Object-Oriented Programming Appendix A

A

What You'll Learn

Characteristics of an object-oriented program

What an object is

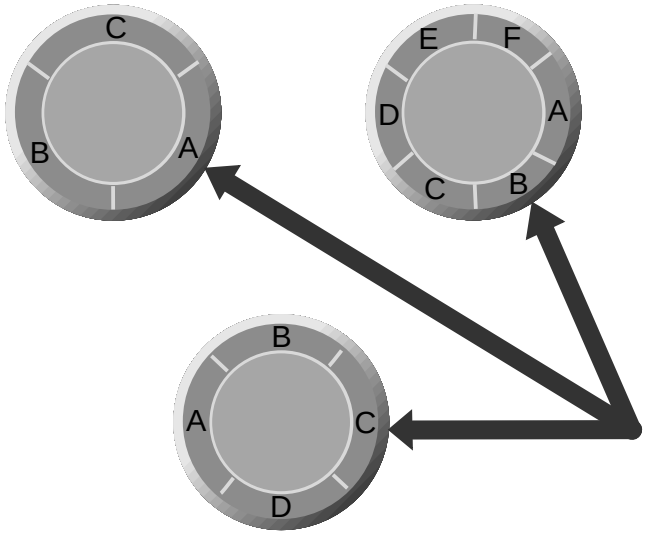
Encapsulation

Messages

What a class is

Inheritance

Categories and protocols

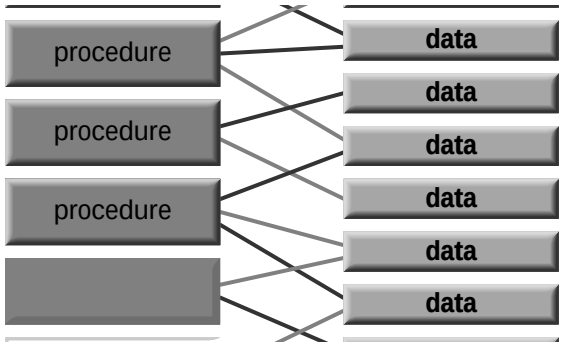


Appendix A
Object-Oriented Programming

You can't get far in Rhapsody or Yellow Box for Windows development without a grasp of the basic concepts of object-oriented programming. For those new to this approach to programming, it might seem strange at first glance, but a common reaction after learning a bit more is "Yes, of course." This appendix presents an overview of object-oriented programming from the particular perspective of Objective-C.

"Object-oriented programming" has become one of the premier buzzwords in the computer industry. To understand why, it's important to cut through the hype and focus on the problem that motivated the object-oriented approach.

In classic *procedural programming* (used with COBOL, Fortran, C, and other languages), programs are made of two fundamental components: *data* and *code*. The data represents what the user needs to manipulate, while the code does the manipulation. To improve project management and maintenance, procedural programs compartmentalize code into *procedures*. However, much of the data is global, and each procedure may manipulate any part of that global data directly.



With the procedural approach, the network of interaction between procedures and data becomes increasingly complex as an application grows. Inevitably, the interrelationships become a hard-to-maintain tangle—spaghetti code. A simple change in a data structure can affect many procedures, many lines of code—a nightmare for those who must maintain and enhance applications. Procedural programming also leads to nasty, hard-to-find bugs in which one function inadvertently changes data that another function relies on.

Objects change all that.

Objects

An object is a self-contained programmatic unit that combines data and the procedures that operate on that data. In the Objective-C language, an object's data comprises its *instance variables*, and its procedures, the functions that affect or make use of the data, are known as *methods*.

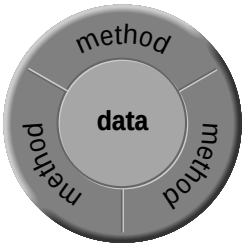
Like objects in the physical world, objects in a program have identifying characteristics and behavior. Often programmatic objects are modeled on real objects. For example, an object such as a button has an analog in the buttons on control devices, such as stereo equipment and telephones. A button object includes the data and code to generate an appearance on the screen that simulates a "real" button and to respond in a familiar way to user actions.

A button object highlights its on-screen representation when the user clicks it.

Encapsulation

Just as procedures compartmentalize code, objects compartmentalize both code *and* data. This results in *data encapsulation*, effectively surrounding data with the procedures for manipulating that data.

Typically, an object is regarded as a "black box," meaning that a program never directly accesses an object's variables. Indeed, a program shouldn't even need to know what variables an object has in order to perform its functions. Instead, the program accesses the object only through its methods. In a sense, the methods surround the data, not only shielding an object's instance variables but mediating access to them:



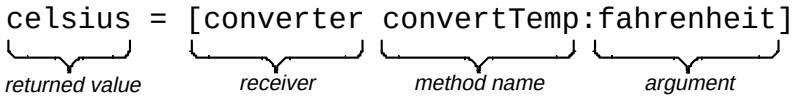
Objects are the basic building blocks of Objective-C applications. By representing a responsibility in the problem domain, each object encapsulates a particular area of functionality that the program needs. The object's methods provide the interface to this functionality. For example, an

object representing a database record both stores data and provides well-defined ways to access that data.

Using this *modularity*, object-oriented programs can be divided into distinct objects for specific data and specific tasks. Programming teams can easily parcel out areas of responsibility among individual members, agreeing on interfaces to the distinct objects while implementing data structures and code in the most efficient way for their specific area of functionality.

Messages

To invoke one of the object's methods you send it a *message*. A message requests an object to perform some functionality or to return a value. In Objective-C, a message expression is enclosed in square brackets, like this:



In this example **converter** is the *receiver*, the object that receives the message. Everything to the right of this term is the message itself; it consists of a method name and any arguments the method requires. The message received by **converter** tells it to convert a temperature from Fahrenheit to Celsius and return that value.

In Objective-C, every message argument is identified with a label. Arguments follow colon-terminated *keywords*, which are considered part of the method name. One argument per keyword is allowed. If a method has more than one argument—as NSString's **rangeOfString:options:** method does, for example—the name is broken apart to accept the arguments:

```
range = [string rangeOfString:@"Rhapsody" options:NSLiteralSearch];
```

Often, but not always, messages return values to the sender of the message. Returned values must be received in a variable of an appropriate type. In the above example, the variable **range** must be of type NSRange. Messages that return values can be *nested*, especially if those returned values are objects. By enclosing one message expression within another, you can use a returned value as an argument or as a receiver without having to declare a variable for it.

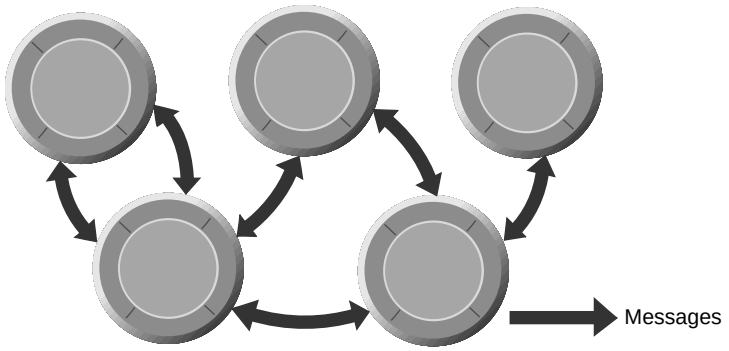
```
newString = [stringOne stringByAppendingString:
    [substringFromRange:[stringTwo rangeOfString:
    @"Rhapsody" at:NSAnchoredSearch]]];
```

The above message nests two other messages, each of which returns a value used as an argument. The inmost message expression is resolved first, then the next nested message expression is resolved, then the third message is sent and a value is returned to **newString**.

An Object-Oriented Program

Object-oriented programming is more than just another way of organizing data and functions. It permits application programmers to conceive and construct solutions to complex programs using a model that resembles—much more so than traditional programs—the way we organize the world around us. The object-oriented model for program structure simplifies problem resolution by clarifying roles and relationships.

You can think of an object-oriented program as a network of objects with well-defined behavior and characteristics, objects that interact through messages.



Different objects in the network play different roles. Some correspond to graphical elements in the user interface. The elements that you can drag from an Interface Builder palette are all objects. In an application, each window is represented by a separate object, as is each button, menu item, or display of text.

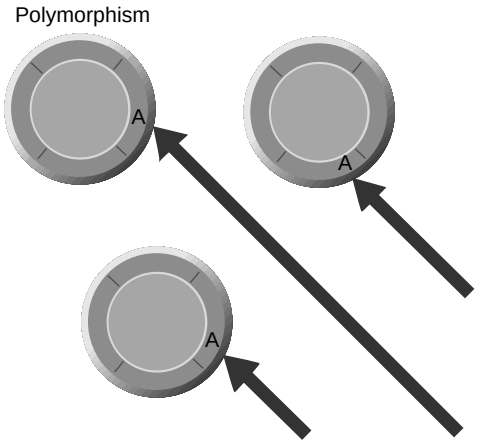
Applications also assign to objects functionality that isn't directly apparent in the interface, giving each object a different area of responsibility. Some of these objects might perform very specific computational tasks while others might manage the display and transfer of data, mediating the interaction between user-interface objects and computational objects.

Once you've defined your objects, creating a program is largely a matter of "hooking up" these objects: creating the connections that objects will use to communicate with each other.

Polymorphism and Dynamic Binding

Although the purpose of a message is to invoke a method, a message isn't the same as a function call. An object "knows about" only those methods that were defined for it or that it inherits. It can't confuse its methods with another object's methods, even if the methods are identically named.

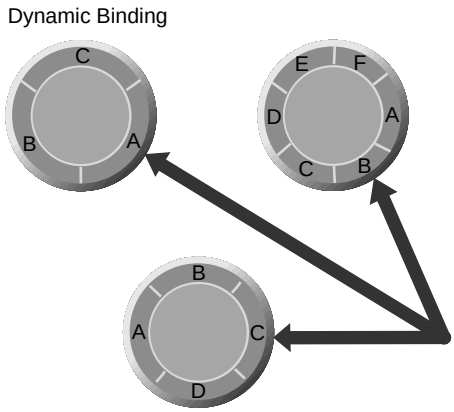
Each object is a self-contained unit, with its own name space (an name space being an area of the program where it is uniquely recognized by name). Just as local variables within a C function are isolated from other parts of a program, so too are the variables and methods of an object. Thus if two different kinds of objects have the same names for their methods, both objects could receive the same message, but each would respond to it differently. The ability of one message to cause different behavior in different receivers is referred to as *polymorphism*.



The advantage polymorphism brings to application developers is significant. It helps improve program flexibility while maintaining code simplicity. You can write code that might have an effect on a variety of objects without having to know at the time you write the code what objects they might be. For example, most user-interface objects respond to the message **display**; you can send **display** to any of these objects in your interface and it will draw itself, in its own way.

Dynamic binding is perhaps even more useful than polymorphism. It means both the object receiving a message and the message that an object receives can be set within your program as it runs. This is particularly important in a graphical, user-driven environment, where one user command—say, Copy or Paste—may apply to any number of user-interface objects.

> The example of **display** highlights the role of inheritance in polymorphism: a subclass often implements an identically named method (that is, *overrides* the method) of its superclass to achieve more specialized behavior. See the following section, "Classes," for details.



In dynamic binding, a run-time process finds the method implementation appropriate for the receiver of the message; it then invokes this implementation and passes it the receiver's data structure. This mechanism makes it easier to structure programs that respond to selections and actions chosen by users at run time. For example, either or both parts of a message expression—the receiver and the method name—can be variables whose values are determined by user actions. A simple message expression can deliver a Cut, Copy, or Paste menu command to whatever object controls the current selection.

Dynamic binding even enables applications to deal with new kinds of objects, ones that were not envisioned when the application itself was built. For example, it lets Interface Builder send messages to objects such as EOModeler when it is loaded into the application by means of custom palettes.

Polymorphism and dynamic binding depend on two other features: *dynamic typing* and *introspection*. The Objective-C language allows you to identify objects *generically* with the data type of **id**. This type defines a pointer to an object and its data structure (that is, instance variables) which, by inheritance from the root class NSObject, include a pointer to the object's class. What this means is that you don't have to type objects strictly by class in your code: the class for the object can be determined at run time through introspection.

Introspection means that an object, even one typed as **id**, can reveal its class and divulge other characteristics at run time. Several introspection methods allow you to ascertain the inheritance relationships of an object, the methods it responds to, and the protocols that it conforms to.

Classes

Some of the objects networked together in an applications are of different kinds, and some might be of the same kind. Objects of the same kind belong to the same *class*. A class is a programmatic entity that creates *instances* of itself—objects. A class defines the structure and interface of its instances and specifies their behavior.

When you want a new kind of object, you define a new class. You can think of a class definition as a type definition for a kind of object. It specifies the data structure that all objects belonging to the class will have and the methods they will use to respond to messages. Any number of objects can be created from a single class definition. In this sense, a class is like a factory for a particular kind of object.

In terms of lines of code, an object-oriented program consists mainly of class definitions. The objects the program will use to do its work are created at run time from class definitions (or, if pre-built with Interface Builder, are loaded at run time from the files where they are stored).

A class is more than just an object "factory," however. It can be assigned methods and receive messages just as an object can. As such it acts as a *class object*.

Object Creation

One of the primary functions of a class is to create new objects of the type the class defines. For example, the NSButton class creates new NSButton objects and the NSArray class creates new NSArrays. Objects are created at run time in a two-step process that first allocates memory for the instance variables of the new object and then initializes those variables. The following code creates a new Country object:

```
id newCountry = [[Country alloc] init];
```

The receiver for the **alloc** message is a class (the Country class from the Travel Advisor tutorial). The **alloc** method dynamically allocates memory for a new instance of the receiving class and returns the new object. The receiver for the **init** message is the new Country object that was dynamically allocated by **alloc**. Once allocated and initialized, this new record is assigned to the variable **newCountry**.

Note: You can create objects in your code with the **alloc** and **init** methods described here. But when you define a class in Interface Builder, that class definition is stored in a nib file. When an application loads that nib file, Interface Builder causes an instance of that class to be created.

After being allocated and initialized, a new object is a fully functional member of its class with its own set of variables. The **newCountry** object has all the behavior of any Country object, so it can receive messages, store values in its instance variables, and do all the other things a Country object does. If you need other Country objects, you create them in the same way from the same class definition.

Objects can be typed as **id**, as in the above example, or can be more restrictively typed, based on their class. Here, **newCountry** is typed as a Country object:

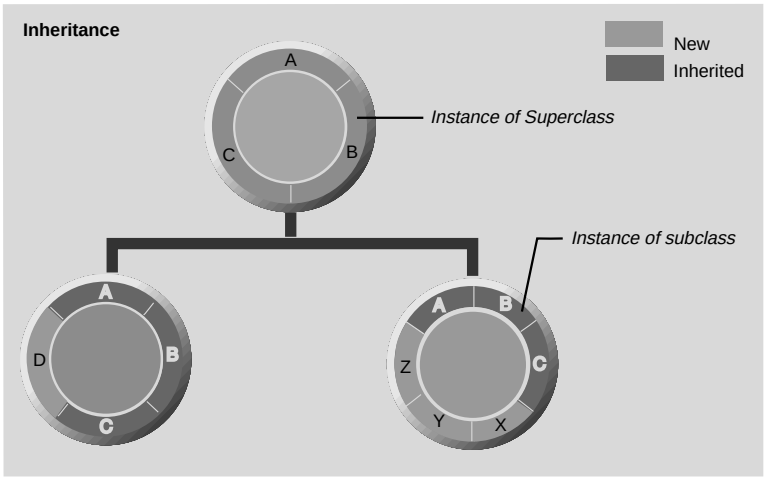
```
Country *newCountry = [[Country alloc] init];
```

The more restrictive typing by class enables the compiler to perform type-checking in assignment statements.

Inheritance

Inheritance is one of the most powerful aspects of object-oriented programming. Just as people inherit traits from their forbearers, instances of a class inherit attributes and behavior from that class's "ancestors." An object's total complement of instance variables and methods derives not only from the class that creates it, but from all the classes that class inherits from.

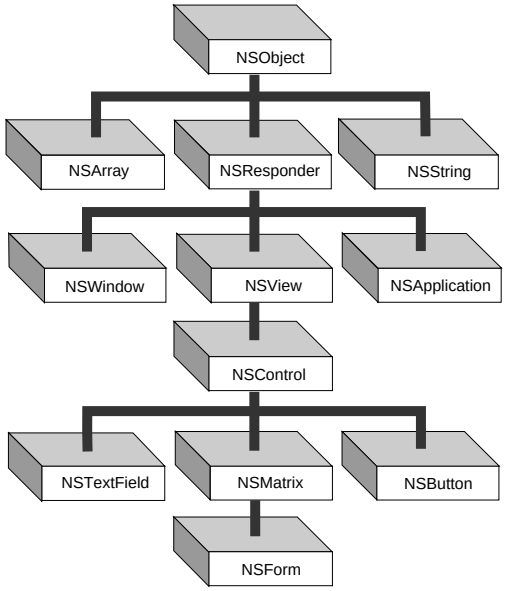
Because of inheritance, an Objective-C class definition doesn't have to specify every method and variable. If there's a class that does almost everything you want, but you need some additional features, you can define a new class that inherits from the existing class. The new class is called a *subclass* of the original class; the class it inherits from its *superclass*.



Creating a new class is often a matter of specialization. Since the new class inherits all its superclass's behavior, you don't need to re-implement the things that work as you want them to. The subclass merely extends the inherited behavior by adding new methods and any variables needed to support the additional methods. All the methods and variables defined for—or inherited by—the superclass are inherited by the subclass. A subclass can also alter superclass behavior by *overriding* an inherited method, re-implementing the method to achieve a behavior different from the superclass's implementation. (The technique for doing this is discussed later.)

The Class Hierarchy and the Root Class

A class can have any number of subclasses, but only one superclass. This means that classes are arranged in a branching hierarchy, with one class at the top—the *root class*—that has no superclass:



Part of the Yellow Box class hierarchy.

NSObject is the root class of this hierarchy, as it is of most Objective-C class hierarchies. From NSObject, other classes inherit the basic functionality that makes messaging work, enables objects to work together, and otherwise invests objects with the ability to behave as objects. Among other things, the root class creates a framework for the creation, initialization, deallocation, introspection, and storage of objects.

Note: Other root classes are possible. In fact, Distributed Objects makes use of another root class, NSProxy.

As noted earlier, you often create a subclass of another class because that superclass provides most, but not all, the behavior you require. But a subclass can have its own unique purpose that does not build on the role of an existing class. To define a new class that doesn't need to inherit any special behavior other than the default behavior of objects, you make it a subclass of the NSObject class. Subclasses of NSObject, because of their general-purpose nature as objects, are very common in Rhapsody applications. They often perform computational or application-specific functions.

Advantages of Inheritance

Inheritance makes it easy to bundle functionality common to a group of classes into a single class definition. For example, every object that draws on the screen—whether it draws an image of a button, a slider, a text display, or a graph of points—must keep track of which window it draws in and where in the window it draws. It must also know when it's appropriate to draw and when to respond to a user action. The code that handles all these details is part of a single class definition (the NSView class in the Application Kit). The specific work of drawing a button, a slider, or a text display can then be entrusted to a subclass.

This bundling of functionality both simplifies the organization of the code that needs to be written for an application and makes it easier to define objects that do complicated things. Each subclass need only implement the things it does differently from its superclass; there's no need to reimplement anything that's already been done.

What's more, hierarchical design assures more robust code. By building on a widely used, well-tested class such as NSView, a subclass inherits a proven foundation of functionality. Because the new code for a subclass is limited to implementing unique behavior, it's easier to test and debug that code.

Any class can be the superclass for a new subclass. Thus inheritance makes every class easily extensible—those provided by the Yellow Box frameworks, those you create, and those offered by third-party vendors.

Defining a Class

You define classes in two parts: One part declares the instance variables and the interface, principally the methods that can be invoked by messages sent to objects belonging to the class, and the other part actually implements those methods. The interface is public. The implementation is private, and can change without affecting the interface or the way the class is used.

The basic procedure for defining a class (using Interface Builder) is covered in the Currency Converter tutorial. However, here is a supplemental list of conventions and other points to remember when you define a class:

- The public interface for a class is usually declared in a header file (with an **.h** extension), the name of which is the name of the class. This header file can be imported into any program that makes use of the class.
- The code implementing a class is usually in a file taking the name of the class and having an extension of **.m**. This code must be present—in the form of a framework, dynamic shared library, static library, or the implementation file itself—when the project containing the class is compiled.
- Method declarations and implementations must begin with a minus (–) sign or a plus (+) sign. The dash indicates that these methods are used by instances of the class; a **+** sign precedes methods that the class object itself uses.
- Method definitions are much like function definitions. Note that methods not only respond to messages, they often initiate messages of their own—just as one function might call another.
- In a method implementation you can refer directly to an object's instance variables, as long as that object belongs to the class the method is defined in. There's no extra syntax for accessing variables or passing the object's data structure. The language keeps all that hidden.
- A method can also refer to the receiving object as **self**. This variable makes it possible for an object, in its method definitions, to send messages to itself.

Overriding a Method

A subclass can not only add new methods to the ones it inherits, it can also replace an inherited method with a new implementation. No special syntax is required; all you do is reimplement the method.

Overriding methods doesn't alter the set of messages that an object can receive; it alters the method implementations that will be used to respond to those messages. As mentioned earlier, this ability of each class to implement its own version of a method is referred to as polymorphism.

It's also possible to extend an inherited method, rather than replace it outright. To do this you override the method but invoke the superclass's same method in the new implementation. This invocation occurs with a message to **super**, which is a special receiver in the Objective-C language. The term **super** indicates that an inherited method should be performed, rather than one defined in the current class.

The Yellow Box Frameworks

When you write an object-oriented program, you rarely do it from scratch. There are almost always class definitions available that you can use. All you need are the class interface files, a library or framework with compiled versions of the class implementations, and some documentation. The task is to fit your pieces with the pieces that are already provided. As you'll realize after awhile, much of the task of writing object-oriented programs is simply implementing methods that respond to system-generated messages.

Categories and Protocols

In addition to subclassing, you can expand an object and make it fit with other classes using two Objective-C mechanisms: categories and protocols.

Categories provide a way to extend classes defined by other implementors—for example, you can add methods to the classes defined in the Yellow Box frameworks. The added methods are inherited by subclasses and are indistinguishable at run time from the original methods of the class. Categories can also be used as a way to distribute the implementation of a class into groups of related methods and to simplify the management of large classes where more than one developer is responsible for components of the code.

Protocols provide a way to declare groups of methods independent of a specific class—methods which any class, and perhaps many classes, might implement. Protocols declare interfaces to objects, leaving the programmer free to choose the implementation most appropriate for a specific class. Protocols free method declarations from dependency on the class hierarchy, so they can be used in ways that subclasses and categories cannot. They allow objects of any class to communicate with each other for a specific purpose.

The Rhapsody APIs provide a number of protocols. For example, the spell-checking protocols and the object-dragging protocols enable other developers to seamlessly integrate their spell-checking and object-dragging implementations into an existing system.