

Objective-C

Succinctly

by Ryan Hodson

Objective-C Succinctly

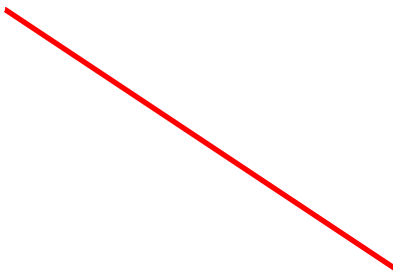
By



Ryan Hodson

Foreword by Daniel Jebaraj





Copyright © 2012 by Syncfusion Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

I mportant licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal, educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

E dited by
This publication was edited by Daniel Jebaraj, vice president, Syncfusion, Inc.

Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	8
Introduction	10
The Objective-C Language	10
Sample Code	11
Setting Up	11
Installation	12
Creating an Application	12
Getting to Know the Xcode IDE	14
Editing Files	15
Compiling Code	15
Summary.....	16
Chapter 1 Hello, Objective-C.....	17
Creating a Class.....	17
Components of a Class.....	18
Defining Methods	19
Instantiating Objects.....	20
Calling Methods	21
Adding Method Parameters	21
Defining Properties	22
Summary.....	23
Chapter 2 Data Types.....	24
Displaying Values.....	24
Primitive Data Types	25
Booleans	25
Chars.....	26
Short Integers	26
“Normal” Integers	26
Long Integers	27
Floats	27
Doubles.....	27
Structs	28
Arrays.....	28

Void	30
nil and NULL	31
Primitive Data Type Summary	31
Foundation Data Structures	31
NSNumber	31
NSDecimalNumber	32
NSString	34
NSMutableString	35
NSArray	36
NSMutableArray	38
NSSet and NSMutableSet	39
NSDictionary and NSMutableDictionary	41
The id Data Type	42
The Class Data Type	43
Foundation Data Structures Summary	43
Chapter 3 Properties	45
Declaring Properties	45
Implementing Properties	45
Instance Variables	46
Customizing Accessors	47
Dot Syntax	48
Summary	49
Chapter 4 Memory Management	50
Manual Memory Management	50
Auto-Releasing Objects	55
Manual Retain-Release Attributes	56
Automatic Reference Counting	57
ARC Attributes	58
Summary	59
Chapter 5 Methods	60
Instance vs. Class Methods	60
The super Keyword	61
Initialization Methods	61
Class Initialization	63

Deallocation Methods	64
Deallocation in MMR.....	64
Deallocation in ARC	65
Private Methods	65
Selectors	67
Method Names and Selectors	69
Performing Selectors	70
Checking for the Existence of Selectors	70
Using Selectors.....	71
Summary.....	73
Chapter 6 Categories and Extensions	74
Categories.....	74
Protected Methods.....	77
Caveats	80
Extensions.....	80
Private Methods	82
Summary.....	83
Chapter 7 Protocols	84
Creating a Protocol	84
Adopting a Protocol.....	85
Advantages of Protocols	86
Protocols As Pseudo-Types	87
Dynamic Conformance Checking	88
Forward-Declaring Protocols	89
Summary.....	90
Chapter 8 Exceptions and Errors	91
Exception Handling	91
The NSError Class	92
Generating Exceptions.....	92
Catching Exceptions	93
Throwing Exceptions.....	94
Error Handling	96
The NSError Class.....	97
Error Domains	97

Capturing Errors.....	98
Custom Errors.....	99
Summary.....	100
Chapter 9 Blocks.....	102
Creating Blocks.....	102
Parameter-less Blocks.....	103
Using Blocks as Callbacks.....	103
Storing and Executing Blocks	105
Parameter-less Block Variables	106
Working with Variables.....	106
Blocks Are Closures.....	107
Mutable Block Variables.....	108
Defining Methods that Accept Blocks	109
Summary.....	110
Conclusion.....	111
iOS Succinctly	111

```

// main.m
#import <Foundation/Foundation.h>
#import "Person.h"
#import "Person+Relations.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *joe = [[Person alloc] init];
        joe.name = @"Joe";
        Person *bill = [[Person alloc] init];
        bill.name = @"Bill";
        Person *mary = [[Person alloc] init];
        mary.name = @"Mary";

        [joe sayHello];
        [joe addFriend:bill];
        [joe addFriend:mary];
        [joe sayHelloToFriends];
    }
    return 0;
}

```

And that's all there is to creating categories in Objective-C.

Protected Methods

To reiterate, *all* Objective-C methods are public—there is no language construct to mark them as either private or protected. Instead of using “true” protected methods, Objective-C programs can combine categories with the interface/implementation paradigm to achieve the same result.

The idea is simple: declare “protected” methods as a category in a separate header file. This gives subclasses the ability to “opt-in” to the protected methods while unrelated classes use the “public” header file as usual. For example, take a standard **Ship** interface:

```

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

@interface Ship : NSObject

- (void)shoot;

@end

```

As we’ve seen many times, this defines a public method called **shoot**. To declare a *protected* method, we need to create a **Ship** category in a dedicated header file:

```

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

```

```

@interface Ship(Protected)

- (void)prepareToShoot;

@end

```

Any classes that need access to the protected methods (namely, the superclass and any subclasses) can simply import **Ship_Protected.h**. For example, the **Ship** implementation should define a default behavior for the protected method:

```

// Ship.m
#import "Ship.h"
#import "Ship_Protected.h"

@implementation Ship {
    BOOL _gunIsReady;
}

- (void)shoot {
    if (!_gunIsReady) {
        [self prepareToShoot];
        _gunIsReady = YES;
    }
    NSLog(@"Firing!");
}

- (void)prepareToShoot {
    // Execute some private functionality.
    NSLog(@"Preparing the main weapon...");
}

@end

```

Note that if we hadn't imported **Ship_Protected.h**, this **prepareToShoot** implementation would be a private method, as discussed in the [Methods](#) chapter. Without a protected category, there would be no way for subclasses to access this method. Let's subclass the **Ship** to see how this works. We'll call it **ResearchShip**:

```

// ResearchShip.h
#import "Ship.h"

@interface ResearchShip : Ship

- (void)extendTelescope;

@end

```

This is a normal subclass interface—it should *not* import the protected header, as this would make the protected methods available to anyone that imports **ResearchShip.h**, which is

precisely what we're trying to avoid. Finally, the implementation for the subclass imports the protected methods and (optionally) overrides them:

```
// ResearchShip.m
#import "ResearchShip.h"
#import "Ship_Protected.h"

@implementation ResearchShip

- (void)extendTelescope {
    NSLog(@"Extending the telescope");
}

// Override protected method
- (void)prepareToShoot {
    NSLog(@"Oh shoot! We need to find some weapons!");
}

@end
```

To enforce the protected status of the methods in **Ship_Protected.h**, other classes aren't allowed to import it. They'll just import the normal "public" interfaces of the superclass and subclass:

```
// main.m
#import <Foundation/Foundation.h>
#import "Ship.h"
#import "ResearchShip.h"

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

        Ship *genericShip = [[Ship alloc] init];
        [genericShip shoot];

        Ship *discoveryOne = [[ResearchShip alloc] init];
        [discoveryOne shoot];

    }
    return 0;
}
```

Since neither **main.m**, **Ship.h**, nor **ResearchShip.h** import the protected methods, this code won't have access to them. Try adding a **[discoveryOne prepareToShoot]** method—it will throw a compiler error, since the **prepareToShoot** declaration is nowhere to be found.

To summarize, protected methods can be emulated by placing them in a dedicated header file and importing that header file into the implementation files that require access to the protected methods. No other files should import the protected header.

While the workflow presented here is a completely valid organizational tool, keep in mind that Objective-C was never meant to support protected methods. Think of this as an alternative way to structure an Objective-C method, rather than a direct replacement for C#/Simula-style protected methods. It's often better to look for another way to structure your classes rather than forcing your Objective-C code to act like a C# program.

Caveats

One of the biggest issues with categories is that you can't reliably override methods defined in categories for the same class. For example, if you defined an **addFriend:** class in **Person(Relations)** and later decided to change the **addFriend:** implementation via a **Person(Security)** category, there is no way for the runtime to know which method it should use since categories are, by definition, a flat organizational structure. For these kinds of cases, you need to revert to the traditional subclassing paradigm.

Also, it's important to note that a category can't add instance variables. This means you can't declare new properties in a category, as they could only be synthesized in the main implementation. Additionally, while a category technically does have access to its classes' instance variables, it's better practice to access them through their public interface to shield the category from potential changes in the main implementation file.

Extensions

Extensions (also called **class extensions**) are a special type of category that requires their methods to be defined in the *main* implementation block for the associated class, as opposed to an implementation defined in a category. This can be used to override publicly declared property attributes. For example, it is sometimes convenient to change a read-only property to a read-write property within a class' implementation. Consider the normal interface for a **Ship** class:

Included code sample: Extensions

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

@interface Ship : NSObject

@property (strong, readonly) Person *captain;

- (id)initWithCaptain:(Person *)captain;

@end
```

It's possible to override the **@property** definition inside of a class extension. This gives you the opportunity to re-declare the property as **readwrite** in the implementation file. Syntactically, an extension looks like an empty category declaration:

```

// Ship.m
#import "Ship.h"

// The class extension.
@interface Ship()

@property (strong, readwrite) Person *captain;

@end

// The standard implementation.
@implementation Ship

@synthesize captain = _captain;

- (id)initWithCaptain:(Person *)captain {
    self = [super init];
    if (self) {
        // This WILL work because of the extension.
        [self setCaptain:captain];
    }
    return self;
}

@end

```

~~Note the () appended to the class name after the @interface directive. This is what marks it as an extension rather than a normal interface or a category. Any properties or methods that appear in the extension *must* be declared in the main implementation block for the class. In this case, we aren't adding any new fields—we're overriding an existing one. But unlike categories, extensions can add extra instance variables to a class, which is why we're able to declare properties in a class extension but not a category.~~

~~Because we re-declared the captain property with a readwrite attribute, the initWithCaptain: method can use the setCaptain: accessor on itself. If you were to delete the extension, the property would return to its read-only status and the compiler would complain. Clients using the Ship class aren't supposed to import the implementation file, so the captain property will remain read-only.~~

```

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

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

        Person *heywood = [[Person alloc] init];
        heywood.name = @"Heywood";
        Ship *discoveryOne = [[Ship alloc] initWithCaptain:heywood];
    }
}

```

```

        NSLog(@"%@", [discoveryOne captain].name);

        Person *dave = [[Person alloc] init];
        dave.name = @"Dave";
        // This will NOT work because the property is still read-only.
        [discoveryOne setCaptain:dave];
    }
    return 0;
}

```

Private Methods

Another common use case for extending private methods. In the previous chapter, we saw how private methods were implemented by simply adding them anywhere in the implementation file. But, prior to Xcode 4.6, there was a different case. The canonical way to create a private method was to forward-declare it in the header file. Let's take a look at this by slightly altering the **Ship** header file example:

```

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

@interface Ship : NSObject

- (void)shoot;

@end

```

Next, we're going to recreate the example we used when we discussed private methods in the [Methods](#) chapter. Instead of simply adding the private **prepareToShoot** method to the implementation, we need to forward-declare it in a class extension.

```

// Ship.m
#import "Ship.h"

// The class extension.
@interface Ship()

- (void)prepareToShoot;

@end

// The rest of the implementation.
@implementation Ship {
    BOOL _gunIsReady;
}

- (void)shoot {
    if (!_gunIsReady) {
        [self prepareToShoot];
    }
}

```

```

        _gunIsReady = YES;
    }
    NSLog(@"Firing!");
}

- (void)prepareToShoot {
    // Execute some private functionality.
    NSLog(@"Preparing the main weapon...");
}

@end

```

The compiler ensures the extension methods are implemented in the main implementation block, which is why it functions as a forward-declaration. Yet because the extension is encapsulated in the implementation, the compiler wouldn't ever know about it, giving us another way to emulate private methods. Compilers save you this trouble, it's still important to understand how this has been a common way to leverage private methods in Objective-C programs until very recently.

APPROVED

Summary

This chapter covered two of the more unique concepts in the Objective-C programming language: categories and extensions. Categories are a way to extend the API of existing classes, and extensions are a way to add *required* methods to the API outside of the main interface file. Both of these were initially designed to ease the burden of maintaining large code bases.

The next chapter continues our journey through Objective-C's organizational structures. We'll learn how to define a protocol, which is an interface that can be implemented by a variety of classes.

Chapter 7 Protocols

In Objective-C, a **protocol** is a group of methods that can be implemented by any class. Protocols are essentially the same as interfaces in C#, and they both have similar goals. They can be used as a pseudo-data type, which is useful for making sure that a dynamically-typed object can respond to a certain set of messages. And, because any class can “adopt” a protocol, they can be used to represent a shared API between completely unrelated classes.

The official documentation discusses both an informal and a formal method for declaring protocols, but informal protocols are really just a unique use of categories and don’t provide nearly as many benefits as formal protocols. With this in mind, this chapter focuses solely on *formal* protocols.

Creating a Protocol

First, let’s take a look at how to declare a formal protocol. Create a new file in Xcode and select the Objective-C protocol icon under **Mac OS X > Cocoa**:

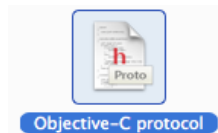


Figure 29: Xcode icon for protocol files

As usual, this will prompt you for a name. Our protocol will contain methods for calculating the coordinates of an object, so let’s call it **CoordinateSupport**:

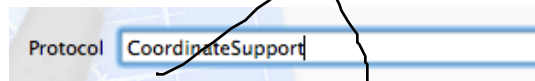


Figure 30: Naming the protocol

Click **Next** and choose the default location for the file. This will create an empty protocol that looks almost exactly like an interface:

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

@protocol CoordinateSupport <NSObject>

@end
```

Of course, instead of the **@interface** directive, it uses **@protocol**, followed by the protocol name. The **<NSObject>** syntax lets us incorporate another protocol into **CoordinateSupport**. In this case, we’re saying that **CoordinateSupport** also includes all of the methods declared in the **NSObject** [protocol](#) (not to be confused with the **NSObject** [class](#)).

Next, let's add a few methods and properties to the protocol. This works the same way as declaring methods and properties in an interface:

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

@protocol CoordinateSupport <NSObject>

@property double x;
@property double y;
@property double z;

- (NSArray *)arrayFromPosition;
- (double)magnitude;

@end
```

Adopting a Protocol

Any class that adopts this protocol is guaranteed to synthesize the **x**, **y**, and **z** properties and implement the **arrayFromPosition** and **magnitude** methods. While this doesn't say *how* they will be implemented, it does give you the opportunity to define a shared API for an arbitrary set of classes.

For example, if we want both **Ship** and **Person** to be able to respond to these properties and methods, we can tell them to adopt the protocol by placing it in angled brackets after the superclass declaration. Also note that, just like using another class, you need to import the protocol file before using it:

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

@interface Person : NSObject <CoordinateSupport>

@property (copy) NSString *name;
@property (strong) NSMutableSet *friends;

- (void)sayHello;
- (void)sayGoodbye;

@end
```

Now, in addition to the properties and methods defined in this interface, the **Person** class is guaranteed to respond to the API defined by **CoordinateSupport**. Xcode will warn you that the **Person** implementation is incomplete until you synthesize **x**, **y**, and **z**, and implement **arrayFromPosition** and **magnitude**:

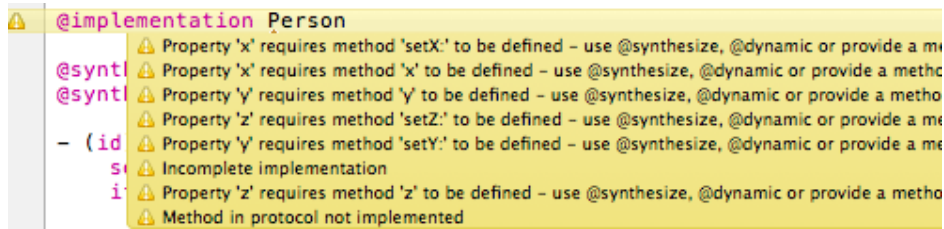


Figure 31: Incomplete implementation warning for **Person** <**CoordinateSupport**>

Likewise, a category can adopt a protocol by adding it after the category. For example, to tell the **Person** class to adopt the **CoordinateSupport** protocol in the **Relations** category, you would use the following line:

```
@interface Person(Relations) <CoordinateSupport>
```

And, if your class needs to adopt more than one protocol, you can separate them with commas:

```
@interface Person : NSObject <CoordinateSupport, SomeOtherProtocol>
```

Advantages of Protocols

Without protocols, we would have two options to ensure both **Ship** and **Person** implemented this shared API:

1. Re-declare the exact same properties and methods in both interfaces.
2. Define the API in an abstract superclass and define **Ship** and **Person** as subclasses.

Neither of these options are particularly appealing: the first is redundant and prone to human error, and the second is severely limiting, especially if they already inherit from different parent classes. It should be clear that protocols are much more flexible and reusable, as they shield the API from being dependent on any particular class.

The fact that *any* class can easily adopt a protocol makes it possible to define horizontal relationships on top of an existing class hierarchy:

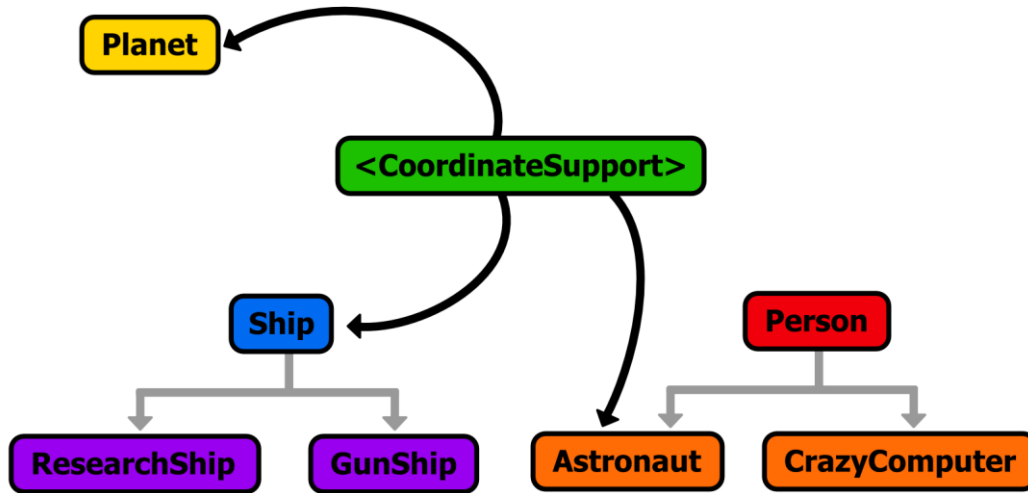


Figure 32: Linking unrelated classes using a protocol

Due to the flexible nature of protocols, the various iOS frameworks make good use of them. For example, user interface controls are often configured using the delegation design pattern, wherein a delegate object is responsible for reacting to user actions. Instead of encapsulating a delegate's responsibilities in an abstract class and forcing delegates to subclass it, iOS defines the necessary API for the delegate in a protocol. This way, it's incredibly easy for *any* object to act as the delegate object. We'll explore this in much more detail in the second half of this series, *iOS Succinctly*.

Protocols as Pseudo-Types

Protocols can be used as pseudo-data types. Instead of making sure a variable is an instance of a class, using a protocol as a type checking tool ensures that the variable always conforms to an arbitrary API. For example, the following **person** variable is guaranteed to implement the **CoordinateSupport** API.

```
Person <CoordinateSupport> *person = [[Person alloc] init];
```

Still, enforcing protocol adoption is often more useful when used with the **id** data type. This lets you assume certain methods and properties while completely disregarding the object's class.

And of course, the same syntax can be used with a method parameter. The following snippet adds a new **getDistanceFromObject:** method to the API whose parameter is required to conform to **CoordinateSupport** protocol:

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

@protocol CoordinateSupport <NSObject>
```

```

@property double x;
@property double y;
@property double z;

- (NSArray *)arrayFromPosition;
- (double)magnitude;
- (double)getDistanceFromObject:(id <CoordinateSupport>)theObject;

@end

```

Note that it's entirely possible to use a protocol in the same file as it is defined.

Dynamic Conformance Checking

In addition to the static type checking discussed in the last section, you can also use the **conformsToProtocol:** method defined by the **NSObject** protocol to dynamically check whether an object conforms to a protocol or not. This is useful for preventing errors when working with dynamic objects (objects typed as **id**).

The following example assumes the **Person** class adopts the **CoordinateSupport** protocol, while the **Ship** class does not. It uses a dynamically typed object called **mysteryObject** to store an instance of **Person**, and then uses **conformsToProtocol:** to check if it has coordinate support. If it does, it's safe to use the **x**, **y**, and **z** properties, as well as the other methods declared in the **CoordinateSupport** protocol:

```

// main.m
#import <Foundation/Foundation.h>
#import "Person.h"
#import "Ship.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        id mysteryObject = [[Person alloc] init];
        [mysteryObject setX:10.0];
        [mysteryObject setY:0.0];
        [mysteryObject setZ:7.5];

        // Uncomment next line to see the "else" portion of conditional.
        //mysteryObject = [[Ship alloc] init];

        if ([mysteryObject
            conformsToProtocol:@protocol(CoordinateSupport)]) {
            NSLog(@"Ok to assume coordinate support.");
            NSLog(@"The object is located at (%0.2f, %0.2f, %0.2f)",
                [mysteryObject x],
                [mysteryObject y],
                [mysteryObject z]);
        } else {
            NSLog(@"Error: Not safe to assume coordinate support.");
            NSLog(@"I have no idea where that object is...");
        }
    }
}

```

```

    }
    return 0;
}

```

If you uncomment the line that reassigns the **mysteryObject** to a **Ship** instance, the **conformsToProtocol:** method will return **NO**, and you won't be able to safely use the API defined by **CoordinateSupport**. If you're not sure what kind of object a variable will hold, this kind of dynamic protocol checking is important to prevent your program from crashing when you try to call a method that doesn't exist.

Also notice the new **@protocol()** directive. This works much like **@selector()**, except instead of a method name, it takes a protocol name. It returns a **Protocol** object, which can be passed to **conformsToProtocol:**, among other built-in methods. The protocol header file does *not* need to be imported for **@protocol()** to work.

Forward-Declaring Protocols

If you end up working with a lot of protocols, you'll eventually run into a situation where two protocols rely on one another. This circular relationship poses a problem for the compiler, since it cannot successfully import either of them without the other. For example, let's say we were trying to abstract out some GPS functionality into a **GPSSupport** protocol, but want to be able to convert between the "normal" coordinates of our existing **CoordinateSupport** and the coordinates used by **GPSSupport**. The **GPSSupport** protocol is pretty simple:

```

#import <Foundation/Foundation.h>
#import "CoordinateSupport.h"

@protocol GPSSupport <NSObject>

- (void)copyCoordinatesFromObject:(id <CoordinateSupport>)theObject;

@end

```

This doesn't pose any problems, that is, until we need to reference the **GPSSupport** protocol from **CoordinateSupport.h**:

```

#import <Foundation/Foundation.h>
#import "GPSSupport.h"

@protocol CoordinateSupport <NSObject>

@property double x;
@property double y;
@property double z;

```

```

- (NSArray *)arrayFromPosition;
- (double)magnitude;
- (double)getDistanceFromObject:(id <CoordinateSupport>)theObject;

- (void)copyGPSCoordinatesFromObject:(id <GPSSupport>)theObject;

@end

```

Now, the **CoordinateSupport.h** file requires the **GPSSupport.h** file to compile correctly, and vice versa. It's a chicken-or-the-egg kind of problem, and the compiler will not like it very much:

 Cannot find protocol declaration for 'CoordinateSupport'

Figure 33: Compiler error caused by circular protocol references

Resolving the recursive relationship is simple. All you need to do is forward-declare one of the protocols instead of trying to import it directly:

```

#import <Foundation/Foundation.h>

@protocol CoordinateSupport;

@protocol GPSSupport <NSObject>

- (void)copyCoordinatesFromObject:(id <CoordinateSupport>)theObject;

@end

```

All **@protocol CoordinateSupport;** says is that **CoordinateSupport** is indeed a protocol and the compiler can assume it exists without importing it. Note the semicolon at the end of the statement. This could be done in either of the two protocols; the point is to remove the circular reference. The compiler doesn't care how you do it.

Summary

Protocols are an incredibly powerful feature of Objective-C. They let you capture relationships between arbitrary classes when it's not feasible to connect them with a common parent class. We'll utilize several built-in protocols in **iOS Succinctly**, as many of the core functions of an iPhone or iPad app are defined as protocols.

The next chapter introduces exceptions and errors, two very important tools for managing the problems that inevitably arise while writing Objective-C programs.

Chapter 8 Exceptions and Errors

In Objective-C, there are two types of errors that can occur while a program is running. *Unexpected* errors are “serious” programming errors that typically cause your program to exit prematurely. These are called **exceptions**, since they represent an exceptional condition in your program. On the other hand, *expected* errors occur naturally in the course of a program’s execution and can be used to determine the success of an operation. These are referred to as **errors**.

You can also approach the distinction between exceptions and errors as a difference in their target audiences. In general, exceptions are used to inform the *programmer* about something that went wrong, while errors are used to inform the *user* that a requested action could not be completed.

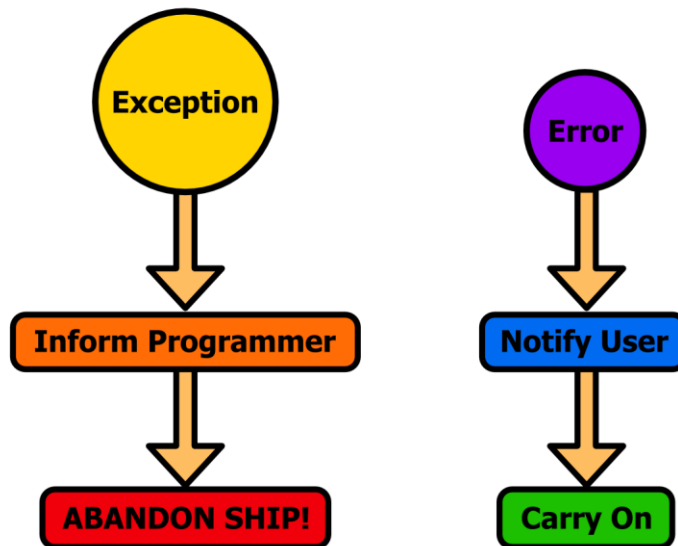


Figure 34: Control flow for exceptions and errors

For example, trying to access an array index that doesn’t exist is an exception (a programmer error), while failing to open a file is an error (a user error). In the former case, something likely went very wrong in the flow of your program and it should probably shut down soon after the exception. In the latter, you would want to tell the user that the file couldn’t be opened and possibly ask to retry the action, but there is no reason your program wouldn’t be able to keep running after the error.

Exception Handling

The main benefit to Objective-C’s exception handling capabilities is the ability to separate the handling of errors from the detection of errors. When a portion of code encounters an exception, it can “throw” it to the nearest error handling block, which can “catch” specific exceptions and handle them appropriately. The fact that exceptions can be thrown from arbitrary locations

eliminates the need to constantly check for success or failure messages from each function involved in a particular task.

The `@try`, `@catch()`, and `@finally` compiler directives are used to catch and handle exceptions, and the `@throw` directive is used to detect them. If you've worked with exceptions in C#, these exception handling constructs should be familiar to you.

It's important to note that in Objective-C, exceptions are relatively slow. As a result, their use should be limited to catching serious programming errors—not for basic control flow. If you're trying to determine what to do based on an *expected* error (e.g., failing to load a file), please refer to the [Error Handling](#) section.

The NSError Class

Exceptions are represented as instances of the **NSError** class or a subclass thereof. This is a convenient way to encapsulate all the necessary information associated with an exception. The three properties that constitute an exception are described as follows:

- **name** – An instance of **NSString** that uniquely identifies the exception.
- **reason** – An instance of **NSString** containing a human-readable description of the exception.
- **userInfo** – An instance of **NSDictionary** that contains application-specific information related to the exception.

The Foundation framework defines several constants that define the “standard” [exception names](#). These strings can be used to check what type of exception was caught.

You can also use the `initWithName:reason:userInfo:` initialization method to create new exception objects with your own values. Custom exception objects can be caught and thrown using the same methods covered in the upcoming sections.

Generating Exceptions

Let's start by taking a look at the default exception-handling behavior of a program. The `objectAtIndex:` method of **NSArray** is defined to throw an **NSRangeException** (a subclass of **NSError**) when you try to access an index that doesn't exist. So, if you request the 10th item of an array that has only three elements, you'll have yourself an exception to experiment with:

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

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

        NSArray *crew = [NSArray arrayWithObjects:
                        @"Dave",
                        @"Heywood",
                        @"Frank", nil];

        // This will throw an exception.
```

```

        NSLog(@"%@", [crew objectAtIndex:10]);
    }
    return 0;
}

```

When it encounters an uncaught exception, Xcode halts the program and points you to the line that caused the problem.

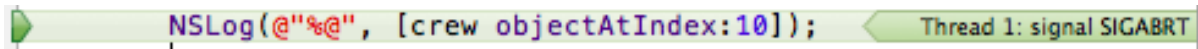


Figure 35: Aborting a program due to an uncaught exception

Next, we'll learn how to catch exceptions and prevent the program from terminating.

Catching Exceptions

To handle an exception, any code that *may* result in an exception should be placed in a **@try** block. Then, you can catch specific exceptions using the **@catch()** directive. If you need to execute any housekeeping code, you can optionally place it in a **@finally** block. The following example shows all three of these exception-handling directives:

```

@try {
    NSLog(@"%@", [crew objectAtIndex:10]);
}
@catch (NSException *exception) {
    NSLog(@"Caught an exception");
    // We'll just silently ignore the exception.
}
@finally {
    NSLog(@"Cleaning up");
}

```

This should output the following in your Xcode console:

```

Caught an exception!
Name: NSRangeException
Reason: *** -[__NSArrayI objectAtIndex:]: index 10 beyond bounds [0 .. 2]
Cleaning up

```

When the program encounters the **[crew objectAtIndex:10]** message, it throws an **NSRangeException**, which is caught in the **@catch()** directive. Inside of the **@catch()** block is where the exception is actually handled. In this case, we just display a descriptive error message, but in most cases, you'll probably want to write some code to take care of the problem.

When an exception is encountered in the **@try** block, the program jumps to the corresponding **@catch()** block, which means any code *after* the exception occurred won't be executed. This

poses a problem if the `@try` block needs some cleaning up (e.g., if it opened a file, that file needs to be closed). The `@finally` block solves this problem, since it is *guaranteed* to be executed regardless of whether an exception occurred. This makes it the perfect place to tie up any loose ends from the `@try` block.

Catching Custom Objects

The parentheses after the `@catch()` directive lets you define what type of exception you're trying to catch. In this case, it's an `NSException`, which is the standard exception class. But, an exception can actually be *any* class—not just an `NSException`. For example, the following `@catch()` directive will handle a generic object:

```
@catch (id genericException)
```

We'll learn how to throw instances of `NSException` as well as generic objects in the next section.

Throwing Exceptions

When you detect an exceptional condition in your code, you create an instance of `NSException` and populate it with the relevant information. Then, you throw it using the aptly named `@throw` directive, prompting the nearest `@try/@catch` block to handle it.

For example, the following example defines a function for generating random numbers between a specified interval. If the caller passes an invalid interval, the function throws a custom error.

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

int generateRandomInteger(int minimum, int maximum) {
    if (minimum >= maximum) {
        // Create the exception.
        NSException *exception = [NSException
            exceptionWithName:@"RandomNumberIntervalException"
            reason:@"*** generateRandomInteger(): "
                "maximum parameter not greater than minimum parameter"
            userInfo:nil];

        // Throw the exception.
        @throw exception;
    }
    // Return a random integer.
    return arc4random_uniform((maximum - minimum) + 1) + minimum;
}

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

        int result = 0;
        @try {
            result = generateRandomInteger(0, 10);
        }
    }
}
```