Technische Universität München
Chair for Applied Software Engineering **2011**

TrafficWatch Tutorial

A Hands-on Approach to Cocoa Touch

Tobias Haeberle

May 25, 2011

Contents

1. Introduction

This document will guide you step by step through the process of developing your first fully functional and useful iPhone application. You will learn many of the basic concepts that are commonly used when writing Cocoa applications. This tutorial states every line of code your app will need in order to run successfully. This means, you can simply copy and paste the code to complete the tutorial. However, you are strongly advised to read every chapter carefully. Much effort has been made to explain the ideas behind the code and provide you with useful tips and additional references for further reading.

The application written in this guide will present recent incidents on German roads using a publicly available RSS feed in the XML format.

If you are new to Cocoa it is quite likely that you will be a little confused at first. But rest assured: You *will* love Cocoa soon!

A good first introduction to some basic Cocoa concepts and language specifics can be found online at `http://www.cocoadevcentral.com/d/learn_objectivec/`. Novice readers are encouraged to read this introduction prior to completing this tutorial.

More advanced users might be a bit bored by the pace of this tutorial. In the beginning of each chapter there is a brief summery of its contents. You can try to implement the code yourself without first reading the chapter and then later compare your results to the example implementation given here. Moreover, many of the more advanced chapters also include some ideas for additional features you can implement yourself without giving you the complete code. Last but not least, you are highly encouraged to extend the app even further with you own ideas.

Online Coding Support

If you find errors in this document or if you have trouble following the instructions, please feel free to let us know. Feedback is greatly appreciated! The preferred way to request help is to post your question in our Coding Support Forum located at

`https://forumbruegge.in.tum.de/ios2011/`.

We will try to respond within 24hrs.

Garching, May 15th, 2011

2. Getting started

Summary Create a new view based application called TrafficWatch and add a table view to its main view controller displaying a single cell with the title `Hello World`!

2.1. Xcode

Xcode is Apple's main software development tool. It is is a highly customizable integrated development environment (IDE) with many features for creating an efficient and effective working environment.

The Xcode application is located at `/Developer/Applications`. You can launch Xcode either by directly double clicking its application icon, by choosing it from the Dock, or by typing in `xcode` in the Spotlight search field which resides in the top right corner of the screen.

2.2. Create a new project

After Xcode has launched, a start up screen is presented. Choose Create a new Xcode project! Alternatively, you can access the File menu and choose New - New Project.... When the panel appears, on the left side in the iOS section click on Application and choose View based Application. Actually, the Navigation based application template would be better suited for our needs, however, for the sake of practice, we will create the view hierarchy ourselves.

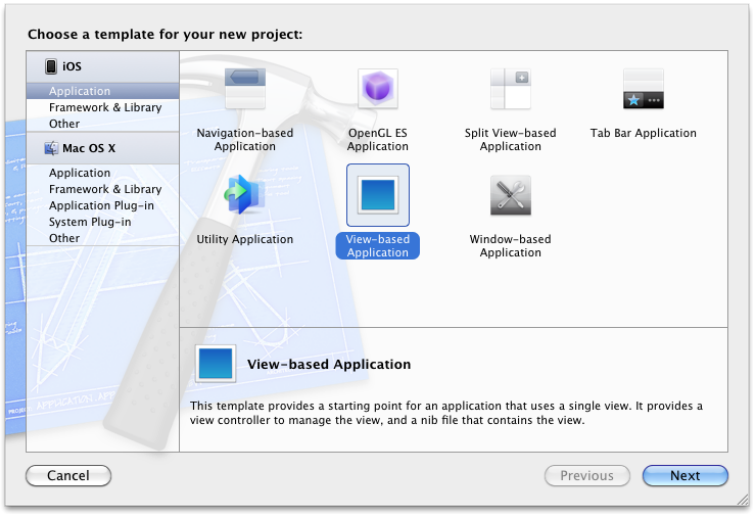


Figure 2.1.: Create a New Project

In the next screen enter TrafficWatch as the product name and choose an appropriate Company Identifier

(or accept the default identifier). After clicking Next your are prompted to save the project on disk. Pick a location of your choice and make sure to uncheck the Create local git repository for your project checkbox.

Xcode creates a new project skeleton, consisting of an *AppDelegate*, a *ViewController* and a *XIB files*. Throughout this tutorial you will extend this skeleton into the source for a complete application.

Looking at the new project, you will see an outline view on the left side of the window. Each item in the outline view represents one type of information that might be useful to a programmer. On top of this view resides a tab bar, where you can specify the type of items you wish to inspect: files, symbols, search results, compiler warnings, messages from the debugger, breakpoints, and logs. You will start by creating and editing files. Therefore, click on the folder icon on the left to activate the Project navigator. If necessary, click on the small triangle to expand the contents of your project. You will see four folders: TrafficWatch, Supporting Files, Frameworks, and Products.

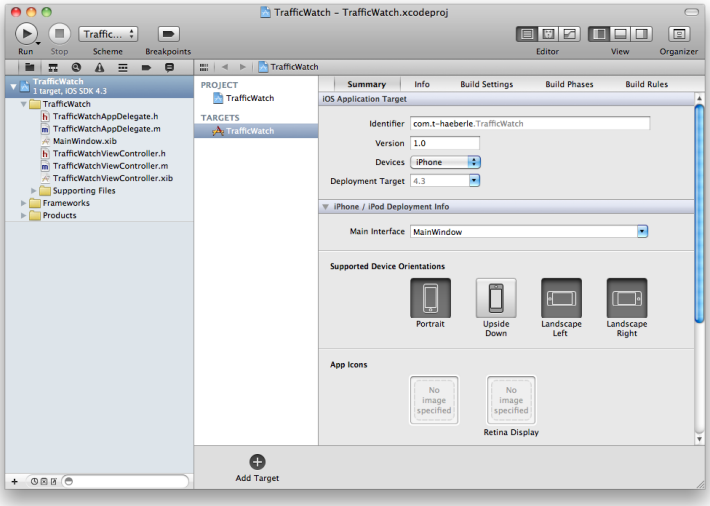


Figure 2.2.: The Project Window

2.3. Creating User Interfaces

User interface elements are usually stored in a *XIB file*. This file can be edited within Xcode. When you start a new project Xcode will already create a few empty XIB files for you. XIB files are stored alongside the class files of your project in the TrafficWatch folder. There, you will find the file RootViewController.xib. Click it to open its editor.

2.3.1. The Interface Editor

When you select a XIB file in Xcode you are presented the Interface editor. On its left side is bar with icons which represent the objects available to your view(s). Click on the small button on the bottom to expand the bar to a more verbose list:

File's Owner: The object that is responsible for loading the XIB file.

First Responder: The object that first responds to actions when the XIB file is loaded. For example a text field.

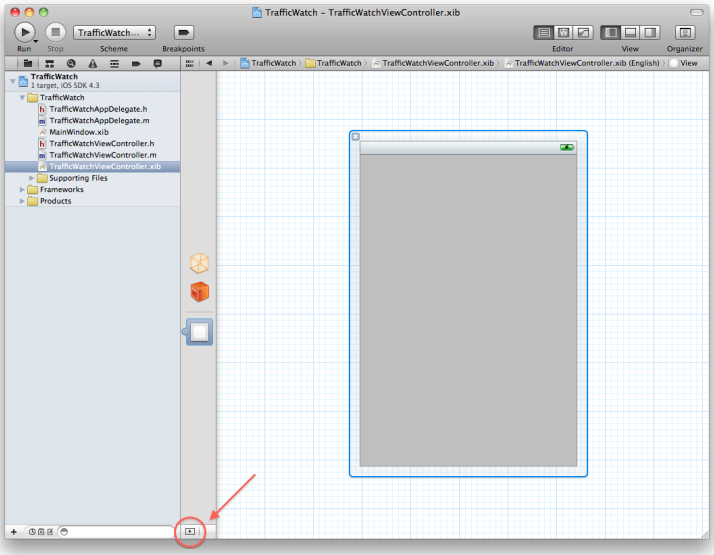


Figure 2.3.: TrafficWatchViewController.xib opened in the Interface editor. Use the small button on the left bottom to expand the objects view.

View: An **`UIView`** object that will contain the user interface objects.

2.3.2. Add a table view to your view

Xcode4 introduced a new all-in-one window concept, i. e. all the tools you will need are always contained within the main window. For this purpose the main window is divided into several areas. The main editor area, a left sidebar, a right sidebar, and a bottom bar. Use the switch button on the top right corner of the main window to show or hide these areas.

Next, we will add a table view to your custom view. Activate the right sidebar, in order to show the Object browser. Type in `table view` in the search field and then drag the table view object on top of the blank view in the editor. Make sure that the table view matches the size of the underlying view as shown in Figure2.5.

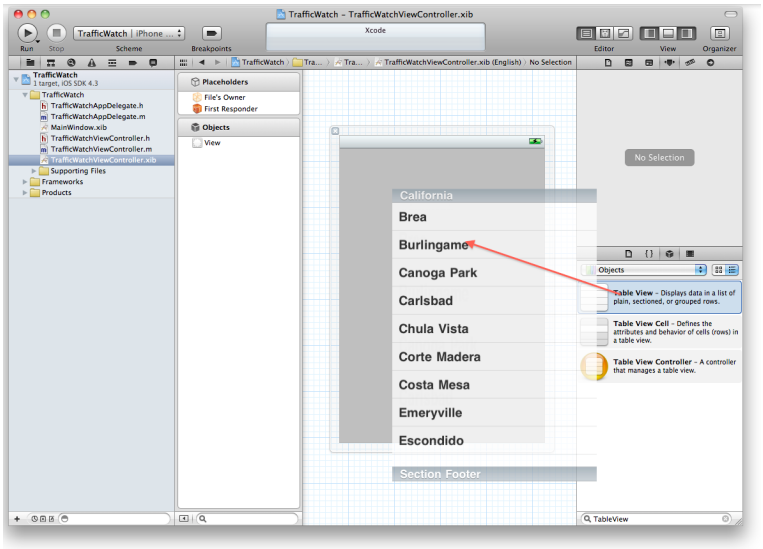


Figure 2.4.: Adding a Table View

Save the XIB file (menu File: Save).

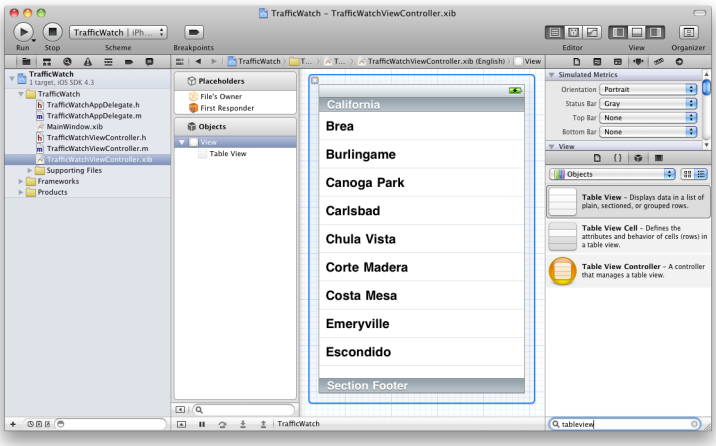


Figure 2.5.: View with a Table View

2.4. Test your application

Choose TrafficWatch - iPhone 4.3 Simulator from the Scheme popup menu and click on Run. Xcode will now compile your application, install it in the *iPhone Simulator*, and start the simulator.

You should see an empty table view. Click on the "Home Button" on the iPhone to quit your application and return to Xcode.

Figure 2.6.: Application Running in iPhone Simulator

2.5. Implement the user interface

In an iPhone application the user interface is usually managed by subclasses of **`UIViewController`**. Xcode has already created a new view controller for us: **`TrafficWatchViewController`**. We will now edit this class to display an empty instance of **`UITableView`**, that will later show the contents of our application.

2.5.1. Define IBOutlets for user interface objects

Click on the TrafficViewController.h item. The editor should now display the empty interface skeleton of the class:

```
1 #import <UIKit/UIKit.h>
2
3 @interface TrafficWatchViewController : UIViewController {
4
5 }
6
7 @end
```

As mentioned above, this view controller's job is to manage an **`UITableView`** object. Therefore, we will add an instance variable **`tableView`** of the type **`UITableView`** and define a *property* for this instance variable. *Properties* are a convenient way to declare setter and getter methods for instance variables.

Change the code to look like this:

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

@interface TrafficWatchViewController : UIViewController {
    UITableView *tableView;
}
@property (nonatomic, retain) IBOutlet UITableView *tableView;
@end
```

With `@property` you only declare that you will implement the appropriate setter and getter methods for the instance variable `tableView` (you will learn more about properties throughout this tutorial). To actually implement them, click on the `TrafficWatchViewController.m` file. Directly under the line starting with `@implementation` add:

```
@synthesize tableView;
```

This line tells the compiler to implement the setter and getter method as declared in the interface.

Because our property is set to **`retain`**, we need to make sure that the object is released when the view controller is removed from memory. Look for the **`dealloc`** method:

```
- (void)dealloc {
    [super dealloc];
}
```

And change it to:

```
- (void)dealloc {
    self.tableView = nil;
    [super dealloc];
}
```

Note, when writing your own dealloc method, always make sure to issue the call to `super` after you released your own member objects. Otherwise, the app will likely crash unexpectedly.

2.5.2. Implement the UITableViewDataSource protocol

The view controller is supposed to provide content for the table view. Every **`UITableView`** object has a reference to an object called **`dataSource`**. This object must conform to a *protocol* named **`UITableViewDataSource`**. The protocol declares several methods the table view will call in order to gather its content. In the `TrafficWatchViewController.m` file add the following two methods somewhere in the `@implementation` block.

```
- (NSInteger)tableView:(UITableView *)aTableView numberOfRowsInSection:(NSInteger)
    section
{
     return 1;
}

- (UITableViewCell *)tableView:(UITableView *)aTableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *MyIdentifier = @"MyIdentifier";

    UITableViewCell *cell = [aTableView dequeueReusableCellWithIdentifier:
        MyIdentifier];
```

```
12      if (cell == nil) {
13          cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
14                                        reuseIdentifier:MyIdentifier];
15          [cell autorelease];
16      }
17
18      cell.textLabel.text = @"Hello World!";
19      return cell;
20  }
```

The first method tells the table view how many rows it contains. The second methods provides a **`UITableViewCell`** object for each row. For now, we only display a single cell.

In order to let other objects know, that this view controller can function as a data source, open the TrafficWatchViewController.h file and edit the interface declaration as follows:

```
1  @interface TrafficWatchViewController : UIViewController <UITableViewDataSource> {
```

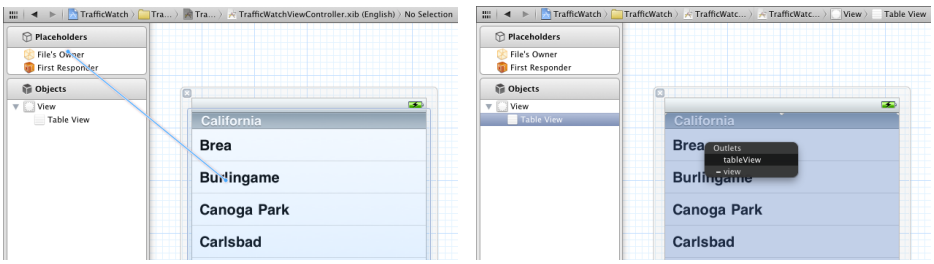
Our view controller is now basically ready. Is has an instance variable to contain an **`UITableView`** object and conforms to the **`UITableViewDataSource`** protocol in order to provide content for its table view.

Next, we need to connect the view controller to the previously created table view in the xib file.

2.6. Connect user interface elements

Choose the TrafficWatchViewController.xib file from the project navigator.

Now, we will connect the outlet we defined in the previous chapter to the actual table view in the xib file. Hold the `ctrl` key on the keyboard and drag a line from File's Owner to the table view. When you release the mouse button while on top of the table view, a small window appears. Choose tableView to establish the connection (see Figure 2.7).



(a) (b)

Figure 2.7.: Dragging and Establishing a Connection

Next, `ctrl`-click on the table view and drag a connection to File's owner. Choose dataSource as the outlet.

Save your xib file and return to Xcode. Build and run your application again. You should now see a single cell in the table view displaying "Hello World!".

3. Make your application useful

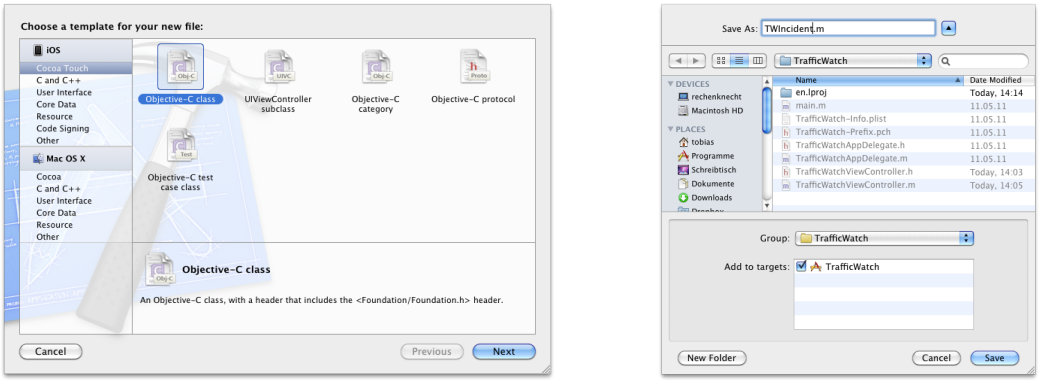
Summary Add a new class to your project called `TWIncident`. Add instance variables and properties named `title`, `summary`, and `weblink`.

Application development with Cocoa Touch should always follow the MVC-design pattern (Model-View-Controller). So far we have implemented a view and a controller for your iPhone app. In this chapter we are going to define a model.

3.1. Create a new class

Our application is supposed to display traffic incidents on roads in Germany. Therefore we will first create a model class named **`TWIncident`**, that will represent such an incident ("TW" stands for "TrafficWatch" and we will use it as a prefix for all our project's classes. See the Coding Conventions Guide for class naming conventions).

In Xcode in the outline view click on the folder TrafficWatch to select it. Then choose New File... from the File menu. On the left side make sure Cocoa Touch is selected and then choose the Objective-C class template. Click Next, enter NSObject as the superclass, click Next again and enter TWIncident.m as the filename (see Figure 3.1).



(a) (b)

Figure 3.1.: Create a New Class

Xcode creates two files for you: an interface declaration file TWIncident.h and the implementation file TWIncident.m.

3.1.1. Define the TWIncident class interface

Select the TWIncident.h file. It should look something like this:

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


@interface TWIncident : NSObject {

}

@end
```

Declare instance variables

An incident is described by the following attributes: *title*, *summary*, and *weblink*.

Each of them will be represented by an **`NSString`** object. Edit the interface as follows:

```
@interface TWIncident : NSObject {
 @private
    NSString *title;
    NSString *summary;
    NSString *weblink;
}
```

Properties

Because Objective-C is an object oriented language, instance variables should only be accessed via designated accessor (and setter) methods. This means you have to declare these methods and then write a suitable implementation. Especially when dealing with large and complex classes, this can be a painful task. Fortunately Objective-C 2.0, which is supported by the iPhone, features the already mentioned properties.

Add three property declarations to your class:

```
@interface TWIncident : NSObject {
 @private
    NSString *title;
    NSString *summary;
    NSString *weblink;
}
@property (nonatomic, copy) NSString *title;
@property (nonatomic, copy) NSString *weblink;
@property (nonatomic, copy) NSString *summary;
```

A property declaration generally comprises the following parts:

`@property ( attributes ) type name`

Please consult *The Objective-C 2.0 Programming Language*[1] document for a detailed explanation. For now, you can go with the following rule of thumb:

> Except you know what you are doing, use the **`nonatomic`** attribute. For **`NSString`** objects always use **`copy`**, for most other objects use **`retain`**.[2]

The `@property` statements only declare the properties. The actual implementation has to be triggered in the implementation file.

[1] `http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/ObjectiveC/`
[2] There exist many exceptions to this simple rule, but **`retain`** is almost always a good idea.

3.1.2. Implement the TWIncident class

Open the `TWIncident.m` file and tell the compiler to implement the properties as declared in the interface:

```
#import "TWIncident.h"


@implementation TWIncident

@synthesize title;
@synthesize weblink;
@synthesize summary;

@end
```

3.1.3. The dealloc method

The iPhone OS does not support garbage collection yet. Therefore, it is your job to keep track of your application's memory usage.

Please refer to appendix A for a brief introduction to memory management in iOS or read the *Memory Management Programming Guide for Cocoa*[3] for a complete overview of the topic.

For the scope of this tutorial, we will assume, that you are already familiar with the basics of memory management in Cocoa.

When an object's retain count reaches zero, its **`dealloc`** method is called to remove it from memory. Although, you *must never* call **`dealloc`** directly (this would contradict the retain/release pattern), you must overwrite the default **`dealloc`** method to release objects your class has previously retained.

The **`copy`** attribute of **`TWIncident`**'s properties implies, that whenever such a property is set to a new string, a copy of that string is created and stored in memory. Since **`TWIncident`** is the creator of that copy, it is its obligation to release it, if it is no longer needed. In other words, when a **`TWIncident`** object is removed from memory, it has to release its member variables.

Add the following method to `TWIncident.m`:

```
@implementation TWIncident

@synthesize title;
@synthesize weblink;
@synthesize summary;

- (void)dealloc
{
    self.title   = nil;
    self.weblink = nil;
    self.summary = nil;
    [super dealloc];
}

@end
```

[3] `http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/MemoryMgmt/`

The so called *dot-notation* `self.title = nil` is absolutely equivalent to calling the setter method `[self setTitle:nil]` and providing `nil` as an argument.

Trick question

Assuming that the setter methods are implemented as shown in the Sidenote below, convince yourself, that setting a variable to `nil` using its setter actually releases this instance variable!

Sidenote: A closer look at properties

In order to better understand why properties are useful and how they work, it is a good idea to understand what Cocoa developers had to do before Objective-C 2.0. For example, consider this simplified incident class with only a single instance variable called `title`:

```
@interface TWSimpleIncident : NSObject {
 @private
    NSString *title;
}

- (void)setTitle:(NSString *)newTitle;   // setter
- (NSString *)title;                     // getter

@end
```

Note, that lines 6 and 7 define two methods, a setter and a getter for the instance variable. Before properties, the developer had to implement these methods himself:

```
#import "TWSimpleIncident.h"

@implementation TWSimpleIncident

- (void)setTitle:(NSString *)newTitle   // setter
{
    if (newTitle != title) {
        [title release];
        title = [newTitle copy];
    }
}

- (NSString *)title                     // getter
{
    return title;
}

@end
```

The setter method first checks if the given string is actually a different object, it than releases its own reference to the old title and assigns the copy of the new title to the *ivar* (instance variable). The getter method simply returns the ivar to the caller.
With properties you can now declare and implement your setter and getter methods with two simple lines: `@property (nonatomic, copy)NSString *title;` and `@synthesize title;`
Also, in former days the header file wont give you any information on whether the setter method retains, copies, or assigns the new value. With properties, the declaration is a far more meaningful.

3.1.4. The init method

The **init** method is called, whenever a new object of your class is created. When defining a model, you should always make sure that **init** sets all member variables to an appropriate standard value. For **TWIncident** such an **init** method could read (don't add this code yet!):

```
- (id)init
{
    self = [super init];
    if (self != nil) {
        self.title = @"Unknown Incident";
        self.weblink = @"Not available.";
        self.summary = @"Not available.";
    }
    return self;
}
```

Sometimes you want to change the standard values of an initializer. In this case you can define your own *designated initializer*. Add the following two methods to the implementation block of TWIncident.m:

```
- (id)initWithTitle:(NSString *)aTitle
            weblink:(NSString *)aWeblink
            summary:(NSString *)aSummary
{
    self = [super init];
    if (self != nil) {
        self.title = aTitle;
        self.weblink = aWeblink;
        self.summary = aSummary;
    }
    return self;
}

- (id)init
{
    // the standard initializer must call the designated initializer
    return [self initWithTitle:@"Unknown Incident"
                       weblink:@"Not available."
                       summary:@"Not available."];
}
```

Finally, let other objects know that your class has a designated initializer by adding its signature to TWIncident.h:

```
@interface TWIncident : NSObject {
 @private
    NSString *title;
    NSString *summary;
    NSString *weblink;
}
@property (nonatomic, copy) NSString *title;
@property (nonatomic, copy) NSString *weblink;
@property (nonatomic, copy) NSString *summary;

- (id)initWithTitle:(NSString *)aTitle
            weblink:(NSString *)aWeblink
            summary:(NSString *)aSummary;
@end
```

3.1.5. The description method

Every subclass of **NSObject** inherits a **description** method:

```
- (NSString *)description;
```

The default implementation returns a **NSString** object composed of the class' name and its address in memory. Subclasses of **NSObject** should always overwrite this method to give a more meaningful description of the object's content, status or purpose. **NSArray** for example returns a list of objects it contains.

Here, we will simply return the values of **TWIncident**'s member variables. Add the following method to the implementation block of TWIncident.m:

```
- (NSString *)description
{
    NSString *theDescription = [[NSString alloc] initWithFormat:
                                    @"< %@: title = %@, weblink = %@, summary = %@
                                        >",
                                    NSStringFromClass([self class]), self.title,
                                    self.weblink, self.summary];
    return [theDescription autorelease];
}
```

3.2. Test the new class

We will now modify our view controller to actually create an instance of **TWIncident** and display that object in the table view.

3.2.1. Import TWIncident.h

First, open TrafficWatchViewController.m. In order to create or use objects of **TWIncident** we must import its header file. Edit the top of the file:

```
#import "TrafficWatchViewController.h"
#import "TWIncident.h"

@implementation TrafficWatchViewController
// implementation continues...
```

Next, edit the `tableView:cellForRowAtIndexPath:` method to create an instance of **TWIncident**:

```
- (UITableViewCell *)tableView:(UITableView *)aTableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *MyIdentifier = @"MyIdentifier";

    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                        reuseIdentifier:MyIdentifier];
        [cell autorelease];
    }

```

```
12      TWIncident *anIncident = [[TWIncident alloc] init];
13      NSLog(@"displaying incident: %@", anIncident);
14      cell.textLabel.text = anIncident.title;
15      [anIncident release];
16
17      return cell;
18  }
```

Note, that this is, of course, a rather stupid way to do it. Next week we will think of a better one. For now, build your project. In case of errors, try to fix them. After that, click on view switch to activate the bottom view. Now run your project. When the table view is loaded, you should see a printout of the standard **`TWIncident`** object in the console (this is what **`NSLog()`** does, see Figure 3.2). The iPhone simulator should correctly display a single "Unknown Incident" cell in a table view.

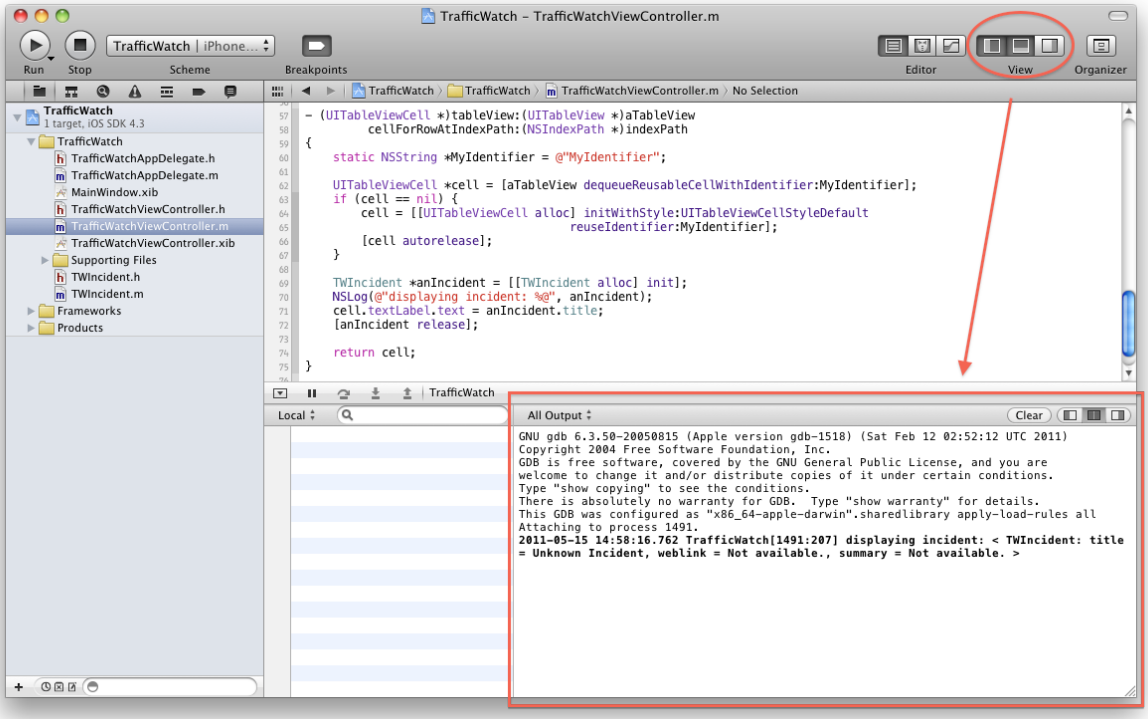


Figure 3.2.: **`NSLog()`** prints console messages.

4. XML Parsing

Summary Use an instance of **NSXMLParser** to parse the XML file located at `http://www.freiefahrt.info/lmst.de_DE.xml`. For each incident, create a **TWIncident** object and print its contents to the console.

4.1. The XML file

The RSS feed which contains all recent incidents on German streets is located at `http://www.freiefahrt.info/lmst.de_DE.xml`. It is comprised of `<entry>` elements, each describing an individual incident.

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
   <updated>2009-04-18T22:00:07.168Z</updated>
   <link href="http://www.freiefahrt.info/lmst.de_DE.xml" rel="self"/>
   <author>
      <name>OneStepAhead AG (www.onestepahead.de)</name>
      <email>info@onestepahead.de</email>
   </author>
   <title>Aktuelle Verkehrslage fuer Deutschland</title>
   <id>http://www.onestepahead.de/</id>
   <entry>
      <updated>2009-04-18T21:58:35Z</updated>
      <expires>2009-04-18T23:58:00Z</expires>
      <title>A4 : Koeln - Olpe</title>
      <id>http://www.freiefahrt.info/?id=83FBBF15-B187-452a-B66E-E3B4FCC408D2...</id>
      <link type="text/html" rel="alternate"
            href="http://www.freiefahrt.info/?id=83FBBF...">http://www.freiefahrt.info/?id=83FBBF...</link>
      <summary>Zwischen AS Wenden und AS Krombach in beiden Richtungen Achtung, Ihnen kommt ein
               Falschfahrer entgegen! Nicht ueberholen! Vorsicht auf beiden Richtungsfahrbahnen.
               Wir melden, wenn die Gefahr vorueber ist.</summary>
      <rights>(c) OneStepAhead</rights>
      <content type="xhtml">
         <div xmlns="http://www.w3.org/1999/xhtml" xmlns:xs="http://www.w3.org/2001/XMLSchema">
            <div>
               <img alt="A4" width="32" src="http://mosaic.onestepahead.de/img/roadsigns/DE/A4.png"/>
               <img alt="83FBBF15-B187-452a-B66E-E3B4FCC408D2"
                    src="http://mosaic.onestepahead.de/img/alertc/ec-101.gif"/>
            </div>A4: Koeln - Olpe<br/>Zwischen AS Wenden und AS Krombach in beiden Richtungen Achtung,
                Ihnen kommt ein Falschfahrer entgegen! Nicht ueberholen! Vorsicht auf beiden
                Richtungsfahrbahnen. Wir melden, wenn die Gefahr vorueber ist.
          </div>
      </content>
   </entry>
   <entry>
     ...
   </entry>
</feed>
```

We will run this feed through a **NSXMLParser** object. Whenever the parser recognizes an `<entry>` section, we will create a new **TWIncident** object and store some of the values from the XML file in the object's properties.

4.1.1. Naming conventions

In a XML file an *element* looks like this:

```
<elementName attributeName1="attribueValue1">contentOfElement</elementName>
```

Where `contentOfElement` can either be a simple string or be comprised of one or many other elements.

4.2. Create a parser object

In your Xcode project create a new subclass of **`NSObject`** named **`TWIncidentsParser`**. Open the TWIncidentsParser.h file and add two member variables and a method. Also make sure make your class conform to the **`NSXMLParserDelegate`** protocol.

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

@class TWIncident;

@interface TWIncidentsParser : NSObject <NSXMLParserDelegate> {
@private
    TWIncident *currentIncidentObject;
    NSMutableString *contentOfCurrentIncidentProperty;
}

- (void)parseFileAtURL:(NSURL *)URL parseError:(NSError **)error;

@end
```

`currentIncidentObject` will represent the currently parsed `<entry>` section. The mutable string **`contentOfCurrentIncidentProperty`** will be used to temporarily store a value of the entry until the parser has read it it completely. Don't worry if you don't understand this yet, it should become clear as soon as you see the actual implementation.

The **`@class TWIncident;`** statement tells the compiler that a **`TWIncident`** class exists. If you removed this line, you would get a warning during build.

4.2.1. Define private properties

This time, we don't want our member variables to be accesible from the outside world. Still, we want to benefit from the convenience of properties to access them. Editing the TWIncidentsParser.m file add the following "private" interface to your class. Note, that you need to import the TWIncident.h file here, in order to use this class.

```
#import "TWIncidentsParser.h"
#import "TWIncident.h"


// these are private properties
@interface TWIncidentsParser ()
@property (nonatomic, retain) TWIncident *currentIncidentObject;
@property (nonatomic, retain) NSMutableString *contentOfCurrentIncidentProperty;
@end

[implementation follows]
```

Next, make sure that the accessor and setter methods are synthesized and release the objects in the `dealloc` method:

```
#import "TWIncidentsParser.h"
#import "TWIncident.h"


// these are private properties
@interface TWIncidentsParser ()
@property (nonatomic, retain) TWIncident *currentIncidentObject;
@property (nonatomic, retain) NSMutableString *contentOfCurrentIncidentProperty;
@end

@implementation TWIncidentsParser
@synthesize currentIncidentObject, contentOfCurrentIncidentProperty;
- (void) dealloc
{
    self.contentOfCurrentIncidentProperty = nil;
    self.currentIncidentObject = nil;
    [super dealloc];
}
@end
```

4.3. Using NSXMLParser

NSXMLParser is quite a powerful XML parser. This tutorial will not explain its behavior in detail but instead will present a simple example that you can use as a starting point for your own needs.

4.3.1. The delegate pattern

When **NSXMLParser** parses a XML file, it informs a *delegate* about its progress. The delegate can then use this information and process it accordingly. This is a very common design pattern in Cocoa. It minimizes the need to implement subclasses. Note, that we are not using a subclass of **NSXMLParser**. Instead **TWIncidentsParser** will be the delegate of a standard **NSXMLParser** in order to process the information found in the XML file.

4.3.2. Start the parsing

Inside the implementation block of **TWIncidentsParser** add the following lines of code:

```
- (void)parseFileAtURL:(NSURL *)URL parseError:(NSError **)error
{
    NSXMLParser *parser = [[NSXMLParser alloc] initWithContentsOfURL:URL];

    [parser setDelegate:self];

    [parser setShouldProcessNamespaces:NO];
    [parser setShouldReportNamespacePrefixes:NO];
    [parser setShouldResolveExternalEntities:NO];

    [parser parse];

    NSError *parseError = [parser parserError];
    if (parseError && error) {
```

```
15          *error = parseError;
16      }
17
18      [parser release];
19  }
```

Let's take a look at this method for a moment: In line 3 you create a new instance of **NSXMLParser**. Then you set your own class **TWIncidentsParser** as the parser's delegate. This will allow you to receive the parser's delegate methods. Lines 7 to 9 set some parsing options (refer to the documentation for clarification) and line 11 finally triggers the parsing. If an error occurs, this error is returned *by reference* to the sender (lines 13 to 16).

4.3.3. Implement some delegate methods

In order to actually use the parser, you need to react to its delegate methods. **NSXMLParser** sends a lot of delegate methods, but you are not required to implement all of them. In this case we only need three.

Add the following method to your implementation block. This method is called whenever the parser starts reading a new element (such as `<entry>`, `<title>`, `<summary>`, etc.):

Listing 4.1: A **NSXMLParser** callback

```
1  -   (void)parser:(NSXMLParser *)parser
2   didStartElement:(NSString *)elementName
3      namespaceURI:(NSString *)namespaceURI
4     qualifiedName:(NSString *)qName
5        attributes:(NSDictionary *)attributeDict
6  {
7      if (qName) {
8          elementName = qName;
9      }
10
11     if ([elementName isEqualToString:@"entry"]) {
12         // An entry in the RSS feed represents an incident, so create an instance of
               it.
13         TWIncident *anIncident = [[TWIncident alloc] init];
14         self.currentIncidentObject =  anIncident;
15         [anIncident release];
16         return;
17     }
18
19     if ([elementName isEqualToString:@"link"]) {
20         NSString *relAtt = [attributeDict valueForKey:@"rel"];
21         if ([relAtt isEqualToString:@"alternate"]) {
22             NSString *link = [attributeDict valueForKey:@"href"];
23             self.currentIncidentObject.weblink = link;
24         }
25     } else if ([elementName isEqualToString:@"title"]) {
26         // Create a mutable string to hold the contents of the 'title' element.
27         // The contents are collected in parser:foundCharacters:.
28         self.contentOfCurrentIncidentProperty = [NSMutableString string];
29     } else if ([elementName isEqualToString:@"summary"]) {
30         // Create a mutable string to hold the contents of the 'summary' element.
31         // The contents are collected in parser:foundCharacters:.
32         self.contentOfCurrentIncidentProperty = [NSMutableString string];
33     } else {
```

```
34          // The element isn't one that we care about, so set the property that holds
                the
35          // character content of the current element to nil. That way, in the parser:
                foundCharacters:
36          // callback, the string that the parser reports will be ignored.
37          self.contentOfCurrentIncidentProperty = nil;
38      }
39  }
```

As you can see, when the parser encounters an `<entry>` element, the delegate creates a new instance of **`TWIncident`** and keeps a reference to it.

If the element represents a property of the `<entry>` we are interested in, the delegate creates an empty string to hold its content. Otherwise `contentOfCurrentIncidentProperty` is set to **`nil`**.

Sidenote: Convenience allocators

Take a look at the code in listing 4.1 on the preceding page again. Notice how the mutable strings are created:

```
self.contentOfCurrentIncidentProperty = [NSMutableString string];
```

The **`string`** method of **`NSMutableString`** is a so called *convenience allocator*. It returns an autoreleased instance of **`NSMutableString`**.
Normally, if you wanted to create a new instance of a class and assign it to a property, you would do something like this:

```
NSMutableString *aString = [[NSMutableString alloc] init];
self.contentOfCurrentIncidentProperty = aString;
[aString release];
```

You create an instance, retain it using the property and then release its local reference.
If a convenience allocator is present, you can use it to save code. Using the convenience allocator is identical to the following code snippet:

```
self.contentOfCurrentIncidentProperty = [[[NSMutableString alloc] init]
    autorelease];
```

The **`alloc init`** combination creates a new instance of **`NSMutableString`** and **`autorelease`** adds this instance to the current autorelease pool. Therefore the string will be released when the autorelease pool is deallocated from memory. Of course the property `contentOfCurrentIncidentProperty` will remain valid because it is set to retain its value.

Make sure that the content of an element we are interested in is appended to `contentOfCurrentIncidentProperty`:

```
1  - (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
2  {
3      if (self.contentOfCurrentIncidentProperty) {
4          // If the current element is one whose content we care about, append 'string
               '
5          // to the property that holds the content of the current element.
6          [self.contentOfCurrentIncidentProperty appendString:string];
7      }
8  }
```

Finally, when the parser ends an element, check if it is an `<entry>` element. If that is the case, simply print the `currentIncidentObject` to the console. If the element is a property we are interested in, store its content in the appropriate property of the `currentIncidentObject`.

```
- (void)parser:(NSXMLParser *)parser
 didEndElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName
{
    if (qName) {
        elementName = qName;
    }
    if ([elementName isEqualToString:@"entry"]) {
        NSLog(@"parsed: %@", self.currentIncidentObject);
        return;
    }
    if ([elementName isEqualToString:@"title"]) {
        self.currentIncidentObject.title = self.contentOfCurrentIncidentProperty;
    } else if ([elementName isEqualToString:@"summary"]) {
        self.currentIncidentObject.summary = self.contentOfCurrentIncidentProperty;
    }
}
```

The parser is now ready to use.

4.4. Use your new parser

Open the file TrafficWatchAppDelegate.m and add the following method:

```
- (void)loadIncidentsData
{
    static NSString *kIncidentsURLString = @"http://www.freiefahrt.info/lmst.de_DE.
        xml";
    NSError *error = nil;

    TWIncidentsParser *incidentsParser = [[TWIncidentsParser alloc] init];
    [incidentsParser parseFileAtURL:[NSURL URLWithString:kIncidentsURLString]
                         parseError:&error];
    [incidentsParser release];
}
```

Look for the `application: didFinishLaunchingWithOptions:` method and call your new method there:

```
- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    // Override point for customization after app launch
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];

        [self loadIncidentsData];

        return YES;
}
```

Also, make sure that you import TWIncidentsParser.h.

Finally, build and run your application. In the console, you should see several incident reports.

Use JSON instead

If you plan to implement your own webservice, you might consider using XML as the format for exchanging data with iOS (and other mobile) devices. While XML parsing can be implemented easily and its performance is good, it is not suitable for every application. Its main advantage is verbosity, i. e. you can easily understand the information stored in an arbitrary XML file without knowing the exact details about the communication protocol. This makes XML a good choice for complex data that has to be understood by many developers. However, this also means that a lot of data in a XML file is consumed by explanatory and structural elements. If sizes matter and speed is the key, you might want to consider using a more stripped down data format such as JSON. There exists a very fast and powerful implementation of JSON for Cocoa called **JSONKit**. Check it out!

`http://github.com/johnezang/JSONKit`

5. Managing application data

Summary In the last chapter, you have implemented a class that downloads and parses an RSS feed creating **`TWIncident`** objects. Now, it is time to store this data somewhere. Read the next section to get an idea of what has to be done in this chapter.

5.1. Application architecture

Of course, there exist many possibilities to manage data in an application. For the scope of this tutorial, we will discuss an approach that is simple to implement and suitable for most iPhone utility applications. If you plan to design a more complex application you might want to take a look at the CoreData Framework.

It is always a good idea to store you application's data, where it can easily be accessed from everywhere inside you navigation hierarchy. In an iPhone application the application delegate is therefore a good place to manage your data.

Our application must handle only one type of data: A bunch of **`TWIncident`** objects. We will store these objects in a **`NSMutableArray`** in our custom **`TrafficWatchAppDelegate`**. Access to this mutable array is provided only through a set of specialized methods, i.e. the array wont be public.

5.2. Key-Value Coding (KVC)

`NSObject` defines two extremely useful methods for every object: one for reading and one for setting variables by name:

```
- (id)valueForKey:(NSString *)attrName;
- (void)setValue:(id)newValue forKey:(NSString *)attrName;
```

The **`valueForKey:`** method allows you to read the value of a variable by the given name. This is done by calling the appropriate accessor method, if present. If no accessor method is found, the instance variable is accessed directly. The **`setValue:forKey:`** method on the other hand uses the setter method to change a variable's value. If no setter is found, the variable is changed directly.

In order for this to work properly, your class has to be *Key-Value Coding* compliant. When using properties, this is ensured automatically.[1] If you are using your own accessor and setter methods you need to make sure, that they are named within the specifications described in the *Key-Value Coding Programming Guide*.[2]

[1] Actually, there is one exception. If you had a property of the type **`BOOL`** named **`writable`**, the KVC compliant accessor to this variable would be **`isWritable`**. Since the standard accessor of the property would simply be **`writable`**, your class would not be fully KVC compliant. You should therefore change the name of the accessor method by using the `getter=isWritable` attribute in the property declaration.

[2] https://developer.apple.com/ios/library/documentation/Cocoa/Conceptual/KeyValueCoding/

Since nearly all of Apple's frameworks use KVC, you should always write your custom classes KVC compliant!

For further information on why KVC is useful and how it works, please read the *Key-Value Coding Programming Guide* from Apple.

5.3. Edit the AppDelegate

As mentioned above, declare a new instance variable named **`incidents`** of the type **`NSMutableArray`** in the TrafficWatchAppDelegate.m file. This time **do not** create a property for this variable. Instead, declare three (KVC compliant) methods, that will encapsulate external access to the array (note, that you need to inform the compiler, that a **`TWIncident`** class actually exists using the `@class` directive):

Listing 5.1: The TrafficWatchAppDelegate.h file.

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

@class TrafficWatchViewController;
@class TWIncident;
@interface TrafficWatchAppDelegate : NSObject <UIApplicationDelegate> {
    NSMutableArray *incidents;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet TrafficWatchViewController *viewController;

- (void)loadIncidentsData;

- (NSUInteger)countOfIncidents;
- (void)addToIncidents:(TWIncident *)newIncident;
- (id)objectInIncidentsAtIndex:(NSUInteger)index;

@end
```

The *Cocoa Coding Guidelines*[3] demand that you should always use accessor and setter methods, even for private variables. Therefore, open the TrafficWatchAppDelegate.m file and add the following private interface before the `@implemnation` block:

```
@interface TrafficWatchAppDelegate ()
@property (nonatomic, retain) NSMutableArray *incidents;
@end
```

Don't forget to `@synthesize` the property and to release the array in the **`dealloc`** method (see Section 3.1.1 on page 10).

Next, add the implementation of the three methods defined earlier:

```
- (NSUInteger)countOfIncidents
{
    return [self.incidents count];
}

- (void)addToIncidents:(TWIncident *)newIncident
{
    [self.incidents addObject:newIncident];
```

[3]The *Coding Guidelines* will soon be available for download on iTunes U.

```
}

- (id)objectInIncidentsAtIndex:(NSUInteger)index
{
    return [self.incidents objectAtIndex:index];
}
```

As you can see, these methods simply forward the desired action to the private mutable array.

Finally, make sure that a new empty **NSMutableArray** is present when the RSS feed is processed. Additionally, once the parser is done, log the incidents array to console:

```
- (void)loadIncidentsData
{
    static NSString *kIncidentsURLString = @"http://www.freiefahrt.info/lmst.de_DE.
        xml";
    NSError *error = nil;

    self.incidents = [NSMutableArray array];

    TWIncidentsParser *incidentsParser = [[TWIncidentsParser alloc] init];
    [incidentsParser parseFileAtURL:[NSURL  URLWithString: kIncidentsURLString]
                            parseError:&error];
    [incidentsParser release];

    NSLog(@"Parser has finished. incidents: %@", self.incidents);
}
```

Trick question

Why would the following line of code be wrong and result in memory being wasted?

```
self.incidents = [[NSMutableArray alloc] init];
```

Save and build your project. Fix the errors and then continue with the next section.

5.4. Edit TWIncidentsParser

We have now successfully created the infrastructure in our application delegate to store **TWIncident** objects. The next step is to actually store every parsed **TWIncident** object in the application delegate.

Open TWIncidentsParser.m and import your application delegate's .h file:

```
#import "TWIncidentsParser.h"
#import "TWIncident.h"
#import "TrafficWatchAppDelegate.h"
```

Then look for the `parser:didEndElement:namespaceURI:qualifiedName:` method. Instead of only logging the parsed object, we will now send it to the application delegate:

```
    if ([elementName isEqualToString:@"entry"]) {
        NSLog(@"parsed: %@", self.currentIncidentObject);
        TrafficWatchAppDelegate *appDelegate = [[UIApplication sharedApplication]
            delegate];
```

```
4           [appDelegate addToIncidents:self.currentIncidentObject];
5           return;
6       }
```

Take a moment to understand what is happening here: **`UIApplication`** is a so called *singelton* class. Such a class only has one shared instance per application. You access this shared instance by sending **`UIApplication`** the **`sharedApplication`** method. In line 3 we obtain a reference to the application's delegate. Then we can accses the **`addToIncidents:`** method.

5.5. Access data from TrafficWatchViewController

In the beginning of this tutorial you prepared the **`TrafficWatchViewController`** to display data in a **`UITableView`**. Open the TrafficWatchViewController.m file and import TrafficWatchAppDelegate.h:

```
#import "TrafficWatchAppDelegate.h"
```

Then, edit the two **`UITableView`** data source methods to access the data stored in the application delegate using the methods we defined in Section 5.3 on page 26.

```
1  - (NSInteger)tableView:(UITableView *)aTableView numberOfRowsInSection:(NSInteger)
       section
2  {
3      TrafficWatchAppDelegate *appDelegate = [[UIApplication sharedApplication]
           delegate];
4      return [appDelegate countOfIncidents];
5  }
6
7  - (UITableViewCell *)tableView:(UITableView *)aTableView
8           cellForRowAtIndexPath:(NSIndexPath *)indexPath
9  {
10     static NSString *MyIdentifier = @"MyIdentifier";
11
12     UITableViewCell *cell = [aTableView dequeueReusableCellWithIdentifier:
           MyIdentifier];
13     if (cell == nil) {
14         cell = [[UITableViewCell alloc] initWithFrame:CGRectZero
15                                       reuseIdentifier:MyIdentifier];
16         [cell autorelease];
17     }
18
19     TrafficWatchAppDelegate *appDelegate = [[UIApplication sharedApplication]
           delegate];
20     TWIncident *anIncident = [appDelegate objectInIncidentsAtIndex:[indexPath row]];
21     cell.textLabel.text = anIncident.title;
22     // Remove the line: [anIncident release];
23     // and remove the NSLog call.
24     return cell;
25 }
```

Build, fix and run your application. The table view should now display a list of incidents.

5.6. Challenge

Examine how the table view displays its content:

```
- (UITableViewCell *)tableView:(UITableView *)aTableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *MyIdentifier = @"MyIdentifier";

    UITableViewCell *cell = [aTableView dequeueReusableCellWithIdentifier:
        MyIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithFrame:CGRectZero
                                       reuseIdentifier:MyIdentifier];
        [cell autorelease];
    }

    TrafficWatchAppDelegate *appDelegate = [[UIApplication sharedApplication]
        delegate];
    TWIncident *anIncident = [appDelegate objectInIncidentsAtIndex:[indexPath row]];
    cell.textLabel.text = anIncident.title;

    return cell;
}
```

For each row, the table asks its **`dataSource`** for an instance of **`UITableViewCell`**. The data source in its implementation of the `tableView:cellForRowAtIndexPath:` method returns a configured cell object that the table view can use to draw a row. For performance reasons, the data source tries to reuse cells as much as possible. It first asks the table view for a specific reusable cell object by sending it `dequeueReusableCellWithIdentifier:` message. If no such object exists, the data source creates it, assigning it a reuse identifier. It sets the cell's content (its text label's text in this example) and returns it. The chapter "A Closer Look at Table-View Cells" in the *Table View Programming Guide for iPhone OS*[4] discusses this data source method and **`UITableViewCell`** objects in more detail.

Your task

Change the style of the returned `UITableViewCell` to display two lines of text. The first line should contain the incident's title and the second (smaller) line should contain the incident's summary.

[4] `https://developer.apple.com/ios/library/documentation/UserExperience/Conceptual/TableView_iPhone/`

6. UINavigationController

Until now, our application has only used a single **UIViewController**. This is fine for applications that display a limited amount of information. However, most of the times you will want to present different views to the user each representing different layers in your application's view hierarchy. Using **UINavigationController** you can easily realize even complex navigational structures, while maintaining a simple and intuitive user experience.

6.1. Change TrafficWatchAppDelegate

Open TrafficWatchAppDelegate.h and exchange the **TrafficWatchViewController** for a standard **UINavigationController**.

```
@interface TrafficWatchAppDelegate : NSObject <UIApplicationDelegate> {
    NSMutableArray *incidents;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UINavigationController *navigationController;
```

Note, that we do not need to explicitly declare an instance variable (*ivar*), instead, we only declare the property, the iOS runtime will automatically create an ivar named `navigationController`.

In the .m file, you need to change the `@synthesize` directive, the `release` call in **dealloc** and the **application:didFinishLaunchingWithOptions:** method (only the last change is shown below):

```
- (BOOL)                    application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // Override point for customization after app launch
    self.window.rootViewController = self.navigationController;
    [self.window makeKeyAndVisible];

    [self loadIncidentsData];

    return YES;
}
```

6.2. Change MainWindow.xib

Select MainWindow.xib to open the file in the interface editor. Right-click (ctrl-click) on the TrafficWatchAppDelegate item and remove the viewController outlet connection by clicking on the small "x" next to it.

If it is not already visible, open the Object Library and search for a UINavigationController. Drag it to the document window (see Figure 6.1).

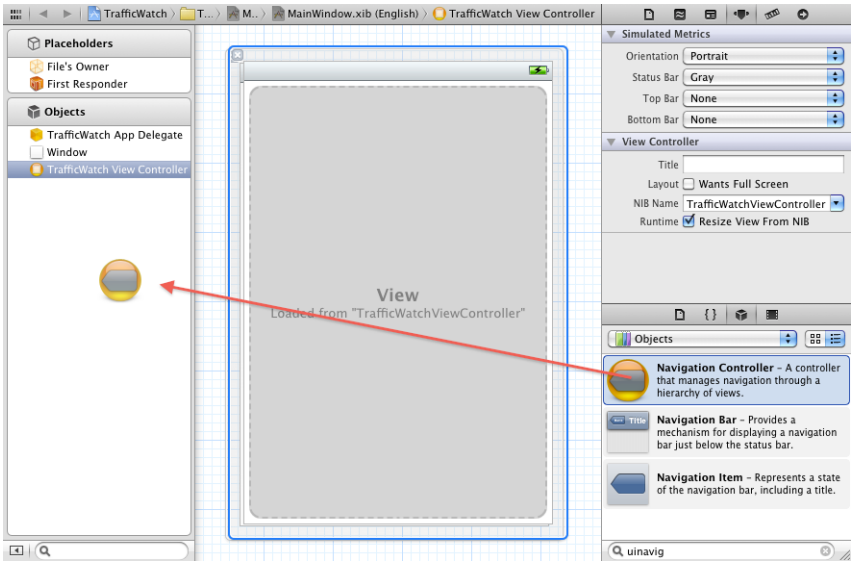


Figure 6.1.: Add a **`UINavigationController`** to your application.

Now, select the Navigation Controller element. A new window appears showing an empty navigation controller layout. From the objects outline view drag the TrafficWatchViewController into the navigation controller you just added (see Figure 6.2).

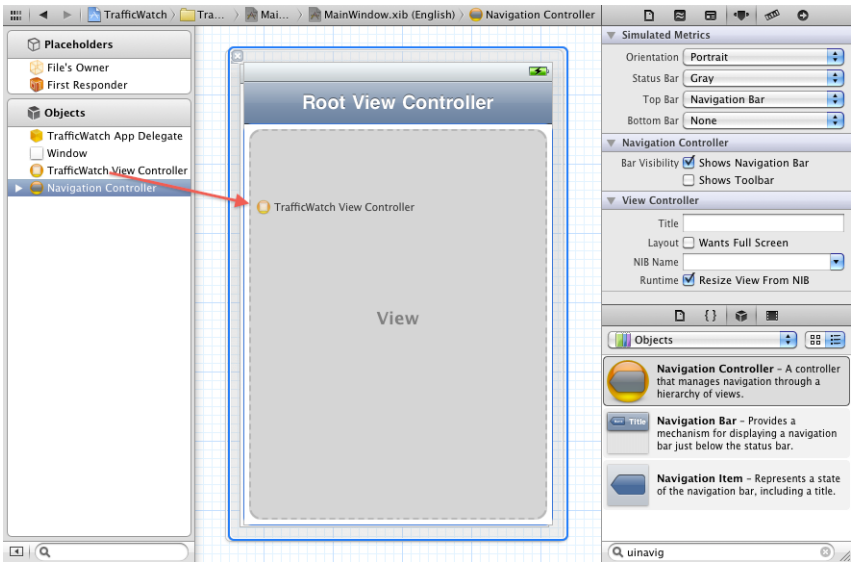


Figure 6.2.: Set your view controller as the root view controller of the navigation controller.

Finally, drag a connection from TrafficWatchAppDelegate to Navigation Controller and choose navigationController as the name of the outlet. The finished XIB file should look as depicted in Figure 6.3.

When you build and run your application you should now see the table view embedded in a navigation controller.

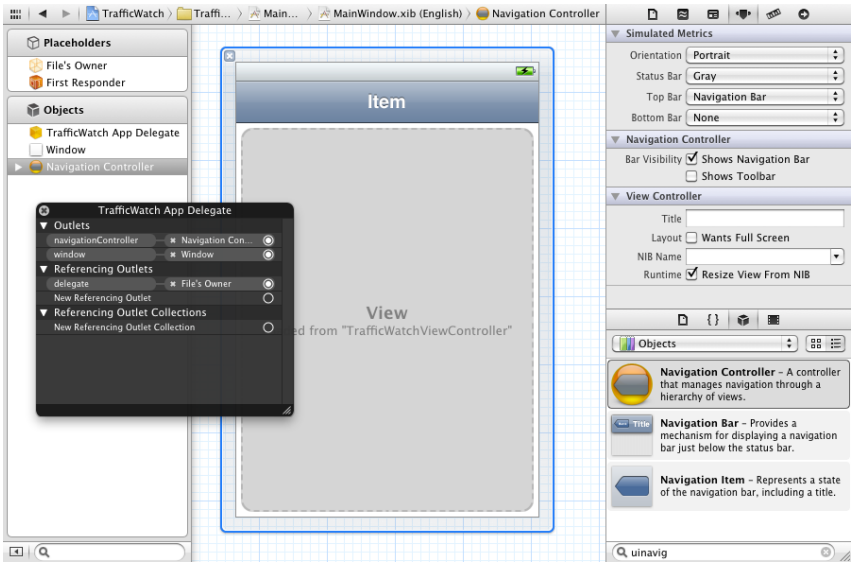


Figure 6.3.: The finished XIB file after adding a navigation controller.

6.3. Change the title attribute

Each view controller has certain attributes that define its appearance when added to a navigation controller hierarchy.

Every object that is created from inside a XIB file is sent the **`awakeFromNib`** message during creation. This is therefore a good place to change properties that affect the appearance of the object.

Add the following method to TrafficWatchViewController.m:

```
1 - (void)awakeFromNib
2 {
3     self.title = @"Traffic Watch";
4 }
```

6.4. Further Reading

Understanding the correct usage of **`UIViewController`** and **`UINavigationController`** is crucial to writing iPhone applications efficiently. It is therefore strongly recommended to read Apple's *View Controller Programming Guide for iOS*[1] to learn more about these concepts.

[1] https://developer.apple.com/iphone/library/featuredarticles/ViewControllerPGforiPhoneOS/

7. Custom table view cells

Table views are a powerful tool when displaying information in your iPhone application. They are extremely customizable and highly optimized.

A table view is constructed of sections and rows. In our application, the table view only has a single section and several rows. Each row is drawn by the table view using a **`UITableViewCell`** object. The standard cell type shipped with Cocoa Touch is already quite versatile and useful to many applications. However, if the standard class does not fill your needs you need to create your own subclass of **`UITableViewCell`**.

If you finished the challenge from chapter 5, you have already changed the cell style to **`UITableViewCellStyleSubtitle`** and your app should look similar to the one in Figure 7.1.

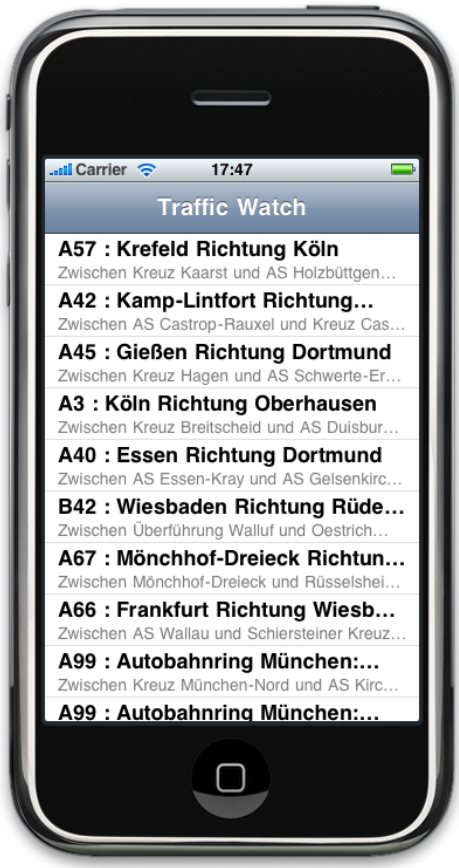


Figure 7.1.: Table view cells with **`UITableViewCellStyleSubtitle`**.

As you can see, although this cell style already improves the usability of your application, it still fails in completely displaying all the information associated with incident objects.

7.1. Anatomy of a `UITableViewCell`

The drawing area of table view cells is devided in different sections. The *Table View Programming Guide for iPhone OS*[1] gives a good overview of this topic and it is highly recommended that you read and understand this guide soon. For now we will stick with a brief introduction.

A table view automatically resizes its cells to fill an entire row. Each cell has a `contentView` which is used to display information or controls. That means, this is the place where you can add additional subviews in order to customize the appearance of your cell.

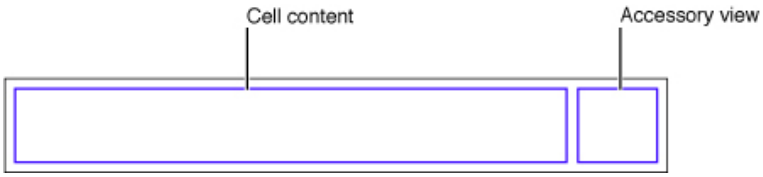


Figure 7.2.: Anatomy of a cell.

7.2. Views and coordinates in UIKit

Coordinates in UIKit are based on a coordinate system whose origin is in the top-left corner and whose coordinate axes extend down and to the right from that point. Coordinate values are represented using floating-point numbers, which allow for precise layout and positioning of content and allow for resolution independence.

Every window and view object maintains its own local coordinate system. All drawing in a view occurs relative to the view's local coordinate system. The frame rectangle for each view, however, is specified using the coordinate system of its parent view, and coordinates delivered as part of an event object are specified relative to the coordinate system of the enclosing window. For convenience, the UIWindow and UIView classes each provide methods to convert back and forth between the coordinate systems of different objects.

A view object tracks its size and location using its `frame`, `bounds`, and `center` properties. The frame property contains a rectangle, the **frame rectangle**, that specifies the view's location and size relative to its parent view's coordinate system. The bounds property contains a rectangle, the **bounds rectangle**, that defines the view's position and size relative to its own local coordinate system. And although the origin of the bounds rectangle is typically set to (0, 0), it need not be. The center property contains the **center point** of the frame rectangle.

If you are not already familiar with the concepts of views and windows in UIKit, you should read the chapter "Window and Views" in the *iPhone Application Programming Guide*[2].

7.3. Challenge: Changing the cell layout

When you create your own customized cell you usually add a few views to it, such as control elements, labels or a text field. You can also change a cell's background color or even provide your own back-

[1] `https://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/TableView_iPhone/`

[2] `http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/WindowsandViews/WindowsandViews.html`

ground view. In our project we will use the text labels that are already on board. Your task will be to change the labels' size in order to fit their entire text at a given size.

Figure 7.3 shows how your application is supposed to look.

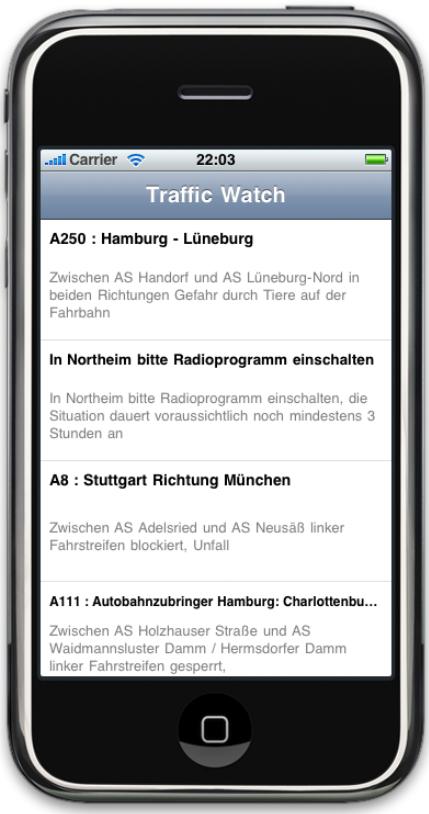


Figure 7.3.: A better cell layout.

7.3.1. Create a new subclass of UITableViewCell

In your project create a new subclass of **`UITableViewCell`** and call it TWIncidentCell. Open the implementation file and remove the predefined `initWithStyle:reuseIdentifier:` method. Instead add the following method:

```
- (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString *)
    reuseIdentifier {
    if (self = [super initWithStyle:UITableViewCellStyleSubtitle
                  reuseIdentifier:reuseIdentifier]) {
        // Edit text labels' properties here
    }
    return self;
}
```

This method is where you can change the font size of the text labels. Check the documentation for **`UILabel`** and **`UIFont`** to learn how you can change the font attributes of your cell's labels.

When the cell is created its size is usually still undefined. Eventually, when the cell is added to a table view and shortly before its drawn, the table view calculates its size and position. Then the cell is asked

to layout its subviews. This is done by calling the `layoutSubviews` method.

Add the following method to your custom cell:

```
- (void)layoutSubviews
{
    [super layoutSubviews];

    CGRect contentRect = self.contentView.bounds;

    // Layout subviews with respect to |contentRect|
}
```

Since you probably still don't have a clue how you are supposed to layout the text labels, here is the code, which puts the main text label in the top left corner and sets its width to fill the entire horizontal space:

```
- (void)layoutSubviews
{
    [super layoutSubviews];

    CGRect contentRect = self.contentView.bounds;

    CGSize sizeOfString = [self.textLabel.text sizeWithFont:self.textLabel.font];

    CGFloat positionX = 8.0f;
    CGFloat positionY = 8.0f;
    CGFloat width  = contentRect.size.width - 2.0f * 8.0f;
    CGFloat height = sizeOfString.height;

    CGRect textLabelFrame = CGRectMake(positionX, positionY, width, height );
    self.textLabel.frame = textLabelFrame;
}
```

Edit your subclass to reflect the layout shown in the drawing in Figure 7.4.

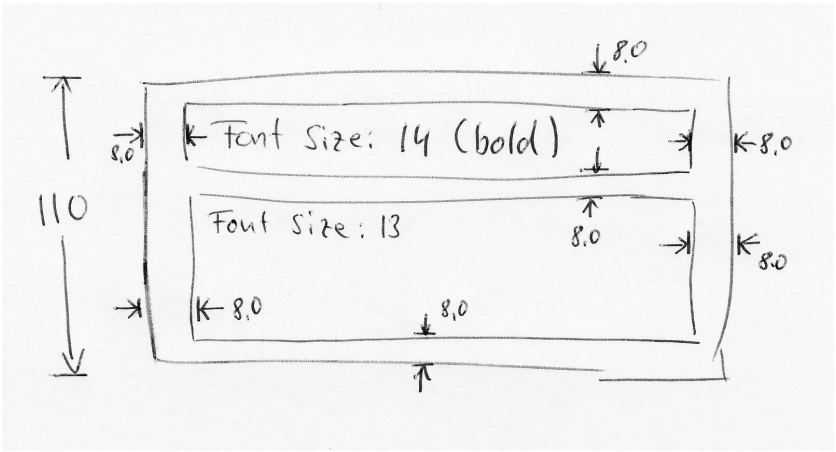


Figure 7.4.: Layout specifications

The height of a cell is not controlled by the cell itself. Instead the table view asks its delegate for the height of a specific row. Therefore you need to define your view controller (`TrafficWatchViewController`) as the table view's delegate object. Open the TrafficWatchViewController.xib file and drag a connection

from the table view to the File's owner placeholder. Then you can implement the following method in TrafficWatchViewController.m and return a suitable value:

```
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath
    *)indexPath;
```

Of course, you also need to actually use your newly created cell. Edit the `tableView:cellForRowAtIndexPath:` method to return an instance of **`TWIncidentCell`**. The `detailTextLabel` of the cell should contain the value stored in the `summary` property of the displayed incident object.

7.4. Challenge: Launch Safari from your app

Each incident has a URL associated with it, that points to a webpage describing the event in more detail. Extend your application to launch this URL in Safari when the user taps a table row.

Be sure to follow the *iPhone Human Interface Guidelines*[3] . Especially make sure your table view reacts on taps correctly (as stated in section "Table Views"):

> [...] a table view automatically highlights a row in response to the user's tap, when the row item supports selection. This brief highlight is a feedback mechanism that allows users to see that they've tapped the row they intended and lets them know their selection is being processed. You should never make this highlight permanent in an effort to indicate the current selection in a list; instead, use the checkmark (described in 'Table-View Elements').

Hints

- Implement the **`UITableViewDelegate`** method `tableView:didSelectRowAtIndexPath:`. Don't forget to connect the table view delegate outlet in TrafficWatchViewController.xib.
- Use **`UIApplication`**'s `openURL:` to launch Safari (see the "Communicating with Other Applications" section in the *iOS Application Programming Guide* [4].

7.5. Further reading

You will soon realize that table views and table view cells are integral parts of almost any iOS app. Efficient use of these classes is crucial to writing apps that perform well. Take a look at the AdvancedTableViewCells sample code to gain some insights on increasing the scrolling performance of complex table view cells.

[3] `http://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/MobileHIG/`
[4] `http://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/`

8. Increasing application performance

As stated in the *iOS Application Programming Guide* [1] you should never "block the main thread". Unfortunately, our implementation breaks this rule by downloading and parsing the RSS feed on the main thread. If your internet connection is slow or even unavailable, the application will hang during startup unable to respond to events or user interaction.

8.1. Concurrency Programming in iOS

To circumvent the problems mentioned above you can spawn a new thread for potentially time consuming operations. As simple as this may sound, threading is a complex topic. When more then one thread is running, you need to take care, that your application's resources stay synchronized between all of them to prevent data corruption.

While you can, of course, spawn new threads using **`NSThread`** and calling its `detachNewThreadSelector:toTarget:withObject:` method, this is not the recommended way. Instead, you should encapsulate your time consuming task in a self-contained instance of **`NSOperation`**. This operation is then added to an instance of **`NSOperationQueue`** and the system will detach a new thread for you. If you add multiple operations to a single operation queue, the system will either execute one operation at the time or spawn multiple threads to execute multiple operations simultaneously. Additionally, an operation queue will make sure that the execution benefits from multiple processor cores, if your system features those (e.g. iPad 2). Last but not least, **`NSOperation`** and **`NSOperationQueue`** provide many useful means to monitor the progress of your tasks easily.

8.2. Rewrite TWIncidentsParser

In this section we will rewrite the XML parser to be a subclass of **`NSOperation`**.

8.2.1. Refractor TWIncidentsParser

First, we will rename `TWIncidentsParser` into `TWIncidentsParseOperation`.

In Xcode open TWIncidentsParser, right-click on the class name `TWIncidentsParser` and choose Refractor... -> Rename from the context menu.

Type in `TWIncidentsParseOperation` and make sure Rename related files is checked. Xcode will present the changes it will make to all files affected by the refractor in a diff editor. Click Save to apply those changes. If you want, you can create a snapshot of your project in order to save the state before the changes are applied. If something goes wrong you can then easily go back.

Finally, change the superclass from `NSObject` to **`NSOperation`**.

[1] `http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/`

8.2.2. Define a delegate protocol

The initial implementation of the parser informed the app delegate every time a new incident had been parsed. This time, we will make this a bit more generic and adopt the delegate pattern. Every parse operation will have a delegate. This delegate will be informed when the parsing is done and an array with all parsed incidents will be passed on to this delegate. If a class wants to be the delegate of the parse operation it must implement the method we will use to inform it. This method is declared in a protocol named `TWIncidentsParseOperationDelegate`.

At the end of the TWIncidentsParseOperation.h file, just below the `@end` statement, add the following lines of code:

```
@protocol TWIncidentsParseOperationDelegate <NSObject>
@optional
- (void)incidentsParseOperation:(TWIncidentsParseOperation *)parseOperation
                       finished:(NSArray *)incidents;
@end
```

8.2.3. Add new instance variables and methods

Add a new instance variable called **`delegate`** of the type `id` to the interface declaration of `TWIncidentsParseOperation`. Since the delegate should implement the previously defined protocol you can add the protocol name enclosed by left and right pointing angle brackets:

```
id <TWIncidentsParseOperationDelegate> delegate;
```

`id` is the most general type in Objective-C. It resolves to every object pointer. Note, that you don't need the asterisk (*) symbol here, because `id` itself already is a pointer.

Also add a property declaration for the delegate. Make sure to set its option to `assign`. A delegate object is by convention never retained.

As mentioned above, before, we informed the app delegate about every new incident as soon as it had been parsed. Even in a multithreaded environment, this will cause a significant performance issue. In fact, it is much faster to first parse the entire XML file and then pass a complete array of all incident objects on to the delegate. Therefore, we need a mutable array which we can populate with incident objects while parsing the XML file.

Add a new instance variable called **`parsedIncidentObjects`** of type `NSMutableArray` to the interface declaration of `TWIncidentsParseOperation`.

Additionally, add another instance variable called **`feedURL`** of type `NSURL` to the interface.

Remove the `parseFileAtURL:parseError:` method and instead declare a new designated initializer:

```
- (id)initWithFeedURL:(NSURL *)feedURL
             delegate:(id <TWIncidentsParseOperationDelegate>)aDelegate;
```

When done, the new header file should look like this:

```
@class TWIncident;
@protocol TWIncidentsParseOperationDelegate;
@interface TWIncidentsParseOperation : NSOperation <NSXMLParserDelegate> {
    id <TWIncidentsParseOperationDelegate> delegate;
    TWIncident *currentIncidentsObject;
    NSMutableString *contentOfCurrentIncidentProperty;
```

```
    NSMutableArray *parsedIncidentObjects;
    NSURL *feedURL;
}
@property (assign) id <TWIncidentsParseOperationDelegate> delegate;
- (id)initWithFeedURL:(NSURL *)feedURL
             delegate:(id <TWIncidentsParseOperationDelegate>)aDelegate;
@end

@protocol TWIncidentsParseOperationDelegate <NSObject>
@optional
- (void)incidentsParseOperation:(TWIncidentsParseOperation *)parseOperation
                       finished:(NSArray *)incidents;
@end
```

8.2.4. Implement the TWIncidentsParseOperation

In the implementation file add private properties for `feedURL` and `parsedIncidentObjects`. Set their attributes to `(nonatomic, retain)` and add appropriate `@synthesize` statements.

Implement the previously defined designated initializer,

```
- (id)initWithFeedURL:(NSURL *)url
             delegate:(id <TWIncidentsParseOperationDelegate>)aDelegate;
{
    if ((self = [super init])) {
        self.feedURL = url;
        self.delegate = aDelegate;
        self.parsedIncidentObjects = [NSMutableArray array];
    }
    return self;
}
```

and complete the `dealloc` method.

```
- (void)dealloc {
    self.contentOfCurrentIncidentProperty = nil;
    self.currentIncidentObject = nil;
    self.delegate = nil;
    self.parsedIncidentObjects = nil;
    self.feedURL = nil;
    [super dealloc];
}
```

When a **`NSOperation`** object is added to an operation queue its **`main`** method is executed. In our case we can simply rename the `parseFileAtURL:parseError:` method to **`main`** and change a few lines of code:

```
- (void)main
{
    NSAutoreleasePool *ap = [[NSAutoreleasePool alloc] init];
    self.parsedIncidentObjects = [NSMutableArray array];
    NSXMLParser *parser = [[NSXMLParser alloc] initWithContentsOfURL:self.feedURL];
    if (parser) {
        [parser setDelegate:self];
        [parser setShouldProcessNamespaces:NO];
        [parser setShouldReportNamespacePrefixes:NO];
        [parser setShouldResolveExternalEntities:NO];
```

```
        [parser parse];
        [parser release];
        if (![self isCancelled]) {
            if (self.delegate && [self.delegate respondsToSelector:
                                                @selector(incidentsParseOperation:
                                                    finished:)]) {
                [self.delegate incidentsParseOperation:self
                                                finished:self.parsedIncidentObjects];
            }
        }
    }
    [ap release];
}
```

The `main` method is called on a secondary thread when the operation is executed. Because autorelease pools can not be shared between different threads you must provide a local autorelease pool. This is done by simply creating a local instance in the beginning of the threaded method and release the pool at the end of the method.

When the parser is done and the operation has not been cancelled already, the delegate - if present - is informed. Note, that we call the delegate from a threaded method here. Therefore, the delegate method will also be called from a secondary thread. We have to keep that in mind later.

For now, change the `parser:didEndElement:namespaceURI:qualifiedName:` method such that parsed incidents are stored in the local array.

```
if ([elementName isEqualToString:@"entry"]) {
    [self.parsedIncidentObjects addObject:self.currentIncidentObject];
    self.currentIncidentObject = nil;
}
```

8.3. Use the new operation

Open TrafficWatchAppDelegate.h and import the TWIncidentsParseOperation.h file. In the interface declaration make the app delegate implement the `TWIncidentsParseOperationDelegate` protocol and add a new instance variable **`NSOperationQueue *operationQueue;`**. Also add a private property for the queue and synthesize the getter and setter methods in the implementation file.

Listing 8.1: TrafficWatchAppDelegate.h

```
#import "TWIncidentsParseOperation.h"
@class TrafficWatchViewController;
@class TWIncident;
@interface TrafficWatchAppDelegate : NSObject <UIApplicationDelegate,
    TWIncidentsParseOperationDelegate> {
    NSMutableArray *incidents;
    NSOperationQueue *operationQueue;
}
```

Listing 8.2: TrafficWatchAppDelegate.m

```
@interface TrafficWatchAppDelegate ()
@property (nonatomic, retain) NSMutableArray *incidents;
@property (nonatomic, retain) NSOperationQueue *operationQueue;
@end
@implementation TrafficWatchAppDelegate
```

```
@synthesize window=_window;
@synthesize navigationController;
@synthesize incidents;
@synthesize operationQueue;
```

Open TrafficWatchAppDelegate.m and change the `loadIncidentsData:` method to use the new operation.

```
- (void)loadIncidentsData
{
    if (self.operationQueue) {
        // download in progress
        return;
    }
    self.incidents = [NSMutableArray array];
    NSOperationQueue *newQueue = [[NSOperationQueue alloc] init];
    self.operationQueue = newQueue;
    [newQueue release];
    static NSString *kIncidentsURLString = @"http://www.freiefahrt.info/lmst.de_DE.
        xml";
    NSURL *feedURL = [NSURL URLWithString:kIncidentsURLString];
    TWIncidentsParseOperation *parseOperation = [[TWIncidentsParseOperation alloc]
        initWithFeedURL:feedURL delegate:self];
    [[UIApplication sharedApplication] setNetworkActivityIndicatorVisible:YES];
    [self.operationQueue addOperation:parseOperation];
    [parseOperation release];
}
```

8.4. Sending messages to the main thread

When the parse operation is finished, it calls the delegate method `parseOperation:finished:`. The second parameter of this method is an array containing all incident objects. We must add these incident objects to the `incidents` array of the app delegate. However, the delegate method is called from a secondary thread. It is very dangerous to access shared memory from multiple threads. The `incidents` array should only be accessed by one thread at a time. Just like any other modern programming language, Objective-C supports the usage of locks to synchronize code execution in crucial parts of your application. However, it is sometimes easier and even better to simply force the execution of a message on the main thread. Whenever you modify application data from a secondary thread you should - if possible - forward this message to the main thread.

First, implement a method that can add entries from a given array to the `incidents` array:

```
- (void)handleLoadedIncidents:(NSArray *)loadedIncidents
{
    if ([loadedIncidents count] > 0) {
        [self.incidents addObjectsFromArray:loadedIncidents];
    }
    [[UIApplication sharedApplication] setNetworkActivityIndicatorVisible:NO];
}
```

Then implement the delegate method, which will call the above method **on the main thread**.

```
- (void)incidentsParseOperation:(TWIncidentsParseOperation *)parseOperation
                       finished:(NSArray *)parsedIncidents
{
```

```
    [self performSelectorOnMainThread:@selector(handleLoadedIncidents:)
                           withObject:parsedIncidents
                        waitUntilDone:NO];
    self.operationQueue = nil;
}
```

When you build, fix and run your application now, you will be surprised to see that no incidents appear in the table. This is because the main thread reloads the table view before the secondary thread has downloaded and processed the RSS feed.

8.5. Key-Value-Observing

In Cocoa objects can observe other object's key-value pairs. If such a pair is changed during runtime, the observing objects receive a notification to react on the change. We will use this method to reload the table view each time the parser adds a new object to the array.

When using KVC compliant properties key-value-observing (KVO) is usually automatically enabled. However, when adding or removing objects from an array, you must tigger the notification mechanism manually.

8.5.1. Trigger change notifications

Open TrafficWatchAppDelegate.m and change the `handleLoadedIncidents:` method as follows:

```
- (void)handleLoadedIncidents:(NSArray *)loadedIncidents
{
    if ([loadedIncidents count] > 0) {
        NSRange changedRange = NSMakeRange([self.incidents count], [loadedIncidents
            count]);
        NSIndexSet *changedIndexes = [NSIndexSet indexSetWithIndexesInRange:
            changedRange];
        [self         willChange:NSKeyValueChangeInsertion
                 valuesAtIndexes:changedIndexes
                          forKey:@"incidents"];
        [self.incidents addObjectsFromArray:loadedIncidents];
        [self          didChange:NSKeyValueChangeInsertion
                 valuesAtIndexes:changedIndexes
                          forKey:@"incidents"];
    }
    [[UIApplication sharedApplication] setNetworkActivityIndicatorVisible:NO];
}
```

These methods are declared in the **`NSKeyValueObserving`** protocol (just in case you want to look them up in the documentation - you should).

8.5.2. Observe key-value changes

The change notifications are only sent to those objects that explicitly registered as an observer for them. In our case **`TrafficWatchViewController`** shall be notified when new incidents arrive in the array. Add the following two methods to TrafficWatchViewController.m:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    [self.tableView reloadData];
        TrafficWatchAppDelegate *appDelegate;
    appDelegate = (TrafficWatchAppDelegate *)[[UIApplication sharedApplication]
        delegate];
    [appDelegate addObserver:self
                  forKeyPath:@"incidents"
                     options:NSKeyValueObservingOptionNew
                     context:NULL];
}

- (void)viewWillDisappear:(BOOL)animated
{
    TrafficWatchAppDelegate *appDelegate;
    appDelegate = (TrafficWatchAppDelegate *)[[UIApplication sharedApplication]
        delegate];
    [appDelegate removeObserver:self forKeyPath:@"incidents"];
    [super viewWillDisappear:animated];
}
```

`viewWillAppear:` and `viewWillDisappear:` are inherited from **`UIViewController`** and are called whenever the view controller's view becomes visible on screen, or disappears respectivley.

8.5.3. Handle the change notification

Add the following method to TrafficWatchViewController.m:

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context
{
    if ([keyPath isEqualToString:@"incidents"]) {
        [self.tableView reloadData];
        return;
    } else {
        // be sure to call the super implementation
        // if the superclass implements it
        [super observeValueForKeyPath:keyPath
                             ofObject:object
                               change:change
                              context:context];
    }
}
```

This will do the trick. When you build and run your application, the incidents feed is downloaded and parsed in the background. When finished the table view will display all incidents at once.

KVO is an extremely versatile and powerful technology. When an object receives a change notification it is also informed what kind of change took place (insertion, deletion, etc.). Additionally, if a variable changes, you can define whether the new value, the old value or both shall be included in the notification message. Furthermore, you can even define an arbitrary *context*, which is relayed to the observer. To

learn more about KVO, read the *Key-Value Observing Programming Guide*[2].

8.6. An alternative: NSNotificationCenter

Besides KVO there exists another notification mechanism in Cocoa: **`NSNotificationCenter`**. Each application has such a notification center built in. You can define a notification object and post it to the notification center. Then, each object in your application that previously registered as an observer for this particular notification will be informed.

You should use the notification center, if KVO is not practical or unnecessarily complicates the code. **`UIDevice`** for example defines **`UIDeviceOrientationDidChangeNotification`** which informs observers about a rotation of the device.

We will now change our code to use the shared notification center instead of KVO. Therefore, save your project, close Xcode and create a copy of the project's folder.

8.6.1. Post a notification

You are going to name the notification `@"TWIncidentsDidChangeNotification"`, but you are going to create a global variable for the constant. Open TrafficWatchAppDelegate.h and add the declaration (right below the `#import` statements:

```
extern NSString * const TWIncidentsDidChangeNotification;
```

In TrafficWatchAppDelegate.m define the constant (again right below the `#import` statements):

```
NSString * const TWIncidentsDidChangeNotification = @"
    TWIncidentsDidChangeNotification";
```

Notifications are usually named in this fashion:

```
<Prefix><Class><Event>Notification
```

Next, change `handleLoadedIncidents:` once more.

```
1  - (void)handleLoadedIncidents:(NSArray *)loadedIncidents
2  {
3      if ([loadedIncidents count] > 0) {
4          [self.incidents addObjectsFromArray:loadedIncidents];
5          NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
6          [nc postNotificationName:TWIncidentsDidChangeNotification
7                            object:nil];
8      }
9      [[UIApplication sharedApplication] setNetworkActivityIndicatorVisible:NO];
10 }
```

8.6.2. Register as an observer

Edit `viewWillAppear:` and `viewWillDisappear:` in TrafficWatchViewController.m:

[2] `http://developer.apple.com/iphone/prerelease/library/documentation/Cocoa/Conceptual/KeyValueObserving/`

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    [self.tableView reloadData];

    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc addObserver:self
           selector:@selector(handleIncidentsDidChange:)
               name:TWIncidentsDidChangeNotification
             object:nil];
}

- (void)viewWillDisappear:(BOOL)animated
{
    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc removeObserver:self
                  name:TWIncidentsDidChangeNotification
                object:nil];

    [super viewWillDisappear:animated];
}
```

8.6.3. Handle the notification

In the code snippet from above you told the notification center, that your observer will handle the notification using a selector `@selector(handleIncidentsDidChange:)`. Add the following method to TrafficWatchViewController.m to keep this promise:

```
- (void) handleIncidentsDidChange:(NSNotification *)theNotification
{
    [self.tableView reloadData];
}
```

Build, fix and run your application again.

8.7. Notifications and Threading

When Cocoa beginners (but not only them) start using threading and notifications, they often think, that notifications are useful to inform the main thread about progress from a secondary thread. Be warned: Notifications should only be sent between objects living on the same thread. The reason for this is obvious: Notifications fly around in your entire application and potentially cause data to be changed. If that happens on multiple threads simultaneously, keeping everything synchronized is very difficult (read: impossible). Therefore, a good rule of thumb is to only post notifications on the main thread. This includes both KVO notifications as well as notifications posted to the default notification center[3].

In our application we follow this rule, because the app delegate forces the execution of `handleLoadedIncidents:` on the main thread:

```
[self performSelectorOnMainThread:@selector(handleLoadedIncidents:)
                       withObject:parsedIncidents
                    waitUntilDone:NO];
```

[3]What you could do, of course, is to write your own thread safe subclass of **`NSNotificationCenter`**.

8.8. Challenges

You now have seen many of the most important techniques in Cocoa.

It is now time to train your new skills . Working independently on your project is the only way to really dive into Cocoa.

We have collected some ideas in that chapter that you might want to implement. They are all pretty easy but will most certainly require some reading in Apple's documentation.

8.8.1. Support autorotation

Mandatory.

That's a one liner: Return `YES` in `shouldAutorotateToInterfaceOrientation:`. Check if your table view cells mess up their layout during rotation - they should not!

8.8.2. Display the street icons in the table view cell

Mandatory.

Some of the RSS feed entries include an url to an icon image. You can download this image, store it in the incident object and then display it in your table view. The cell type you are using already features an image view. All you need to figure out, is how to get the image URL from the RSS feed. Take a look at how the web link is found and then use `[NSString rangeOfString:aString];` to identify the correct URL. You can download an image using the following code snippet:

```
- (UIImage *)downloadImageAtURL:(NSURL *)imageURL
{
    NSURLRequest *URLRequest = [NSURLRequest requestWithURL:imageURL];
    NSURLResponse *URLResponse;
    NSError *error;
    NSData *imageData = [NSURLConnection sendSynchronousRequest:URLRequest
                                              returningResponse:&URLResponse
                                                          error:&error];
    if (imageData && !error) {
        UIImage *icon = [UIImage imageWithData:imageData];
        return icon;
    }
    return nil;
}
```

However, the code above uses the synchronous API of **`NSURLConnection`**, i.e. the current thread will be blocked while the image is downloaded. As a result, if you execute this method on the main thread, the UI will not respond to user events during download. If you call this method on the parser thread, the parsing will take considerably longer. In either case, the user will not be very happy.

Here are some requirements you should consider, when implementing the icon download.

- The icon download should be triggered by **`TWTrafficWatchViewController`**.
- Use the asynchronous API of **`NSConnection`** to perform the downloads in the background while the UI remains responsive.

- New downloads should be triggered only when the table view is not currently scrolling and only for those cells that are currently visible on the screen. Otherwise the downloads would significantly impact the scrolling performance.

There actually is an Apple Sample Code available, which presents a similiar solution to a different problem. You can check it out, if you need to.[4]

In order to make the image view visible on your cell, you need to adjust the layout a bit. Adding 80 pixels to the left spacing of your text fields should do the trick. Then you can display the image (assuming `roadIcon` is a property of type `UIImage` that holds the image):

```
cell.imageView.image = incidentForRow.roadIcon;
```

8.8.3. Sort the incidents by distance

Optional.

The web link URL includes the GPS coordinates of the incidents. You can use this information, to sort the incidents in the order of their distance from your current location. The **`CoreLocation`** framework includes all you need for this. The framework needs to be added to your project, before you can use it. To do so, follow these steps:

1. Select your project in the Navigation area (left) to open the project editor.
2. Select the TrafficWatch target and click Build Phases at the top of the project editor.
3. Open the Link Binary With Libraries section.
4. Click the Add (+) button to add a library or framework.
5. Look for CoreLoaction.framework and click Add.

In order to use classes and methods from this framework, you need to import its header files.

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

Sorting is done using **`NSArray`**'s `sortedArrayUsingDescriptors`. Read the documentation on sort descriptors! They take full advantage of key-value coding.

CoreLocation can calculate the distance between two GPS coordinates.

8.8.4. Sort incidents by date

Optional.

Same as above, but using the incidents' timestamp information. You can convert the string encoded date to an actual **`NSDate`** object to simplify this task (this might be really tricky).

[4]The sample code is called LazyTableImages.

A. Memory Management

For a very good introduction to memory management see `http://www.cocoadevcentral.com/d/learn_objectivec/`, chapter Memory Management.

The **iPhone OS** does not support garbage collection yet. Therefore it is your job to keep track of your application's memory usage. This is not as difficult as it might sound in the first place.

A.1. retain and release

Every object in Objective-C has a *retain count*. This is simply a positive integer. When an object is created in memory with the **`alloc`** method, its retain count is set to 1. If an object's retain count becomes zero, the object is removed (deallocated) from memory. To increase an object's retain count by one, send it the **`retain`** message. You decrement the retain count by sending **`release`** to the object.

An object's retain count represents the number of objects, that have a reference to that object.

For example, take the simple model in Figure A.1: The book object is referenced by the author, a publisher and a reader. Each of them has a reference to the book. As a result the book's retain count must be 3. When the reader and the publisher are no longer interested in the book, they will send it the **`release`** message, decreasing the retain count to 1. Eventually, even the author looses interest in his own work by sending the **`release`** message to the book object. The book's retain count now is zero and the book is deallocated from memory.

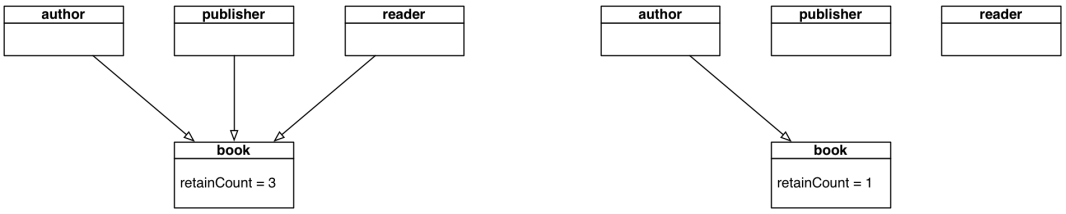


Figure A.1.: The Retain Count

A.2. Pitfalls

Using this method of memory management, two problems can occur:

1. An object is never sent a release message and therefore stays in memory until the application finishes (memory leak).

2. An object is sent the release message too often and is therefore removed from memory although another object still has a reference to it (over-release).

The first problem will cause your application to waste memory. In the worst case, e. g. if the leak occurs in a repeating block of code such as a loop, this leakage can cause your application to run out of memory. Remember that the iPhone only has limited resources, therefore memory leaks should generally be avoided!

The second problem will with almost certainty cause your application to crash as soon as some object tries to access the over-released object.

The basic rules to avoid these problems are:

1. If you **retain**'ed an object, it is your obligation to **release** it.
2. If you **copy**'ed an object, it is also your obligation to **release** it.
3. Only **release** an object, if you **retain**'ed or **copy**'ed it before.

Some actions automatically retain an object. For example, when you create a new object using the **alloc** method, its initial retain count is set to 1. Therefore you are responsible for releasing the object, if you don't need it anymore.

A.2.1. Example: memory leak

Take a look at the following method:

```
- (void) exampleMethod
{
    NSMutableArray *numbers;
    int i;

    numbers = [[NSMutableArray alloc] init];

    for (i = 0; i < 10; i++) {
       NSNumber *aNumber = [[NSNumber alloc] initWithInt:i];
       [numbers addObject:aNumber];
    }

    // do something with |numbers|

    [numbers release];
}
```

This code fragment creates an instance of **NSMutableArray** and adds 10 **NSNumber** objects to it. When the method is done, it releases the `numbers` array correctly. However, the for-loop leaks a lot of **NSNumber** objects. Note, that we use **alloc** to create a number in each step of the loop, but never release it. This is a violation of the first rule. To fix this, we must release the **NSNumber** object in each step:

```
    for (i = 0; i < 10; i++) {
       NSNumber *aNumber = [[NSNumber alloc] initWithInt:i];
       [numbers addObject:aNumber];
       [aNumber release]
    }
```

Note that when adding the number to the array (`[numbers addObject:aNumber];`), the array will of course retain the number in order to ensure a valid reference. In general objects that are added to an array, are released when the array is removed from memory.

A.2.2. Example: over-release

Imagine your application has the method **`setupCar:`** which takes a car object as the first argument and edits some of its properties. Another method, **`exampleMethod`**, calls **`setupCar:`**.

```
1  - (void) setupCar:(Car *)theCar
2  {
3      [theCar setVendor:@"BMW"];
4      [theCar setLicencePlate:@"M TU 101"];
5      [theCar release];            // WRONG
6  }
7
8  - (void) exampleMethod
9  {
10     Car *newCar = [[Car alloc] init];
11     [self setupCar:newCar];
12     [newCar doSomething];        // CRASH
13     [newCar release];
14 }
```

In line 12 this example will crash, because **`newCar`** isn't valid anymore. The line `[theCar release];` in **`setupCar:`** is wrong. **`theCar`** is not sent **`retain`** in **`setupCar:`** therefore you must not release it there.

A.3. autorelease

Sometimes you cannot fulfill your duty to release a previously retained object. In this case you should send an **`autorelease`** message to the object, adding it to an autorelease pool. An autorelease pool is an instance of **`NSAutoreleasePool`** that "contains" other objects that have received an **`autorelease`** message; when the autorelease pool is deallocated it sends a **`release`** message to each of those objects. An object can be put into an autorelease pool several times, and receives a **`release`** message for each time it was put into the pool. Thus, sending **`autorelease`** instead of **`release`** to an object extends the lifetime of that object at least until the pool itself is released (the object may survive longer if it is retained in the interim).

Think of **`autorelease`** as: *Release, but not now!*

A.3.1. Example: Returning autoreleased objects

The **`description`** method returns a **`NSString`** describing the sender.

```
1  - (NSString *) description
2  {
3      NSString *theDescription = [[NSString alloc] initWithFloat:self.topSpeed];
4      return theDescription;
5  }
```

In this example you create the object **theDescription** but fail to release it, thus violating the second rule. But you cannot use **[theDescription release]** here either, because this would remove your object from memory before it is return to the receiver. Instead use **[theDescription autorelease]**:

```
- (NSString *) description
{
    NSString *theDescription = [[NSString alloc] initWithFloat:self.topSpeed];
    return [theDescription autorelease];
}
```

A.4. Further reading

For a more detailed instruction on memory management, including many code examples, it is strongly recommended to read Apple's *Memory Management Programming Guide for Cocoa*.[1]

[1] http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/

B. Document History

This table describes the changes to the *TrafficWatch Tutorial*.

Date	Notes
2011-05-25	Updated chapter 8 for Xcode 4 replacing **`NSThread`** by **`NSOperation`**.
2011-05-19	Updated chapters 5-7 for Xcode 4
2011-05-15	Updated chapters 1-4 for Xcode 4
	Added appendix *Memory Management*.
2010-05-06	Respects new API: `application:didFinishLaunchingWithOptions:` (thanks to F. Kaser).
	Corrected several typos (thanks to F. Kaser).
2010-05-04	Changed deprecated call in Listing 3.2.1, using `cell.textLabel.text` instead of `cell.text`.
2010-04-22	Updated contact information.
2010-04-22	Updated document for new course in summer semester 2010.
	Added sidenote on properties.
	Fixed minor typos.
2009-08-05	Fixed listing 8.5.1 in section 8.5.1 to trigger a **`NSKeyValueChangeInsertion`** notification.
2009-05-22	Added chapter 8.
	Fixed some typos here and there.
2009-05-16	Added chapter 6.
	Added chapter 7.
2009-05-09	Added chapter 5.
2009-05-04	Corrected a typo in the method signature in section 4.4 on page 22.
2009-05-01	Added chapter 4.
2009-04-27	Initial release