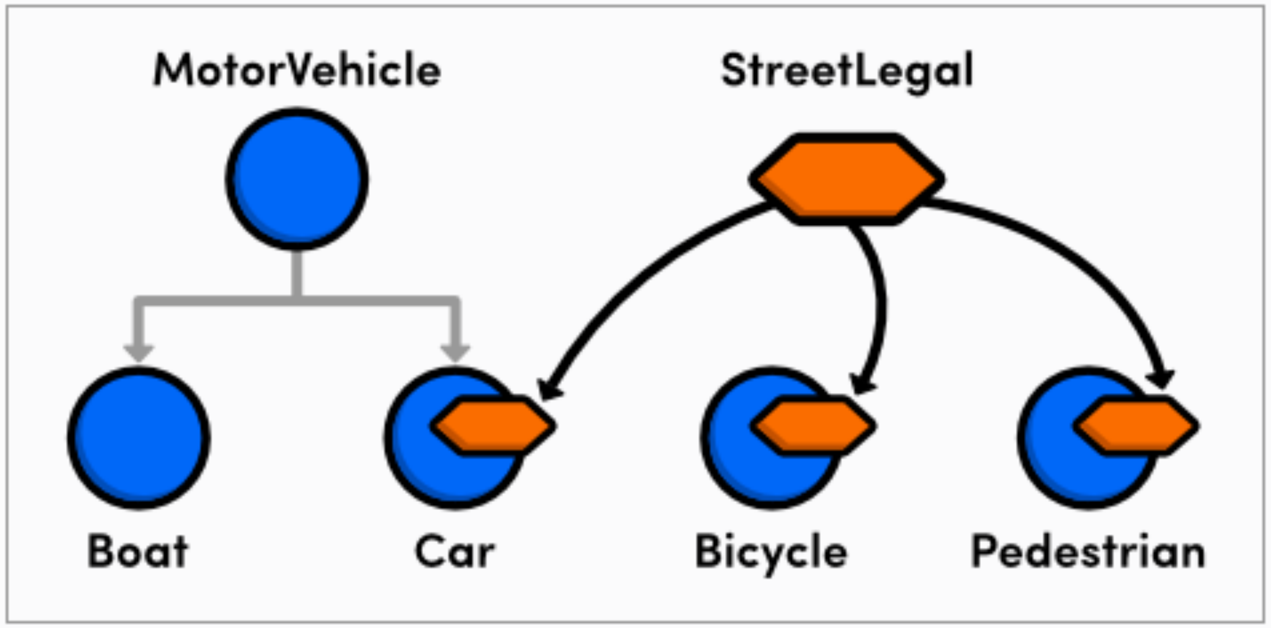
- Constructor in objective C is technically just “init” method.
- Default constructor for every object is

```
-(id) init {
                    self = [super init] ;
                    return self ;
               }
```

- Like Java , Objective C has only parent class , i,e NSObject. So, we access constructor of super class by [super init];

Protocols

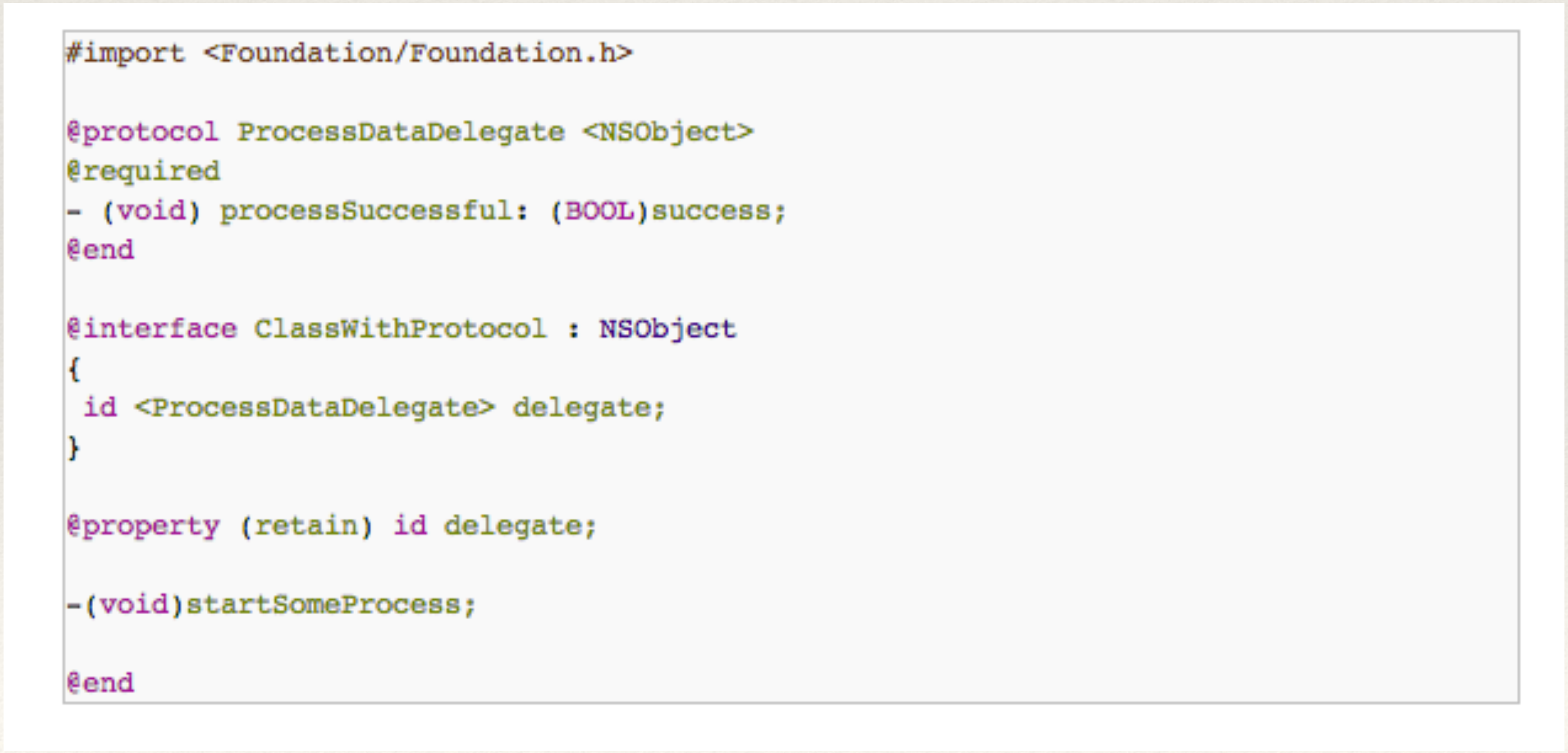
- A protocol is a group of related properties and methods that can be implemented by any class.
- They are more flexible than a normal class interface, since they let you reuse a single API declaration in completely unrelated classes.



Unrelated classes adopting the `StreetLegal` *protocol*

Protocol Definition

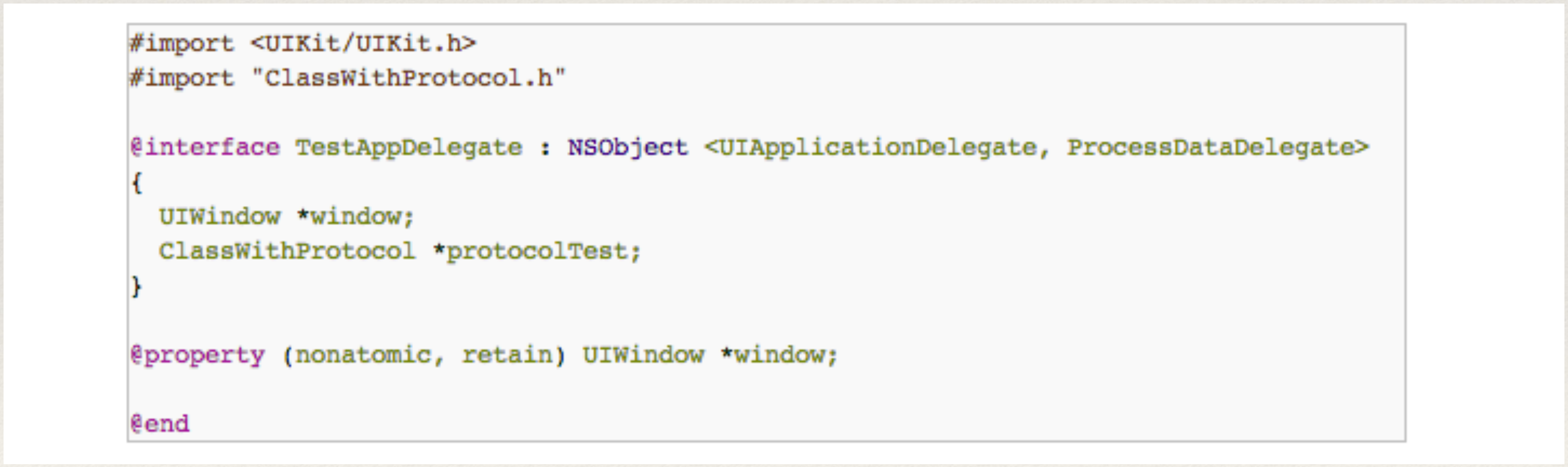
- Here is an example of a protocol which includes one method, notice the instance variable delegate is of type id, as it will be unknown at compile time the type of class that will adopt this protocol.



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

@protocol ProcessDataDelegate <NSObject>
@required
- (void) processSuccessful: (BOOL)success;
@end

@interface ClassWithProtocol : NSObject
{
 id <ProcessDataDelegate> delegate;
}

@property (retain) id delegate;

-(void)startSomeProcess;

@end
```

Adopting the Protocol

- To keep the example short, I am using the application delegate as the class that adopts the protocol. Here is how the app delegate looks:



```
#import <UIKit/UIKit.h>
#import "ClassWithProtocol.h"

@interface TestAppDelegate : NSObject <UIApplicationDelegate, ProcessDataDelegate>
{
  UIWindow *window;
  ClassWithProtocol *protocolTest;
}

@property (nonatomic, retain) UIWindow *window;

@end
```

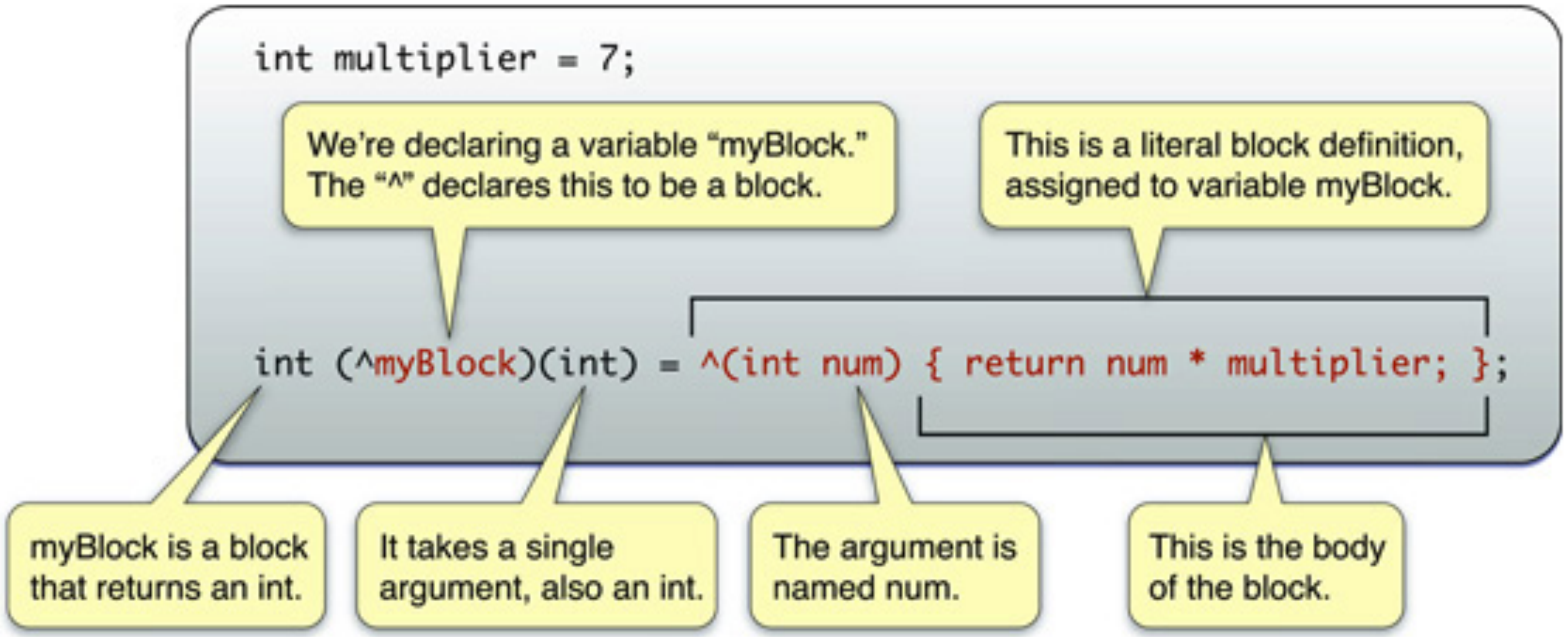
Blocks

- An Objective-C class defines an object that combines data with related behaviour. Sometimes, it makes sense just to represent a single task or unit of behaviour, rather than a collection of methods.
- Blocks are a language-level feature added to C, Objective-C and C++, which allow you to create distinct segments of code that can be passed around to methods or functions as if they were values.
- Blocks are Objective-C objects, which means they can be added to collections like NSArray or NSDictionary.

Blocks

```
int multiplier = 7;
int (^myBlock)(int) = ^(int num) {
    return num * multiplier;
};
```

The example is explained in the following illustration:



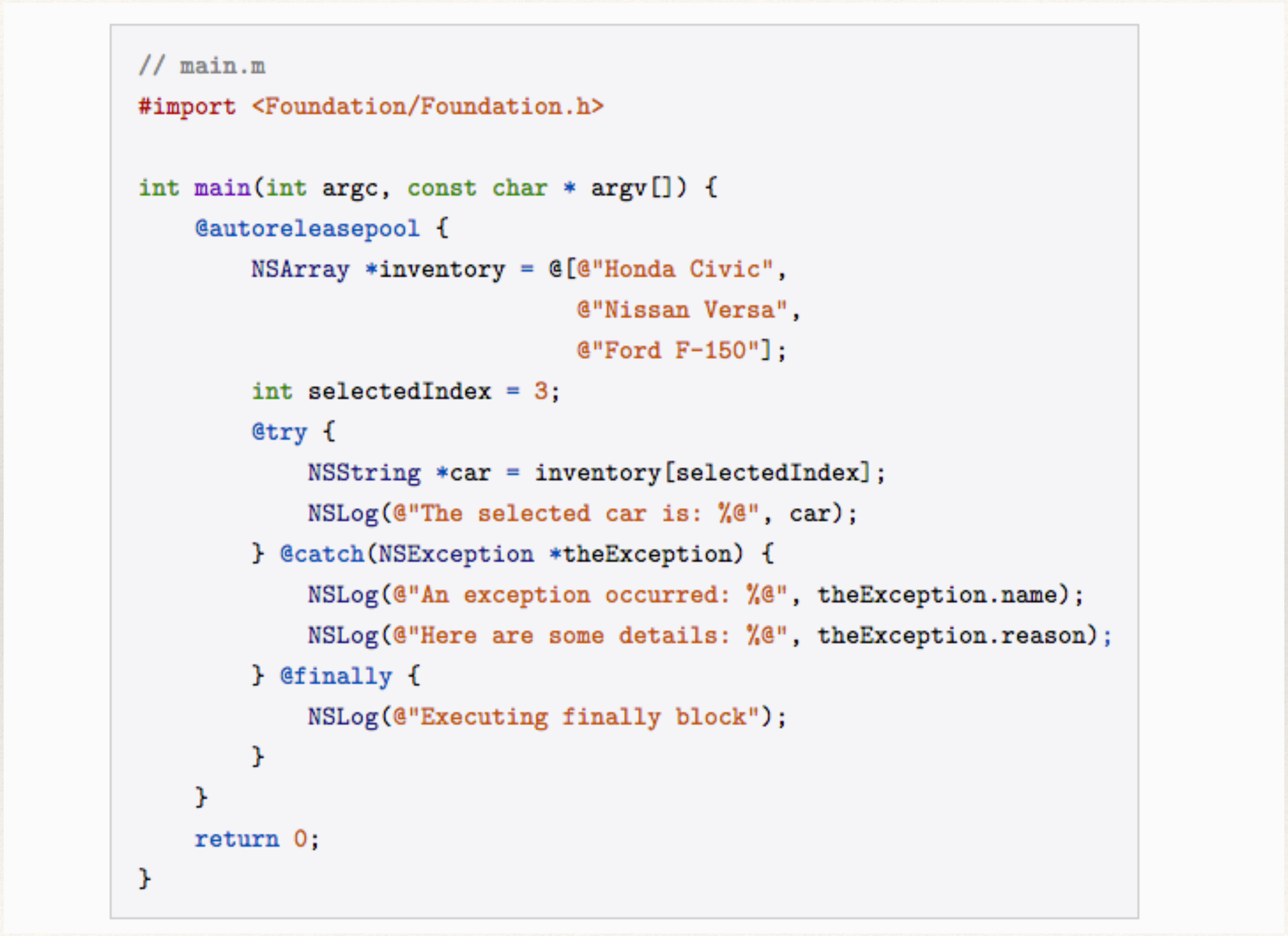
```
int multiplier = 7;
int (^myBlock)(int) = ^(int num) { return num * multiplier; };
```

We're declaring a variable "myBlock." The "^" declares this to be a block.
This is a literal block definition, assigned to variable myBlock.
myBlock is a block that returns an int.
It takes a single argument, also an int.
The argument is named num.
This is the body of the block.

Exceptions & Errors

- Exceptions can be handled using the standard try-catch-finally pattern found in most other high-level programming languages. First, you need to place any code that might result in an exception in an @try block. Then, if an exception is thrown, the corresponding @catch() block is executed to handle the problem. The @finally block is called afterwards, regardless of whether or not an exception occurred.

Exceptions & Errors



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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSArray *inventory = @[@"Honda Civic",
                               @"Nissan Versa",
                               @"Ford F-150"];
        int selectedIndex = 3;
        @try {
            NSString *car = inventory[selectedIndex];
            NSLog(@"The selected car is: %@", car);
        } @catch(NSException *theException) {
            NSLog(@"An exception occurred: %@", theException.name);
            NSLog(@"Here are some details: %@", theException.reason);
        } @finally {
            NSLog(@"Executing finally block");
        }
    }
    return 0;
}
```

“Thank You”