The Run-Time System

The Objective-C language defers as many decisions as it can from compile time and link time to run time. Whenever possible, it does things dynamically. This means that the language requires not just a compiler, but also a run-time system to execute the compiled code. The run-time system acts as a kind of operating system for the Objective-C language; it's what makes the language work.

Objective-C programs interact with the run-time system at three distinct levels:

- Through Objective-C source code. For the most part, the run-time system works automatically and behind the scenes. You use it just by writing and compiling Objective-C source code.

  It's up to the compiler to produce the data structures that the run-time system requires and to arrange the run-time function calls that carry out language instructions. The data structures capture information found in class and category definitions and in protocol declarations; they include the class and protocol objects discussed earlier, as well as method selectors, instance variable templates, and other information distilled from source code. The principal run-time function is the one that sends messages, as described under ªHow Messaging Worksº in Chapter 2. It's invoked by source-code message expressions.

- Through a method interface defined in the NSObject class. Every object inherits from the NSObject class, so every object has access to the methods it defines. Most NSObject methods interact with the run-time system.

  Some of these methods simply query the system for information. The preceding chapters, for example, mentioned the **class** method, which asks an object to identify its class, **isKindOfClass:** and **isMemberOfClass:**, which test an object's position in the inheritance hierarchy, **respondsToSelector:**, which checks whether an object can accept a particular message, **conformsToProtocol:**, which checks whether it conforms to a protocol, and **methodForSelector:**, which asks for the address of a method implementation. Methods like these give an object the ability to introspect about itself.

  Other methods set the run-time system in motion. For example, **perform:** and its companions initiate messages, and **alloc** produces a new object properly connected to its class.

  All these methods were mentioned in previous chapters and are described in detail in the NSObject class specification in the *Foundation Framework Reference*.

- Through direct calls to run-time functions. The run-time system has a public interface, consisting mainly of a set of functions. Many are functions that duplicate what you get automatically by writing Objective-C code or what the NSObject class provides with a method interface. Others manipulate low-level run-time processes and data structures. These functions make it possible to develop other interfaces to the run-time system and produce tools that augment the development environment; they're not needed when programming in Objective-C.

  However, a few of the run-time functions might on occasion be useful when writing an Objective-C program. These functionsÐsuch as **sel_getUid()**, which returns a method selector for a method name, and **objc_msgSend()**, which sends a message to an objectÐare defined in the Objective-C run time system described at various places in the text of this manual.

Because the NSObject class is at the root of all inheritance hierarchies, the methods it defines are inherited by all classes. Its methods therefore establish behaviors that are inherent to every instance and every class object. However, in a few cases, the NSObject class merely defines a framework for how something should be done; it doesn't provide all the necessary code itself.

For example, the NSObject class defines a **description** method that should return an NSString associated with the receiver. If you define a class of named objects, you must implement a **description** method to return the

specific character string associated with the receiver. NSObject's version of the method can't know what that name will be, so it merely returns the class name as a default.

This chapter looks at three areas where the NSObject class provides a framework and defines conventions, but where you may need to write code to fill in the details:

- Allocating and initializing new instances of a class, and deallocating instances when they're no longer needed
- Forwarding messages to another object
- Dynamically loading new modules into a running program

Other conventions of the NSObject class are described in the NSObject class specification in the *Foundation Framework Reference*.

Allocation and Initialization

It takes two steps to create an object in Objective-C. You must both:

- Dynamically allocate memory for the new object, and
- Initialize the newly allocated memory to appropriate values.

An object isn't fully functional until both steps have been completed. As discussed in Chapter 2, each step is accomplished by a separate method, but typically in a single line of code:

```
id anObject = [[Rectangle alloc] init];
```

Separating allocation from initialization gives you individual control over each step so that each can be modified independently of the other. The following sections look first at allocation and then at initialization, and discuss how they are in fact controlled and modified.

Allocating Memory For Objects

In Objective-C, memory for new objects is allocated using class methods defined in the NSObject class. NSObject defines two principal methods for this purpose, **alloc** and **allocWithZone:**.

```
+ (id)alloc;
+ (id)allocWithZone:(NSZone *)zone;
```

These methods allocate enough memory to hold all the instance variables for an object belonging to the receiving class. They don't need to be overridden and modified in subclasses.

Initializing New Objects

The **alloc** and **allocWithZone:** methods initialize a new object's **isa** instance variable so that it points to the object's class (the class object). All other instance variables are set to 0. Usually, an object needs to be more specifically initialized before it can be safely used.

This initialization is the responsibility of class-specific instance methods that, by convention, begin with the abbreviation ªinitº. If the method takes no arguments, the method name is just those four letters, **init**. If it takes arguments, labels for the arguments follow the ªinitº prefix. For example, an NSView can be initialized with an **initWithFrame:** method.

Every class that declares instance variables must provide an **init...** method to initialize them. The NSObject class declares the **isa** variable and defines an **init** method. However, since **isa** is initialized when memory for a new object is allocated, all NSObject's **init** method does is return **self**. NSObject declares the method mainly to establish the naming convention described above.

The Returned Object

An **init...** method normally initializes the instance variables of the receiver, then returns it. It's the responsibility of the method to return an object that can be used without error.

However, in some cases, this responsibility can mean returning a different object than the receiver. For example, if a class keeps a list of named objects, it might provide an **initWithName:** method to initialize new instances. If there can be no more than one object per name, **initWithName:** might refuse to assign the same name to two objects. When asked to assign a new instance a name that's already being used by another object, it might free the newly allocated instance and return the other objectÐthus ensuring the uniqueness of the name while at the same time providing what was asked for, an instance with the requested name.

In a few cases, it might be impossible for an **init...** method to do what it's asked to do. For example, an **initFromFile:** method might get the data it needs from a file passed as an argument. If the file name it's passed

doesn't correspond to an actual file, it won't be able to complete the initialization. In such a case, the **init...** method could free the receiver and return **nil**, indicating that the requested object can't be created.

Because an **init...** method might return an object other than the newly allocated receiver, or even return **nil**, it's important that programs use the value returned by the initialization method, not just that returned by **alloc** or **allocWithZone:**. The following code is very dangerous, since it ignores the return of **init**.

```
id anObject = [SomeClass alloc];
[anObject init];
[anObject someOtherMessage];
```

It's recommended that you combine allocation and initialization messages:

```
id anObject = [[SomeClass alloc] init];
[anObject someOtherMessage];
```

If there's a chance that the **init...** method might return **nil**, the return value should be checked before proceeding:

```
id anObject = [[SomeClass alloc] init];
if ( anObject )
    [anObject someOtherMessage];
else
    . . .
```

Arguments

An **init...** method must ensure that all of an object's instance variables have reasonable values. This doesn't mean that it needs to provide an argument for each variable. It can set some to default values or depend on the fact that (except for **isa**) all bits of memory allocated for a new object are set to 0. For example, if a class requires its instances to have a name and a data source, it might provide an **initWithName:fromFile:** method, but set nonessential instance variables to arbitrary values or allow them to have the null values set by default. It could then rely on methods like **setEnabled:**, **setFriend:**, and **setDimensions:** to modify default values after the initialization phase had been completed.

Any **init...** method that takes arguments must be prepared to handle cases where an inappropriate value is

passed. One option is to substitute a default value, and to let a null argument explicitly evoke the default.

Coordinating Classes

Every class that declares instance variables must provide an **init...** method to initialize them (unless the variables require no initialization). The **init...** methods the class defines initialize only those variables declared in the class. Inherited instance variables are initialized by sending a message to **super** to perform an initialization method defined somewhere farther up the inheritance hierarchy:

```
- initWithName:(char *)string
{
    if ( self = [super init] ) {
        name = (char *)NSZoneMalloc([self zone],
€€€€€€€€€€€€strlen(string) + 1);
        strcpy(name, string);
        return self;
    }
    return nil;
}
```

The message to **super** chains together initialization methods in all inherited classes. Because it comes first, it ensures that superclass variables are initialized before those declared in subclasses. For example, a Rectangle object must be initialized as an NSObject, a Graphic, and a Shape before it's initialized as a Rectangle. (See Chapter 2 for a figure illustrating the Rectangle inheritance hierarchy.)

The connection between the **initWithName:** method illustrated above and the inherited **init** method it incorporates is diagrammed in the figure below:

InitSimple.eps ¬

A class must also make sure that all inherited initialization methods work. For example, if class A defines an **init** method and its subclass B defines an **initWithName:** method, as shown in the figure above, B must also make sure that an **init** message will successfully initialize B instances. The easiest way to do that is to replace the inherited **init** method with a version that invokes **initWithName:**.

```
- init
{
```

```
    return [self initWithName:"default"];
}
```

The **initWithName:** method would, in turn, invoke the inherited method, as was shown in the example and figure above. That figure can be modified to include B's version of **init**, as shown below:

InitA_B.eps ¬

Covering inherited initialization methods makes the class you define more portable to other applications. If you leave an inherited method uncovered, someone else may use it to produce incorrectly initialized instances of your class.

The Designated Initializer

In the example above, **initWithName:** would be the *designated initializer* for its class (class B). The designated initializer is the method in each class that guarantees inherited instance variables are initialized (by sending a message to **super** to perform an inherited method). It's also the method that does most of the work, and the one that other initialization methods in the same class invoke. It's an OPENSTEP convention that the designated initializer is always the method that allows the most freedom to determine the character of a new instance (the one with the most arguments).

It's important to know the designated initializer when defining a subclass. For example, suppose we define class C, a subclass of B, and implement an **initWithName:fromFile:** method. In addition to this method, we have to make sure that the inherited **init** and **initWithName:** methods also work for instances of C. This can be done just by covering B's **initWithName:** with a version that invokes **initWithName:fromFile:**.

```
- initWithName:(char *)string
{
    return [self initWithName:string fromFile:NULL];
}
```

For an instance of the C class, the inherited **init** method will invoke this new version of **initWithName:** which will invoke **initWithName:fromFile:**. The relationship between these methods is diagrammed below.

InitB_C.eps ¬

This figure omits an important detail. The **initWithName:fromFile:** method, being the designated initializer for the

C class, will send a message to **super** to invoke an inherited initialization method. But which of B's methods should it invoke, **init** or **initWithName:**? It can't invoke **init**, for two reasons:

- Circularity would result (**init** invokes C's **initWithName:**, which invokes **initWithName:fromFile:**, which invokes **init** again).
- It won't be able to take advantage of the initialization code in B's version of **initWithName:**.

Therefore, **initWithName:fromFile:** must invoke **initWithName:**.

```
- initWithName:(char *)string fromFile:(char *)pathname
{
    if ( self = [super initWithName:string] )
        . . .
}
```

The general principle is this:

> *The designated initializer in one class must, through a message to* ***super****, invoke the designated initializer in an inherited class.*

Designated initializers are chained to each other through messages to **super**, while other initialization methods are chained to designated initializers through messages to **self**.

The figure below shows how all the initialization methods in classes A, B, and C are linked. Messages to **self** are shown on the left and messages to **super** are shown on the right.

InitAll.eps ¬

Note that B's version of **init** sends a message to **self** to invoke the **initWithName:** method. Therefore, when the receiver is an instance of the B class, it will invoke B's version of **initWithName:**, and when the receiver is an instance of the C class, it will invoke C's version.

Combining Allocation and Initialization

By convention, in OPENSTEP classes define creation methods that combine the two steps of allocating and initializing to return new, initialized instances of the class. These methods typically take the form + *className*... where *className* is the name of the class. For instance, NSString has the following methods

(among others):

```
+ (NSString *)stringWithCString:(const char *)bytes;
+ (NSString *)stringWithFormat:(NSString *)format, ...;
```

Similarly, NSArray defines the following class methods that combine allocation and initialization:

```
+ (id)array;
+ (id)arrayWithObject:(id)anObject;
+ (id)arrayWithObjects:(id)firstObj, ...;
```

Instances created with any of these methods will be deallocated automatically, so you don't have to release them unless you first retain them.

Methods that combine allocation and initialization are particularly valuable if the allocation must somehow be informed by the initialization. For example, if the data for the initialization is taken from a file, and the file might contain enough data to initialize more than one object, it would be impossible to know how many objects to allocate until the file is opened. In this case, you might implement a **listFromFile:** method that takes the name of the file as an argument. It would open the file, see how many objects to allocate, and create a List object large enough to hold all the new objects. It would then allocate and initialize the objects from data in the file, put them in the List, and finally return the List.

It also makes sense to combine allocation and initialization in a single method if you want to avoid the step of blindly allocating memory for a new object that you might not use. As mentioned under ªThe Returned Objectº above, an **init...** method might sometimes substitute another object for the receiver. For example, when **initWithName:** is passed a name that's already taken, it might free the receiver and in its place return the object that was previously assigned the name. This means, of course, that an object is allocated and freed immediately without ever being used.

If the code that checks whether the receiver should be initialized is placed inside the method that does the allocation instead of inside **init...**, you can avoid the step of allocating a new instance when one isn't needed.

In the following example, the **soloist** method ensures that there's no more than one instance of the Soloist class. It allocates and initializes an instance only once:

```
+ soloist
```

```
{
    static Soloist *instance = nil;

    if ( instance == nil )
        instance = [[self alloc] init];
    return instance;
}
```

Deallocation

The NSObject class defines a **dealloc** method that relinquishes the memory that was originally allocated for an object. You rarely invoke **dealloc** directly, however, because OPENSTEP provides a mechanism for the automatic disposal of objects (which makes use of **dealloc**). For more information on this automatic object disposal mechanism, see the introduction to the *Foundation Framework Reference*.

The purpose of a **dealloc** message is to deallocate all the memory occupied by the receiver. NSObject's version of the method deallocates the receiver's instance variables, but doesn't follow any variable that points to other memory. If the receiver allocated any additional memoryÐto store a character string or an array of structures, for exampleÐthat memory must also be deallocated (unless it's shared by other objects). Similarly, if the receiver is served by another object that would be rendered useless in its absence, that object must also be deallocated.

Therefore, it's necessary for subclasses to override NSObject's version of **dealloc** and implement a version that deallocates all of the other memory the object occupies. Every class that has its objects allocate additional memory must have its own **dealloc** method. Each version of **dealloc** ends with a message to **super** to perform an inherited version of the method, as illustrated in the following example:

```
- dealloc {
    [companion release];
    free(privateMemory);
    vm_deallocate(task_self(), sharedMemory, memorySize);
    [super dealloc];
}
```

By working its way up the inheritance hierarchy, every **dealloc** message eventually invokes NSObject's version of the of the method.

Forwarding

It's an error to send a message to an object that can't respond to it. However, before announcing the error, the run-time system gives the receiving object a second chance to handle the message. It sends the object a **forwardInvocation:** message with an NSInvocation object as its sole argumentÐthe NSInvocation object encapsulates the original message and the arguments that were passed with it.

You can implement a **forwardInvocation:** method to give a default response to the message, or to avoid the error in some other way. As its name implies, **forwardInvocation:** is commonly used to forward the message to another object.

To see the scope and intent of forwarding, imagine the following scenarios: Suppose, first, that you're designing an object that can respond to a **negotiate** message, and you want its response to include the response of another kind of object. You could accomplish this easily by passing a **negotiate** message to the other object somewhere in the body of the **negotiate** method you implement.

Take this a step further, and suppose that you want your object's response to a **negotiate** message to be exactly the response implemented in another class. One way to accomplish this would be to make your class inherit the method from the other class. However, it might not be possible to arrange things this way. There may be good reasons why your class and the class that implements **negotiate** are in different branches of the inheritance hierarchy.

Even if your class can't inherit the **negotiate** method, you can still ªborrowº it by implementing a version of the method that simply passes the message on to an instance of the other class:

```
- negotiate
{
    if ( [someOtherObject respondsTo:@selector(negotiate)] )
        return [someOtherObject negotiate];
    return self;
}
```

This way of doing things could get a little cumbersome, especially if there were a number of messages you wanted your object to pass on to the other object. You'd have to implement one method to cover each method

you wanted to borrow from the other class. Moreover, it would be impossible to handle cases where you didn't know, at the time you wrote the code, the full set of messages that you might want to forward. That set might depend on events at run time, and it might change as new methods and classes are implemented in the future.

The second chance offered by a **forwardInvocation:** message provides a less ad hoc solution to this problem, and one that's dynamic rather than static. It works like this: When an object can't respond to a message because it doesn't have a method matching the selector in the message, the run-time system informs the object by sending it a **forwardInvocation:** message. Every object inherits a **forwardInvocation:** method from the NSObject class. However, NSObject's version of the method simply invokes **doesNotRecognizeSelector:** due to the unrecognized message. By overriding NSObject's version and implementing your own, you can take advantage of the opportunity that the **forwardInvocation:** message provides to forward messages to other objects.

To forward a message, all a **forwardInvocation:** method needs to do is:

- Determine where the message should go, and
- Send it there with its original arguments.

The message can be sent with the **invokeWithTarget:** method:

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    if ([someOtherObject respondsToSelector:
€€€€€€€€€€€[anInvocation selector]])
€€€€€€€€[anInvocation invokeWithTarget:someOtherObject];
    else
      € [self doesNotRecognizeSelector:[anInvocation selector]];
}
```

The return value of the message that's forwarded is returned to the original sender. All types of return values can be delivered to the sender, including **id**s, structures, and double-precision floating point numbers.

A **forwardInvocation:** method can act as a distribution center for unrecognized messages, parceling them out to different receivers. Or it can be a transfer station, sending all messages to the same destination. It can translate one message into another, or simply ªswallowº some messages so there's no response and no error. A **forwardInvocation:** method can also consolidate several messages into a single response. What

forwardInvocation: does is up to the implementor. However, the opportunity it provides for linking objects in a forwarding chain opens up possibilities for program design.

Note: The **forwardInvocation:** method gets to handle messages only if they don't invoke an existing method in the nominal receiver. If, for example, you want your object to forward **negotiate** messages to another object, it can't have a **negotiate** method of its own. If it does, the message will never reach **forwardInvocation:**.

For more information on forwarding and invocations, see the NSInvocation class specification in the *Foundation Framework Reference*.

Forwarding and Multiple Inheritance

Forwarding mimics inheritance, and can be used to lend some of the effects of multiple inheritance to Objective-C programs. As shown in the figure below, an object that responds to a message by forwarding it appears to borrow or ªinheritº a method implementation defined in another class.

Forwarding.eps ¬

In this illustration, an instance of the Warrior class forwards a **negotiate** message to an instance of the Diplomat class. The Warrior will appear to negotiate like a Diplomat. It will seem to respond to the **negotiate** message, and for all practical purposes it does respond (although it's really a Diplomat that's doing the work).

The object that forwards a message thus ªinheritsº methods from two branches of the inheritance hierarchyÐits own branch and that of the object that responds to the message. In the example above, it will appear as if the Warrior class inherits from Diplomat as well as its own superclass.

Forwarding addresses most needs that lead programmers to value multiple inheritance. However, there's an important difference between the two: Multiple inheritance combines different capabilities in a single object. It tends toward large, multifaceted objects. Forwarding, on the other hand, assigns separate responsibilities to separate objects. It decomposes problems into smaller objects, but associates those objects in a way that's transparent to the message sender.

Surrogate Objects

Forwarding not only mimics multiple inheritance, it also makes it possible to develop lightweight objects that represent or ªcoverº more substantial objects. The surrogate stands in for the other object and funnels messages to it.

The proxy discussed under ªRemote Messagingº in Chapter 3 is such an object. A proxy takes care of the administrative details of forwarding messages to a remote receiver, making sure argument values are copied and retrieved across the connection, and so on. But it doesn't attempt to do much else; it doesn't duplicate the functionality of the remote object but simply gives the remote object a local address, a place where it can receive messages in another application.

Other kinds of surrogate objects are also possible. Suppose, for example, that you have an object that manipulates a lot of dataÐperhaps it creates a complicated image or reads the contents of a file on disk. Setting this object up could be time-consuming, so you prefer to do it lazilyÐwhen it's really needed or when system resources are temporarily idle. At the same time, you need at least a placeholder for this object in order for the other objects in the application to function properly.

In this circumstance, you could initially create, not the full-fledged object, but a lightweight surrogate for it. This object could do some things on its own, such as answer questions about the data, but mostly it would just hold a place for the larger object and, when the time came, forward messages to it. When the surrogate's **forwardInvocation:** method first receives a message destined for the other object, it would check to be sure that the object existed and would create it if it didn't All messages for the larger object go through the surrogate, so as far as the rest of the program is concerned, the surrogate and the larger object would be the same.

Making Forwarding Transparent

Although forwarding mimics inheritance, the NSObject class never confuses the two. Methods like **respondsToSelector:** and **isKindOfClass:** look only at the inheritance hierarchy, never at the forwarding chain. If, for example, a Warrior object is asked whether it responds to a **negotiate** message,

```
if ( [aWarrior respondsToSelector:@selector(negotiate)] )
    . . .
```

the answer will be NO, even though it can receive **negotiate** messages without error and respond to them, in a sense, by forwarding them to a Diplomat. (See the previous figure.)

In many cases, NO is the right answer. But it may not be. If you use forwarding to set up a surrogate object or to extend the capabilities of a class, the forwarding mechanism should probably be as transparent as inheritance. If you want your objects to act as if they truly inherited the behavior of the objects they forward messages to, you'll need to re-implement the **respondsToSelector:** and **isKindOfClass:** methods to include your

forwarding algorithm:

```
- (BOOL)respondsToSelector:(SEL)aSelector
{
    if ( [super respondsToSelector:aSelector] )
        return YES;
    else {
        /* Here, test whether the aSelector message can     *
         * be forwarded to another object and whether that  *
         * object can respond to it.  Return YES if it can. */
    }
    return NO;
}
```

In addition to **respondsToSelector:** and **isKindOfClass:**, the **instancesRespondToSelector:** method should also mirror the forwarding algorithm. This method rounds out the set. If protocols are used, the **conformsToProtocol:** method should likewise be added to the list. Similarly, if an object forwards any remote messages it receives, it should have a version of **methodSignatureForSelector:** that can return accurate descriptions of the methods that ultimately respond to the forwarded messages.

You might consider putting the forwarding algorithm somewhere in private code and have all these methods, **forwardInvocation:** included, call it.

Note: The methods mentioned above are described in the NSObject class specification in the *Foundation Framework Reference*. For information on **invokeWithTarget:**, see the NSInvocation class specification in the *Foundation Framework Reference*.

Dynamic Loading

An Objective-C program can load and link new classes and categories while it's running. The new code is incorporated into the program and treated identically to classes and categories loaded at the start.

Dynamic loading can be used to do a lot of different things. For example, device drivers are dynamically loaded into the kernel. Adaptors for database servers are dynamically loaded by the Enterprise

Objects™Framework.

In the OpenStep environment, dynamic loading is commonly used to allow applications to be customized. Others can write modules that your program will load at run timeÐmuch as Interface Builder loads custom palettes, OPENSTEP for Mach's Preferences application loads custom displays, and its Workspace Manager loads data inspectors. The loadable modules extend what your application can do. They contribute to it in ways that you permit, but could not have anticipated or defined yourself. You provide the framework, but others provide the code.

Although there are run-time functions that enable dynamic loading (**objc_loadModules()** and **objc_unloadModules()**, defined in **objc/objc-load.h**), OPENSTEP's NSBundle class provides a significantly more convenient interface for dynamic loadingÐone that's object-oriented and integrated with related services. See the NSBundle class specification in the *Foundation Framework Reference* for information on the NSBundle class and its use.