;C_TravelAdvisor_DefineClasses.rtfd;;¬ Previous Section ;E_TravelAdvisor_ImplemTAController.rtfd;;¬ Next Section

3. Travel Advisor Tutorial

Implementing the Country Class

Although it has no outlets, the Country class defines a number of instance variables that correspond to the fields of Travel Advisor.

1 Declare instance variables.

In Project Builder, click Headers in the project browser, then select **Country.h**.

Add the declarations shown between the braces below.

```
@interface Country : NSObject <NSCoding>      /* 1 */
{
    NSString *name;                           /* 2 */
    NSString *airports;
    NSString *airlines;
    NSString *transportation;
    NSString *hotels;
    NSString *languages;
    BOOL     englishSpoken;
    NSString *currencyName;
    float    currencyRate;                    /* 3 */
    NSString *comments;
}
```

1. Declares that the Country class adopts the NSCoding protocol

   When a class adopts a protocol, it asserts that it implements the methods the protocol declares. Classes that archive or serialize their data must adopt the NSCoding protocol. See *Object-Oriented Programming and the Objective-C Language* for more on protocols.

2. Explicitly types the instance variable as ªa pointer to class NSStringºÐor a NSString object. See below for more about the NSString class.

3. Declare non-object instance variables the same way you declare them in C programs. In this case, **currencyRate** is of type **float**.

Related Concept: ;TravelAdvisorConcepts.rtfd;linkMarkername NSString:AStringforAllCountries;, NSString: A String for All Countries

Country.h also declares a dozen or more methods. Most of these are *accessor methods*. Accessor methods fetch and set the values of instance variables. They are a critical part of an object's interface.

2 Declare methods.

After the instance variables, add the declarations listed here.

```
/* initializtion and de-allocation */
- (id)init;                                    /* 1 */
- (void)dealloc;
/* archiving and unarchiving */
- (void)encodeWithCoder:(NSCoder *)coder;     /* 2 */
- (id)initWithCoder:(NSCoder *)coder;
/* accessor methods */
- (NSString *)name;                            /* 3 */
- (void)setName:(NSString *)str;
- (NSString *)airports;
```

```
- (void)setAirports:(NSString *)str;
- (NSString *)airlines;
- (void)setAirlines:(NSString *)str;
/* ...other accessor method declarations follow... */
```

1. **Object initialization and deallocation**. In OpenStep you usually create an object by allocating it (**alloc**) and then initializing it (**init** or **init**... variant):

   ```
   Country *aCountry = [[Country alloc] init];
   ```

   When Country's **init** method is invoked, it initializes its instance variables to known values and completes other start-up tasks. Similarly, when an object is deallocated, its **dealloc** method is invoked, giving it the opportunity to release objects it's created, free **malloc**'d memory, and so on. You'll learn more about **init** and **dealloc** shortly.

2. **Object archiving and unarchiving**. The **encodeWithCoder:** declaration indicates that objects of this class are to be archived. Archiving encodes an object's class and state (typically instance variables) in a file that is often stored within the application wrapper (that is, the ªhiddenº application directory). Unarchiving, through **initWithCoder:**, reads the encoded class and state data and restores the object to its previous state. There's more on this topic in the following pages.

3. **Accessor methods**. The declaration for accessor methods that *return* values is, by convention, the name of the instance variable preceded by the type of the returned value in parentheses. Accessor methods that *set* the value of instance variables begin with ªsetº prepended to the name of the instance variable (initial letter capitalized). The ªsetº method's argument takes the type of the instance variable and the method itself returns void.

975498_TableRule.eps ¬***Before You Go On***

If you don't want to allow an instance variable's value to be changed by anyone outside of your class, *don't* provide a set method for the instance variable. If you do provide a set method, make sure objects of your own

class use it when specifying a value for the instance variables. This has important implications for subclasses of your class.

Exercise: The previous example shows the declarations for only a few accessor methods. Every instance variable of the Country class should have an accessor method that returns a value and one that sets a value. Complete the remaining declarations.
316044_TableRule.eps ¬
Related Concept: ;TravelAdvisorConcepts.rtfd;linkMarkername TheFoundationFramework:Capabilities,Concepts,andParadigms;, The Foundation Framework: Capabilities, Concepts, and Paradigms

Now that you've declared the Country class's accessor methods, implement them.

3 Implement the accessor methods.

Select **Country.m** in the project browser.

Write the code that fetches and sets the values of instance variables.

```
- (NSString *)name                          /* 1 */
{
    return name;
}

- (void)setName:(NSString *)str             /* 2 */
{
    [name autorelease];
    name = [str copy];
}

/* more accessor method implementations follow */
```

1. For ªgetº accessor methods (at least when the instance variables, like Travel Advisor's, hold immutable objects) simply return the instance variable.

2. For accessor methods that set *object* values, first send **autorelease** to the current instance variable, then **copy** (or **retain**) the passed-in value to the variable. The **autorelease** message causes the previously assigned object to be released at the end of the current event loop, keeping current references to the object valid until then.

   If the instance variable has a non-object value (such as an integer or float value), you don't need to **autorelease** and **copy**; just assign the new value.

   In many situations you can send **retain** instead of **copy** to keep an object around. But for ªvalueº type objects, such as Country's instance variables, **copy** is better. For the reason why, and for more on **autorelease**, **retain**, **copy**, and related messages for object disposal and object retention, see ``Object Ownership, Retention, and Disposal.''

Related Concept: ;TravelAdvisorConcepts.rtfd;linkMarkername ObjectOwnership,Retention,andDisposal;, Object Ownership, Retention, and Disposal

401819_TableRule.eps ¬

Before You Go On

Exercise: The example above shows the implementation of the accessor methods for the **name** instance variable. Implement the remaining accessor methods.

565943_TableRule.eps ¬

4 Write the object-initialization and object-deallocation code.

Implement the **init** method, as shown here.

Implement the **dealloc** method, following the suggestions in the Required Exercise, below.

```
- (id)init
{
    [super init];                /* 1 */
```

```
        name = @"";                     /* 2 */
        airports = @"";
        airlines = @"";
        transportation = @"";
        hotels = @"";
        languages = @"";
        currencyName = @"";
        comments = @"";

        return self;                    /* 3 */
    }
```

1. Invokes **super**'s (the superclass's) **init** method to have inherited instance variables initialized. Always do this first in an **init** method.
2. Initializes an NSString instance variable to an empty string. @°° is a compiler-supported construction that creates an immutable NSString object from the text enclosed by the quotes. You could have just as well typed:

```
name = @°Howdy Doody°;
```

But that wouldn't have been practical as an initial value. You don't need to initialize instance variables to null values because the run-time system does it for you; it assigns **nil** to objects, zeroes to integers and floats, and NULL to **char ***'s if they're not explicitly initialized. However, you should initialize instance variables that take other starting values.

Don't substitute nil when empty objects are expected, and vice versa.The Objective-C keyword nil represents an ªobjectº with an **id** (value) of zero. An empty object (such as @``") is a true object; it just has no content of its given type. To learn more about Objective-C keywords, see *Object-Oriented Programming and the Objective-C Language*.

3. By returning **self** you're returning a true instance of your object; up until this point, the instance is considered undefined.

549033_TableRule.eps ¬***Before You Go On***

Implement the **dealloc** method. In this method you release (that is, send **release** or **autorelease** to) objects that you've created, copied, or retained (which don't have an impending **autorelease**). For the Country class, release all objects held as instance variables. If you had other retained objects, you would release them, and if you had dynamically allocated data, you would free it. When this method completes, the Country object is deallocated. The **dealloc** method should send **dealloc** to **super** as the *last* thing it does, so that the Country object isn't released by its superclass before it's had the chance to release all objects it owns.

> Note that **release** itself doesn't deallocate objects, but it leads to their deallocation. For more on **release** and **autorelease**, see ``Object Ownership, Retention, and Disposal'' ;TravelAdvisorConcepts.rtfd;linkMarkername ObjectOwnership,Retention,andDisposal;, .

836878_TableRule.eps ¬

You want the Country objects created by the Travel Advisor application to be *persistent*. That is, you want them to ªrememberº their state between sessions. Archiving lets you do this by encoding the state of application objects in a file along with their class membership. The NSCoding protocol defines two methods that enable archiving for a class: **encodeWithCoder:** and **initWithCoder:**.

5 Implement the methods that archive and unarchive the object.

Implement the **encodeWithCoder:** method, as shown below.

```
- (void)encodeWithCoder:(NSCoder *)coder
{
    [coder encodeObject:name];                          /* 1 */
    [coder encodeObject:airports];
    [coder encodeObject:airlines];
    [coder encodeObject:transportation];
```

```
    [coder encodeObject:hotels];
    [coder encodeObject:languages];
    [coder encodeValueOfObjCType:"s" at:&englishSpoken]; /* 2 */
    [coder encodeObject:currencyName];
    [coder encodeValueOfObjCType:"f" at:&currencyRate];
    [coder encodeObject:comments];
}
```

1. The **encodeObject:** method encodes a single object in the archival file.
2. For both object and non-object types, you can use **encodeValueOfObjCType:at:**.

Implement the **initWithCoder:** method, as shown below.

```
- (id)initWithCoder:(NSCoder *)coder
{
    name = [[coder decodeObject] copy];                  /* 1 */
    airports = [[coder decodeObject] copy];
    airlines = [[coder decodeObject] copy];
    transportation = [[coder decodeObject] copy];
    hotels = [[coder decodeObject] copy];
    languages = [[coder decodeObject] copy];
    [coder decodeValueOfObjCType:"s" at:&englishSpoken];
    currencyName = [[coder decodeObject] copy];
    [coder decodeValueOfObjCType:"f" at:&currencyRate];
    comments = [[coder decodeObject] copy];

    return self;                                         /* 2 */
 }
```

1. The order of decoding should be the same as the order of encoding; since **name** is encoded first it should be decoded first. Use **copy** when you assign value-type objects to instance variables (see the concept ``Object Ownership, Retention, and Disposal'' ;TravelAdvisorConcepts.rtfd;linkMarkername ObjectOwnership,Retention,andDisposal;,). NSCoder defines **decode...** methods that correspond the **encode...** methods, which you should use.

2. As in any **init...** method, end by returning **self**Đan initialized instance.