```
// Single-line comments start with //

/*
Multi-line comments look like this
*/

// XCode supports pragma mark directive that improve jump bar readability
#pragma mark Navigation Functions // New tag on jump bar named 'Navigation Funct
ions'
#pragma mark - Navigation Functions // Same tag, now with a separator

// Imports the Foundation headers with #import
// Use <> to import global files (in general frameworks)
// Use "" to import local files (from project)
#import <Foundation/Foundation.h>
#import "MyClass.h"

// If you enable modules for iOS >= 7.0 or OS X >= 10.9 projects in
// Xcode 5 you can import frameworks like that:
@import Foundation;

// Your program's entry point is a function called
// main with an integer return type
int main (int argc, const char * argv[])
{
    // Create an autorelease pool to manage the memory into the program
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // If using automatic reference counting (ARC), use @autoreleasepool instead
:
    @autoreleasepool {

    // Use NSLog to print lines to the console
    NSLog(@"Hello World!"); // Print the string "Hello World!"

    ////////////////////////////////////
    // Types & Variables
    ////////////////////////////////////

    // Primitive declarations
    int myPrimitive1  = 1;
    long myPrimitive2 = 234554664565;

    // Object declarations
    // Put the * in front of the variable names for strongly-typed object declar
ations
    MyClass *myObject1 = nil;  // Strong typing
    id       myObject2 = nil;  // Weak typing
    // %@ is an object
    // 'description' is a convention to display the value of the Objects
    NSLog(@"%@ and %@", myObject1, [myObject2 description]); // prints => "(null
) and (null)"

    // String
    NSString *worldString = @"World";
    NSLog(@"Hello %@!", worldString); // prints => "Hello World!"
    // NSMutableString is a mutable version of the NSString object
    NSMutableString *mutableString = [NSMutableString stringWithString:@"Hello"]
;
    [mutableString appendString:@" World!"];
    NSLog(@"%@", mutableString); // prints => "Hello World!"

    // Character literals
    NSNumber *theLetterZNumber = @'Z';
    char theLetterZ            = [theLetterZNumber charValue]; // or 'Z'
    NSLog(@"%c", theLetterZ);

    // Integral literals
    NSNumber *fortyTwoNumber = @42;
    int fortyTwo             = [fortyTwoNumber intValue]; // or 42
```



```
    NSLog(@"%i", fortyTwo);

    NSNumber *fortyTwoUnsignedNumber = @42U;
    unsigned int fortyTwoUnsigned    = [fortyTwoUnsignedNumber unsignedIntValue]
; // or 42
    NSLog(@"%u", fortyTwoUnsigned);

    NSNumber *fortyTwoShortNumber = [NSNumber numberWithShort:42];
    short fortyTwoShort           = [fortyTwoShortNumber shortValue]; // or 42
    NSLog(@"%hi", fortyTwoShort);

    NSNumber *fortyOneShortNumber   = [NSNumber numberWithShort:41];
    unsigned short fortyOneUnsigned = [fortyOneShortNumber unsignedShortValue];
// or 41
    NSLog(@"%u", fortyOneUnsigned);

    NSNumber *fortyTwoLongNumber = @42L;
    long fortyTwoLong            = [fortyTwoLongNumber longValue]; // or 42
    NSLog(@"%li", fortyTwoLong);

    NSNumber *fiftyThreeLongNumber   = @53L;
    unsigned long fiftyThreeUnsigned = [fiftyThreeLongNumber unsignedLongValue];
 // or 53
    NSLog(@"%lu", fiftyThreeUnsigned);

    // Floating point literals
    NSNumber *piFloatNumber = @3.141592654F;
    float piFloat           = [piFloatNumber floatValue]; // or 3.141592654f
    NSLog(@"%f", piFloat); // prints => 3.141592654
    NSLog(@"%5.2f", piFloat); // prints => " 3.14"

    NSNumber *piDoubleNumber = @3.1415926535;
    double piDouble          = [piDoubleNumber doubleValue]; // or 3.1415926535
    NSLog(@"%f", piDouble);
    NSLog(@"%4.2f", piDouble); // prints => "3.14"

    // NSDecimalNumber is a fixed-point class that's more precise than float or
double
    NSDecimalNumber *oneDecNum = [NSDecimalNumber decimalNumberWithString:@"10.9
9"];
    NSDecimalNumber *twoDecNum = [NSDecimalNumber decimalNumberWithString:@"5.00
2"];
    // NSDecimalNumber isn't able to use standard +, -, *, / operators so it pro
vides its own:
    [oneDecNum decimalNumberByAdding:twoDecNum];
    [oneDecNum decimalNumberBySubtracting:twoDecNum];
    [oneDecNum decimalNumberByMultiplyingBy:twoDecNum];
    [oneDecNum decimalNumberByDividingBy:twoDecNum];
    NSLog(@"%@", oneDecNum); // prints => 10.99 as NSDecimalNumber is immutable

    // BOOL literals
    NSNumber *yesNumber = @YES;
    NSNumber *noNumber  = @NO;
    // or
    BOOL yesBool = YES;
    BOOL noBool  = NO;
    NSLog(@"%i", yesBool); // prints => 1

    // Array object
    // May contain different data types, but must be an Objective-C object
    NSArray *anArray      = @[@1, @2, @3, @4];
    NSNumber *thirdNumber = anArray[2];
    NSLog(@"Third number = %@", thirdNumber); // prints => "Third number = 3"
    // Since Xcode 7, NSArray objects can be typed (Generics)
    NSArray<NSString *> *stringArray = @[@"hello", @"world"];
    // NSMutableArray is a mutable version of NSArray, allowing you to change
    // the items in the array and to extend or shrink the array object.
    // Convenient, but not as efficient as NSArray.
    NSMutableArray *mutableArray = [NSMutableArray arrayWithCapacity:2];
```

```
    [mutableArray addObject:@"Hello"];
    [mutableArray addObject:@"World"];
    [mutableArray removeObjectAtIndex:0];
    NSLog(@"%@", [mutableArray objectAtIndex:0]); // prints => "World"

    // Dictionary object
    NSDictionary *aDictionary = @{ @"key1" : @"value1", @"key2" : @"value2" };
    NSObject *valueObject     = aDictionary[@"A Key"];
    NSLog(@"Object = %@", valueObject); // prints => "Object = (null)"
    // Since Xcode 7, NSDictionary objects can be typed (Generics)
    NSDictionary<NSString *, NSNumber *> *numberDictionary = @{@"a": @1, @"b": @
2};
    // NSMutableDictionary also available as a mutable dictionary object
    NSMutableDictionary *mutableDictionary = [NSMutableDictionary dictionaryWith
Capacity:2];
    [mutableDictionary setObject:@"value1" forKey:@"key1"];
    [mutableDictionary setObject:@"value2" forKey:@"key2"];
    [mutableDictionary removeObjectForKey:@"key1"];

    // Change types from Mutable To Immutable
    //In general [object mutableCopy] will make the object mutable whereas [obje
ct copy] will make the object immutable
    NSMutableDictionary *aMutableDictionary = [aDictionary mutableCopy];
    NSDictionary *mutableDictionaryChanged = [mutableDictionary copy];


    // Set object
    NSSet *set = [NSSet setWithObjects:@"Hello", @"Hello", @"World", nil];
    NSLog(@"%@", set); // prints => {(Hello, World)} (may be in different order)
    // Since Xcode 7, NSSet objects can be typed (Generics)
    NSSet<NSString *> *stringSet = [NSSet setWithObjects:@"hello", @"world", nil
];
    // NSMutableSet also available as a mutable set object
    NSMutableSet *mutableSet = [NSMutableSet setWithCapacity:2];
    [mutableSet addObject:@"Hello"];
    [mutableSet addObject:@"Hello"];
    NSLog(@"%@", mutableSet); // prints => {(Hello)}

    ////////////////////////////////////
    // Operators
    ////////////////////////////////////

    // The operators works like in the C language
    // For example:
    2 + 5; // => 7
    4.2f + 5.1f; // => 9.3f
    3 == 2; // => 0 (NO)
    3 != 2; // => 1 (YES)
    1 && 1; // => 1 (Logical and)
    0 || 1; // => 1 (Logical or)
    ~0x0F; // => 0xF0 (bitwise negation)
    0x0F & 0xF0; // => 0x00 (bitwise AND)
    0x01 << 1; // => 0x02 (bitwise left shift (by 1))

    ////////////////////////////////////
    // Control Structures
    ////////////////////////////////////

    // If-Else statement
    if (NO)
    {
        NSLog(@"I am never run");
    } else if (0)
    {
        NSLog(@"I am also never run");
    } else
    {
        NSLog(@"I print");
    }
```



```
    // Switch statement
    switch (2)
    {
        case 0:
        {
            NSLog(@"I am never run");
        } break;
        case 1:
        {
            NSLog(@"I am also never run");
        } break;
        default:
        {
            NSLog(@"I print");
        } break;
    }

    // While loops statements
    int ii = 0;
    while (ii < 4)
    {
        NSLog(@"%d,", ii++); // ii++ increments ii in-place, after using its val
ue
    } // prints => "0,"
      //           "1,"
      //           "2,"
      //           "3,"

    // For loops statements
    int jj;
    for (jj=0; jj < 4; jj++)
    {
        NSLog(@"%d,", jj);
    } // prints => "0,"
      //           "1,"
      //           "2,"
      //           "3,"

    // Foreach statements
    NSArray *values = @[@0, @1, @2, @3];
    for (NSNumber *value in values)
    {
        NSLog(@"%@,", value);
    } // prints => "0,"
      //           "1,"
      //           "2,"
      //           "3,"

    // Object for loop statement. Can be used with any Objective-C object type
    for (id item in values) {
        NSLog(@"%@,", item);
    } // prints => "0,"
      //           "1,"
      //           "2,"
      //           "3,"

    // Try-Catch-Finally statements
    @try
    {
        // Your statements here
        @throw [NSException exceptionWithName:@"FileNotFoundException"
                                       reason:@"File Not Found on System" userInfo:nil];
    } @catch (NSException * e) // use: @catch (id exceptionName) to catch all ob
jects.
    {
        NSLog(@"Exception: %@", e);
    } @finally
    {
```

```
        NSLog(@"Finally. Time to clean up.");
    } // prints => "Exception: File Not Found on System"
      //           "Finally. Time to clean up."

    // NSError objects are useful for function arguments to populate on user mis
takes.
    NSError *error = [NSError errorWithDomain:@"Invalid email." code:4 userInfo:
nil];

    ////////////////////////////////////////
    // Objects
    ////////////////////////////////////////

    // Create an object instance by allocating memory and initializing it
    // An object is not fully functional until both steps have been completed
    MyClass *myObject = [[MyClass alloc] init];

    // The Objective-C model of object-oriented programming is based on message
    // passing to object instances
    // In Objective-C one does not simply call a method; one sends a message
    [myObject instanceMethodWithParameter:@"Steve Jobs"];

    // Clean up the memory you used into your program
    [pool drain];

    // End of @autoreleasepool
    }

    // End the program
    return 0;
}

////////////////////////////////////////
// Classes And Functions
////////////////////////////////////////

// Declare your class in a header file (MyClass.h):
// Class declaration syntax:
// @interface ClassName : ParentClassName <ImplementedProtocols>
// {
//    type name; <= variable declarations;
// }
// @property type name; <= property declarations
// -/+ (type) Method declarations; <= Method declarations
// @end
@interface MyClass : NSObject <MyProtocol> // NSObject is Objective-C's base obj
ect class.
{
    // Instance variable declarations (can exist in either interface or implemen
tation file)
    int count; // Protected access by default.
    @private id data; // Private access (More convenient to declare in implement
ation file)
    NSString *name;
}
// Convenient notation for public access variables to auto generate a setter met
hod
// By default, setter method name is 'set' followed by @property variable name
@property int propInt; // Setter method name = 'setPropInt'
@property (copy) id copyId; // (copy) => Copy the object during assignment
// (readonly) => Cannot set value outside @interface
@property (readonly) NSString *roString; // Use @synthesize in @implementation t
o create accessor
// You can customize the getter and setter names instead of using default 'set'
name:
@property (getter=lengthGet, setter=lengthSet:) int length;

// Methods
+/- (return type)methodSignature:(Parameter Type *)parameterName;
```



```
// + for class methods:
+ (NSString *)classMethod;
+ (MyClass *)myClassFromHeight:(NSNumber *)defaultHeight;

// - for instance methods:
- (NSString *)instanceMethodWithParameter:(NSString *)string;
- (NSNumber *)methodAParameterAsString:(NSString*)string andAParameterAsNumber:(
NSNumber *)number;

// Constructor methods with arguments:
- (id)initWithDistance:(int)defaultDistance;
// Objective-C method names are very descriptive. Always name methods according
to their arguments

@end // States the end of the interface


// To access public variables from the implementation file, @property generates
a setter method
// automatically. Method name is 'set' followed by @property variable name:
MyClass *myClass = [[MyClass alloc] init]; // create MyClass object instance
[myClass setCount:10];
NSLog(@"%d", [myClass count]); // prints => 10
// Or using the custom getter and setter method defined in @interface:
[myClass lengthSet:32];
NSLog(@"%i", [myClass lengthGet]); // prints => 32
// For convenience, you may use dot notation to set and access object instance v
ariables:
myClass.count = 45;
NSLog(@"%i", myClass.count); // prints => 45

// Call class methods:
NSString *classMethodString = [MyClass classMethod];
MyClass *classFromName = [MyClass myClassFromName:@"Hello"];

// Call instance methods:
MyClass *myClass = [[MyClass alloc] init]; // Create MyClass object instance
NSString *stringFromInstanceMethod = [myClass instanceMethodWithParameter:@"Hell
o"];

// Selectors
// Way to dynamically represent methods. Used to call methods of a class, pass m
ethods
// through functions to tell other classes they should call it, and to save meth
ods
// as a variable
// SEL is the data type. @selector() returns a selector from method name provide
d
// methodAParameterAsString:andAParameterAsNumber: is method name for method in
MyClass
SEL selectorVar = @selector(methodAParameterAsString:andAParameterAsNumber:);
if ([myClass respondsToSelector:selectorVar]) { // Checks if class contains meth
od
    // Must put all method arguments into one object to send to performSelector
function
    NSArray *arguments = [NSArray arrayWithObjects:@"Hello", @4, nil];
    [myClass performSelector:selectorVar withObject:arguments]; // Calls the met
hod
} else {
    // NSStringFromSelector() returns a NSString of the method name of a given s
elector
    NSLog(@"MyClass does not have method: %@", NSStringFromSelector(selectedVar)
);
}

// Implement the methods in an implementation (MyClass.m) file:
@implementation MyClass {
    long distance; // Private access instance variable
```

```
    NSNumber height;
}

// To access a public variable from the interface file, use '_' followed by vari
able name:
_count = 5; // References "int count" from MyClass interface
// Access variables defined in implementation file:
distance = 18; // References "long distance" from MyClass implementation
// To use @property variable in implementation, use @synthesize to create access
or variable:
@synthesize roString = _roString; // _roString available now in @implementation

// Called before calling any class methods or instantiating any objects
+ (void)initialize
{
    if (self == [MyClass class]) {
        distance = 0;
    }
}

// Counterpart to initialize method. Called when an object's reference count is
zero
- (void)dealloc
{
    [height release]; // If not using ARC, make sure to release class variable o
bjects
    [super dealloc];  // and call parent class dealloc
}

// Constructors are a way of creating instances of a class
// This is a default constructor which is called when the object is initialized.
- (id)init
{
    if ((self = [super init])) // 'super' used to access methods from parent cla
ss
    {
        self.count = 1; // 'self' used for object to call itself
    }
    return self;
}
// Can create constructors that contain arguments:
- (id)initWithDistance:(int)defaultDistance
{
    distance = defaultDistance;
    return self;
}

+ (NSString *)classMethod
{
    return @"Some string";
}

+ (MyClass *)myClassFromHeight:(NSNumber *)defaultHeight
{
    height = defaultHeight;
    return [[self alloc] init];
}

- (NSString *)instanceMethodWithParameter:(NSString *)string
{
    return @"New string";
}

- (NSNumber *)methodAParameterAsString:(NSString*)string andAParameterAsNumber:(
NSNumber *)number
{
    return @42;
}
```



```
// Objective-C does not have private method declarations, but you can simulate t
hem.
// To simulate a private method, create the method in the @implementation but no
t in the @interface.
- (NSNumber *)secretPrivateMethod {
    return @72;
}
[self secretPrivateMethod]; // Calls private method

// Methods declared into MyProtocol
- (void)myProtocolMethod
{
    // statements
}

@end // States the end of the implementation

////////////////////////////////////////
// Categories
////////////////////////////////////////
// A category is a group of methods designed to extend a class. They allow you t
o add new methods
// to an existing class for organizational purposes. This is not to be mistaken
with subclasses.
// Subclasses are meant to CHANGE functionality of an object while categories in
stead ADD
// functionality to an object.
// Categories allow you to:
// -- Add methods to an existing class for organizational purposes.
// -- Allow you to extend Objective-C object classes (ex: NSString) to add your
own methods.
// -- Add ability to create protected and private methods to classes.
// NOTE: Do not override methods of the base class in a category even though you
 have the ability
// to. Overriding methods may cause compiler errors later between different cate
gories and it
// ruins the purpose of categories to only ADD functionality. Subclass instead t
o override methods.

// Here is a simple Car base class.
@interface Car : NSObject

@property NSString *make;
@property NSString *color;

- (void)turnOn;
- (void)accelerate;

@end

// And the simple Car base class implementation:
#import "Car.h"

@implementation Car

@synthesize make = _make;
@synthesize color = _color;

- (void)turnOn {
    NSLog(@"Car is on.");
}
- (void)accelerate {
    NSLog(@"Accelerating.");
}

@end

// Now, if we wanted to create a Truck object, we would instead create a subclas
s of Car as it would
```

```
// be changing the functionality of the Car to behave like a truck. But lets say
 we want to just add
// functionality to this existing Car. A good example would be to clean the car.
 So we would create
// a category to add these cleaning methods:
// @interface filename: Car+Clean.h (BaseClassName+CategoryName.h)
#import "Car.h" // Make sure to import base class to extend.

@interface Car (Clean) // The category name is inside () following the name of t
he base class.

- (void)washWindows; // Names of the new methods we are adding to our Car object
.
- (void)wax;

@end

// @implementation filename: Car+Clean.m (BaseClassName+CategoryName.m)
#import "Car+Clean.h" // Import the Clean category's @interface file.

@implementation Car (Clean)

- (void)washWindows {
    NSLog(@"Windows washed.");
}
- (void)wax {
    NSLog(@"Waxed.");
}

@end

// Any Car object instance has the ability to use a category. All they need to d
o is import it:
#import "Car+Clean.h" // Import as many different categories as you want to use.
#import "Car.h" // Also need to import base class to use it's original functiona
lity.

int main (int argc, const char * argv[]) {
    @autoreleasepool {
        Car *mustang = [[Car alloc] init];
        mustang.color = @"Red";
        mustang.make = @"Ford";

        [mustang turnOn]; // Use methods from base Car class.
        [mustang washWindows]; // Use methods from Car's Clean category.
    }
    return 0;
}

// Objective-C does not have protected method declarations but you can simulate
them.
// Create a category containing all of the protected methods, then import it ONL
Y into the
// @implementation file of a class belonging to the Car class:
@interface Car (Protected) // Naming category 'Protected' to remember methods ar
e protected.

- (void)lockCar; // Methods listed here may only be created by Car objects.

@end
//To use protected methods, import the category, then implement the methods:
#import "Car+Protected.h" // Remember, import in the @implementation file only.

@implementation Car

- (void)lockCar {
    NSLog(@"Car locked."); // Instances of Car can't use lockCar because it's no
t in the @interface.
}
```



```
@end

////////////////////////////////////
// Extensions
////////////////////////////////////
// Extensions allow you to override public access property attributes and method
s of an @interface.
// @interface filename: Shape.h
@interface Shape : NSObject // Base Shape class extension overrides below.

@property (readonly) NSNumber *numOfSides;

- (int)getNumOfSides;

@end
// You can override numOfSides variable or getNumOfSides method to edit them wit
h an extension:
// @implementation filename: Shape.m
#import "Shape.h"
// Extensions live in the same file as the class @implementation.
@interface Shape () // () after base class name declares an extension.

@property (copy) NSNumber *numOfSides; // Make numOfSides copy instead of readon
ly.
-(NSNumber)getNumOfSides; // Make getNumOfSides return a NSNumber instead of an
int.
-(void)privateMethod; // You can also create new private methods inside of exten
sions.

@end
// The main @implementation:
@implementation Shape

@synthesize numOfSides = _numOfSides;

-(NSNumber)getNumOfSides { // All statements inside of extension must be in the
@implementation.
    return _numOfSides;
}
-(void)privateMethod {
    NSLog(@"Private method created by extension. Shape instances cannot call me.
");
}

@end

// Starting in Xcode 7.0, you can create Generic classes,
// allowing you to provide greater type safety and clarity
// without writing excessive boilerplate.
@interface Result<__covariant A> : NSObject

- (void)handleSuccess:(void(^)(A))success
              failure:(void(^)(NSError *))failure;

@property (nonatomic) A object;

@end

// we can now declare instances of this class like
Result<NSNumber *> *result;
Result<NSArray *> *result;

// Each of these cases would be equivalent to rewriting Result's interface
// and substituting the appropriate type for A
@interface Result : NSObject
- (void)handleSuccess:(void(^)(NSArray *))success
              failure:(void(^)(NSError *))failure;
@property (nonatomic) NSArray * object;
```

```
@end

@interface Result : NSObject
- (void)handleSuccess:(void(^)(NSNumber *))success
              failure:(void(^)(NSError *))failure;
@property (nonatomic) NSNumber * object;
@end

// It should be obvious, however, that writing one
//  Class to solve a problem is always preferable to writing two

// Note that Clang will not accept generic types in @implementations,
// so your @implemnation of Result would have to look like this:

@implementation Result

- (void)handleSuccess:(void (^)(id))success
              failure:(void (^)(NSError *))failure {
  // Do something
}

@end


////////////////////////////////////
// Protocols
////////////////////////////////////
// A protocol declares methods that can be implemented by any class.
// Protocols are not classes themselves. They simply define an interface
// that other objects are responsible for implementing.
// @protocol filename: "CarUtilities.h"
@protocol CarUtilities <NSObject> // <NSObject> => Name of another protocol this
 protocol includes.
    @property BOOL engineOn; // Adopting class must @synthesize all defined @pro
perties and
    - (void)turnOnEngine; // all defined methods.
@end
// Below is an example class implementing the protocol.
#import "CarUtilities.h" // Import the @protocol file.

@interface Car : NSObject <CarUtilities> // Name of protocol goes inside <>
    // You don't need the @property or method names here for CarUtilities. Only
@implementation does.
- (void)turnOnEngineWithUtilities:(id <CarUtilities>)car; // You can use protoco
ls as data too.
@end
// The @implementation needs to implement the @properties and methods for the pr
otocol.
@implementation Car : NSObject <CarUtilities>

@synthesize engineOn = _engineOn; // Create a @synthesize statement for the engi
neOn @property.

- (void)turnOnEngine { // Implement turnOnEngine however you would like. Protoco
ls do not define
    _engineOn = YES; // how you implement a method, it just requires that you do
 implement it.
}
// You may use a protocol as data as you know what methods and variables it has
implemented.
- (void)turnOnEngineWithCarUtilities:(id <CarUtilities>)objectOfSomeKind {
    [objectOfSomeKind engineOn]; // You have access to object variables
    [objectOfSomeKind turnOnEngine]; // and the methods inside.
    [objectOfSomeKind engineOn]; // May or may not be YES. Class implements it h
owever it wants.
}

@end
// Instances of Car now have access to the protocol.
```



```
Car *carInstance = [[Car alloc] init];
[carInstance setEngineOn:NO];
[carInstance turnOnEngine];
if ([carInstance engineOn]) {
    NSLog(@"Car engine is on."); // prints => "Car engine is on."
}
// Make sure to check if an object of type 'id' implements a protocol before cal
ling protocol methods:
if ([myClass conformsToProtocol:@protocol(CarUtilities)]) {
    NSLog(@"This does not run as the MyClass class does not implement the CarUti
lities protocol.");
} else if ([carInstance conformsToProtocol:@protocol(CarUtilities)]) {
    NSLog(@"This does run as the Car class implements the CarUtilities protocol.
");
}
// Categories may implement protocols as well: @interface Car (CarCategory) <Car
Utilities>
// You may implement many protocols: @interface Car : NSObject <CarUtilities, Ca
rCleaning>
// NOTE: If two or more protocols rely on each other, make sure to forward-decla
re them:
#import "Brother.h"

@protocol Brother; // Forward-declare statement. Without it, compiler will throw
 error.

@protocol Sister <NSObject>

- (void)beNiceToBrother:(id <Brother>)brother;

@end

// See the problem is that Sister relies on Brother, and Brother relies on Siste
r.
#import "Sister.h"

@protocol Sister; // These lines stop the recursion, resolving the issue.

@protocol Brother <NSObject>

- (void)beNiceToSister:(id <Sister>)sister;

@end


////////////////////////////////////
// Blocks
////////////////////////////////////
// Blocks are statements of code, just like a function, that are able to be used
 as data.
// Below is a simple block with an integer argument that returns the argument pl
us 4.
^(int n) {
    return n + 4;
}
int (^addUp)(int n); // Declare a variable to store a block.
void (^noParameterBlockVar)(void); // Example variable declaration of block with
 no arguments.
// Blocks have access to variables in the same scope. But the variables are read
only and the
// value passed to the block is the value of the variable when the block is crea
ted.
int outsideVar = 17; // If we edit outsideVar after declaring addUp, outsideVar
is STILL 17.
__block long mutableVar = 3; // __block makes variables writable to blocks, unli
ke outsideVar.
addUp = ^(int n) { // Remove (int n) to have a block that doesn't take in any pa
rameters.
    NSLog(@"You may have as many lines in a block as you would like.");
```

```
    NSSet *blockSet; // Also, you can declare local variables.
    mutableVar = 32; // Assigning new value to __block variable.
    return n + outsideVar; // Return statements are optional.
}
int addUp = addUp(10 + 16); // Calls block code with arguments.
// Blocks are often used as arguments to functions to be called later, or for ca
llbacks.
@implementation BlockExample : NSObject

 - (void)runBlock:(void (^)(NSString))block {
    NSLog(@"Block argument returns nothing and takes in a NSString object.");
    block(@"Argument given to block to execute."); // Calling block.
 }

 @end


////////////////////////////////////
// Memory Management
////////////////////////////////////
/*
For each object used in an application, memory must be allocated for that object
. When the application
is done using that object, memory must be deallocated to ensure application effi
ciency.
Objective-C does not use garbage collection and instead uses reference counting.
 As long as
there is at least one reference to an object (also called "owning" an object), t
hen the object
will be available to use (known as "ownership").

When an instance owns an object, its reference counter is increments by one. Whe
n the
object is released, the reference counter decrements by one. When reference coun
t is zero,
the object is removed from memory.

With all object interactions, follow the pattern of:
(1) create the object, (2) use the object, (3) then free the object from memory.
*/

MyClass *classVar = [MyClass alloc]; // 'alloc' sets classVar's reference count
to one. Returns pointer to object
[classVar release]; // Decrements classVar's reference count
// 'retain' claims ownership of existing object instance and increments referenc
e count. Returns pointer to object
MyClass *newVar = [classVar retain]; // If classVar is released, object is still
 in memory because newVar is owner
[classVar autorelease]; // Removes ownership of object at end of @autoreleasepoo
l block. Returns pointer to object

// @property can use 'retain' and 'assign' as well for small convenient definiti
ons
@property (retain) MyClass *instance; // Release old value and retain a new one
(strong reference)
@property (assign) NSSet *set; // Pointer to new value without retaining/releasi
ng old (weak reference)

// Automatic Reference Counting (ARC)
// Because memory management can be a pain, Xcode 4.2 and iOS 4 introduced Autom
atic Reference Counting (ARC).
// ARC is a compiler feature that inserts retain, release, and autorelease autom
atically for you, so when using ARC,
// you must not use retain, release, or autorelease
MyClass *arcMyClass = [[MyClass alloc] init];
// ... code using arcMyClass
// Without ARC, you will need to call: [arcMyClass release] after you're done us
ing arcMyClass. But with ARC,
// there is no need. It will insert this release statement for you
```



```

// As for the 'assign' and 'retain' @property attributes, with ARC you use 'weak
' and 'strong'
@property (weak) MyClass *weakVar; // 'weak' does not take ownership of object.
If original instance's reference count
// is set to zero, weakVar will automatically receive value of nil to avoid appl
ication crashing
@property (strong) MyClass *strongVar; // 'strong' takes ownership of object. En
sures object will stay in memory to use

// For regular variables (not @property declared variables), use the following:
__strong NSString *strongString; // Default. Variable is retained in memory unti
l it leaves it's scope
__weak NSSet *weakSet; // Weak reference to existing object. When existing objec
t is released, weakSet is set to nil
__unsafe_unretained NSArray *unsafeArray; // Like __weak, but unsafeArray not se
t to nil when existing object is released
```