Objects, Classes, and Messaging

This chapter describes the fundamentals of objects, classes, and messaging as used and implemented by the Objective-C language. It also introduces the Objective-C runtime.

The Runtime System

The Objective-C language defers as many decisions as it can from *compile time* and *link time* to *runtime*. Whenever possible, it dynamically performs operations such as creating objects and determining what method to invoke. Therefore, the language requires not just a compiler, but also a runtime system to execute the compiled code. The runtime system acts as a kind of operating system for the Objective-C language; it's what makes the language work. Typically, however, you don't need to interact with the runtime directly. To understand more about the functionality it offers, though, see *Objective-C Runtime Programming Guide*.

Objects

As the name implies, object-oriented programs are built around *objects*. An object associates data with the particular operations that can use or affect that data. Objective-C provides a data type to identify an object variable without specifying a particular class of the object.

Object Basics

An object associates data with the particular operations that can use or affect that data. In Objective-C, these operations are known as the object's *methods*; the data they affect are its *instance variables* (in other environments they may be referred to as *ivars* or *member variables*). In essence, an object bundles a data structure (instance variables) and a group of procedures (methods) into a self-contained programming unit.

In Objective-C, an object's instance variables are internal to the object; generally, you get access to an object's state only through the object's methods (you can specify whether subclasses or other objects can access instance variables directly by using scope directives, see The Scope of Instance Variables). For others to find out something about an object, there has to be a method to supply the information. For example, a rectangle would have methods that reveal its size and position.

Moreover, an object sees only the methods that were designed for it; it can't mistakenly perform methods intended for other types of objects. Just as a C function protects its local variables, hiding them from the rest of the program, an object hides both its instance variables and its method implementations.

id

In Objective-C, object identifiers are of a distinct data type: `id`. This type is the general type for any kind of object regardless of class and can be used for instances of a class and for class objects themselves.

```
id anObject;
```

For the object-oriented constructs of Objective-C, such as method return values, `id` replaces `int` as the default data type. (For strictly C constructs, such as function return values, `int` remains the default type.)

The keyword `nil` is defined as a null object, an `id` with a value of `0`. `id`, `nil`, and the other basic types of Objective-C are defined in the header file `objc/objc.h`.

`id` is defined as pointer to an object data structure:

```
typedef struct objc_object {
    Class isa;
} *id;
```

Every object thus has an `isa` variable that tells it of what class it is an instance. Since the `Class` type is itself defined as a pointer:

```
typedef struct objc_class *Class;
```

the `isa` variable is frequently referred to as the "`isa` pointer."

Dynamic Typing

The `id` type is completely nonrestrictive. By itself, it yields no information about an object, except that it is an object. At some point, a program typically needs to find more specific information about the objects it contains. Since the `id` type designator can't supply this specific information to the compiler, each object has to be able to supply it at runtime.

The `isa` instance variable identifies the object's *class*—what kind of object it is. Objects with the same behavior (methods) and the same kinds of data (instance variables) are members of the same class.

Objects are thus *dynamically typed* at runtime. Whenever it needs to, the runtime system can find the exact class that an object belongs to, just by asking the object. (To learn more about the runtime, see *Objective-C Runtime Programming Guide*.) Dynamic typing in Objective-C serves as the foundation for dynamic binding, discussed later.

The `isa` variable also enables objects to perform introspection—to find out about themselves (or other objects). The compiler records information about class definitions in data structures for the runtime system to use. The functions of the runtime system use `isa` to find this information at runtime. Using the runtime system, you can, for example, determine whether an object implements a particular method or discover the name of its superclass.

Object classes are discussed in more detail under Classes.

It's also possible to give the compiler information about the class of an object by statically typing it in source code using the class name. Classes are particular kinds of objects, and the class name can serve as a type name. See Class Types and Enabling Static Behavior.

Memory Management

In any program, it is important to ensure that objects are deallocated when they are no longer needed—otherwise your application's memory footprint becomes larger than necessary. It is also important to ensure that you do not deallocate objects while they're still being used.

Objective-C offers three mechanisms for memory management that allow you to meet these goals:

- *Automatic Reference Counting* (ARC), where the compiler reasons about the lifetimes of objects.
- *Manual Reference Counting* (MRC, sometimes referred to as MRR for "manual retain/release"), where you are ultimately responsible for determining the lifetime of objects.

  Manual reference counting is described in *Advanced Memory Management Programming Guide*.
- *Garbage collection*, where you pass responsibility for determining the lifetime of objects to an automatic "collector."

  Garbage collection is described in *Garbage Collection Programming Guide*. (Not available for iOS—you cannot access this document through the iOS Dev Center.)

Object Messaging

This section explains the syntax of sending messages, including how you can nest message expressions. It also discusses the scope or “visibility” of an object’s instance variables, and the concepts of polymorphism and dynamic binding.

Message Syntax

To get an object to do something, you send it a *message* telling it to apply a method. In Objective-C, *message expressions* are enclosed in brackets:

```
[receiver message]
```

The *receiver* is an object, and the message tells it what to do. In source code, the message is simply the name of a method and any parameters that are passed to it. When a message is sent, the runtime system selects the appropriate method from the receiver’s repertoire and invokes it.

For example, this message tells the `myRectangle` object to perform its `display` method, which causes the rectangle to display itself:

```
[myRectangle display];
```

The message is followed by a “`;`” as is normal for any statement in C.

Because the method name in a message serves to “select” a method implementation, method names in messages are often referred to as *selectors*.

Methods can also take parameters, sometimes called *arguments*. A message with a single parameter affixes a colon (`:`) to the name and puts the parameter right after the colon:

```
[myRectangle setWidth:20.0];
```

For methods with multiple parameters, Objective-C's method names are interleaved with the parameters such that the method’s name naturally describes the parameters expected by the method. The imaginary message below tells the `myRectangle` object to set its origin to the coordinates (30.0, 50.0):

```
[myRectangle setOriginX: 30.0 y: 50.0]; // This is a good example of
                                        // multiple parameters
```

A selector name includes all the parts of the name, including the colons, so the selector in the preceding example is named `setOriginX:y:`. It has two colons, because it takes two parameters. The selector name does not, however, include anything else, such as return type or parameter types.

Important: The subparts of an Objective-C selector name are not optional, nor can their order be varied. In some languages, the terms “named parameters” and “keyword parameters” carry the implications that the parameters can vary at runtime, can have default values, can be in a different order, and can possibly have additional named parameters. None of these characteristics about parameters are true for Objective-C.

For all intents and purposes, an Objective-C method declaration is simply a C function that prepends two additional parameters (see Messaging in the *Objective-C Runtime Programming Guide*). Thus, the structure of an Objective-C method declaration differs from the structure of a method that uses named or keyword parameters in a language like Python, as the following Python example illustrates:

```
def func(a, b, NeatMode=SuperNeat, Thing=DefaultThing):
    pass
```

In this Python example, `Thing` and `NeatMode` might be omitted or might have different values when called.

In principle, a `Rectangle` class could instead implement a `setOrigin::` method with no label for the second parameter, which would be invoked as follows:

```
[myRectangle setOrigin:30.0 :50.0]; // This is a bad example of multiple parameters
```

While syntactically legal, `setOrigin::` does not interleave the method name with the parameters. Thus, the second parameter is effectively unlabeled and it is difficult for a reader of this code to determine the kind or purpose of the method's parameters.

Methods that take a variable number of parameters are also possible, though they're somewhat rare. Extra parameters are separated by commas after the end of the method name. (Unlike colons, the commas are not considered part of the name.) In the following example, the imaginary `makeGroup:` method is passed one required parameter (`group`) and three parameters that are optional:

```
[receiver makeGroup:group, memberOne, memberTwo, memberThree];
```

Like standard C functions, methods can return values. The following example sets the variable `isFilled` to `YES` if `myRectangle` is drawn as a solid rectangle, or `NO` if it's drawn in outline form only.

```
BOOL isFilled;

isFilled = [myRectangle isFilled];
```

Note that a variable and a method can have the same name.

One message expression can be nested inside another. Here, the color of one rectangle is set to the color of another:

```
[myRectangle setPrimaryColor:[otherRect primaryColor]];
```

Objective-C also provides a dot (`.`) operator that offers a compact and convenient syntax for invoking an object's accessor methods. The dot operator is often used in conjunction with the declared properties feature (see Declared Properties) and is described in Dot Syntax.

Sending Messages to nil

In Objective-C, it is valid to send a message to `nil`—it simply has no effect at runtime. There are several patterns in Cocoa that take advantage of this fact. The value returned from a message to `nil` may also be valid:

- If the method returns an object, then a message sent to `nil` returns `0` (`nil`). For example:

  ```
  Person *motherInLaw = [[aPerson spouse] mother];
  ```

  If the `spouse` object here is `nil`, then `mother` is sent to `nil` and the method returns `nil`.
- If the method returns any pointer type, any integer scalar of size less than or equal to `sizeof(void*)`, a `float`, a `double`, a `long double`, or a `long long`, then a message sent to `nil` returns `0`.
- If the method returns a `struct`, as defined by the *OS X ABI Function Call Guide* to be returned in registers, then a message sent to `nil` returns `0.0` for every field in the `struct`. Other `struct` data types will not be filled with zeros.
- If the method returns anything other than the aforementioned value types, the return value of a message sent to `nil` is undefined.

The following code fragment illustrates a valid use of sending a message to `nil`.

```
id anObjectMaybeNil = nil;


// this is valid
if ([anObjectMaybeNil methodThatReturnsADouble] == 0.0)
{
    // implementation continues...
}
```

The Receiver's Instance Variables

A method has automatic access to the receiving object's instance variables. You don't need to pass them to the method as parameters. For example, the `primaryColor` method illustrated above takes no parameters, yet it can find the primary color for `otherRect` and return it. Every method assumes the receiver and its instance variables, without having to declare them as parameters.

This convention simplifies Objective-C source code. It also supports the way object-oriented programmers think about objects and messages. Messages are sent to receivers much as letters are delivered to your home. Message parameters bring information from the outside to the receiver; they don't need to bring the receiver to itself.

A method has automatic access only to the receiver's instance variables. If it requires information about a variable stored in another object, it must send a message to the object asking it to reveal the contents of the variable. The `primaryColor` and `isFilled` methods shown earlier are used for just this purpose.

See Defining a Class for more information on referring to instance variables.

Polymorphism

As the earlier examples illustrate, messages in Objective-C appear in the same syntactic positions as function calls in standard C. But, because methods "belong to" an object, messages don't behave in the same way that function calls do.

In particular, an object can be operated on by only those methods that were defined for it. It can't confuse them with methods defined for other kinds of object, even if another object has a method with the same name. Therefore, two objects can respond differently to the same message. For example, each kind of object that receives a `display` message could display itself in a unique way. A `Circle` and a `Rectangle` would respond differently to identical instructions to track the cursor.

This feature, referred to as *polymorphism*, plays a significant role in the design of object-oriented programs. Together with dynamic binding, it permits you to write code that might apply to any number of different kinds of objects, without you having to choose at the time you write the code what kinds of objects they might be. They might even be objects that will be developed later, by other programmers working on other projects. If you write code that sends a `display` message to an `id` variable, any object that has a `display` method is a potential receiver.

Dynamic Binding

A crucial difference between function calls and messages is that a function and its parameters are joined together in the compiled code, but a message and a receiving object aren't united until the program is running and the message is sent. Therefore, the exact method invoked to respond to a message can be determined only at runtime, not when the code is compiled.

When a message is sent, a runtime messaging routine looks at the receiver and at the method named in the message. It locates the receiver's implementation of a method matching the name, "calls" the method, and passes it a pointer to the receiver's instance variables. (For more on this routine, see Messaging in *Objective-C Runtime Programming Guide*.)

This *dynamic binding* of methods to messages works hand in hand with polymorphism to give object-oriented programming much of its flexibility and power. Because each object can have its own version of a method, an Objective-C statement can achieve a variety of results, not by varying the message but by varying the object that receives the message. Receivers can be decided as the program runs; the choice of receiver can be made dependent on factors such as user actions.

When executing code based upon the Application Kit (*AppKit*), for example, users determine which objects receive messages from menu commands such as Cut, Copy, and Paste. The message goes to whatever object controls the current selection. An object that displays text would react to a `copy` message differently from an object that displays scanned images. An object that represents a set of shapes would respond differently to a `copy` message than a `Rectangle` would. Because messages do not select methods until runtime (from another perspective, because binding of methods to messages does not occur until runtime), these differences in behavior are isolated to the methods themselves. The code that sends the message doesn't have to be concerned with them; it doesn't even have to enumerate the possibilities. An application's objects can each respond in its own way to `copy` messages.

Objective-C takes dynamic binding one step further and allows even the message that's sent (the method selector) to be a variable determined at runtime. This mechanism is discussed in the section Messaging in *Objective-C Runtime Programming Guide*.

Dynamic Method Resolution

You can provide implementations of class and instance methods at runtime using dynamic method resolution. See Dynamic Method Resolution in *Objective-C Runtime Programming Guide* for more details.

Dot Syntax

Objective-C provides a dot (`.`) operator that offers an alternative to square bracket notation (`[]`) to invoke accessor methods. Dot syntax uses the same pattern that accessing C structure elements uses:

```
myInstance.value = 10;
printf("myInstance value: %d", myInstance.value);
```

When used with objects, however, dot syntax acts as "syntactic sugar"—it is transformed by the compiler into an invocation of an accessor method. Dot syntax does *not* directly get or set an instance variable. The code example above is exactly equivalent to the following:

```
[myInstance setValue:10];
printf("myInstance value: %d", [myInstance value]);
```

As a corollary, if you want to access an object's own instance variable using accessor methods, you must explicitly call out `self`, for example:

```
self.age = 10;
```

or the equivalent:

```
[self setAge:10];
```

If you do not use `self.`, you access the instance variable directly. In the following example, the set accessor method for `age` is *not* invoked:

```
age = 10;
```

If a `nil` value is encountered during property traversal, the result is the same as sending the equivalent message to `nil`. For example, the following pairs are all equivalent:

```
// Each member of the path is an object.
```

```
x = person.address.street.name;
x = [[[person address] street] name];


// The path contains a C struct.
// This will crash if window is nil or -contentView returns nil.
y = window.contentView.bounds.origin.y;
y = [[window contentView] bounds].origin.y;


// An example of using a setter.
person.address.street.name = @"Oxford Road";
[[[person address] street] setName: @"Oxford Road"];
```

Classes

An object-oriented program is typically built from a variety of objects. A program based on the Cocoa frameworks might use `NSMatrix` objects, `NSWindow` objects, `NSDictionary` objects, `NSFont` objects, `NSText` objects, and many others. Programs often use more than one object of the same kind or class—several `NSArray` objects or `NSWindow` objects, for example.

In Objective-C, you define objects by defining their class. The class definition is a prototype for a kind of object; it declares the instance variables that become part of every member of the class, and it defines a set of methods that all objects in the class can use.

The compiler creates just one accessible object for each class, a *class object* that knows how to build new objects belonging to the class. (For this reason it's traditionally called a *factory object*.) The class object is the compiled version of the class; the objects it builds are *instances* of the class. The objects that do the main work of your program are instances created by the class object at runtime.

All instances of a class have the same set of methods, and they all have a set of instance variables cut from the same mold. Each object gets its own instance variables, but the methods are shared.

By convention, class names begin with an uppercase letter (such as `Rectangle`); the names of instances typically begin with a lowercase letter (such as `myRectangle`).

Inheritance

Class definitions are additive; each new class that you define is based on another class from which it inherits methods and instance variables. The new class simply adds to or modifies what it inherits. It doesn't need to duplicate inherited code.

Inheritance links all classes together in a hierarchical tree with a single class at its root. When writing code that is based upon the Foundation framework, that root class is typically `NSObject`. Every class (except a root class) has a *superclass* one step nearer the root, and any class (including a root class) can be the superclass for any number of *subclasses* one step farther from the root. Figure 1-1 illustrates the hierarchy for a few of the classes used in a drawing program.

Figure 1-1 Some drawing program classes

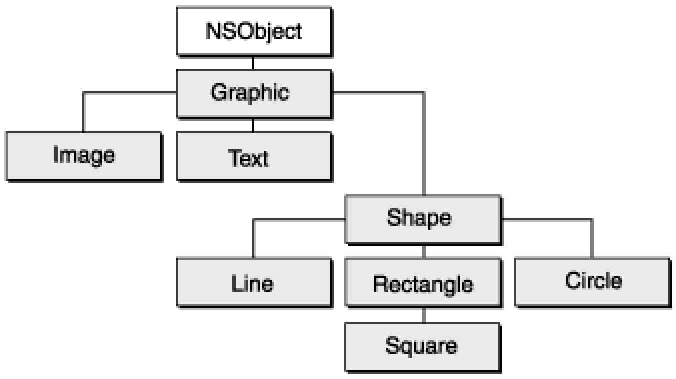


Figure 1-1 shows that the `Square` class is a subclass of the `Rectangle` class, the `Rectangle` class is a subclass of `Shape`, `Shape` is a subclass of `Graphic`, and `Graphic` is a subclass of `NSObject`. Inheritance is cumulative. So a `Square` object has the methods and instance variables defined for `Rectangle`, `Shape`, `Graphic`, and `NSObject`, as well as those defined specifically for `Square`. This is simply to say that an object of type `Square` isn't only a square, it's also a rectangle, a shape, a graphic, and an object of type `NSObject`.

Every class but `NSObject` can thus be seen as a specialization or an adaptation of another class. Each successive subclass further modifies the cumulative total of what's inherited. The `Square` class defines only the minimum needed to turn a rectangle into a square.

When you define a class, you link it to the hierarchy by declaring its superclass; every class you create must be the subclass of another class (unless you define a new root class). Plenty of potential superclasses are available. Cocoa includes the `NSObject` class and several frameworks containing definitions for more than 250 additional classes. Some are classes that you can use off the shelf and incorporate them into your program as is. Others you might want to adapt to your own needs by defining a subclass.

Some framework classes define almost everything you need, but leave some specifics to be implemented in a subclass. You can thus create very sophisticated objects by writing only a small amount of code and reusing work done by the programmers of the framework.

The NSObject Class

`NSObject` is a root class, and so doesn't have a superclass. It defines the basic framework for Objective-C objects and object interactions. It imparts to the classes and instances of classes that inherit from it the ability to behave as objects and cooperate with the runtime system.

A class that doesn't need to inherit any special behavior from another class should nevertheless be made a subclass of the `NSObject` class. Instances of the class must at least have the ability to behave like Objective-C objects at runtime. Inheriting this ability from the `NSObject` class is much simpler and much more reliable than reinventing it in a new class definition.

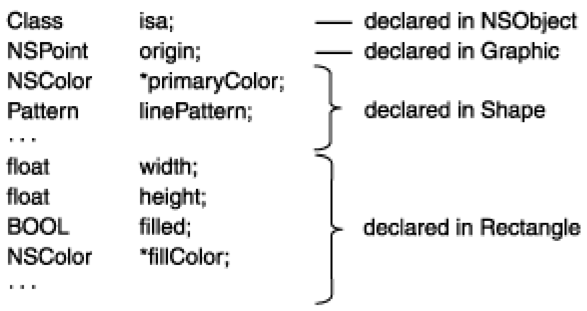
Note: Implementing a new root class is a delicate task and one with many hidden hazards. The class must duplicate much of what the `NSObject` class does, such as allocate instances, connect them to their class, and identify them to the runtime system. For this reason, you should generally use the `NSObject` class provided with Cocoa as the root class. For more information, see *NSObject Class Reference* and the *NSObject Protocol Reference*.

Inheriting Instance Variables

When a class object creates a new instance, the new object contains not only the instance variables that were defined for its class but also the instance variables defined for its superclass and for its superclass's superclass, all the way back to the root class. Thus, the `isa` instance variable defined in the `NSObject` class becomes part of every object. `isa` connects each object to its class.

Figure 1-2 shows some of the instance variables that could be defined for a particular implementation of a `Rectangle` class and where they may come from. Note that the variables that make the object a rectangle are added to the ones that make it a shape, and the ones that make it a shape are added to the ones that make it a graphic, and so on.

Figure 1-2 Rectangle instance variables



```
Class      isa;            — declared in NSObject
NSPoint    origin;         — declared in Graphic
NSColor    *primaryColor;  ⎫
Pattern    linePattern;    ⎬ declared in Shape
. . .                      ⎭
float      width;          ⎫
float      height;         ⎪
BOOL       filled;         ⎬ declared in Rectangle
NSColor    *fillColor;     ⎪
. . .                      ⎭
```

A class doesn't have to declare instance variables. It can simply define new methods and rely on the instance variables it inherits, if it needs any instance variables at all. For example, `Square` might not declare any new instance variables of its own.

Inheriting Methods

An object has access not only to the methods defined for its class but also to methods defined for its superclass, and for its superclass's superclass, all the way back to the root of the hierarchy. For instance, a `Square` object can use methods defined in the `Rectangle`, `Shape`, `Graphic`, and `NSObject` classes as well as methods defined in its own class.

Any new class you define in your program can therefore make use of the code written for all the classes above it in the hierarchy. This type of inheritance is a major benefit of object-oriented programming. When you use one of the object-oriented frameworks provided by Cocoa, your programs can take advantage of the basic functionality coded into the framework classes. You have to add only the code that customizes the standard functionality to your application.

Class objects also inherit from the classes above them in the hierarchy. But because they don't have instance variables (only instances do), they inherit only methods.

Overriding One Method with Another

There's one useful exception to inheritance: When you define a new class, you can implement a new method with the same name as one defined in a class farther up the hierarchy. The new method overrides the original; instances of the new class perform it rather than the original, and subclasses of the new class inherit it rather than the original.

For example, `Graphic` defines a `display` method that `Rectangle` overrides by defining its own version of `display`. The `Graphic` method is available to all kinds of objects that inherit from the `Graphic` class—but not to `Rectangle` objects, which instead perform the `Rectangle` version of `display`.

Although overriding a method blocks the original version from being inherited, other methods defined in the new class can skip over the redefined method and find the original (see Messages to self and super to learn how).

A redefined method can also incorporate the very method it overrides. When it does, the new method serves only to refine or modify the method it overrides, rather than replace it outright. When several classes in the hierarchy define the same method, but each new version incorporates the version it overrides, the implementation of the method is effectively spread over all the classes.

Although a subclass can override inherited methods, it can't override inherited instance variables. Because an object has memory allocated for every instance variable it inherits, you can't override an inherited variable by declaring a new one with the same name. If you try, the compiler will complain.

Abstract Classes

Some classes are designed only or primarily so that other classes can inherit from them. These *abstract classes* group methods and instance variables that can be used by a number of subclasses into a common definition. The abstract class is typically incomplete by itself, but contains useful code that reduces the implementation burden of its subclasses. (Because abstract classes must have subclasses to be useful, they're sometimes also called *abstract superclasses*.)

Unlike some other languages, Objective-C does not have syntax to mark classes as abstract, nor does it prevent you from creating an instance of an abstract class.

The `NSObject` class is the canonical example of an abstract class in Cocoa. You never use instances of the `NSObject` class in an application—it wouldn't be good for anything; it would be a generic object with the ability to do nothing in particular.

The `NSView` class, on the other hand, provides an example of an abstract class, instances of which you might occasionally use directly.

Abstract classes often contain code that helps define the structure of an application. When you create subclasses of these classes, instances of your new classes fit effortlessly into the application structure and work automatically with other objects.

Class Types

A class definition is a specification for a kind of object. The class, in effect, defines a data type. The type is based not just on the data structure the class defines (instance variables), but also on the behavior included in the definition (methods).

A class name can appear in source code wherever a type specifier is permitted in C—for example, as an argument to the `sizeof` operator:

```
int i = sizeof(Rectangle);
```

Static Typing

You can use a class name in place of `id` to designate an object's type:

```
Rectangle *myRectangle;
```

Because this way of declaring an object type gives the compiler information about the kind of object it is, it's known as *static typing*. Just as `id` is actually a pointer, objects are statically typed as pointers to a class. Objects are always typed by a pointer. Static typing makes the pointer explicit; `id` hides it.

Static typing permits the compiler to do some type checking—for example, to warn if an object could receive a message that it appears not to be able to respond to—and to loosen some restrictions that apply to objects generically typed `id`. In addition, it can make your intentions clearer to others who read your source code. However, it doesn't defeat dynamic binding or alter the dynamic determination of a receiver's class at runtime.

An object can be statically typed to its own class or to any class that it inherits from. For example, because inheritance makes a `Rectangle` object a kind of `Graphic` object (as shown in the example hierarchy in Figure 1-1), a `Rectangle` instance can be statically typed to the `Graphic` class:

```
Graphic *myRectangle;
```

Static typing to the superclass is possible here because a `Rectangle` object is a `Graphic` object. In addition, it's more than that because it also has the instance variables and method capabilities of `Shape` and `Rectangle` objects, but it's a `Graphic` object nonetheless. For purposes of type checking, given the declaration described here, the compiler considers `myRectangle` to be of type `Graphic`. At runtime, however, if the `myRectangle` object is allocated and initialized as an instance of `Rectangle`, it is treated as one.

See Enabling Static Behavior for more on static typing and its benefits.

Type Introspection

Instances can reveal their types at runtime. The `isMemberOfClass:` method, defined in the `NSObject` class, checks whether the receiver is an instance of a particular class:

```
if ( [anObject isMemberOfClass:someClass] )

    ...
```

The `isKindOfClass:` method, also defined in the `NSObject` class, checks more generally whether the receiver inherits from or is a member of a particular class (whether it has the class in its inheritance path):

```
if ( [anObject isKindOfClass:someClass] )

    ...
```

The set of classes for which `isKindOfClass:` returns `YES` is the same set to which the receiver can be statically typed.

Introspection isn't limited to type information. Later sections of this chapter discuss methods that return the class object, report whether an object can respond to a message, and reveal other information.

See *NSObject Class Reference* for more on `isKindOfClass:`, `isMemberOfClass:`, and related methods.

Class Objects

A class definition contains various kinds of information, much of it about instances of the class:

- The name of the class and its superclass
- A template describing a set of instance variables
- The declarations of method names and their return and parameter types
- The method implementations

This information is compiled and recorded in data structures made available to the runtime system. The compiler creates just one object, a *class object*, to represent the class. The class object has access to all the information about the class, which means mainly information about what instances of the class are like. It's able to produce new instances according to the plan put forward in the class definition.

Although a class object keeps the prototype of a class instance, it's not an instance itself. It has no instance variables of its own and it can't perform methods intended for instances of the class. However, a class definition can include methods intended specifically for the class object—*class methods* as opposed to *instance methods*. A class object inherits class methods from the classes above it in the hierarchy, just as instances inherit instance methods.

In source code, the class object is represented by the class name. In the following example, the `Rectangle` class returns the class version number using a method inherited from the `NSObject` class:

```
int versionNumber = [Rectangle version];
```

However, the class name stands for the class object only as the receiver in a message expression. Elsewhere, you need to ask an instance or the class to return the class `id`. Both respond to a `class` message:

```
id aClass = [anObject class];

id rectClass = [Rectangle class];
```

As these examples show, class objects can, like all other objects, be typed `id`. But class objects can also be more specifically typed to the `Class` data type:

```
Class aClass = [anObject class];
Class rectClass = [Rectangle class];
```

All class objects are of type `Class`. Using this type name for a class is equivalent to using the class name to statically type an instance.

Class objects are thus full-fledged objects that can be dynamically typed, receive messages, and inherit methods from other classes. They're special only in that they're created by the compiler, lack data structures (instance variables) of their own other than those built from the class definition, and are the agents for producing instances at runtime.

Note: The compiler also builds a metaclass object for each class. It describes the class object just as the class object describes instances of the class. But while you can send messages to instances and to the class object, the metaclass object is used only internally by the runtime system.

Creating Instances

A principal function of a class object is to create new instances. This code tells the `Rectangle` class to create a new rectangle instance and assign it to the `myRectangle` variable:

```
id  myRectangle;
myRectangle = [Rectangle alloc];
```

The `alloc` method dynamically allocates memory for the new object's instance variables and initializes them all to `0`—all, that is, except the `isa` variable that connects the new instance to its class. For an object to be useful, it generally needs to be more completely initialized. That's the function of an `init` method. Initialization typically follows immediately after allocation:

```
myRectangle = [[Rectangle alloc] init];
```

This line of code, or one like it, would be necessary before `myRectangle` could receive any of the messages that were illustrated in previous examples in this chapter. The `alloc` method returns a new instance and that instance performs an `init` method to set its initial state. Every class object has at least one method (like `alloc`) that enables it to produce new objects, and every instance has at least one method (like `init`) that prepares it for use. Initialization methods often take parameters to allow particular values to be passed and have keywords to label the parameters (`initWithPosition:size:`, for example, is a method that might initialize a new `Rectangle` instance), but every initialization method begins with "`init`".

Customization with Class Objects

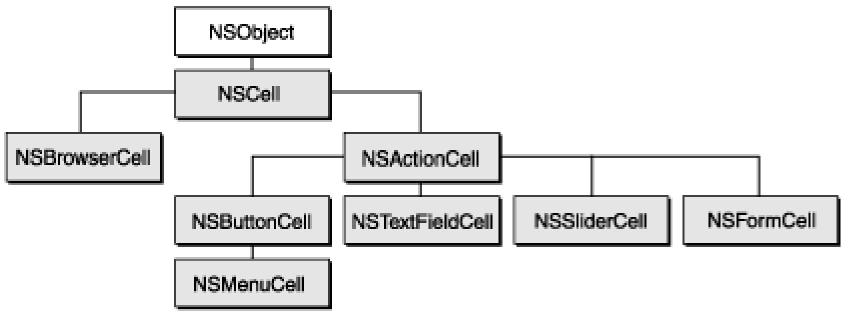
It's not just a whim of the Objective-C language that classes are treated as objects. It's a choice that has intended, and sometimes surprising, benefits for design. It's possible, for example, to customize an object with a class, where the class belongs to an open-ended set. In AppKit, for example, an `NSMatrix` object can be customized with a particular kind of `NSCell` object.

An `NSMatrix` object can take responsibility for creating the individual objects that represent its cells. It can do this when the matrix is first initialized and later when new cells are needed. The visible matrix that an `NSMatrix` object draws on the screen can grow and shrink at runtime, perhaps in response to user actions. When it grows, the matrix needs to be able to produce new objects to fill the new slots that are added.

But what kind of objects should they be? Each matrix displays just one kind of `NSCell`, but there are many different kinds. The inheritance hierarchy in Figure 1-3 shows some of those provided by

AppKit. All inherit from the generic `NSCell` class.

Figure 1-3 The inheritance hierarchy for NSCell



When a matrix creates `NSCell` objects, should they be `NSButtonCell` objects to display a bank of buttons or switches, `NSTextFieldCell` objects to display fields where the user can enter and edit text, or some other kind of `NSCell`? The `NSMatrix` object must allow for any kind of cell, even types that haven't been invented yet.

One solution to this problem would be to define the `NSMatrix` class as abstract and require everyone who uses it to declare a subclass and implement the methods that produce new cells. Because they would be implementing the methods, users could make certain that the objects they created were of the right type.

But this solution would require users of the `NSMatrix` class to do work that ought to be done in the `NSMatrix` class itself, and it unnecessarily proliferates the number of classes. Because an application might need more than one kind of matrix, each with a different kind of cell, it could become cluttered with `NSMatrix` subclasses. Every time you invented a new kind of `NSCell`, you'd also have to define a new kind of `NSMatrix`. Moreover, programmers on different projects would be writing virtually identical code to do the same job, all to make up for the failure of `NSMatrix` to do it.

A better solution, and the solution the `NSMatrix` class adopts, is to allow `NSMatrix` instances to be initialized with a kind of `NSCell`—that is, with a class object. The `NSMatrix` class also defines a `setCellClass:` method that passes the class object for the kind of `NSCell` object an `NSMatrix` should use to fill empty slots:

```
[myMatrix setCellClass:[NSButtonCell class]];
```

The `NSMatrix` object uses the class object to produce new cells when it's first initialized and whenever it's resized to contain more cells. This kind of customization would be difficult if classes weren't objects that could be passed in messages and assigned to variables.

Variables and Class Objects

When you define a new class, you can specify instance variables. Every instance of the class can maintain its own copy of the variables you declare—each object controls its own data. There is, however, no class variable counterpart to an instance variable. Only internal data structures, initialized from the class definition, are provided for the class. Moreover, a class object has no access to the instance variables of any instances; it can't initialize, read, or alter them.

For all the instances of a class to share data, you must define an external variable of some sort. The simplest way to do this is to declare a variable in the class implementation file:

```
int MCLSGlobalVariable;


@implementation MyClass
// implementation continues
```

In a more sophisticated implementation, you can declare a variable to be `static`, and provide class methods to manage it. Declaring a variable `static` limits its scope to just the class—and to just the part of the class that's implemented in the file. (Thus unlike instance variables, static variables cannot be inherited by, or directly manipulated by, subclasses.) This pattern is commonly used to define shared instances of a class (such as singletons; see Creating a Singleton Instance in *Cocoa Fundamentals Guide*).

```
static MyClass *MCLSSharedInstance;


@implementation MyClass


+ (MyClass *)sharedInstance
{
    // check for existence of shared instance
    // create if necessary
    return MCLSSharedInstance;
}
// implementation continues
```

Static variables help give the class object more functionality than just that of a *factory* producing instances; it can approach being a complete and versatile object in its own right. A class object can be used to coordinate the instances it creates, dispense instances from lists of objects already created, or manage other processes essential to the application. In the case when you need only one object of a particular class, you can put all the object's state into static variables and use only class methods. This saves the step of allocating and initializing an instance.

Note: It is also possible to use external variables that are not declared `static`, but the limited scope of static variables better serves the purpose of encapsulating data into separate objects.

Initializing a Class Object

If you want to use a class object for anything besides allocating instances, you may need to initialize it just as you would an instance. Although programs don't allocate class objects, Objective-C does provide a way for programs to initialize them.

If a class makes use of static or global variables, the `initialize` method is a good place to set their initial values. For example, if a class maintains an array of instances, the `initialize` method could set up the array and even allocate one or two default instances to have them ready.

The runtime system sends an `initialize` message to every class object before the class receives any other messages and after its superclass has received the `initialize` message. This sequence gives the class a chance to set up its runtime environment before it's used. If no initialization is required, you don't need to write an `initialize` method to respond to the message.

Because of inheritance, an `initialize` message sent to a class that doesn't implement the `initialize` method is forwarded to the superclass, even though the superclass has already received the `initialize` message. For example, assume class A implements the `initialize` method, and class B inherits from class A but does not implement the `initialize` method. Just before class B is to receive its first message, the runtime system sends `initialize` to it. But, because class B doesn't implement `initialize`, class A's `initialize` is executed instead. Therefore, class A should ensure that its initialization logic is performed only once, and for the appropriate class.

To avoid performing initialization logic more than once, use the template in Listing 1-1 when implementing the `initialize` method.

Listing 1-1 Implementation of the initialize method

```
+ (void)initialize
```

```
{
  if (self == [ThisClass class]) {
        // Perform initialization here.
        ...
    }
}
```

Note: Remember that the runtime system sends `initialize` to each class individually. Therefore, in a class's implementation of the `initialize` method, you must not send the `initialize` message to its superclass.

Methods of the Root Class

All objects, classes and instances alike, need an interface to the runtime system. Both class objects and instances should be able to introspect about their abilities and to report their place in the inheritance hierarchy. It's the province of the `NSObject` class to provide this interface.

So that `NSObject` methods don't have to be implemented twice—once to provide a runtime interface for instances and again to duplicate that interface for class objects—class objects are given special dispensation to perform instance methods defined in the root class. When a class object receives a message that it can't respond to with a class method, the runtime system determines whether there's a root instance method that can respond. The only instance methods that a class object can perform are those defined in the root class, and only if there's no class method that can do the job.

For more on this peculiar ability of class objects to perform root instance methods, see *NSObject Class Reference*.

Class Names in Source Code

In source code, class names can be used in only two very different contexts. These contexts reflect the dual role of a class as a data type and as an object:

- The class name can be used as a type name for a kind of object. For example:

  ```
  Rectangle *anObject;
  ```

  Here `anObject` is statically typed to be a pointer to a `Rectangle` object. The compiler expects it to have the data structure of a `Rectangle` instance and to have the instance methods defined and inherited by the `Rectangle` class. Static typing enables the compiler to do better type checking and makes source code more self-documenting. See Enabling Static Behavior for details.

  Only instances can be statically typed; class objects can't be, because they aren't members of a class, but rather belong to the `Class` data type.

- As the receiver in a message expression, the class name refers to the class object. This usage was illustrated in several of the earlier examples. The class name can stand for the class object only as a message receiver. In any other context, you must ask the class object to reveal its `id` (by sending it a `class` message). This example passes the `Rectangle` class as a parameter in an `isKindOfClass:` message:

  ```
  if ( [anObject isKindOfClass:[Rectangle class]] )
      ...
  ```

  It would have been illegal to simply use the name "Rectangle" as the parameter. The class name can only be a receiver.

  If you don't know the class name at compile time but have it as a string at runtime, you can use `NSClassFromString` to return the class object:

```
NSString *className;
    ...
if ( [anObject isKindOfClass:NSClassFromString(className)] )
    ...
```

This function returns `nil` if the string it's passed is not a valid class name.

Class names exist in the same namespace as global variables and function names. A class and a global variable can't have the same name. Class names are the only names with global visibility in Objective-C.

Testing Class Equality

You can test two class objects for equality using a direct pointer comparison. It is important, though, to get the correct class. There are several features in the Cocoa frameworks that dynamically and transparently subclass existing classes to extend their functionality (for example, key-value observing and Core Data do this—see *Key-Value Observing Programming Guide* and *Core Data Programming Guide* respectively). In a dynamically-created subclass, the `class` method is typically overridden such that the subclass masquerades as the class it replaces. When testing for class equality, you should therefore compare the values returned by the `class` method rather than those returned by lower-level functions. Put in terms of API, the following inequalities pertain for dynamic subclasses:

```
[object class] != object_getClass(object) != *((Class*)object)
```

You should therefore test two classes for equality as follows:

```
if ([objectA class] == [objectB class]) { //...
```