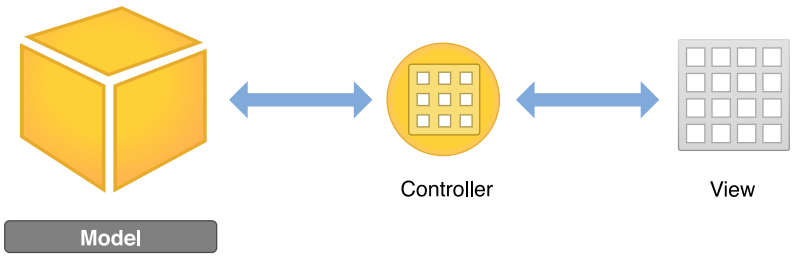
Implementing an App

Incorporating the Data

Your app's **data model** is composed of your data structures and (optionally) custom business logic needed to keep that data in a consistent state. You never want to design your data model in total isolation from your app's user interface. You do, however, want to implement your data model objects separately, without relying on the presence of specific views or view controllers. When you keep your data separate from your user interface, you'll find it easier to implement a universal app—one that can run on both iPad and iPhone—and easier to reuse portions of your code later.



Designing Your Model

If you simply need to store a small amount of data, Foundation framework classes may be your best option. Research existing Foundation classes to see what behaviors are available for you to use instead of attempting to implement the same thing on your own. For example, if your app only needs to keep track of a list of strings, you can rely on `NSArray` and `NSString` to do the job for you. You'll learn more about these and other Foundation classes in "Working with Foundation" (page 75).

If your data model requires custom business logic in addition to just storing data, you can write a custom class. Consider how you can incorporate existing framework classes into the implementation of your own classes. It's beneficial to use existing framework classes within your custom classes instead of trying to reinvent them. For example, a custom class might use `NSMutableArray` to store information—but define its own features for working with that information.

When you design your data model, here are some questions to keep in mind:

What types of data do you need to store? Whether you're storing text, documents, large images, or another type of information, design your data model to handle that particular type of content appropriately.

What data structures can you use? Determine where you can use framework classes and where you need to define classes with custom functionality.

How will you supply data to the user interface? Your model shouldn't communicate directly with your interface. To handle the interaction between the model and the interface, you'll need to add logic to your controllers.

Implementing Your Model

To write good, efficient code, you need to learn more about Objective-C and its capabilities. Although this guide teaches you how to build a simple app, you'll want to become familiar with the language before writing your own fully functional app.

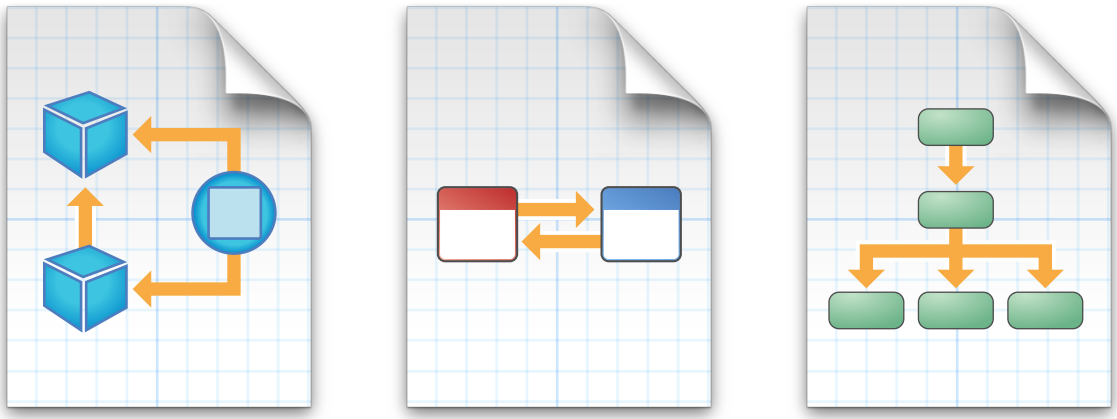
There are several good approaches to learning Objective-C. Some people learn the concepts by reading *Programming with Objective-C* and then writing a number of small test apps to solidify their understanding of the language and to practice writing good code.

Others jump right into programming and look for more information when they don't know how to accomplish something. If you prefer this approach, keep *Programming with Objective-C* as a reference and make it an exercise to learn concepts and apply them to your app as you develop it.

The most important goal in developing your first data model is to get something that works. Think carefully about the structure of your data model, but don't worry about making it perfect. Don't be afraid to iterate and refine your model after you begin implementing it.

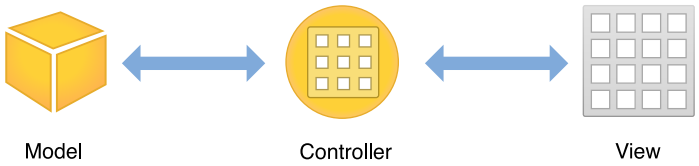
Using Design Patterns

A **design pattern** solves a common software engineering problem. Patterns are abstract designs, not code. When you adopt a design, you adapt its general pattern to your specific needs. No matter what type of app you're creating, it's good to know the fundamental design patterns used in the frameworks. Understanding design patterns helps you use frameworks more effectively and allows you to write apps that are more reusable, more extensible, and easier to change.



MVC

Model-View-Controller (MVC) is central to a good design for any iOS app. MVC assigns the objects in an app to one of three roles: model, view, or controller. In this pattern, models keep track of your app's data, views display your user interface and make up the content of an app, and controllers manage your views. By responding to user actions and populating the views with content, controllers serve as a gateway for communication between models and views.

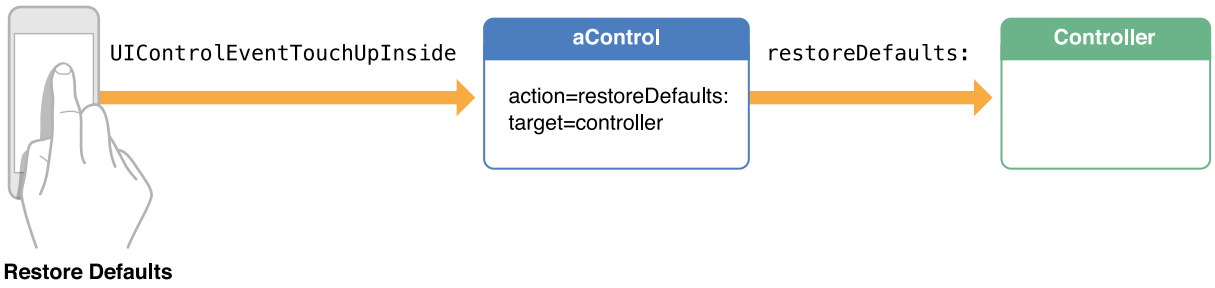


As you've been building your ToDoList app, you've followed an MVC-centric design. The interface you built in storyboards makes up the view layer. `XYZAddToDoItemViewController` and `XYZToDoListViewController` are the controllers that manage your views. In "Tutorial: Add Data" (page 93), you'll be incorporating a data model to work with the views and controllers in your app. When you begin designing your own app, it's important to keep MVC at the center of your design.

Target-Action

Target-action is a conceptually simple design in which one object sends a message to another object when a specific event occurs. The **action message** is a selector defined in source code, and the **target**—the object that receives the message—is an object capable of performing the action, typically a view controller. The object that sends the action message is usually a control—such as a button, slider, or switch—that can trigger an event in response to user interaction such as tap, drag, or value change.

For example, imagine that you want to restore default settings in your app whenever a user taps the Restore Defaults button (which you create in your user interface). First, you implement an action, `restoreDefaults:`, to perform the logic to restore default settings. Next, you register the button's Touch Up Inside event to send the `restoreDefaults:` action method to the view controller that implements that method.

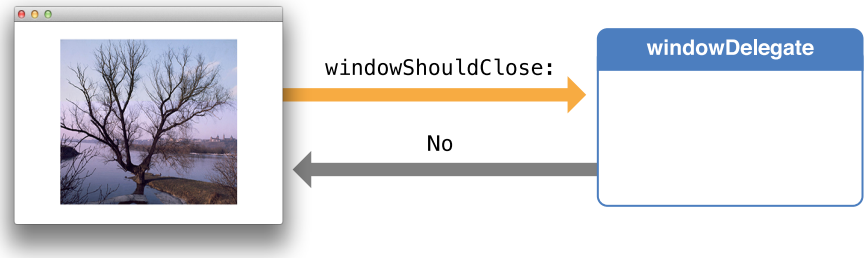


You've already used target-action in your ToDoList app. When a user taps the Done button in the `XYZAddToDoItemViewController`, it triggers the `unwindToList:` action. In this case, the Done button is the object sending the message, the target object is the `XYZToDoListViewController`, the action message is `unwindToList:`, and the event that triggers the action message to be sent is a user tapping the Done button. Target-action is a powerful mechanism for defining interaction and sending information between different parts of your app.

Delegation

Delegation is a simple and powerful pattern in which one object in an app acts on behalf of, or in coordination with, another object. The delegating object keeps a reference to the other object—the delegate—and at the appropriate time sends a message to it. The message informs the delegate of an event that the delegating

object is about to handle or has just handled. The delegate may respond to the message by updating the appearance (or state) of itself or of other objects in the app, and in some cases it will return a value that affects how an impending event is handled.



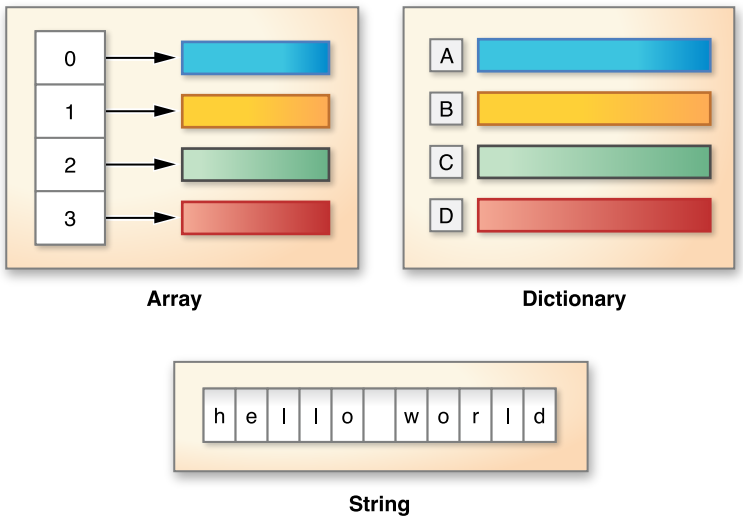
The delegate pattern is prevalent in existing framework classes, but you can also implement delegation between two custom objects in an app. A common design uses delegation as a means for allowing a child view controller to communicate some value (typically a user-entered value) to its parent view controller.

You haven't worked with delegation yet, but in "Tutorial: Add Data" (page 93), you'll see an example of it when you add additional behavior to your `XYZToDoListViewController` class.

These are a few of the most common design patterns that you'll encounter during iOS development, but there are many more. As you learn more about Objective-C, you'll spot other design patterns that you can apply in your app.

Working with Foundation

As you begin writing code for your app, you'll find that there are many Objective-C frameworks that you can take advantage of. Of particular importance is the **Foundation framework**, which provides basic services for all apps. The Foundation framework includes **value classes** representing basic data types such as strings and numbers, as well as **collection classes** for storing other objects. You'll be relying on value and collection classes to write much of the code for your ToDoList app.



Value Objects

The Foundation framework provides you with classes that generate value objects for strings, binary data, dates and times, numbers, and other values.

A **value object** is an object that encapsulates a primitive value (of a C data type) and provides services related to that value. You frequently encounter value objects as the parameters and return values of methods and functions that your app calls. Different parts of a framework—or even different frameworks—can exchange data by passing value objects.

Some examples of value objects in the Foundation framework are:

- `NSString` and `NSMutableString`
- `NSData` and `NSMutableData`

- `NSDate`
- `NSNumber`
- `NSValue`

Because value objects represent scalar values, you can use them in collections and wherever else objects are required. Value objects have an advantage over the primitive types they encapsulate: They let you perform certain operations on the encapsulated value simply and efficiently. The `NSString` class, for example, has methods for searching for and replacing substrings, for writing strings to files or (preferably) URLs, and for constructing file-system paths.

You create a value object from data of the primitive type (and then perhaps pass it in a method parameter). In your code, you later access the encapsulated data from the object. The `NSNumber` class provides the clearest example of this approach.

```
int n = 5; // Value assigned to primitive type
NSNumber *numberObject = [NSNumber numberWithInt:n]; // Value object created from
 primitive type
int y = [numberObject intValue]; // Encapsulated value obtained from value object
 (y == n)
```

Most value classes create their instances by declaring both initializers and class factory methods. **Class factory methods**—implemented by a class as a convenience for clients—combine allocation and initialization in one step and return the created object. For example, the `NSString` class declares a `string` class method that allocates and initializes a new instance of the class and returns it to your code.

```
NSString *string = [NSString string];
```

In addition to creating value objects and letting you access their encapsulated values, most value classes provide methods for simple operations such as object comparison.

Strings

Objective-C supports the same conventions for specifying strings as does C: Single characters are enclosed by single quotes, and strings of characters are surrounded by double quotes. But Objective-C frameworks typically don't use C strings. Instead, they use `NSString` objects.

The NSString class provides an object wrapper for strings, offering advantages such as built-in memory management for storing arbitrary-length strings, support for different character encodings (particularly Unicode), and utilities for string formatting. Because you commonly use such strings, Objective-C provides a shorthand notation for creating NSString objects from constant values. To use this NSString literal, just precede a double-quoted string with the at sign (@), as shown in the following examples:

```
// Create the string "My String" plus carriage return.
NSString *myString = @"My String\n";
// Create the formatted string "1 String".
NSString *anotherString = [NSString stringWithFormat:@"%d %@", 1, @"String"];
// Create an Objective-C string from a C string.
NSString *fromCString = [NSString stringWithCString:"A C string"
encoding:NSUTF8StringEncoding];
```

Numbers

Objective-C offers a shorthand notation for creating NSNumber objects, removing the need to call initializers or class factory methods to create such objects. Simply precede the numeric value with the at sign (@) and optionally follow it with a value type indicator. For example, to create NSNumber objects encapsulating an integer value and a double value, you could write the following:

```
NSNumber *myIntValue    = @32;
NSNumber *myDoubleValue = @3.22346432;
```

You can even use NSNumber literals to create encapsulated Boolean and character values.

```
NSNumber *myBoolValue = @YES;
NSNumber *myCharValue = @'V';
```

You can create NSNumber objects representing unsigned integers, long integers, long long integers, and float values by appending the letters U, L, LL, and F, respectively, to the notated value. For example, to create an NSNumber object encapsulating a float value, you can write the following:

```
NSNumber *myFloatValue = @3.2F
```

Collection Objects

Most **collection objects** in Objective-C code are instances of one of the basic collection classes—`NSArray`, `NSSet`, and `NSDictionary`. These classes are used to manage groups of objects, so any item you want to add to a collection must be an instance of an Objective-C class. If you need to add a scalar value, you must first create a suitable `NSNumber` or `NSValue` instance to represent it.

Any object you add to a collection will be kept alive at least as long as the collection is kept alive. That's because collection classes use strong references to keep track of their contents. In addition to keeping track of their contents, each of the collection classes makes it easy to perform certain tasks, such as enumeration, accessing specific items, or finding out whether a particular object is part of the collection.

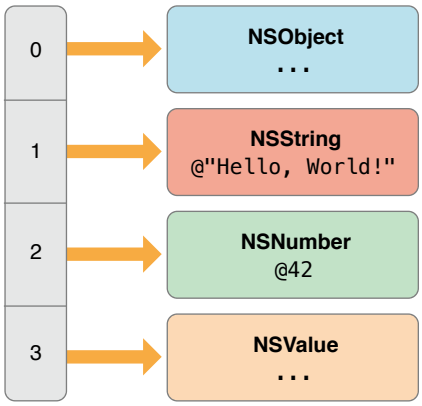
The contents of the `NSArray`, `NSSet`, and `NSDictionary` classes are set at creation. Because they can't be changed over time, they're called **immutable**. Each one also has a subclass that's **mutable** to allow you to add or remove objects at will. Different types of collections organize their contained objects in distinctive ways:

- `NSArray` and `NSMutableArray`—An array is an ordered collection of objects. You access an object by specifying its position (that is, its index) in the array. The first element in an array is at index 0 (zero).
- `NSSet` and `NSMutableSet`—A set stores an unordered collection of objects, with each object occurring only once. You generally access objects in the set by applying tests or filters to objects in the set.
- `NSDictionary` and `NSMutableDictionary`—A dictionary stores its entries as key-value pairs; the key is a unique identifier, usually a string, and the value is the object you want to store. You access this object by specifying the key.

Arrays

An **array** (`NSArray`) is used to represent an ordered list of objects. The only requirement is that each item be an Objective-C object—there's no requirement for each object to be an instance of the same class.

To maintain order in the array, each element is stored at a zero-based index.



Creating Arrays

As with the value classes described earlier in this chapter, you can create an array through allocation and initialization, class factory methods, or array literals.

There are a variety of different initialization and factory methods available, depending on the number of objects.

```
+ (id)arrayWithObject:(id)anObject;
+ (id)arrayWithObjects:(id)firstObject, ...;
- (id)initWithObjects:(id)firstObject, ...;
```

Because the `arrayWithObjects:` and `initWithObjects:` methods both take a `nil`-terminated, variable number of arguments, you must include `nil` as the last value.

```
NSArray *someArray =
[NSArray arrayWithObjects:someObject, someString, someNumber, someValue, nil];
```

This example creates an array like the one shown earlier. The first object, `someObject`, will have an array index of `0`; the last object, `someValue`, will have an index of `3`.

It's possible to truncate the list of items unintentionally if one of the provided values is `nil`.

```
id firstObject = @"someString";
id secondObject = nil;
id thirdObject = @"anotherString";
NSArray *someArray =
  [NSArray arrayWithObjects:firstObject, secondObject, thirdObject, nil];
```

In this case, `someArray` contains only `firstObject`, because `secondObject`, which is `nil`, is interpreted as the end of the list of items.

It's possible to create an array literal using a compact syntax.

```
NSArray *someArray = @[firstObject, secondObject, thirdObject];
```

When using this syntax, don't terminate the list of objects with `nil`—in fact, `nil` is an invalid value. For example, you'll get an exception at runtime if you try to execute the following code:

```
id firstObject = @"someString";
```

```
id secondObject = nil;
NSArray *someArray = @[firstObject, secondObject];
// exception: "attempt to insert nil object"
```

Querying Array Objects

After you've created an array, you can query it for information—such as how many objects it has or whether it contains a given item.

```
NSUInteger numberOfItems = [someArray count];

if ([someArray containsObject:someString]) {
    ...
}
```

You can also query the array for an item at a given index. If you attempt to request an invalid index, you'll get an out-of-bounds exception at runtime. To avoid getting an exception, always check the number of items first.

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", [someArray objectAtIndex:0]);
}
```

This example checks to see whether the number of items is greater than zero. If it is, the Foundation function `NSLog` logs a description of the first item, which has an index of `0`.

As an alternative to using `objectAtIndex:`, you can also query the array using a subscript syntax, which is just like accessing a value in a standard C array. The previous example can be rewritten like this:

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", someArray[0]);
}
```

Sorting Array Objects

The `NSArray` class offers a variety of methods to sort its collected objects. Because `NSArray` is immutable, each of these methods returns a new array containing the items in the sorted order.

For example, you can sort an array of strings by calling `compare:` on each string.

```
NSArray *unsortedStrings = @[@"gammaString", @"alphaString", @"betaString"];
NSArray *sortedStrings =
         [unsortedStrings sortedArrayUsingSelector:@selector(compare:)];
```

Mutability

Although the `NSArray` class itself is immutable, it can nevertheless contain mutable objects. For example, if you add a mutable string to an immutable array, like this:

```
NSMutableString *mutableString = [NSMutableString stringWithString:@"Hello"];
NSArray *immutableArray = @[mutableString];
```

there's nothing to stop you from mutating the string.

```
if ([immutableArray count] > 0) {
    id string = immutableArray[0];
    if ([string isKindOfClass:[NSMutableString class]]) {
        [string appendString:@" World!"];
    }
}
```

If you want to add or remove objects from an array after initial creation, use `NSMutableArray`, which adds a variety of methods to add, remove, or replace one or more objects.

```
NSMutableArray *mutableArray = [NSMutableArray array];
[mutableArray addObject:@"gamma"];
[mutableArray addObject:@"alpha"];
[mutableArray addObject:@"beta"];

[mutableArray replaceObjectAtIndex:0 withObject:@"epsilon"];
```

This example creates an array made up of the objects `@"epsilon"`, `@"alpha"`, and `@"beta"`.

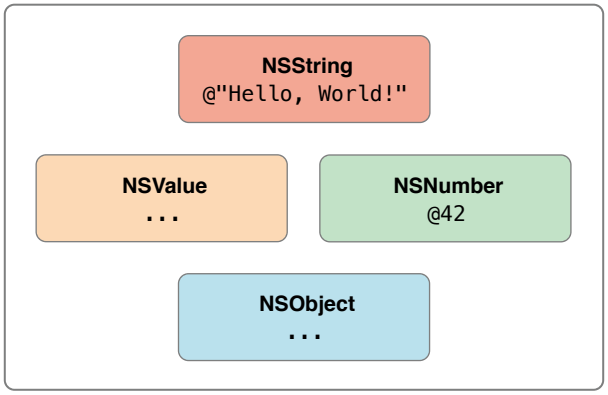
It's also possible to sort a mutable array in place, without creating a secondary array.

```
[mutableArray sortUsingSelector:@selector(caseInsensitiveCompare:)];
```

In this case, the contained items are sorted into the ascending, case-insensitive order `@"alpha"`, `@"beta"`, and `@"epsilon"`.

Sets

A **set** (`NSSet`) object is similar to an array, but it maintains an unordered group of distinct objects.



Because sets don't maintain order, they offer faster performance than arrays do when it comes to testing for membership.

Because the basic `NSSet` class is immutable, its contents must be specified at creation, using either allocation and initialization or a class factory method.

```
NSSet *simpleSet =
  [NSSet setWithObjects:@"Hello, World!", @42, aValue, anObject, nil];
```

As with `NSArray`, the `initWithObjects:` and `setWithObjects:` methods both take a `nil`-terminated, variable number of arguments. The name of the mutable `NSSet` subclass is `NSMutableSet`.

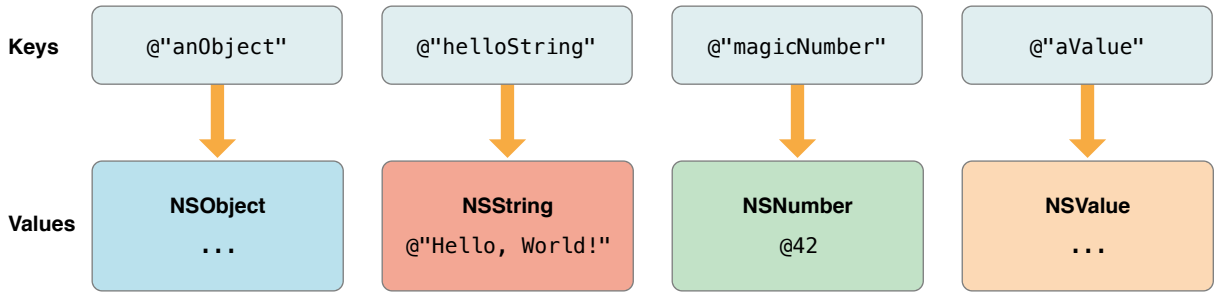
Sets store only one reference to an individual object, even if you try adding an object more than once.

```
    NSNumber *number = @42;
    NSSet *numberSet =
      [NSSet setWithObjects:number, number, number, number, nil];
    // numberSet only contains one object
```

Dictionaries

Rather than simply maintaining an ordered or unordered collection of objects, a **dictionary** (`NSDictionary`) stores objects associated with given keys, which can then be used for retrieval.

The best practice is to use string objects as dictionary keys.



Although you can use other objects as keys, keep in mind that each key is copied for use by a dictionary and so must support `NSCopying`. If you want to use key-value coding, however, you must use string keys for dictionary objects (to learn more, see *Key-Value Coding Programming Guide*).

Creating Dictionaries

You can create dictionaries using either allocation and initialization, or class factory methods, like this:

```
NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
                someObject, @"anObject",
          @"Hello, World!", @"helloString",
                      @42, @"magicNumber",
                 someValue, @"aValue",
                       nil];
```

For the `dictionaryWithObjectsAndKeys:` and `initWithObjectsAndKeys:` methods, each object is specified before its key, and the list of objects and keys must be `nil` terminated.

Objective-C offers a concise syntax for dictionary literal creation.

```
NSDictionary *dictionary = @{
               @"anObject" : someObject,
            @"helloString" : @"Hello, World!",
            @"magicNumber" : @42,
                 @"aValue" : someValue
```

```
};
```

For dictionary literals, the key is specified before its object, and the list of objects and keys is not `nil` terminated.

Querying Dictionaries

After you've created a dictionary, you can ask it for the object stored against a given key.

```
NSNumber *storedNumber = [dictionary objectForKey:@"magicNumber"];
```

If the object isn't found, the `objectForKey:` method returns `nil`.

There's also a subscript syntax alternative to using `objectForKey:`.

```
NSNumber *storedNumber = dictionary[@"magicNumber"];
```

Mutability

If you need to add or remove objects from a dictionary after creation, use the `NSMutableDictionary` subclass.

```
[dictionary setObject:@"another string" forKey:@"secondString"];
[dictionary removeObjectForKey:@"anObject"];
```

Represent nil with NSNull

It's not possible to add `nil` to the collection classes described in this section because `nil` in Objective-C means "no object." If you need to represent "no object" in a collection, use the `NSNull` class.

```
NSArray *array = @[ @"string", @42, [NSNull null] ];
```

With `NSNull`, the `null` method always returns the same instance. Classes that behave in this way are called **singleton classes**. You can check to see whether an object in an array is equal to the shared `NSNull` instance like this:

```
for (id object in array) {
    if (object == [NSNull null]) {
        NSLog(@"Found a null object");
    }
```

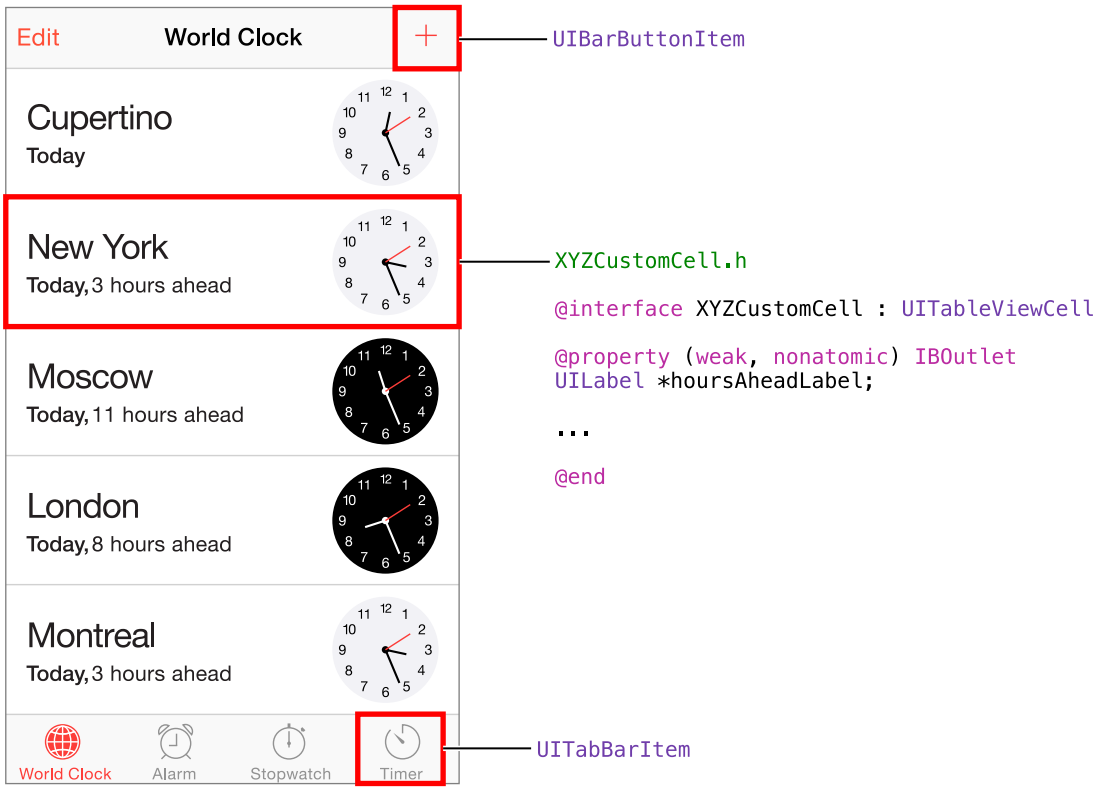
```
}
```

Although the Foundation framework contains many more capabilities than are described here, you don't need to know every single detail right away. If you do want to learn more about Foundation, take a look at *Foundation Framework Reference*. For now, you have enough information to continue implementing your ToDoList app, which you'll do by writing a custom data class.

Writing a Custom Class

As you develop iOS apps, you'll find many occasions when you need to write your own custom classes. Custom classes are useful when you need to package custom behavior together with data. In a custom class, you can define your own behaviors for storing, manipulating, and displaying your data.

For example, consider the World Clock tab in the iOS Clock app. The cells in this table view need to display more content than a standard table view cell. This is a good opportunity to implement a subclass that extends the behavior of `UITableViewCell` to let you display additional custom data for a given table view cell. If you were designing this custom class, you might add outlets for a label to display the hours ahead information and an image view to display the custom clock on the right of the cell.



This chapter teaches you what you need to know about Objective-C syntax and class structure to finish implementing the behavior of your ToDoList app. It discusses the design of `XYZToDoItem`, the custom class that will represent a single item on your to-do list. In the third tutorial, you'll actually implement this class and add it to your app.

Declaring and Implementing a Class

The specification of a class in Objective-C requires two distinct pieces: the interface and the implementation. The **interface** specifies exactly how a given type of object is intended to be used by other objects. In other words, it defines the public interface between instances of the class and the outside world. The **implementation** includes the executable code for each method declared in the interface.

An object should be designed to hide the details of its internal implementation. In Objective-C, the interface and implementation are usually placed in separate files so that you need to make only the interface public. As with C code, you define header files and source files to separate public declarations from the implementation details of your code. Interface files have a `.h` extension, and implementation files have a `.m` extension. (You'll actually create these files for the `XYZToDoItem` class in "Tutorial: Add Data" (page 93)—for now, just follow along as all of the pieces are introduced.)

Interface

The Objective-C syntax used to declare a class interface looks like this:

```
@interface XYZToDoItem : NSObject


@end
```

This example declares a class named `XYZToDoItem`, which inherits from `NSObject`.

The public properties and behavior are defined inside the `@interface` declaration. In this example, nothing is specified beyond the superclass, so the only behavior expected to be available on instances of `XYZToDoItem` is the behavior inherited from `NSObject`. All objects are expected to have a minimum behavior, so by default, they must inherit from `NSObject` (or one of its subclasses).

Implementation

The Objective-C syntax used to declare a class implementation looks like this:

```
#import "XYZToDoItem.h"


@implementation XYZToDoItem


@end
```

If you declare any methods in the class interface, you'll need to implement them inside this file.

Properties Store an Object's Data

Consider what information the to-do item needs to hold. You probably need to know its name, when it was created, and whether it's been completed. In your custom `XYZToDoItem` class, you'll store this information in **properties**.

Declarations for these properties reside inside the interface file (`XYZToDoItem.h`). Here's what they look like:

```
@interface XYZToDoItem : NSObject

@property NSString *itemName;
@property BOOL completed;
@property NSDate *creationDate;

@end
```

In this example, the `XYZToDoItem` class declares three public properties. These properties are available for full public access. With public access, other objects can both read and change the values of the properties.

You may decide to declare that a property shouldn't be changed (that is, that it should be read-only). To indicate whether a property is intended to be read-only—among other things—Objective-C property declarations include **property attributes**. For example, if you don't want the creation date of an `XYZToDoItem` to be changeable, you might update the `XYZToDoItem` class interface to look like this:

```
@interface XYZToDoItem : NSObject

@property NSString *itemName;
@property BOOL completed;
@property (readonly) NSDate *creationDate;

@end
```

Properties can be private or public. Sometimes it makes sense to make a property private so that other classes can't see or access it. For example, if you want to keep track of a property that represents the date an item was marked as completed without giving other classes access to this information, make the property private by putting it in a **class extension** at the top of your implementation file (`XYZToDoItem.m`).

```
#import "XYZToDoItem.h"
```

```
@interface XYZToDoItem ()
@property NSDate *completionDate;
@end

@implementation XYZToDoItem

@end
```

You access properties using getters and setters. A **getter** returns the value of a property, and a **setter** changes it. A common syntactical shorthand for accessing getters and setters is **dot notation**. For properties with read and write access, you can use dot notation for both getting and setting a property's value. If you have an object `toDoItem` of class `XYZToDoItem`, you can do the following:

```
toDoItem.itemName = @"Buy milk";                //Sets the value of itemName
NSString *selectedItemName = toDoItem.itemName;  //Gets the value of itemName
```

Methods Define an Object's Behavior

Methods define what an object can do. A **method** is a piece of code that you define to perform a task or subroutine in a class. Methods have access to data stored in the class and can use that information to perform some kind of operation.

For example, to give a to-do item (`XYZToDoItem`) the ability to get marked as complete, you can add a `markAsCompleted` method to the class interface. Later, you'll implement this method's behavior in the class implementation, as described in "Implementing Methods" (page 91).

```
@interface XYZToDoItem : NSObject

@property NSString *itemName;
@property BOOL completed;
@property (readonly) NSDate *creationDate;
- (void)markAsCompleted;

@end
```

The minus sign (–) at the front of the method name indicates that it is an **instance method**, which can be called on an object of that class. This minus sign differentiates it from class methods, which are denoted with a plus sign (+). **Class methods** can be called on the class itself. A common example of class methods are class factory methods, which you learned about in "Working with Foundation" (page 75). You can also use class methods to access some piece of shared information associated with the class.

The `void` keyword is used inside parentheses at the beginning of the declaration to indicate that the method doesn't return a value. In this case, the `markAsCompleted` method takes in no parameters. Parameters are discussed in more detail in "Method Parameters" (page 90).

Method Parameters

You declare methods with **parameters** to pass along some piece of information when you call a method.

For example, you can revise the `markAsCompleted` method from the previous code snippet to take in a single parameter that will determine whether the item gets marked as completed or uncompleted. This way, you can toggle the completion state of the item instead of setting it only as completed.

```
@interface XYZToDoItem : NSObject

@property NSString *itemName;
@property BOOL completed;
@property (readonly) NSDate *creationDate;
- (void)markAsCompleted:(BOOL)isComplete;

@end
```

Now, your method takes in one parameter, `isComplete`, which is of type `BOOL`.

When you refer to a method with a parameter by name, you include the colon as part of the method name, so the name of the updated method is now `markAsCompleted:`. If a method has multiple parameters, the method name is broken up and interspersed with the parameter names. If you wanted to add another parameter to this method, its declaration would look like this:

```
- (void)markAsCompleted:(BOOL)isComplete onDate:(NSDate *)date;
```

Here, the method's name is written as `markAsCompleted:onDate:`. The names `isComplete` and `date` are used in the implementation to access the values supplied when the method is called, as if these names were variables.

Implementing Methods

Method implementations use braces to contain the relevant code. The name of the method must be identical to its counterpart in the interface file, and the parameter and return types must match exactly.

Here is a simple implementation of the `markAsCompleted:` method you added to your `XYZToDoItem` class interface:

```
@implementation XYZToDoItem
- (void)markAsCompleted:(BOOL)isComplete {
    self.completed = isComplete;
}
@end
```

Like properties, methods can be private or public. Public methods are declared in the public interface and so can be seen and called by other objects. Their corresponding implementation resides in the implementation file and can't be seen by other objects. Private methods have only an implementation and are internal to the class, meaning they're only available to call inside the class implementation. This is a powerful mechanism for adding internal behavior to a class without allowing other objects access to it.

For example, say you want to keep a to-do item's `completionDate` updated. If the to-do item gets marked as completed, set `completionDate` to the current date. If it gets marked as uncompleted, set `completionDate` to `nil`, because it hasn't been completed yet. Because updating the to-do item's `completionDate` is a self-contained task, the best practice is to write its own method for it. However, it's important to make sure that other objects can't call this method—otherwise, another object could set the to-do item's `completionDate` to anything at any time. For this reason, you make this method private.

Now, update the implementation of `XYZToDoItem` to include the private method `setCompletionDate` that gets called inside `markAsCompleted:` to update the to-do item's `completionDate` whenever it gets marked as completed or uncompleted. Notice that you're not adding anything to the interface file, because you don't want other objects to see this method.

```
@implementation XYZToDoItem
- (void)markAsCompleted:(BOOL)isComplete {
    self.completed = isComplete;
    [self setCompletionDate];
}
- (void)setCompletionDate {
    if (self.completed) {
        self.completionDate = [NSDate date];
```

```
    } else {
        self.completionDate = nil;
    }
}
@end
```

At this point, you've defined a basic representation of a to-do list item using the `XYZToDoItem` class. `XYZToDoItem` stores information about itself—name, creation date, completion state—in the form of properties, and it defines what it can do—get marked as completed or uncompleted—using a method. This is the extent of the features you need to finish implementing your ToDoList app in the next tutorial. However, you can always experiment by adding your own properties and methods to the class to integrate new behavior into your app.

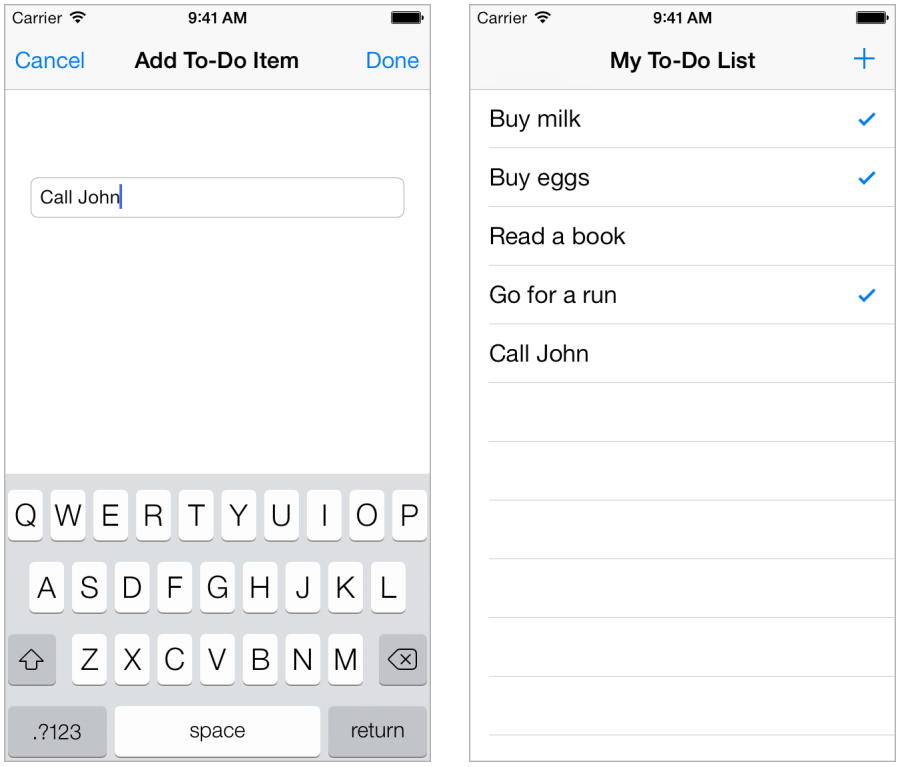
Tutorial: Add Data

This tutorial builds on the project you created in the second tutorial ("Tutorial: Storyboards" (page 47)). You'll use what you learned about using design patterns, working with Foundation, and writing a custom class to add support for dynamic data to your ToDoList app.

This tutorial teaches you how to:

- Work with common Foundation classes
- Create custom data classes
- Implement a delegate and data source protocol
- Pass data between view controllers

After you complete all the steps in this tutorial, you'll have an app that looks something like this:



Create a Data Class

To get started, open your existing project in Xcode.

At this point, you have an interface and a navigation scheme for your ToDoList app using storyboards. Now, it's time to add data storage and behavior with model objects.

The goal of your app is to create a list of to-do items, so first you'll create a custom class, `XYZToDoItem`, to represent an individual to-do item. As you recall, the `XYZToDoItem` class was discussed in "Writing a Custom Class" (page 86).

To create the XYZToDoItem class

1. Choose File > New > File (or press Command-N).

   A dialog appears that prompts you to choose a template for your new file.
2. On the left, select Cocoa Touch under iOS.
3. Select Objective-C Class, and click Next.
4. In the Class field, type `ToDoItem` after the `XYZ` prefix.
5. Choose NSObject from the "Subclass of" pop-up menu.

   If you've been following along with the tutorials exactly, the Class title probably said `XYZToDoItemViewController` prior to this step. When you choose NSObject as the "Subclass of," Xcode knows you're making a normal custom class and removes the `ViewController` text that it was adding previously.
6. Click Next.
7. The save location defaults to your project directory. Leave that as is.
8. The Group option defaults to your app name, ToDoList. Leave that as is.
9. The Targets section defaults to having your app selected and the tests for your app unselected. That's perfect, so leave that as is.
10. Click Create.

The `XYZToDoItem` class is straightforward to implement. It has properties for its name, creation date, and whether the item has been completed. Go ahead and add these properties to the `XYZToDoItem` class interface.

To configure the XYZToDoItem class

1. In the project navigator, select `XYZToDoItem.h`.
2. Add the following properties to the interface so that the declaration looks like this:

```
@interface XYZToDoItem : NSObject

@property NSString *itemName;
@property BOOL completed;
@property (readonly) NSDate *creationDate;

@end
```

Checkpoint: Build your project by choosing Product > Build (or pressing Command-B). You're not using your new class for anything yet, but building it gives the compiler a chance to verify that you haven't made any typing mistakes. If you have, fix them by reading through the warnings or errors that the compiler provides, and then look back over the instructions in this tutorial to make sure everything looks the way it's described here.

Load the Data

You now have a class from which you can create and store the data for individual list items. You also need to keep a list of those items. The natural place to track this is in the `XYZToDoListViewController` class—view controllers are responsible for coordinating between the model and the view, so they need a reference to the model.

The Foundation framework includes a class, `NSMutableArray`, that works well for tracking lists of items. It's important to use a mutable array so that the user can add items to the array. The immutable version, `NSArray`, doesn't allow you to add items to it after it's initialized.

To use an array you need to both declare it and create it. You do this by allocating and initializing the array.

To allocate and initialize the array

1. In the project navigator, select `XYZToDoListViewController.m`.

   Because the array of items is an implementation detail of your table view controller, you declare it in the .m file instead of the .h file. This makes it private to your custom class.

2. Add the following property to the interface category Xcode created in your custom table view controller class. The declaration should look like this:

```
@interface XYZListViewController ()
```

```
@property NSMutableArray *toDoItems;

@end
```

3. Allocate and initialize the `toDoItems` array in the `viewDidLoad` method:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.toDoItems = [[NSMutableArray alloc] init];
}
```

The actual code for `viewDidLoad` includes some additional lines—inserted by Xcode when it created `XYZListViewController`—that are commented out. Feel free to leave them in.

At this point, you have an array that you can add items to. You'll do this in a separate method, `loadInitialData`, which you'll call from `viewDidLoad`. This code goes in its own method because it's a modular task, and you can improve code readability by making this method separate. In a real app this method would load the data from some sort of persistent store, such as a file. For now, the goal is to see how a table view works with custom data items, so you'll create some test data to experiment with.

Create an item in the way you created the array: Allocate and initialize. Then, give the item a name. This is the name that will be shown in the table view. Do this for a couple of items.

To load initial data

1. Add a new method, `loadInitialData`, below the `@implementation` line.

```
- (void)loadInitialData {
}
```

2. In this method, create a few list items, and add them to the array.

```
- (void)loadInitialData {
    XYZToDoItem *item1 = [[XYZToDoItem alloc] init];
    item1.itemName = @"Buy milk";
    [self.toDoItems addObject:item1];
```

```
    XYZToDoItem *item2 = [[XYZToDoItem alloc] init];
    item2.itemName = @"Buy eggs";
    [self.toDoItems addObject:item2];
    XYZToDoItem *item3 = [[XYZToDoItem alloc] init];
    item3.itemName = @"Read a book";
    [self.toDoItems addObject:item3];
}
```

3. Call the `loadInitialData` in the `viewDidLoad` method.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.toDoItems = [[NSMutableArray alloc] init];
    [self loadInitialData];
}
```

Checkpoint: Build your project by choosing Product > Build. You should see numerous errors for the lines of your `loadInitialData` method. The key to what's gone wrong is the first line, which should say "Use of undeclared identifier XYZToDoItem." This means that the compiler doesn't know about your `XYZToDoItem` when it's compiling `XYZToDoListViewController`. Compilers are very particular and need to be told explicitly what to pay attention to.

To tell the compiler to pay attention to your custom list item class

1. Find the `#import "XYZToDoListViewController.h"` line near the top of the `XYZToDoListViewController.m` file.
2. Add the following line immediately below it:

```
#import "XYZToDoItem.h"
```

Checkpoint: Build your project by choosing Product > Build. It should build without errors.

Display the Data

At this point, your table view has a mutable array that's prepopulated with some sample to-do items. Now you need to display the data in the table view.

You'll do this by making `XYZToDoListViewController` a data source of the table view. To make something a data source of the table view, it needs to implement the `UITableViewDataSource` protocol. It turns out that the methods you need to implement are exactly the ones you commented out in the second tutorial. To have a functioning table view requires three methods. The first of these is `numberOfSectionsInTableView:`, which tells the table view how many sections to display. For this app, you want the table view to display a single section, so the implementation is straightforward.

To display a section in your table

1. In the project navigator, select `XYZToDoListViewController.m`.
2. If you commented out the table view data source methods in the second tutorial, remove those comment markers now.
3. Find the section of the template implementation that looks like this.

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
#warning Potentially incomplete method implementation.
    // Return the number of sections.
    return 0;
}
```

   You want a single section, so you want to remove the warning line and change the return value from 0 to 1.

4. Change the `numberOfSectionsInTableView:` data source method to return a single section, like this:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    // Return the number of sections.
    return 1;
}
```

The next method, `tableView:numberOfRowsInSection:`, tells the table view how many rows to display in a given section. You have a single section in your table, and each to-do item should have its own row in the table view. That means that the number of rows should be the number of `XYZToDoItem` objects in your `toDoItems` array.

To return the number of rows in your table

1. In the project navigator, select `XYZToDoListViewController.m`.
2. Find the section of the template implementation that looks like this:

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
#warning Incomplete method implementation.
    // Return the number of rows in the section.
    return 0;
}
```

   You want to return the number of list items you have. Fortunately, `NSArray` has a handy method called `count` that returns the number of items in the array, so the number of rows is `[self.toDoItems count]`.

3. Change the `tableView:numberOfRowsInSection:` data source method to return the appropriate number of rows.

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    // Return the number of rows in the section.
    return [self.toDoItems count];
}
```

The last method, `tableView:cellForRowAtIndexPath:`, asks for a cell to display for a given row. Up until now, you've been working with code only, but the cell to display for a row is very much part of your interface. Fortunately, Xcode makes it easy to design custom cells in Interface Builder. The first task is to design your cell and to tell the table view that instead of using static content, it's going to be using prototype cells with dynamic content.

To configure your table view

1. Open your storyboard.

2. Select the table view in the outline.
3. With the table view selected, open the Attributes inspector in the utility area.
4. In the Attributes inspector, change the table view's Content attribute from Static Cells to Dynamic Prototypes.

Interface Builder takes the static cells you configured and converts them all into prototypes. Prototype cells, as the name implies, are cells that are configured with text styles, colors, images, or other attributes as you want them to be displayed but that get their data from the data source at runtime. The data source loads a prototype cell for each row and then configures that cell to display the data for the row.

To load the correct cell, the data source needs to know what it's called, and that name must also be configured in the storyboard.

While you're setting the prototype cell name, you'll also configure another property—the cell selection style, which determines a cell's appearance when a user taps it. Set the cell selection style to None so that the cell won't be highlighted when a user taps it. This is the behavior you want your cells to have when a user taps an item in the to-do list to mark it as completed or uncompleted—a feature you'll implement later in this tutorial.

To configure the prototype cell

1. Select the first table view cell in your table.
2. In the Attributes inspector, locate the Identifier field and type `ListPrototypeCell`.
3. In the Attributes inspector, locate the Selection field and choose None.

You could also change the font or other attributes of the prototype cell. The basic configuration is easy to work with, so you'll keep that.

The next step is to teach your data source how to configure the cell for a given row by implementing `tableView:cellForRowAtIndexPath:`. This data source method is called by the table view when it wants to display a given row. For table views with a small number of rows, all rows may be onscreen at once, so this method gets called for each row in your table. But table views with a large number of rows display only a small fraction of their total items at a given time. It's most efficient for table views to only ask for the cell for rows that are being displayed, and that's what `tableView:cellForRowAtIndexPath:` allows the table view to do.

For any given row in the table, fetch the corresponding entry in the `toDoItems` array and then set the cell's text label to the item's name.

To display cells in your table

1. In the project navigator, select `XYZToDoListViewController.m`.

2. Find the `tableView:cellForRowAtIndexPath:` data source method. The template implementation looks like this:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath

{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];


    // Configure the cell...


    return cell;

}
```

   The template performs several tasks. It creates a variable to hold the identifier for the cell, asks the table view for a cell with that identifier, adds a comment about where code to configure the cell should go, and then returns the cell.

   To make this code work for your app, you'll need to change the identifier to the one you set in the storyboard and then add code to configure the cell.

3. Change the cell identifier to the one you set in the storyboard. To avoid typos, copy and paste from the storyboard to the implementation file. The cell identifier line should now look like this:

```
static NSString *CellIdentifier = @"ListPrototypeCell";
```

4. Just before the return statement, add the following lines of code:

```
XYZToDoItem *toDoItem = [self.toDoItems objectAtIndex:indexPath.row];

cell.textLabel.text = toDoItem.itemName;
```

Your `tableView:cellForRowAtIndexPath:` method should look like this:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath

{
    static NSString *CellIdentifier = @"ListPrototypeCell";
```

```
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];
    XYZToDoItem *toDoItem = [self.toDoItems objectAtIndex:indexPath.row];
    cell.textLabel.text = toDoItem.itemName;
    return cell;
}
```

Checkpoint: Run your app. The list of items you added in `loadInitialData` should show up as cells in your table view.

Mark Items as Completed

A to-do list isn't much good if you can never mark items as completed. Now, you'll add support for that. A simple interface would be to have the completion state toggle when the user taps the cell and to display completed items with a checkmark next to them. Fortunately, table views come with some built-in behavior that you can take advantage of to implement this simple interface—specifically, table views notify their delegate when the user taps a cell. So the task is to write the code that will respond to the user tapping a to-do item in the table.

Xcode already made `XYZToDoListViewController` the delegate of the table view when you configured it in the storyboard. All you have to do is implement the `tableView:didSelectRowAtIndexPath:` delegate method to respond to user taps and update your to-do list items appropriately.

When a cell gets selected, the table view calls the `tableView:didSelectRowAtIndexPath:` delegate method to see how it should handle the selection. In this method, you'll write code to update the to-do item's completion state.

To mark an item as completed or uncompleted

1. In the project navigator, select `XYZToDoListViewController.m`.
2. Add the following lines to the end of the file, just above the `@end` line:

```
#pragma mark - Table view delegate


- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
```

```
}
```

Try typing the second line instead of just copying and pasting. You'll find that **code completion** is one of the great time-saving features of Xcode. When Xcode brings up the list of potential completions, scroll through the list until you find the one you want and then press Return. Xcode inserts the whole line for you.

3. You want to respond to taps but not actually leave the cell selected. Add the following code to deselect the cell immediately after selection:

```
[tableView deselectRowAtIndexPath:indexPath animated:NO];
```

4. Find the corresponding `XYZToDoItem` in your `toDoItems` array.

```
XYZToDoItem *tappedItem = [self.toDoItems objectAtIndex:indexPath.row];
```

5. Toggle the completion state of the tapped item.

```
tappedItem.completed = !tappedItem.completed;
```

6. Tell the table view to reload the row whose data you just updated.

```
[tableView reloadRowsAtIndexPaths:@[indexPath]
withRowAnimation:UITableViewRowAnimationNone];
```

Your `tableView:didSelectRowAtIndexPath:` method should look like this:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:NO];
    XYZToDoItem *tappedItem = [self.toDoItems objectAtIndex:indexPath.row];
    tappedItem.completed = !tappedItem.completed;
    [tableView reloadRowsAtIndexPaths:@[indexPath]
withRowAnimation:UITableViewRowAnimationNone];
}
```

Checkpoint: Run your app. The list of items you added in `loadInitialData` is visible as cells in your table view. But when you tap items, nothing seems to happen. Why not?

The reason is that you haven't configured the table view cell to display the completion state of an item. To do so, you need to go back to the `tableView:cellForRowAtIndexPath:` method and configure the cell to display an indicator when an item is completed.

One way to indicate that an item is completed is to put a checkmark next to it. Luckily, table view cells can have a cell accessory on the right side. By default, there's no accessory; however, you can change the cell to display a different accessory, one of which is a checkmark. All you have to do is set the cell accessory based on the completion state of the to-do item.

To display an item's completion state

1. Go to the `tableView:cellForRowAtIndexPath:` method.
2. Add the following code just below the line that sets the text label of the cell:

```
if (toDoItem.completed) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
} else {
    cell.accessoryType = UITableViewCellAccessoryNone;
}
```

Your `tableView:cellForRowAtIndexPath:` method should now look like this:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"ListPrototypeCell";
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];
    XYZToDoItem *toDoItem = [self.toDoItems objectAtIndex:indexPath.row];
    cell.textLabel.text = toDoItem.itemName;
    if (toDoItem.completed) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    } else {
        cell.accessoryType = UITableViewCellAccessoryNone;
    }
```

```
    return cell;
}
```

Checkpoint: Run your app. The list of items you added in `loadInitialData` is visible as cells in your table view. When you tap an item, a checkmark should appear next to it. If you tap the same item again, the checkmark disappears.

Add New Items

The final step in creating the to-do list app's functionality is implementing the ability to add an item. When a user enters an item name in the text field on the `XYZAddToDoItemViewController` scene and taps the Done button, you want the view controller to create a new list item and pass it back to the `XYZToDoListViewController` to display in the to-do list.

First, you need to have a list item to configure. Just as with the table view, the view controller is the logical place to connect the interface to the model. Give the `XYZAddToDoItemViewController` a property to hold the new to-do item.

To add an XYZToDoItem to the XYZAddToDoItemViewController class

1. In the project navigator, select `XYZToDoListViewController.h`.

   Because you'll need to access the list item from your table view controller later on, it's important to make this a public property. That's why you declare it in the interface file, `XYZAddToDoItemViewController.h`, instead of in the implementation file, `XYZAddToDoItemViewController.m`.

2. Add an import declaration to the `XYZToDoItem` class above the `@interface` line.

   ```
   #import "XYZToDoItem.h"
   ```

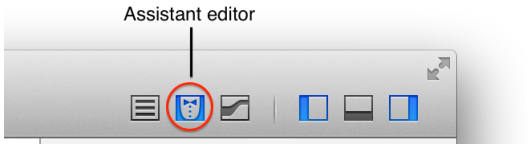
3. Add a `toDoItem` property to the interface.

   ```
   @interface XYZAddToDoItemViewController : UIViewController


   @property XYZToDoItem *toDoItem;


   @end
   ```

To get the name of the new item, the view controller needs access to the contents of the text field where the user enters the name. To do this, create a connection from the `XYZAddToDoItemViewController` class that connects to the text field in your storyboard.

To connect the text field to your view controller

1. In the outline view, select the `XYZAddToDoItemViewController` object.
2. Click the Assistant button in the upper right of the window's toolbar to open the assistant editor.

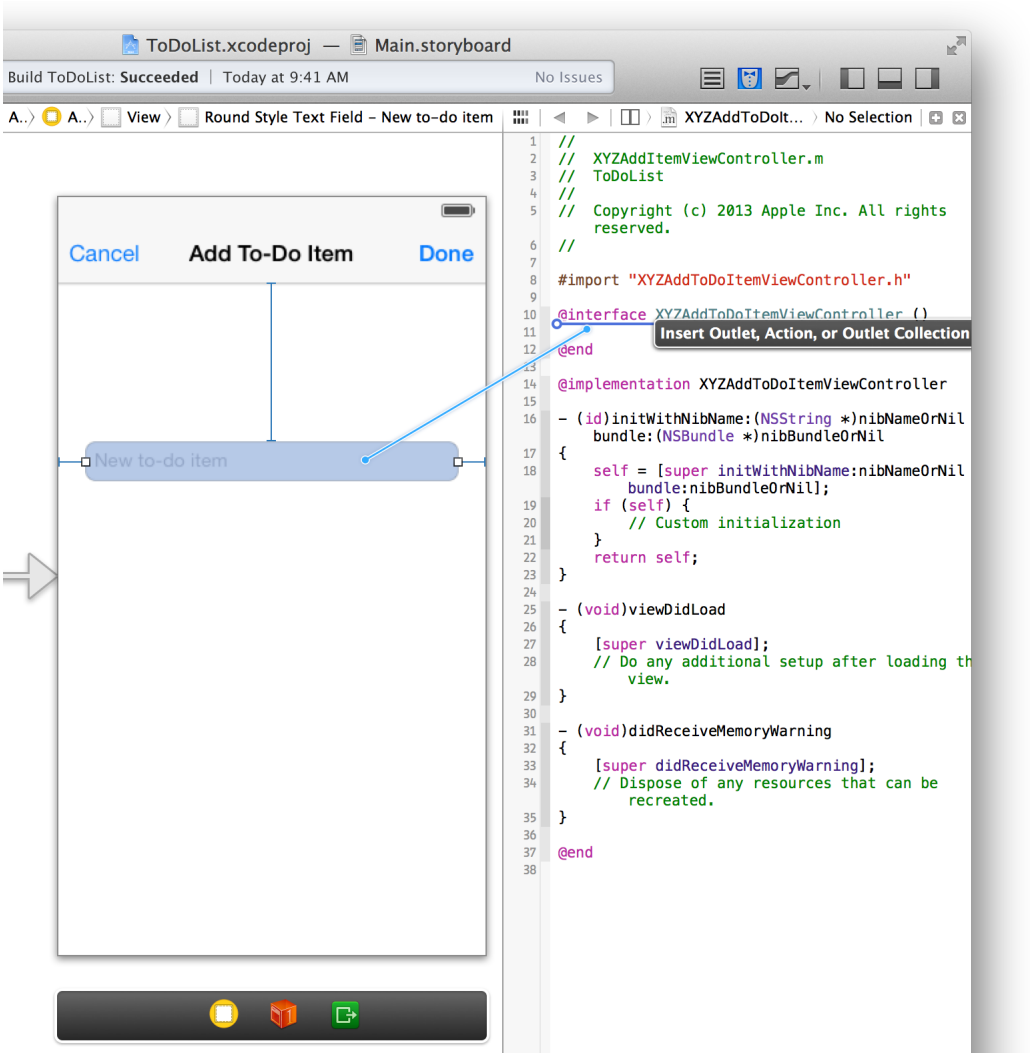


   The editor on the right should appear with `XYZAddToDoItemViewController.m` displayed. If it isn't displayed, click the filename in the editor on the right and choose `XYZAddToDoItemViewController.m`.

   The assistant editor allows you to have two files open at once, making it possible to perform operations between them—for example, tying a property in your source file with an object in your interface.

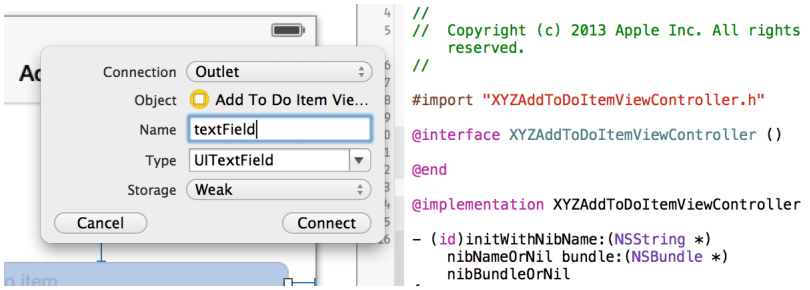
3. Select the text field in your storyboard.

4. Control-drag from the text field on your canvas to the code display in the editor on the right, stopping the drag at the line just below the `@interface` line in `XYZAddToDoItemViewController.m`.



5. In the dialog that appears, for Name, type `textField`.

   Leave the rest of the options as they are. Your dialog should look like this:



6. Click Connect.

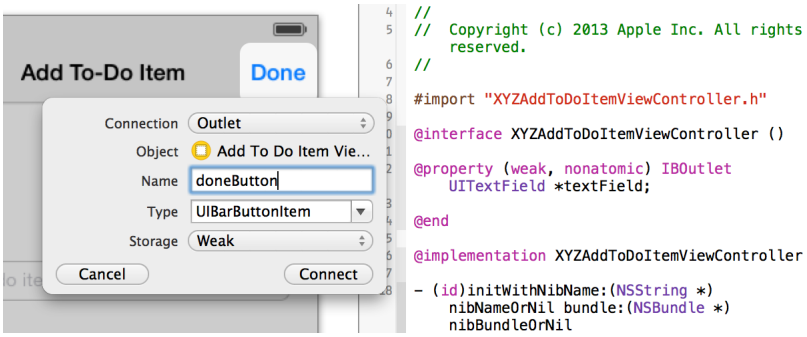
   Xcode adds the necessary code to `XYZAddToDoItemViewController.m` to store a pointer to the text field and configures the storyboard to set up that connection.

Additionally, you need to know when to create the item. You want to create the item only if the Done button was tapped. To do this, add the Done button as an outlet.

To connect the Done button to your view controller

1. In your storyboard, open the assistant editor, and set the rightmost window to `XYZAddToDoItemViewController.m`.
2. Select the Done button in your storyboard.
3. Control-drag from the Done button on your canvas to the code display in the editor on the right, stopping the drag at the line just below your `textField` property in `XYZAddToDoItemViewController.m`.
4. In the dialog that appears, for Name, type `doneButton`.

   Leave the rest of the options as they are. Your dialog should look like this:



5. Click Connect.

You now have a way to identify the Done button. Because you want to create an item when the Done button is tapped, you need to know when that happens.

When the user taps the Done button, it kicks off an unwind segue back to the to-do list—that's the interface you configured in the second tutorial. Before a segue executes, the system gives the view controller involved a chance to prepare by calling `prepareForSegue:`. This is exactly the point at which you want to check to see whether the user tapped the Done button, and if so, create a new to-do item. You can check which one of the buttons got tapped, and if it was the Done button, create the item.

To create an item after tapping the Done button

1. Select `XYZAddToDoItemViewController.m` in the project navigator.
2. Add the `prepareForSegue:` method below the `@implementation` line:

```
- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
}
```

3. In the method, see whether the Done button was tapped.

   If it wasn't, instead of saving the item, you want the method to return without doing anything else.

```
if (sender != self.doneButton) return;
```

4. See whether there's text in the text field.

```
if (self.textField.text.length > 0) {
}
```

5. If there's text, create a new item and give it the name of the text in the text field. Also, ensure that the completed state is set to `NO`.

```
self.toDoItem = [[XYZToDoItem alloc] init];
self.toDoItem.itemName = self.textField.text;
self.toDoItem.completed = NO;
```

   If there isn't text, you don't want to save the item, so you won't do anything else.

Your `prepareForSegue:` method should look like this:

```
- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if (sender != self.doneButton) return;
    if (self.textField.text.length > 0) {
        self.toDoItem = [[XYZToDoItem alloc] init];
        self.toDoItem.itemName = self.textField.text;
        self.toDoItem.completed = NO;
    }
}
```

Now that you've created a new item, you need to pass the item back to `XYZToDoListViewController` so that it can add the item to the to-do list. To accomplish this, you need to revisit the `unwindToList:` method that you wrote in the second tutorial. This method gets called when the `XYZAddToDoItemViewController` scene closes, which happens when the user taps either the Cancel or the Done button.

The `unwindToList:` method takes a segue as a parameter, like all methods that are used as targets for an unwind segue. The segue parameter is the segue that unwinds from `XYZAddToDoItemViewController` back to `XYZToDoListViewController`. Because a segue is a transition between two view controllers, it is aware of its source view controller—`XYZAddToDoItemViewController`. By asking the segue object for its source view controller, you can access any data stored in the source view controller in the `unwindToList:` method. In this case, you want to access `toDoItem`. If it's `nil`, the item was never created—either the text field had no text or the user tapped the Cancel button. If there's a value for `toDoItem`, you retrieve the item, add it to your `toDoItems` array, and display it in the to-do list by reloading the data in the table view.

To store and display the new item

1. In the project navigator, select `XYZToDoListViewController.m`.
2. Add an import declaration to the `XYZAddToDoItemViewController` class above the `@interface` line.

```
#import "XYZAddToDoItemViewController.h"
```

3. Find the `unwindToList:` method you added in the second tutorial.
4. In this method, retrieve the source view controller—the controller you're unwinding from, `XYZAddToDoItemViewController`.

```
XYZAddToDoItemViewController *source = [segue sourceViewController];
```

5. Retrieve the controller's to-do item.

```
XYZToDoItem *item = source.toDoItem;
```

   This is the item that was created when the Done button was tapped.

6. See whether the item exists.

```
if (item != nil) {
}
```

   If it's `nil`, either the Cancel button closed the screen or the text field had no text, so you don't want to save the item.

   If it does exist, add the item to your `toDoItems` array.

```
[self.toDoItems addObject:item];
```

7. Reload the data in your table.

   Because the table view doesn't keep track of its data, it's the responsibility of the data source—in this case, your table view controller—to notify the table view when there's new data for it to display.

```
[self.tableView reloadData];
```

Your `unwindToList:` method should look like this:

```
- (IBAction)unwindToList:(UIStoryboardSegue *)segue
{
    XYZAddToDoItemViewController *source = [segue sourceViewController];
    XYZToDoItem *item = source.toDoItem;
    if (item != nil) {
        [self.toDoItems addObject:item];
        [self.tableView reloadData];
    }
}
```

Checkpoint: Run your app. Now when you click the Add button (+) and create a new item, you should see it in your to-do list. Congratulations! You've created an app that takes input from the user, stores it in an object, and passes that object between two view controllers. This is the foundation of moving data between scenes in a storyboard-based app.

Recap

You're almost done with this introductory tour of developing apps for iOS. The final section gives you more information about how to find your away around the documentation, and it suggests some next steps you might take as you learn how to create more advanced apps.