Chapter

10

Object

Smalltalk is a *rooted* system, which means that there is a root superclass from which all other classes are descended. That root superclass is *Object*. (Actually, there other root classes, and you can see them by inspecting `Class rootsOfTheWorld`. However, for most application development you can consider Object to be the single root class.)

Object provides a lot of behavior that is inherited by all its subclasses, although sometimes they will override that behavior. It's worth looking through all the protocols and methods of Object to become familiar with the general behavior that is inherited by all classes. Let's take a look at some of the instance side protocols of Object. We won't look at all protocols, or even all methods in the protocols we do look at, but we'll cover enough to give you a feel for what your classes will be inheriting from Object.

Accessing

The message `at:` will give the object at the specified index, while `at:put:` puts an object at that index. The `size` message returns the number of indexable fields in the object. These messages only work with indexable objects; ie, collection type objects. Similar methods, `basicAt:`, `basicAt:put:`, and `basicSize` do the same things, but should never be overridden by subclasses. The intent is that you can always rely on the `basic` methods to provide the functionality without worrying that some intermediate superclass might have overridden them. Finally, `yourself` returns the receiver of the message. It is often used in cascades to make sure that the correct object is returned. For example, in `list := List new add: 1; yourself`, the `yourself` makes sure that the variable contains the list rather than the number one.

Changing

The message `changed`, `changed:`, and `changed:with:` are all used to set off dependency notification. The typical use is that an object sends one of these messages to itself when it has changed, and all dependents of the object are informed. We'll see more about the dependency mechanism in Chapter 19, The Dependency Mechanism. The `changeRequest` message is sent when an application wants to change and is asking its dependents if they agree to the change. Your application can override `changeRequest` to put in any application specific checking.

Class membership

The message `isMemberOf:` tells if the receiver is an instance of the specified class, `isKindOf:` tells if the receiver is an instance of the specified class or one of its subclasses, and `respondsTo:` tells if the receiver understands the specified message. The use of all three methods often indicates that you are thinking in procedural terms, that you are not using polymorphism well. These three methods should generally be avoided. The message `class` gives you the class of which the receiver is an instance.

Comparing

The message == tells you if the receiver and parameter are the *same* object, if they are identically equal. The message = tells you if the receiver and parameter are equal. For example, two strings with the same characters are not the same object, but they are equal, so == returns *false* and = returns *true*. Since different classes have different criteria for judging their instances to be equal (for example, two Employee objects may be equal when the social security numbers are the same), the default implementation of = is simply to invoke ==, leaving subclasses to override = with more appropriate behavior. (You should never override ==.) Corresponding to = and == are ~= and `~~`, which are not equal and identically not equal. The message `hash` returns a hash value for an object, and should be overridden whenever you override = because two objects that are equal should have the same hash value. For more information on all this, see Chapter 6, Special Variables, Characters, and Symbols.

Converting

The message `->` returns an Association with the receiver as the key and the parameter as the value. Associations are useful when you want a pair of objects, and they are used in Dictionaries, which hold all the key-value pairs in Associations. The message `asValue` puts the receiver in a ValueHolder and returns this. ValueHolders are widely used by the user interface dependency mechanism, which wants a level of indirection when referring to an object.

Copying

The `copy` message copies the receiver. It is implemented by sending `shallowCopy` then `postCopy`. The `shallowCopy` method makes a *shallow* copy of the receiver — ie, it doesn't make copies of any of the instance variables, leaving both the original object and its copy sharing instance variables. The message `postCopy` is a hook that gives you an opportunity to copy instance variables by overriding `postCopy`. For more information, see Chapter 22, Common Errors and Chapter 25, Hooks into the System.

Dependents access

The messages addDependent: and removeDependent: provide the basic mechanism for setting up and closing down a dependency relationship on another object. The messages expressInterestIn:for:sendBack: and retractInterestIn:for: provide a more sophisticated mechanism. There is more information on these messages in Chapter 19, The Dependency Mechanism.

Error handling

The message subclassResponsibility is sent in methods that should be overridden by subclasses. The typical use is for the superclass method to do self subclassResponsibility. If a subclass fails to override the method, an exception is raised. The opposite message, shouldNotImplement is sent in a subclass method to show that although the subclass has inherited the method, it is not appropriate behavior for this subclass. The messages halt, halt:, error:, and notify: are all used to generate exceptions and raise a Notifier/Debugger window (for example, self halt. The messages errorSignal and messageNotUnderstoodSignal return signals that can be used in signal handlers to trap exceptions. The message doesNotUnderstand: is sent when a message is not understood by the receiver, and its default behavior is to raise a Notifier/Debugger window. Override this method if you want to trap messages that were not understood so you can pass them to another object. We use this mechanism in Chapter 21, Debugging.

File In/Out

The messages binaryRepresentationVersion and representBinaryOn: provide specialized behavior for use with the BOSS facility when storing and retrieving binary objects. In particular, by overriding representBinaryOn:, you can specify how you want an object to be stored. Returning *nil*, which is the default implementation, says to store the default representation of the receiver. Returning 0 says to not store the object at all. The other common return is a *MessageSend*, which provides a way to store the object in one form and reconstitute it in a different form when it is read back.

Initialize-release

The release message breaks any dependencies that may have been established on this object. It is sent when the object is no longer needed and allows the object to be garbage-collected.

Message handling

The perform: family of messages allow you to create a method name at runtime then invoke the method. It is widely used when you don't know the name of the method at compile time and have to wait until runtime. For example, the user interface uses it heavily since the names of the accessor methods for variables and menus are specified by the user when building the user interface, but the underlying user interface code is part of the system classes. We'll talk more about the perform: messages in Chapter 28, Eliminating Procedural Code and we'll use it in Chapter 30, Testing.

Printing

The most commonly used methods in the printing protocol are `printString`, `printOn:`, and `displayString`. You send `printString` to an object to get a printable representation, one that can be shown on the Transcript. The `printString` message invokes `printOn:`, so to override what `printString` returns, you override `printOn:`. The `displayString` message returns a more suitable representation of the receiver for use in labels and list boxes (SequenceViews). It defaults to `printString`, but is overridden by classes such as String.

System primitives

The messages `allOwners` and `allOwnersWeakly:` return collections of objects that reference the receiver. The messages `firstOwner` and `ownerAfter:` return individual objects from those collections. The `become:` message swaps the references of two objects. One use of `become:` is when collections grow in size. A new collection is created, the elements are copied across, and the new collection becomes the old collection. The messages `instVarAt:` and `instVarAt:put:` allow you direct access to instance variables without having to name them. However, since use of these two methods violates encapsulation, they should be used sparingly, if at all.

Testing

This protocol contains methods that return *true* or *false*. Most of the methods return *false*, allowing them to be overridden when appropriate. For example, `isInteger` returns *false*, but is overridden in Integer to return *true*. The `isBehavior` message tells if the receiver can be the class of another object. The `isImmediate` message tells if the receiver is an immediate object such as a SmallInteger or a Character that can be represented in thirty-two bits, or whether the receiver is being represented by an object pointer. The messages `isNil` and `notNil` tell whether the object is *nil* and are overridden by UndefinedObject . The message `isSequenceable` tells if the object is a sequenceable collection — ie, a subclass of SequenceableCollection. The message `isString` tells if the object is a String, and `isSymbol` tells if it is a Symbol. The `respondsToArithmetic` message tells if the object belongs to a subclass of ArithmeticValue.

Use of testing messages allows you to avoid messages such as `isKindOf:` and `isMemberOf:`. In your own classes, use `isXxx` types of messages if you need to determine if an object is of a particular type. For example, suppose you have a Response class with subclasses of SuccessResponse and FailureResponse. In Response, implement the methods `isResponse`, `isSuccessResponse` and `isFailureReponse`, with the first returning *true* and the next two returning *false*. SuccessResponse should override `isSuccessReponse` to return *true*, and FailureResponse should override `isFailureResponse` to return *true*. If you have many different types of domain objects for which you want to implement `isXxx` type messages, consider creating an ApplicationObject that is subclassed off Object, where you write a series of `isXxx` methods, each returning *false*. Then in each domain class you override the corresponding testing method, returning *true*.

Updating

The methods in this protocol that you are most likely to be interested in are the `update:` family — `update:`, `update:with:`, and `update:with:from:`. These are all used in the dependency mechanism to inform dependents about changes. An object that changes sends itself a message from the `changed:` family, and eventually these messages send a message from the `update:` family to each dependent. Dependents therefore need to override one of the `update:` messages if they want to take action based on the notification. We talk more about the dependency mechanism in Chapter 19, The Dependency Mechanism.

User interface

The `browse` message opens a class Browser on the class of the receiver. The `inspect` message opens an Inspector on the receiver. This message is overridden by some classes to provide an Inspector more suitable for the class. For example OrderedCollection overrides `inspect` to hide the internal implementation details of the collection. To see the internal details you can send the message `basicInspect`, which shows all the instance variables of an object. Figure 10-1 shows an instance of OrderedCollection inspected by sending first `inspect` and then `basicInspect`.

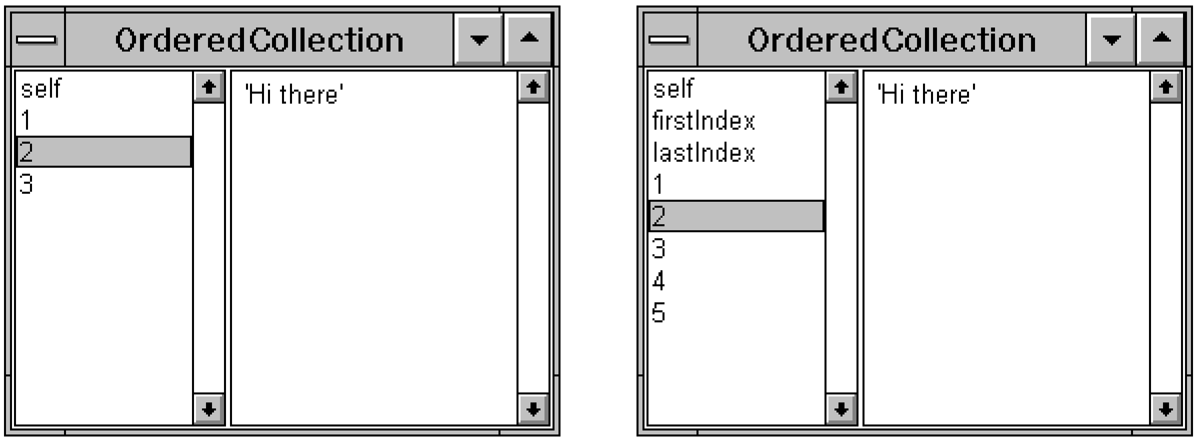

Figure 10-1. Inspecting using `inspect` and `basicInspect`.