# OBJECTIVE-C

# OBJECTIVE-C

- Invented by Brand Cox and Tom Love in the early 80's

- Main language used by Apple for the OS X and iOS operating systems and their libraries

- High-level, object- oriented programming language that adds Smalltalk-style messaging to the C programming language

# OBJECTIVE-C

- Strict superset of C (that means pointers :D)
- Dynamic typing for Objective-C objects
- Static typing for C types
- Automatic Reference Counting (Garbage collecting -ish)

# FILE EXTENSIONS

- Implementation files have the **.m** extension
- Header files have the **.h** extension
- Files mixed with C++ have the **.mm** extension

# IMPORTS

- Another improvement over C

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

- #import
  - Smart #include
  - No more #ifndef and #define __MYHEADER_H__

# GOOD TO KNOW

- Boolean keyword is 'BOOL'
- Boolean values are 'YES' and 'NO'
- Null is called 'Nil'
- @"This is a new NSString object which is a wrapper for the normal C string"

```
BOOL whut = NO;
SomeClass theObject = nil;
NSString *string = @"Great";
```

# FLOW CONTROL

```
if (answer == 42)
{
    moveAlongSir = YES;
}
else
{
    return 0;
}
```

```
for (int i = 0; i < number; i++)
{
    someFunction();
}

while (YES)
{
    point = whereIsThisGoing();
}
```

```
switch (number) {
    case 0:
        someFunction();
        break;

    case 1:
    case 2:
        someOtherFunction();
        break;

    default:
        moveAlongSir = NO;
        break;
}
```

Just like in C

# CONST AND ENUM

- Two ways of defining constants
  - const
  - #define
- Naming convention is ‘kConstName’
- Enums can be anonymous or declared as types

```
const CGFloat kSpeed = 5.5f;
#define SPEED 5.5f

typedef enum
{
    kLeft,
    kRight
} Direcion;
```

# OOP

- No multiple inheritance
- No method overloading
- Keyword '**self**' instead of 'this'
- Keyword '**super**' denotes the parent class (inheritance)
- Sending **messages** instead of calling methods
- Messages sent to a **nil** object are ignored instead of crashing the program
- All classes inherit from **NSObject**

# THE ID TYPE

- Provides dynamic typing
- Every object is of implicit type **id**
- **id** is not a class, it's a type
- Typing object as id tells compiler: "This will eventually be some Objective-C object"
- Powerful because you can send any message to an id object and nothing happens till runtime

# METHODS

## Java/C++/C#

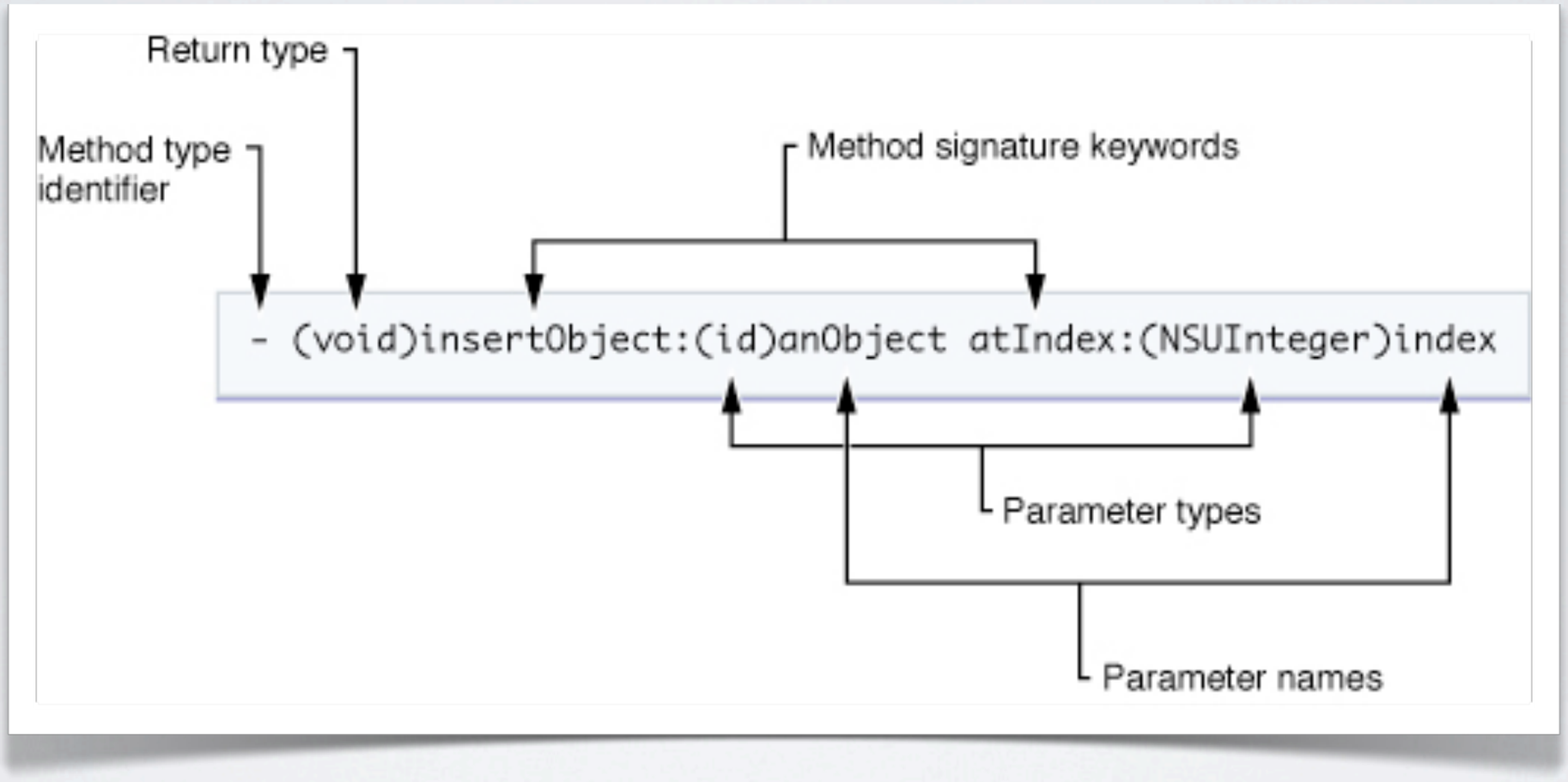
```
someObject.simpleStuff();
someObject.doSomethingCool(argument1, argument2);
```

## Objective-C

```
[someObject simpleStuff];
[someObject doSomethingCool:argument1 withData:argument2];
```

Named arguments! :D

# METHODS


Return type
Method type identifier
Method signature keywords
- (void)insertObject:(id)anObject atIndex:(NSUInteger)index
Parameter types
Parameter names

# METHODS

- - denotes a method on the object (instance method)
- + denotes a method on the class (static method)
- - (returnType)nameOfMethod:(ArgumentType)argument

```
- (void)runInCircles:(NSInteger)numberOfCircles
{
    for (int i = 0; i < numberOfCircles; i++)
    {
        [self walkInCircle:kLeft withSpeed:10];
    }
}
```

# CLASSES

## Split into interface (.h) and implementation (.m)

```
const CGFloat kSpeed = 5.5f;
typedef enum
{
    kLeft,
    kRight
} Direction;

@interface Elephant : NSObject
{
    NSString *_name;
    CGFloat *_currentSpeed;
}

- (void)runInCircles:(NSInteger)circles;
- (void)walkInCircle:(Direction)direction
           withSpeed:(CGFloat)speed;

@end
```

### Using the class

```
Elephant *el =
[[Elephant alloc] init];
```

```
#import "Elephant.h"

@implementation Elephant

- (id)init
{
    self = [super init];
    if (self != nil)
    {
        // Initial setup
        _name = @"Marco";
        _currentSpeed = 0;
    }
    return self;
}

- (void)runInCircles:(NSInteger)numberOfCircles
{
    for (int i = 0; i < numberOfCircles; i++)
    {
        [self walkInCircle:kLeft withSpeed:10];
    }
}

- (void)walkInCircle:(Direction)direction
           withSpeed:(CGFloat)speed
{
    // TODO: implement this
}
@end
```

# PROPERTIES

- Defining a property automatically creates setters and getters

- Automatically generates an instance variable _name;

```
const CGFloat kSpeed = 5.5f;
typedef enum
{
    kLeft,
    kRight
} Direction;

@interface Elephant : NSObject
{
    // Look, No instance variables!
}

- (void)runInCircles:(NSInteger)circles;
- (void)walkInCircle:(Direction)direction
           withSpeed:(CGFloat)speed;

@property (nonatomic, retain) NSString *name;
@property (readonly) CGFloat currentSpeed;

@end
```

# PROTOCOLS

- Similar to interfaces in Java/C#
- A class that implements a protocol promises to implement defined methods

```
@protocol Jumper<NSObject>

- (void)jump;

@end
```

```
@interface Elephant : NSObject<Jumper>
{
    // ..
}
```

# THE INIT METHOD

- The init method is kind of a constructor
- Returns the object
- Called manually

```
- (id)initWithName:(NSString *)name
{
    self = [super init];
    if (self != nil)
    {
        // Initial setup
        _name = name;
        _currentSpeed = 0;
    }
    return self;
}
```

```
Elephant *el = [[Elephant alloc] initWithName:@"Marco"];
```

# ALTERNATIVE TO INIT

- Class method that takes care of alloc and init
- Most built in classes implement this pattern
- Used to be different in terms of memory management until recently (coming up)

```
@interface Elephant : NSObject<Jumper>
{

}

+ (id)elephantWithName:(NSString *)name;

// ...
@end
```

```
@implementation Elephant

+ (id)elephantWithName:(NSString *)name
{
    return [[Elephant alloc] initWithName:name];
}

// ..
@end
```

# FOUNDATION FRAMEWORK

- NSArray & NSMutableArray
- NSDictionary & NSDictionary
- NSString
- NSInteger / NSUInteger
- NSNumber / NSValue
- NSLog

# BUILT IN CLASSES

- NSArray / NSMutableArray
- NSDictionary / NSMutableDictionary
- NSException
- NSDate, NSDateFormatter
- And many many more..

```
NSDate *date = [[NSDate alloc] initWithTimeIntervalSince1970:0];
NSMutableDictionary *dict = [[NSMutableDictionary alloc] init];
[dict setValue:date forKey:@"first"];
NSLog(@"%@", [dict objectForKey:@"first"]);
```

# BUILT IN TYPES

## NSInteger and NSUInteger

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

## CGPoint

```
struct CGPoint {
  CGFloat x;
  CGFloat y;
};
typedef struct
CGPoint CGPoint;
```

And a lot more..

# NSLOG

```
NSLog(@"%d, %@", number, someObject);
```

- Logs to the console (good for debuggin)
- Similar string formatting to printf(..) in C
- Further format specifiers:
  http://goo.gl/8Ab0Z

# NSASSERT, NSEXCEPTION & NSERROR

- Assert for sanity checking
- NSExceptions for internal errors
- NSError for errors that should be displayed to the user

# LITERALS

- New feature in Objective-C
- Saves time when using commonly used classes

```
// Previously
NSArray *array = [NSArray arrayWithObjects:a, b, c, nil];

// Now
array = @[ a, b, c ];

// Previously
NSDictionary *dict = [NSDictionary dictionaryWithObjects:@[o1, o2, o3]
                                                 forKeys:@[k1, k2, k3]];

// Now
dict = @{ k1 : o1, k2 : o2, k3 : o3 };

// Previously
NSNumber *number;
number = [NSNumber numberWithChar:'X'];
number = [NSNumber numberWithInt:12345];

// Now
NSNumber *number;
number = @'X';
number = @12345;
```

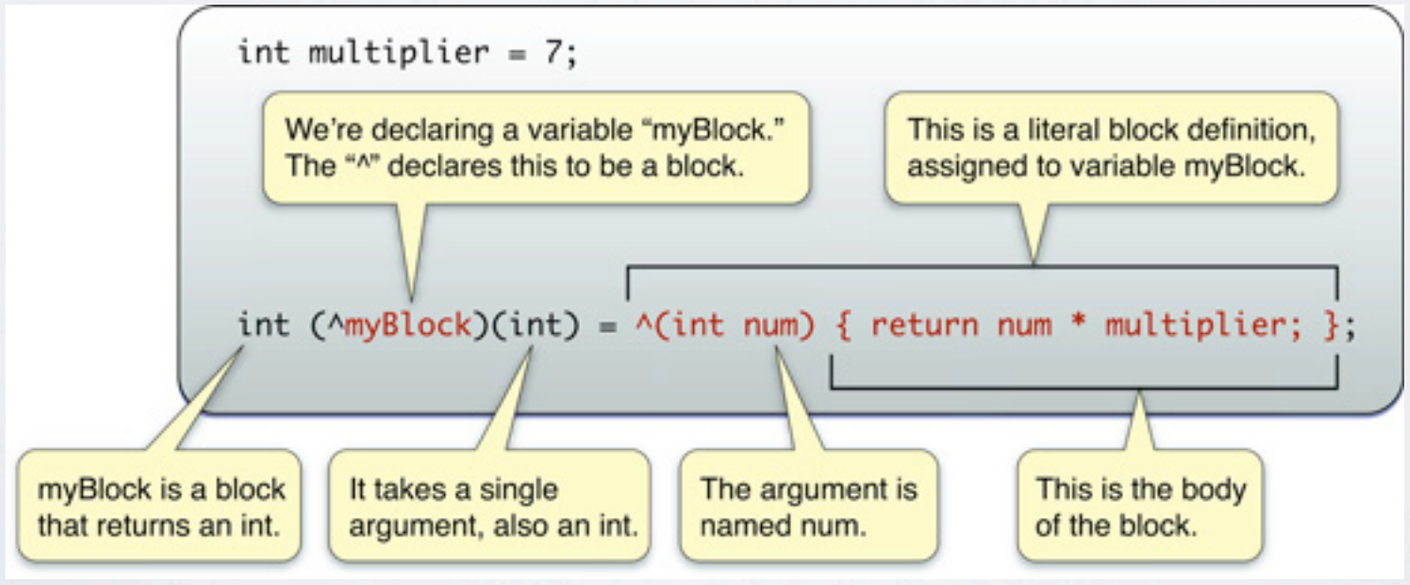
# SELECTORS

```
Elephant *el = [[Elephant alloc] init];
SEL selector = @selector(jump);
if ([el respondsToSelector:selector])
{
    [el performSelector:selector];
}
```

A selector describes a message, based upon its name

# BLOCKS

- Block objects are a C-level syntactic and runtime feature
- "Anonymous functions"
- Great for callbacks
- Can make use of variables in the scope it was created in



# SIMPLE BLOCK EXAMPLE

```objc
@class Job;
@interface Worker : NSObject
- (id)initWithJob:(Job *)job;
- (void)startWithCallback:(void (^)(void))callbackBlock;
@end
```

```objc
Worker *worker = [[Worker alloc] initWithJob:[self getNextJob]];
[worker startWithCallback:^{
    NSLog(@"Yaaay! Work complete!");
}];
```

# THE WRAPPING PROBLEM

- NSArray and other collections only accept objects that derive from NSObject

- C structs, NSIntegers, BOOL's are not objects

- But there is a solution (NSValue and NSNumber)

```
// Some variables
BOOL whut = YES;
NSInteger number = 42;
CGPoint point = CGPointMake(0.5f, 1337.0f);

// Array containing wrapper objects
NSMutableArray *array = [[NSMutableArray alloc] init];
[array addObject:[NSNumber numberWithBool:whut]];
[array addObject:[NSNumber numberWithInteger:number]];
[array addObject:[NSValue valueWithPoint:point]];
[array addObject:[NSValue valueWithPointer:&number]];

// Retreving from the array
NSInteger sameNumber = [array[1] integerValue];
CGPoint samePoint = [array[2] pointValue];
```

# MEMORY MANAGEMENT

- Until recently Objective-C only had no type of garbage collection in iOS

- Every object needed to be retained and released by it's owners. Then deallocated when everyone had released it.

- Memory leaks were common because of human errors

- Locating memory leaks is a tedious task in large projects

# AUTOMATIC REFERENCE COUNTING (ARC)

- Never need to call release or autorelese
- Runs at compile time so there is no garbage collector taking resources at runtime like in Java/C# (more efficient that GC)
- Lot of libraries and old code still uses manual reference counting so be careful
- Does not manage C code, you still have to call malloc and free

# DELEGATION PATTERN