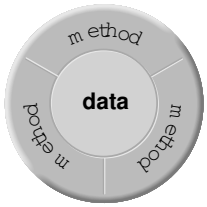
Many of the advantages of the NeXTSTEP development environment are made possible by object-oriented programming. The basis of the development environment is the Objective C language, a superset of standard C that adds a small amount of syntax to the language to support object-oriented design. Thanks to Objective C, NeXTSTEP provides a coding model that's simple yet extensible.

This section discusses some of the fundamentals of object-oriented programming and Objective C. It explains how the language lets you take advantage of the powerful application building blocks supplied by NeXTSTEP, and how to incorporate object-oriented techniques into your own applications.

OBJECTS

In the Objective C language, an *object* is a self-contained unit that groups a data structure (variables) with *methods*, the functions that affect or make use of the data.

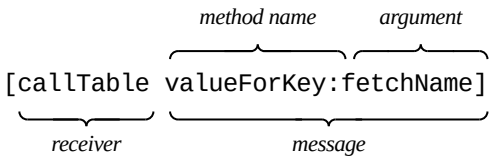
Typically, an object is regarded as a "black box," meaning that a program never directly accesses an object's variables. Indeed, a program shouldn't even need to know what variables an object has in order to perform its functions. Instead, the program accesses the object only through its methods. In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



Objects are the basic building blocks of Objective C applications. Each object encapsulates a particular area of functionality that the program needs. The interface to this functionality is provided by the object's methods.

Messages

To invoke one of the object's methods, you send it a *message*. In Objective C, a message expression is enclosed in square brackets, like this:



The term on the left is the object that receives the message, the *receiver*—here it's the HashTable object that YourCall uses to store records. Everything on the right is the message itself; it consists of a method name and any arguments the method requires. The message received by the HashTable object above tells it to search for and return the record for a customer name.

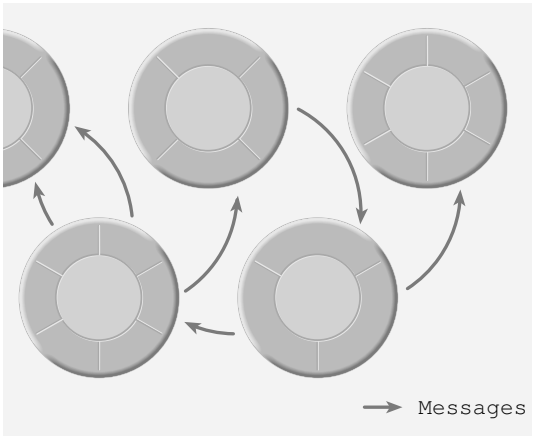
In Objective C, arguments follow colons, which are considered part of the method name. If a method has more than one argument—as Form's setStringValue:at: method does, for example—the name is broken apart to accept the arguments:

```
[customerForm
      setStringValue:NULL at:0];
```

Thus every argument can be given an identifying label within the method name—although sometimes a label isn't used and the colons stand next to each other. (The setStringValue:at: message shown above is used by the CallController's clearForm: method to clear the Name field of the Customer Form.)

Program Structure

Object-oriented programming is more than just another way of organizing data and functions. It permits complex programs to be conceived and constructed using a model that resembles—much more so than traditional programs—the way we organize the world around us. An object-oriented program can be thought of as a network of objects with well-defined behavior and characteristics, objects that interact through messages.



Every object in the network has a separate role to play. 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 divide functionality that doesn't have a graphical counterpart on-screen into a series of objects, assigning each one a different area of responsibility. Four such objects play prominent roles in the YourCall program: An Application object to run the program, a CallRecord object to represent each record, a HashTable object to store CallRecords, and a CallController object to move data between the user interface and storage.

Each object is a self-contained unit. 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 two very different kinds of objects might use the same names for their variables or have identically named methods. Both objects could receive the same message, but each would respond to it differently, in a way that's appropriate for its role in the application. The ability of one method to assume different forms in different objects is referred to as *polymorphism*.

Dynamic Binding

Although the purpose of a message is to invoke a method, a message isn't the same as a function call. In Objective C, both the message being sent

and the object to receive that message can be selected as the program runs. A run-time process finds the method implementation appropriate for the receiver of the message; it then "calls" this implementation and passes it the receiver's data structure.

This *dynamic binding* 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.

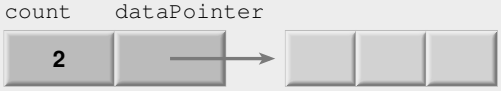
Dynamic binding even enables applications to deal with new kinds of objects, ones that were not envisioned when the application itself was built. For example, it lets Interface Builder send messages to objects such as DBModule loaded into the application by means of custom palettes.

CLASSES

Applications are composed of many different kinds of objects and often have more than one object of the same kind. YourCall, for example, has three buttons (each a ButtonCell object) and

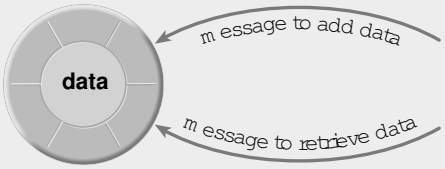
THE OBJECT-ORIENTED ADVANTAGE

Object-oriented programming delivers its greatest benefits to large and complex programs. But its advantages can also be demonstrated with a simple data structure such as might be used in any application.



With procedural programming techniques, the application is directly responsible for data manipulation. One problem with this is illustrated in the picture above: It shows a data structure consisting of a **count** variable and a data pointer. Since the application directly manipulates the data, it has the opportunity to introduce inconsistencies. Here, it has added an item to the data, but has forgotten to increment the count; the **count** variable says there are still only two data elements when in fact there are three. The structure has become inconsistent and unreliable.

Another problem is that all parts of the application must have intimate knowledge about the structure of the data. If the allocation of data elements were changed from a statically allocated array to a dynamically allocated linked list, it would affect every part of the application that accesses, adds, or deletes elements from the list.



With an object-oriented programming paradigm, the application as a whole wouldn't directly manipulate the data structure; rather, that task is entrusted to a particular object. Since the application doesn't directly access the data, it can't introduce inconsistencies. Note also that it's possible to change the implementation of the object without breaking other parts of the application. For example, the data storage method could be changed to optimize performance. So long as the object responds to the same messages, other parts of the application are unaffected by internal implementation details.

several different menu items (each an individual MenuCell object).

Objects of the same kind belong to the same *class*. When you want a new kind of object, you define a new class.

A class definition can be thought of 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.

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 specification, however. It can be assigned methods and receive messages just as an object can. As such it acts as a *class object*.

One of the primary functions of a class is to create new objects of the type the class defines. For example, the Button class creates new Button objects and the MenuCell class creates new MenuCells. 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 it. The following code is used in CallController's saveCall: method to create a new CallRecord object:

```
newRecord =
      [[CallRecord alloc] init];
```

The receiver for the alloc message is the CallRecord class. The alloc method dynamically allocates memory for a new object of the receiving class and returns the new object. The receiver for the init message is the new CallRecord object that was dynamically allocated by alloc. Once allocated and initialized, this new record is assigned to the local variable newRecord.

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

The Objective C type for an object, regardless of what class it belongs to, is id (internally, a pointer to the object's private data structure). The newRecord variable in the code sample above could be declared to be an id:

```
id  newRecord;
```

Objects can also be more restrictively typed, based on their class. For example, in the actual code, newRecord is typed as a CallRecord object:

```
CallRecord *newRecord;
newRecord =
      [[CallRecord alloc] init];
```

To take advantage of polymorphism, variables that reference objects are often of type id. The more restrictive typing by class enables the compiler to perform type-checking in assignment statements.

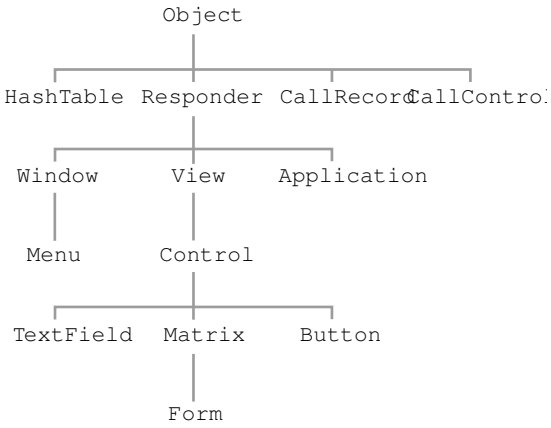
This discussion illustrates the dual nature of a class. In program code, the class defines a type of object. In the run-time system, the class acts as an object itself to receive messages like alloc.

Inheritance

An Objective C class definition doesn't have to specify every method and variable; it can inherit from other classes. 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 is its *superclass*.

The new class inherits all its superclass's behavior, so you don't need to reimplement 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 inherited methods. The technique for doing this is discussed later.)

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 root. The hierarchy of the principal classes used in YourCall is shown below:



Object is the root class of this hierarchy, as it is of all Objective C class hierarchies. From Object, other classes inherit the basic functionality that makes messaging work and enables objects to work together. To define a new class that doesn't need to inherit any of the special behavior encoded into an existing class, you make it a subclass of the Object class.

Inheritance makes it easy to bundle common functionality 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 View 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 not only simplifies the organization of the code that needs to be written for an application, it 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 View, 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.

Your applications can use subclassing to make greater use of objects in the Application Kit. When an application needs an object to do drawing, it implements a subclass of View. When it needs an object that both draws and provides simple user interaction, it implements a subclass of Control.

Inheritance makes every class easily extensible—those provided by NeXTSTEP, those you create, and those offered by third party vendors. Any class can be the superclass for a new subclass.

Defining a Class

Classes are defined in two parts: One part declares its 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; it can change without affecting the interface or the way the class is used.

The public interface for a class is usually declared in a header file that can be included in any program that makes use of the class. The declaration begins with the directive @interface and ends with @end.

```
@interface CallController : Object
/*
 * variable and method declarations
 */
@end
```

Here the @interface line declares that CallController inherits from the Object class; Object is the superclass for CallController.

Method declarations serve the same purpose as function prototypes. Here are declarations for two of CallController's methods:

```
- clearForm:sender;
- retrieveCall:sender;
```

The default argument and return type for a method is id. The initial '–' sign indicates that these methods are used by objects belonging to the class; a '+' precedes methods to be used by the class object itself.

The @implementation directive announces the start of a class definition.

```
@implementation CallController
/*
 * method definitions
 */
@end
```

Method definitions are much like function definitions. For example, this is CallController's implementation of the clearForm: method:

```
- clearForm:sender;
{
[queryText setStringValue:NULL];
[responseText setStringValue:NULL];
[customerForm
  setStringValue:NULL at:0];
[customerForm
  setStringValue:NULL at:1];
[customerForm
  setStringValue:NULL at:2];
[customerForm
  setStringValue:NULL at:3];
[customerForm
  setStringValue:NULL at:4];
return self;
}
```

Note that methods not only respond to messages, they often initiate messages of their own—just as one function might call another.

This example also points out two important differences between functions and methods:

- A method can refer directly to the receiving object's variables. (Here the clearForm: method refers to the CallController's queryText, responseText, and customerForm instance variables.) 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. (Here the method uses self to return the id of the receiver—the standard return value for Objective C methods.)

Because the default return value for a message is an id (usually self), messages can frequently be *nested*, resulting in code that's compact and easy to read:

```
newRecord =
   [[CallRecord alloc] init];
```

In this example, the object returned by alloc becomes the receiver for the init message.

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.

For example, if its superclass had a clearForm: method, CallController's version would override

the inherited version. Any CallController object would use the new method, rather than the inherited one. The new method would also be inherited by CallController subclasses.

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. This is done by overriding the method but including the old version in the new implementation.

For example, the CallRecord class inherits a write: method that archives Object's single instance variable to a typed stream. However, a CallRecord must also save the instance variables that it declares in order to store the data for a record in the archive.

The CallRecord class therefore implements a new version of write:, but incorporates the superclass's version through a message to super:

```
- write:(NXTypedStream *)stream
{
  [super write:stream];
  NXWriteTypes(stream,
  "*******", &name,
  &street, &city,
  &state, &phone,
  &query, &response);
  return self;
}
```

super is a special receiver in the Objective C language. It indicates that an inherited method should be performed, rather than one defined in the current class. After the message to super is sent, the CallRecord object has written the instance variable it inherits from Object to the stream. The NXWriteTypes() function then writes the instance variables that are unique to the stream.

USING AN OBJECT-ORIENTED KIT

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 with compiled versions of the class implementations, and some documentation.

In NeXTSTEP, groups of object-oriented classes are organized as kits. Each kit provides a complete framework for a particular functionality, with specific behavior organized in well-defined classes. The Application Kit, for example, provides classes that implement basic application structure and that provide "glue" to connect your custom objects to an application. In addition, the Application Kit provides a variety of ready-to-use application features, with classes for everything from editing text and managing windows to interacting with the pasteboard.

Other kits offer similar frameworks. The Database Kit implements, in a vendor-independent way, objects for accessing data in commercial database management systems. The Indexing Kit provides standardized behavior for organizing and accessing text and other large, heterogeneous data items. In addition, there are smaller groupings of objects that provide widely used functionality, such as the "common classes" like HashTable, Storage, and List for data manipulation.

When you use the Application Kit and other components of NeXTSTEP, you are in effect building your application in partnership with the programmers at NeXT. Kit classes are supported by NeXT and are continually improved with each new release. It's likely that most of the objects in your application will come from Kit classes. Only two of the classes shown earlier in this section in the inheritance hierarchy for YourCall—CallController and CallRecord—were actually defined in that application. The rest were provided by NeXTSTEP.

Implementing Methods

No matter how many ready-made classes there are or how much of a program framework the Application Kit provides, you must write code to do the things that are unique to your application. The task is to fit your pieces with the pieces that are already provided.

Because you will be developing classes in the context of existing class definitions, you'll find yourself implementing methods to respond to messages sent from other parts of the application—parts that you don't write yourself. Although your code implements the methods, it won't initiate any messages to invoke them. Much of the task of writing object-oriented programs is simply implementing methods that respond to system-generated messages.

For example, although CallRecord implements a read: method for unarchiving its instance variables, the code of YourCall never invokes that method. Instead, the application unarchives the data by invoking a function NXReadObjects() on a HashTable. Through this function, the HashTable in turn sends each of the objects it contains a read: message. All you need do is implement the application-specific method to respond to these messages.

Delegation

One way to have the code you write "fit" an object-oriented kit is to develop subclasses of kit classes, as described above. Another way is to define classes for objects that have a special relationship to kit objects. The most general type of relationship is for your object to accept responsibilities delegated to it by the kit object. The *delegate* acts on behalf of the other object and thus can extend or modify its behavior.

In the discussion of inter-application services in "Extending the Advantage," you saw how the Application object requires a delegate to implement a service. The Application object implements the basic mechanism for initiating the response to a request for services, while its delegate provides access to the service.

Delegation can eliminate the need to define a subclass of a complex class, such as Application. The delegate receives messages notifying it of significant happenings in the other object—for example, when text is edited, a file needs to be opened, or a window becomes the one the user is working in—and it responds appropriately. Application-specific code is thus confined to methods defined for the delegate, rather than subclass methods that override methods in the complex class.

Sometimes an object will simply forward a message it receives to its delegate. Thus it's possible for an application to implement a response to the message in a class definition for the delegate, rather than in a subclass of the Kit class.

Delegation is often used to coordinate a group of objects. A single object can act as the delegate for several other objects, even objects belonging to different classes. Since the delegate is notified of significant happenings in all the objects it is the delegate for, it can take appropriate action to keep its objects synchronized.

Application Kit classes must be used by many different applications, so it makes sense for the Kit to adopt a strategy that assigns application-specific code to a delegate. If you want any of your own classes to be reusable in other applications, it may make sense for you to adopt the same strategy.

Categories and Protocols

In addition to subclassing and delegation, 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 NeXTSTEP software kits. 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 to 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.

One use of protocols is to allow a concealed object to identify its interface to others. This use is illustrated in the Distributed Objects discussion in "Extending the Advantage." There, the server application publishes a protocol, CallProvider, that it conforms to. The client application need only know about the protocol in order to send the appropriate messages to the server.

Protocols are also useful for declaring methods that other objects must implement, without specifying the exact implementations. For example, the CallData protocol specified methods for retrieving and setting data in a record, without specifying their implementation. NeXTSTEP provides a number of such protocols. For example, the spell-checking protocols enable other developers to implement spell-checking objects that can plug into the Application Kit's Text object.

Summary

Subclasses of existing classes, methods that respond to system-generated messages, delegates, categories, protocols—these are all mechanisms you can use to adapt the code you write to the framework provided by NeXTSTEP (and to adapt the classes provided with NeXTSTEP to the needs of your application).

To understand how to use these object-oriented techniques when writing an application for NeXTSTEP, you need to learn more about the program framework it provides for event-driven applications. The next section discusses this framework and the facilities provided by the Application Kit and other components of NeXTSTEP.