Key-Value Coding Programming Guide

Developer

Contents

Figures, Tables, and Listings

Introduction

This document describes the `NSKeyValueCoding` informal protocol, which defines a mechanism allowing applications to access the properties of an object indirectly by name (or key), rather than directly through invocation of an accessor method or as instance variables.

You should read this document to gain an understanding of how to use key-value coding in your applications and how to make your classes key-value coding compliant for interacting with other technologies. Key-value coding is a fundamental technology when working with key-value observing, Cocoa bindings, Core Data, and making your application AppleScript-able. You are expected to be familiar with the basics of Cocoa development and the Objective-C language.

Organization of This Document

Key-Value Coding includes these articles:

- What Is Key-Value Coding? (page 8) provides an overview of key-value coding.
- Terminology (page 11) defines the terms commonly used to refer to an object's properties.
- Key-Value Coding Fundamentals (page 12) describes the basic principles required to use key-value coding.
- Key-Value Coding Accessor Methods (page 15) describes the accessor methods your classes should implement.
- Key-Value Validation (page 25) tells you how to implement property validation.
- Ensuring KVC Compliance (page 29) describes the capabilities a class must implement to be key-value coding compliant.
- Scalar and Structure Support (page 31) describes the data types supported by key-value coding.
- Collection Operators (page 35) lists the available collection operators and describes their use.
- Accessor Search Implementation Details (page 43) explains how the appropriate accessor method or instance variable is determined.
- Describing Property Relationships (page 48) describes the use of meta-data to define the relationships between objects and their properties.
- Performance Considerations (page 49) describes the performance considerations when using key-value coding.

See Also

There are other technologies, not covered in this document, that are related to key-value coding.

- *Key-Value Observing Programming Guide* describes the features of the key-value observing protocol that allows objects to observe changes in other objects.

What Is Key-Value Coding?

Key-value coding is a mechanism for accessing an object's properties indirectly, using strings to identify properties, rather than through invocation of an accessor method or accessing them directly through instance variables. In essence, key-value coding defines the patterns and method signatures that your application's accessor methods implement.

Accessor methods, as the name suggests, provide access to your application's data model's property values. There are two basic forms of accessor—get accessors and set accessors. Get accessors, also referred to as getters, return the values of a property. Set accessors, also referred to as setters, set the value of a property. There are getter and setter variants for dealing with both object attributes and to-many relationships.

Implementing key-value coding compliant accessors in your application is an important design principle. Accessors help to enforce proper data encapsulation and facilitate integration with other technologies such as key-value observing, Core Data, Cocoa bindings, and scriptability. Key-value coding methods can, in many cases, also be utilized to simplify your application's code.

The essential methods for key-value coding are declared in the `NSKeyValueCoding` Objective-C informal protocol and default implementations are provided by `NSObject`.

Key-value coding supports properties with object values, as well as the scalar types and structs. Non-object parameters and return types are detected and automatically wrapped, and unwrapped, as described in Scalar and Structure Support (page 31).

Key-Value Coding and Scripting

The scripting support in Cocoa is designed so that an application can easily implement scripting by accessing its model objects—the objects that encapsulate the application's data. When the user executes an AppleScript command that targets your application, the goal is for that command to go directly to the application's model objects to get the work done.

Scripting in OS X relies heavily on key-value coding to provide automatic support for executing AppleScript commands. In a scriptable application, a model object defines a set of keys that it supports. Each key represents a property of the model object. Some examples of scripting-related keys are `words`, `font`, `documents`, and `color`. The key-value coding API provides a generic and automatic way to query an object for the values of its keys and to set new values for those keys.

As you design the objects of your application, you should define the set of keys for your model objects and implement the corresponding accessor methods. Then when you define the script suites for your application, you can specify the keys that each scriptable class supports. If you support key-value coding, you will get a great deal of scripting support "for free."

In AppleScript, object hierarchies define the structure of the model objects in an application. Most AppleScript commands specify one or more objects within your application by drilling down this object hierarchy from parent container to child element. You can define the relationships between properties available through key-value coding in class descriptions. See Describing Property Relationships (page 48) for more details.

Cocoa's scripting support takes advantage of key-value coding to get and set information in scriptable objects. The method in the Objective-C informal protocol `NSScriptKeyValueCoding` provides additional capabilities for working with key-value coding, including getting and setting key values by index in multi-value keys and coercing (or converting) a key-value to the appropriate data type.

Using Key-Value Coding to Simplify Your Code

You can use key-value coding methods in your own code to generalize implementations. For example, in OS X `NSTableView` and `NSOutlineView` objects both associate an identifier string with each of their columns. By making this identifier the same as the key for the property you wish to display, you can significantly simplify your code.

Listing 1 shows an implementation of an `NSTableView` delegate method without using key-value coding. Listing 2 shows an implementation that takes advantage of key-value coding to return the appropriate value using the column identifier as the key.

Listing 1 Implementation of data-source method without key-value coding

```
- (id)tableView:(NSTableView *)tableview
      objectValueForTableColumn:(id)column row:(NSInteger)row {

    ChildObject *child = [childrenArray objectAtIndex:row];
    if ([[column identifier] isEqualToString:@"name"]) {
        return [child name];
    }
    if ([[column identifier] isEqualToString:@"age"]) {
        return [child age];
    }
    if ([[column identifier] isEqualToString:@"favoriteColor"]) {
```

```
        return [child favoriteColor];
    }
    // And so on.
}
```

Listing 2 Implementation of data-source method with key-value coding

```
- (id)tableView:(NSTableView *)tableview
      objectValueForTableColumn:(id)column row:(NSInteger)row {

    ChildObject *child = [childrenArray objectAtIndex:row];
    return [child valueForKey:[column identifier]];
}
```

Terminology

In addition to overloading existing terms, key-value coding defines some unique terminology of its own.

Key-value coding can be used to access three different types of object values: attributes, to-one relationships, and to-many relationships. The term *property* refers to any of these types of values.

An *attribute* is a property that is a simple value, such as a scalar, string, or Boolean value. Value objects such as `NSNumber` and other immutable types such as as `NSColor` are also considered attributes.

A property that specifies a *to-one relationship* is an object that has properties of its own. These underlying properties can change without the object itself changing. For example, an `NSView` instance's superview is a to-one relationship.

Finally, a property that specifies a *to-many relationship* consists of a collection of related objects. An instance of `NSArray` or `NSSet` is commonly used to hold such a collection. However, key-value coding allows you to use custom classes for collections and still access them as if they were an `NSArray` or `NSSet` by implementing the key-value coding accessors discussed in Collection Accessor Patterns for To-Many Properties (page 17).

Key-Value Coding Fundamentals

This article describes the basic principles of key-value coding.

Keys and Key Paths

A key is a string that identifies a specific property of an object. Typically, a key corresponds to the name of an accessor method or instance variable in the receiving object. Keys must use ASCII encoding, begin with a lowercase letter, and may not contain whitespace.

Some example keys would be `payee`, `openingBalance`, `transactions` and `amount`.

A key path is a string of dot separated keys that is used to specify a sequence of object properties to traverse. The property of the first key in the sequence is relative to the receiver, and each subsequent key is evaluated relative to the value of the previous property.

For example, the key path `address.street` would get the value of the `address` property from the receiving object, and then determine the `street` property relative to the `address` object.

Getting Attribute Values Using Key-Value Coding

The method `valueForKey:` returns the value for the specified key, relative to the receiver. If there is no accessor or instance variable for the specified key, then the receiver sends itself a `valueForUndefinedKey:` message. The default implementation of `valueForUndefinedKey:` raises an `NSUndefinedKeyException`; subclasses can override this behavior.

Similarly, `valueForKeyPath:` returns the value for the specified key path, relative to the receiver. Any object in the key path sequence that is not key-value coding compliant for the appropriate key receives a `valueForUndefinedKey:` message.

The method `dictionaryWithValuesForKeys:` retrieves the values for an array of keys relative to the receiver. The returned NSDictionary contains values for all the keys in the array.

> **Note:** Collection objects, such as `NSArray`, `NSSet`, and `NSDictionary`, can't contain `nil` as a value. Instead, you represent `nil` values using a special object, `NSNull`. `NSNull` provides a single instance that represents the `nil` value for object properties. The default implementations of `dictionaryWithValuesForKeys:` and `setValuesForKeysWithDictionary:` translates between `NSNull` and `nil` automatically, so your objects don't have to explicitly test for `NSNull` values.

When a value is returned for a key path that contains a key for a to-many property, and that key is not the last key in the path, the returned value is a collection containing all the values for the keys to the right of the to-many key. For example, requesting the value of the key path `transactions.payee` returns an array containing all the `payee` objects, for all the transactions. This also works for multiple arrays in the key path. The key path `accounts.transactions.payee` would return an array with all the payee objects, for all the transactions, in all the accounts.

Setting Attribute Values Using Key-Value Coding

The method `setValue:forKey:` sets the value of the specified key, relative to the receiver, to the provided value. The default implementation of `setValue:forKey:` automatically unwraps `NSValue` objects that represent scalars and structs and assigns them to the property. See Scalar and Structure Support (page 31) for details on the wrapping and unwrapping semantics.

If the specified key does not exist, the receiver is sent a `setValue:forUndefinedKey:` message. The default implementation of `setValue:forUndefinedKey:` raises an `NSUndefinedKeyException`; however, subclasses can override this method to handle the request in a custom manner.

The method `setValue:forKeyPath:` behaves in a similar fashion, but it is able to handle a key path as well as a single key.

Finally, `setValuesForKeysWithDictionary:` sets the properties of the receiver with the values in the specified dictionary, using the dictionary keys to identify the properties. The default implementation invokes `setValue:forKey:` for each key-value pair, substituting `nil` for `NSNull` objects as required.

One additional issue that you should consider is what happens when an attempt is made to set a non-object property to a nil value. In this case, the receiver sends itself a `setNilValueForKey:` message. The default implementation of `setNilValueForKey:` raises an `NSInvalidArgumentException`. Your application can override this method to substitute a default value or a marker value, and then invoke `setValue:forKey:` with the new value.

Dot Syntax and Key-Value Coding

Objective-C's dot syntax and key-value coding are orthogonal technologies. You can use key-value coding whether or not you use the dot syntax, and you can use the dot syntax whether or not you use KVC. Both, though, make use of a "dot syntax." In the case of key-value coding, the syntax is used to delimit elements in a key path. Remember that when you access a property using the dot syntax, you invoke the receiver's standard accessor methods.

You can use key-value coding methods to access a property, for example, given a class defined as follows:

```
@interface MyClass
@property NSString *stringProperty;
@property NSInteger integerProperty;
@property MyClass *linkedInstance;
@end
```

you could access the properties in an instance using KVC:

```
MyClass *myInstance = [[MyClass alloc] init];
NSString *string = [myInstance valueForKey:@"stringProperty"];
[myInstance setValue:@2 forKey:@"integerProperty"];
```

To illustrate the difference between the properties dot syntax and KVC key paths, consider the following.

```
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
myInstance.linkedInstance.integerProperty = 2;
```

This has the same result as:

```
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
[myInstance setValue:@2 forKeyPath:@"linkedInstance.integerProperty"];
```

Key-Value Coding Accessor Methods

In order for key-value coding to locate the accessor methods to use for invocations of `valueForKey:`, `setValue:forKey:`, `mutableArrayValueForKey:`, and `mutableSetValueForKey:`, you need to implement the key-value coding accessor methods.

> **Note:** The accessor patterns in this section are written in the form `–set<Key>:` or `–<key>`. The <key> text is a placeholder for the name of your property. Your implementation of the corresponding method should substitute the property name for <Key> or <key> respecting the case specified by key. For example, for the property `name`, `–set<Key>:` would expand to `–setName:`, `–<key>` would simply be `–name`.

Commonly Used Accessor Patterns

The format for an accessor method that returns a property is `–<key>`. The `–<key>` method returns an object, scalar, or a data structure. The alternate naming form `–is<Key>` is supported for Boolean properties.

The example in Listing 1 shows the method declaration for the `hidden` property using the typical convention, and Listing 2 shows the alternate format.

Listing 1 Accessor naming variations for a hidden property key

```
- (BOOL)hidden {
   // Implementation specific code.
   return ...;
}
```

Listing 2 Alternate form accessor for a hidden property key

```
- (BOOL)isHidden {
   // Implementation specific code.
   return ...;
}
```

In order for an attribute or to-one relationship property to support `setValue:forKey:`, an accessor in the form `set<Key>:` must be implemented. Listing 3 shows an example accessor method for the `hidden` property key.

Listing 3 Accessor naming convention to support a hidden property key

```
- (void)setHidden:(BOOL)flag {
   // Implementation specific code.
   return;
}
```

If the attribute is a non-object type, you must also implement a suitable means of representing a `nil` value. The key-value coding method `setNilValueForKey:` method is called when you attempt to set an attribute to `nil`. This provides the opportunity to provide appropriate default values for your application, or handle keys that don't have corresponding accessors in the class.

The following example sets the `hidden` attribute to `YES` when an attempt is made to set it to `nil`. It creates an `NSNumber` instance containing the Boolean value and then uses `setValue:forKey:` to set the new value. This maintains encapsulation of the model and ensures that any additional actions that should occur as a result of setting the value will actually occur. This is considered better practice than calling an accessor method or setting an instance variable directly.

```
- (void)setNilValueForKey:(NSString *)theKey {

    if ([theKey isEqualToString:@"hidden"]) {
        [self setValue:@YES forKey:@"hidden"];
    }
    else {
        [super setNilValueForKey:theKey];
    }
}
```

Collection Accessor Patterns for To-Many Properties

Although your application can implement accessor methods for to-many relationship properties using the `-<key>` and `-set<Key>:` accessor forms, you should typically only use those to create the collection object. For manipulating the contents of the collection it is best practice to implement the additional accessor methods referred to as the collection accessor methods. You then use the collection accessor methods, or a mutable collection proxy returned by `mutableArrayValueForKey:` or `mutableSetValueForKey:`.

Implementing the collection accessor methods, instead of, or in addition to, the basic getter for the relationship, can have many advantages:

- Performance can be increased in the case of mutable to-many relationships, often significantly.
- To-many relationships can be modeled with classes other than `NSArray` or `NSSet` by implementing the appropriate collection accessors. Implementing collection accessor methods for a property makes that property indistinguishable from an array or set when using key-value coding methods.
- You can use the collection accessor methods directly to make modifications to the collection in a key-value observing compliant way. See *Key-Value Observing Programming Guide* for more information on key-value observing.

There are two variations of collection accessors: indexed accessors for ordered to-many relationships (typically represented by `NSArray`) and unordered accessors for relationships that don't require an order to the members (represented by `NSSet`).

Indexed Accessor Pattern

The indexed accessor methods define a mechanism for counting, retrieving, adding, and replacing objects in an ordered relationship. Typically this relationship is an instance of `NSArray` or `NSMutableArray`; however, any object can implement these methods and be manipulated just as if it was an array. You are not restricted to simply implementing these methods, you can also invoke them as well to interact directly with objects in the relationship.

There are indexed accessors which return data from the collection (the getter variation) and mutable accessors that provide an interface for `mutableArrayValueForKey:` to modify the collection.

Getter Indexed Accessors

In order to support read-only access to an ordered to-many relationship, implement the following methods:

- `-countOf<Key>`. Required. This is the analogous to the `NSArray` primitive method `count`.
- `-objectIn<Key>AtIndex:` or `-<key>AtIndexes:`. One of these methods must be implemented. They correspond to the `NSArray` methods `objectAtIndex:` and `objectsAtIndexes:`.

- `–get<Key>:range:`. Implementing this method is optional, but offers additional performance gains. This method corresponds to the `NSArray` method `getObjects:range:`.

An implementation of the `–countOf<Key>` method simply returns the number of objects in the to-many relationship as an `NSUInteger`. The code fragment in Listing 4 illustrates the `–countOf<Key>` implementation for the to-many relationship property `employees`.

Listing 4 Example `–count<Key>` implementation

```
– (NSUInteger)countOfEmployees {
    return [self.employees count];
}
```

The `–objectIn<Key>AtIndex:` method returns the object at the specified index in the to-many relationship. The `–<key>AtIndexes:` accessor form returns an array of objects at the indexes specified by the `NSIndexSet` parameter. Only one of these two methods must be implemented.

The code fragment in Listing 5 shows `–objectIn<Key>AtIndex:` and `–<key>AtIndexes:` implementations for a to-many relationship property `employees`.

Listing 5 Example `–objectIn<Key>AtIndex:` and `–<key>AtIndexes:` implementations

```
– (id)objectInEmployeesAtIndex:(NSUInteger)index {
    return [employees objectAtIndex:index];
}

– (NSArray *)employeesAtIndexes:(NSIndexSet *)indexes {
    return [self.employees objectsAtIndexes:indexes];
}
```

If benchmarking indicates that performance improvements are required, you can also implement `–get<Key>:range:`. Your implementation of this accessor should return in the buffer given as the first parameter the objects that fall within the range specified by the second parameter.

Listing 6 shows an example implementation of the `–get<Key>:range:` accessor pattern for the to-many `employees` property.

Listing 6 Example `–get<Key>:range:` implementation

```
- (void)getEmployees:(Employee * __unsafe_unretained *)buffer range:(NSRange)inRange
 {
    // Return the objects in the specified range in the provided buffer.
    // For example, if the employees were stored in an underlying NSArray
    [self.employees getObjects:buffer range:inRange];
}
```

Mutable Indexed Accessors

Supporting a mutable to-many relationship with indexed accessors requires implementing additional methods. Implementing the mutable indexed accessors allow your application to interact with the indexed collection in an easy and efficient manner by using the array proxy returned by `mutableArrayValueForKey:`. In addition, by implementing these methods for a to-many relationship your class will be key-value observing compliant for that property (see *Key-Value Observing Programming Guide*).

> **Note:** You are strongly advised to implement these mutable accessors rather than relying on an accessor that returns a mutable set directly. The mutable accessors are much more efficient when making changes to the data in the relationship.

In order to be key-value coding compliant for a mutable ordered to-many relationship you must implement the following methods:

- `–insertObject:in<Key>AtIndex:` or `–insert<Key>:atIndexes:`. At least one of these methods must be implemented. These are analogous to the `NSMutableArray` methods `insertObject:atIndex:` and `insertObjects:atIndexes:`.
- `–removeObjectFrom<Key>AtIndex:` or `–remove<Key>AtIndexes:`. At least one of these methods must be implemented. These methods correspond to the `NSMutableArray` methods `removeObjectAtIndex:` and `removeObjectsAtIndexes:` respectively.
- `–replaceObjectIn<Key>AtIndex:withObject:` or `–replace<Key>AtIndexes:with<Key>:`. Optional. Implement if benchmarking indicates that performance is an issue.

The `–insertObject:in<Key>AtIndex:` method is passed the object to insert, and an `NSUInteger` that specifies the index where it should be inserted. The `–insert<Key>:atIndexes:` method inserts an array of objects into the collection at the indices specified by the passed `NSIndexSet`. You are only required to implement one of these two methods.

Listing 7 shows sample implementations of both insert accessors for the to-many `employee` property.

Listing 7 Example `-insertObject:in<Key>AtIndex:` and `-insert<Key>:atIndexes:` accessors

```
- (void)insertObject:(Employee *)employee inEmployeesAtIndex:(NSUInteger)index {
    [self.employees insertObject:employee atIndex:index];
    return;
}

- (void)insertEmployees:(NSArray *)employeeArray atIndexes:(NSIndexSet *)indexes
{
    [self.employees insertObjects:employeeArray atIndexes:indexes];
    return;
}
```

The `-removeObjectFrom<Key>AtIndex:` method is passed an `NSUInteger` value specifying the index of the object to be removed from the relationship. The `-remove<Key>AtIndexes:` is passed an `NSIndexSet` specifying the indexes of the objects to be removed from the relationship. Again, you are only required to implement one of these methods.

Listing 8 shows sample implementations of `-removeObjectFrom<Key>AtIndex:` and `-remove<Key>AtIndexes:` implementations for the to-many `employee` property.

Listing 8 Example `-removeObjectFrom<Key>AtIndex:` and `-remove<Key>AtIndexes:` accessors

```
- (void)removeObjectFromEmployeesAtIndex:(NSUInteger)index {
    [self.employees removeObjectAtIndex:index];
}

- (void)removeEmployeesAtIndexes:(NSIndexSet *)indexes {
    [self.employees removeObjectsAtIndexes:indexes];
}
```

If benchmarking indicates that performance improvements are required, you can also implement one or both of the optional replace accessors. Your implementation of either `-replaceObjectIn<Key>AtIndex:withObject:` or `-replace<Key>AtIndexes:with<Key>:` are called when an object is replaced in a collection, rather than doing a remove and then insert.

Listing 9 shows sample implementations of `-replaceObjectIn<Key>AtIndex:withObject:` and `-replace<Key>AtIndexes:with<Key>:` implementations for the to-many `employee` property.

Listing 9 Example `-replaceObjectIn<Key>AtIndex:withObject:` and `-replace<Key>AtIndexes:with<Key>:` accessors

```
- (void)replaceObjectInEmployeesAtIndex:(NSUInteger)index
                             withObject:(id)anObject {

    [self.employees replaceObjectAtIndex:index withObject:anObject];
}

- (void)replaceEmployeesAtIndexes:(NSIndexSet *)indexes
                     withEmployees:(NSArray *)employeeArray {

    [self.employees replaceObjectsAtIndexes:indexes withObjects:employeeArray];
}
```

Unordered Accessor Pattern

The unordered accessor methods provide a mechanism for accessing and mutating objects in an unordered relationship. Typically, this relationship is an instance of `NSSet` or `NSMutableSet`. However, by implementing these accessors, any class can by used to model the relationship and be manipulated using key-value coding just as if it was an instance of `NSSet`.

Getter Unordered Accessors

The getter variations of the unordered accessor methods provide simple access to the relationship data. The methods return the number of objects in the collection, an enumerator to iterate over the collection objects, and a method to compare an object with the contents of the collection to see if it is already present.

> **Note:** It's rare to have to implement the getter variations of the unordered accessors. To-many unordered relationships are most often modeled using instance of NSSet or a subclass. In that case the key-value coding will, if it doesn't find these accessor patterns for the property, directly access the set. Typically, you only implement these methods if you are using a custom collection class that needs to be accessed as if it was a set.

In order to support read-only access to an unordered to-many relationship, you would implement the following methods:

- `-countOf<Key>`. Required. This method corresponds to the `NSSet` method `count`.
- `-enumeratorOf<Key>`. Required. Corresponds to the `NSSet` method `objectEnumerator`.

- `-memberOf<Key>:`. Required. This method is the equivalent of the `NSSet` method `member:`

Listing 10 shows simple implementations of the necessary getter accessors that simply pass the responsibilities to the `transactions` property.

Listing 10 Example `-countOf<Key>`, `-enumeratorOf<Key>`, and `-memberOf<Key>:` accessors

```
- (NSUInteger)countOfTransactions {
    return [self.transactions count];
}

- (NSEnumerator *)enumeratorOfTransactions {
    return [self.transactions objectEnumerator];
}

- (Transaction *)memberOfTransactions:(Transaction *)anObject {
    return [self.transactions member:anObject];
}
```

The `-countOf<Key>` accessor implementation should simply return the number of items in the relationship. The `-enumeratorOf<Key>` method implementation must return an `NSEnumerator` instance that is used to iterate over the items in the relationship. See Enumeration: Traversing a Collection's Elements in *Collections Programming Topics* for more information about enumerators.

The `-memberOf<Key>:` accessor must compare the object passed as a parameter with the contents of the collection and returns the matching object as a result, or `nil` if no matching object is found. Your implementation of this method may use `isEqual:` to compare the objects, or may compare objects in another manner. The object returned may be a different object than that tested for membership, but it should be the equivalent as far as content is concerned.

Mutable Unordered Accessors

Supporting a mutable to-many relationship with unordered accessors requires implementing additional methods. Implementing the mutable unordered accessors for your application to interact with the collection in an easy and efficient manner through the use of the array proxy returned by `mutableSetValueForKey:`. In addition, by implementing these methods for a to-many relationship your class will be key-value observing compliant for that property (see *Key-Value Observing Programming Guide*).

> **Note:** You are strongly advised to implement these mutable accessors rather than relying on an accessor that returns a mutable array directly. The mutable accessors are much more efficient when making changes to the data in the relationship.

In order to be key-value coding complaint for a mutable unordered to-many relationship you must implement the following methods:

- `–add<Key>Object:` or `–add<Key>:`. At least one of these methods must be implemented. These are analogous to the `NSMutableSet` method `addObject:`.
- `–remove<Key>Object:` or `–remove<Key>:`. At least one of these methods must be implemented. These are analogous to the `NSMutableSet` method `removeObject:`.
- `–intersect<Key>:`. Optional. Implement if benchmarking indicates that performance is an issue. It performs the equivalent action of the NSSet method `intersectSet:`.

The `–add<Key>Object:` and `–add<Key>:` implementations add a single item or a set of items to the relationship. You are only required to implement one of the methods. When adding a set of items to the relationship you should ensure that an equivalent object is not already present in the relationship. Listing 11 shows an example pass-through implementation for the `transactions` property.

Listing 11 Example `–add<Key>Object:` and `–add<Key>:` accessors

```
- (void)addTransactionsObject:(Transaction *)anObject {
    [self.transactions addObject:anObject];
}

- (void)addTransactions:(NSSet *)manyObjects {
    [self.transactions unionSet:manyObjects];
}
```

Similarly, the `–remove<Key>Object:` and `–remove<Key>:` implementations remove a single item or a set of items from the relationship. Again, implementation of only one of the methods is required. Listing 12 shows an example pass-through implementation for the `transactions` property.

Listing 12 Example `–remove<Key>Object:` and `–remove<Key>:` accessors

```
- (void)removeTransactionsObject:(Transaction *)anObject {
    [self.transactions removeObject:anObject];
}
```

```
- (void)removeTransactions:(NSSet *)manyObjects {
    [self.transactions minusSet:manyObjects];
}
```

If benchmarking indicates that performance improvements are required, you can also implement the `-intersect<Key>:` or `-set<Key>:` methods (see Listing 13).

The implementation of `-intersect<Key>:` should remove from the relationship all the objects that aren't common to both sets. This is the equivalent of the `NSMutableSet` method `intersectSet:`.

Listing 13 Example `-intersect<Key>:` and `-set<Key>:` implementations

```
- (void)intersectTransactions:(NSSet *)otherObjects {
    return [self.transactions intersectSet:otherObjects];
}
```

Key-Value Validation

Key-value coding provides a consistent API for validating a property value. The validation infrastructure provides a class the opportunity to accept a value, provide an alternate value, or deny the new value for a property and give a reason for the error.

Validation Method Naming Convention

Just as there are conventions for naming accessor methods, there is a convention for naming a property's validation method. A validation method has the format `validate<Key>:error:`. The example in Listing 1 shows the validation method declaration for the property `name`.

Listing 1 Validation method declaration for the property name

```
-(BOOL)validateName:(id *)ioValue error:(NSError * __autoreleasing *)outError {
    // Implementation specific code.
    return ...;
}
```

Implementing a Validation Method

Validation methods are passed two parameters by reference: the value object to validate and the NSError used to return error information.

There are three possible outcomes from a validation method:

1. The object value is valid, so `YES` is returned without altering the value object or the error.
2. The object value is not valid and a valid value cannot be created and returned. In this case `NO` is returned after setting the error parameter to an `NSError` object that indicates the reason validation failed.
3. A new object value that is valid is created and returned. In this case `YES` is returned after setting the value parameter to the newly created, valid value. The error is returned unaltered. You must return a new object, rather than just modifying the passed `ioValue`, even if it is mutable.

   This variant is typically discouraged because it can be difficult to ensure values are propagated consistently.

The example in Listing 2 implements a validation method for a `name` property that ensures that the value object is a string and that the name is capitalized correctly.

Listing 2 Validation method for the name property

```
-(BOOL)validateName:(id *)ioValue error:(NSError * __autoreleasing *)outError{

    // The name must not be nil, and must be at least two characters long.
    if ((*ioValue == nil) || ([(NSString *)*ioValue length] < 2)) {
        if (outError != NULL) {
            NSString *errorString = NSLocalizedString(
                    @"A Person's name must be at least two characters long",
                    @"validation: Person, too short name error");
           NSDictionary *userInfoDict = @{ NSLocalizedDescriptionKey : errorString
 };
            *outError = [[NSError alloc] initWithDomain:PERSON_ERROR_DOMAIN
                                                   code:PERSON_INVALID_NAME_CODE
                                               userInfo:userInfoDict];
        }
        return NO;
    }
    return YES;
}
```

Important: A validation method that returns `NO` must first check whether the `outError` parameter is `NULL`; if the `outError` parameter is not `NULL`, the method should set the `outError` parameter to a valid `NSError` object.

Invoking Validation Methods

You can call validation methods directly, or by invoking `validateValue:forKey:error:` and specifying the key. The default implementation of validateValue:forKey:error: searches the class of the receiver for a validation method whose name matches the pattern `validate<Key>:error:`. If such a method is found, it is invoked and the result is returned. If no such method is found, `validateValue:forKey:error:` returns `YES`, validating the value.

Warning: An implementation of `-set<Key>:` for a property should never call the validation methods.

Automatic Validation

In general, key-value coding does not perform validation automatically—it is your application's responsibility to invoke the validation methods.

Some technologies do perform validation automatically in some circumstances: Core Data automatically performs validation when the managed object context is saved; in OS X, Cocoa bindings allow you to specify that validation should occur automatically (see *Cocoa Bindings Programming Topics* for more information).

Validation of Scalar Values

Validation methods expect the value parameter to be an object, and as a result values for non-object properties are wrapped in an `NSValue` or `NSNumber` object as discussed in Scalar and Structure Support (page 31). The example in Listing 3 demonstrates a validation method for the scalar property `age`.

Listing 3 Validation method for a scalar property

```
-(BOOL)validateAge:(id *)ioValue error:(NSError * __autoreleasing *)outError {

    if (*ioValue == nil) {
        // Trap this in setNilValueForKey.
        // An alternative might be to create new NSNumber with value 0 here.
        return YES;
    }
    if ([*ioValue floatValue] <= 0.0) {
        if (outError != NULL) {
            NSString *errorString = NSLocalizedStringFromTable(
                @"Age must be greater than zero", @"Person",
                @"validation: zero age error");
          NSDictionary *userInfoDict = @{ NSLocalizedDescriptionKey : errorString
};
            NSError *error = [[NSError alloc] initWithDomain:PERSON_ERROR_DOMAIN
                code:PERSON_INVALID_AGE_CODE
                userInfo:userInfoDict];
```

```
            *outError = error;
        }
        return NO;
    }
    else {
        return YES;
    }
    // ...
```

Ensuring KVC Compliance

In order for a class to be considered KVC compliant for a specific property, it must implement the methods required for `valueForKey:` and `setValue:forKey:` to work for that property.

Attribute and To-One Relationship Compliance

For properties that are an attribute or a to-one relationship, this requires that your class:

- Implement a method named `-<key>`, `-is<Key>`, or have an instance variable `<key>` or `_<key>`.

  Although key names frequently begin with a lowercase letter, KVC also supports key names that begin with an uppercase letter, such as `URL`.
- If the property is mutable, then it should also implement `-set<Key>:`.
- Your implementation of the `-set<Key>:` method should not perform validation.
- Your class should implement `-validate<Key>:error:` if validation is appropriate for the key.

Indexed To-Many Relationship Compliance

For indexed to-many relationships, KVC compliance requires that your class:

- Implement a method named `-<key>` that returns an array.
- Or have an array instance variable named `<key>` or `_<key>`.
- Or implement the method `-countOf<Key>` and one or both of `-objectIn<Key>AtIndex:` or `-<key>AtIndexes:`.
- Optionally, you can also implement `-get<Key>:range:` to improve performance.

For a mutable indexed ordered to-many relationship, KVC compliance requires that your class also:

- Implement one or both of the methods `-insertObject:in<Key>AtIndex:` or `-insert<Key>:atIndexes:`.
- Implement one or both of the methods `-removeObjectFrom<Key>AtIndex:` or `-remove<Key>AtIndexes:`.

- Optionally, you can also implement `-replaceObjectIn<Key>AtIndex:withObject:` or `-replace<Key>AtIndexes:with<Key>:` to improve performance.

Unordered To-Many Relationship Compliance

For unordered to-many relationships, KVC compliance requires that your class:

- Implement a method named `-<key>` that returns a set.
- Or have a set instance variable named `<key>` or `_<key>`.
- Or implement the methods `-countOf<Key>`, `-enumeratorOf<Key>`, and `-memberOf<Key>:`.

For a mutable unordered to-many relationship, KVC compliance requires that your class also:

- Implement one or both of the methods `-add<Key>Object:` or `-add<Key>:`.
- Implement one or both of the methods `-remove<Key>Object:` or `-remove<Key>:`.
- Optionally, you can also implement `-intersect<Key>:` and `-set<Key>:` to improve performance.

Scalar and Structure Support

Key-value coding provides support for scalar values and data structures by automatically wrapping and unwrapping `NSNumber` and `NSValue` instance values.

Representing Non-Object Values

The default implementations of `valueForKey:` and `setValue:forKey:` provide support for automatic object wrapping of the non-object data types, both scalars and structs.

Once `valueForKey:` has determined the specific accessor method or instance variable that is used to supply the value for the specified key, it examines the return type or the data type. If the value to be returned is not an object, an `NSNumber` or `NSValue` object is created for that value and returned in its place.

Similarly, `setValue:forKey:` determines the data type required by the appropriate accessor or instance variable for the specified key. If the data type is not an object, then the value is extracted from the passed object using the appropriate `–<type>Value` method.

Handling nil Values

An additional issue arises when `setValue:forKey:` is invoked with `nil` passed as the value for a non-object property. There is no generalized action that is appropriate. The receiver is sent a `setNilValueForKey:` message when `nil` is passed as the value for a non-object property. The default implementation of `setNilValueForKey:` raises an `NSInvalidArgumentException` exception. A subclass can override this method to provide the appropriate implementation specific behavior.

The example code in Listing 1 responds to an attempt to set a person's age, a float value, to a `nil` value by instead setting the age to 0.

Note: For backward binary compatibility, `unableToSetNilForKey:` is invoked instead of `setNilValueForKey:` if the receiver's class has overridden the NSObject implementation of `unableToSetNilForKey:`.

Listing 1 Example implementation of setNilValueForKey:

```
- (void)setNilValueForKey:(NSString *)theKey
{
    if ([theKey isEqualToString:@"age"]) {
        [self setValue:[NSNumber numberWithFloat:0.0] forKey:@”age”];
    } else
        [super setNilValueForKey:theKey];
}
```

Wrapping and Unwrapping Scalar Types

Table 1 lists the scalar types that are wrapped using `NSNumber` instances.

Table 1 Scalar types as wrapped in `NSNumber` objects

Data type	Creation method	Accessor method
`BOOL`	`numberWithBool:`	`boolValue`
`char`	`numberWithChar:`	`charValue`
`double`	`numberWithDouble:`	`doubleValue`
`float`	`numberWithFloat:`	`floatValue`
`int`	`numberWithInt:`	`intValue`
`long`	`numberWithLong:`	`longValue`
`long long`	`numberWithLongLong:`	`longLongValue`
`short`	`numberWithShort:`	`shortValue`
`unsigned char`	`numberWithUnsignedChar:`	`unsignedChar`
`unsigned int`	`numberWithUnsignedInt:`	`unsignedInt`

Data type	Creation method	Accessor method
`unsigned long`	`numberWithUnsignedLong:`	`unsignedLong`
`unsigned long long`	`numberWithUnsignedLongLong:`	`unsignedLongLong`
`unsigned short`	`numberWithUnsignedShort:`	`unsignedShort`

Wrapping and Unwrapping Structs

Table 2 (page 33) shows the creation and accessor methods use for wrapping the common `NSPoint`, `NSRange`, `NSRect`, and `NSSize` structs.

Table 2 Common struct types as wrapped using `NSValue`.

Data type	Creation method	Accessor method
`NSPoint`	`valueWithPoint:`	`pointValue`
`NSRange`	`valueWithRange:`	`rangeValue`
`NSRect`	`valueWithRect:` (OS X only).	`rectValue`
`NSSize`	`valueWithSize:`	`sizeValue`

Automatic wrapping and unwrapping is not confined to `NSPoint`, `NSRange`, `NSRect`, and `NSSize`—structure types (that is, types whose Objective-C type encoding strings start with `{`) can be wrapped in an `NSValue` object. For example, if you have a structure and class like this:

```
typedef struct {
    float x, y, z;
} ThreeFloats;

@interface MyClass
- (void)setThreeFloats:(ThreeFloats)threeFloats;
- (ThreeFloats)threeFloats;
@end
```

Sending an instance of `MyClass` the message valueForKey: with the parameter `@"threeFloats"` will invoke the `MyClass` method `threeFloats` and return the result wrapped in an `NSValue`. Likewise sending the instance of `MyClass` a `setValue:forKey:` message with an `NSValue` object wrapping a `ThreeFloats` struct will invoke `setThreeFloats:` and pass the result of sending the `NSValue` object a `getValue:` message.

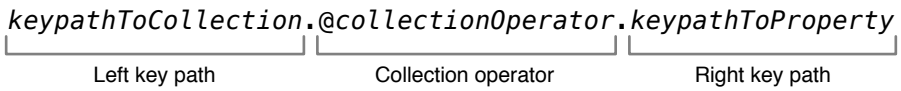
> **Note:** This mechanism doesn't take reference counting or garbage collection into account, so take care when using with object-pointer-containing structure types.

Collection Operators

Collection operators allow actions to be performed on the items of a collection using key path notation and an action operator. This article describes the available collection operators, example key paths, and the results they'd produce.

Collection operators are specialized key paths that are passed as the parameter to the `valueForKeyPath:` method. The operator is specified by a string preceded by an at sign (@). The key path on the left side of the collection operator, if present, determines the array or set, relative to the receiver, that is used in the operation. The key path on the right side of the operator specifies the property of the collection that the operator uses. Figure 1 illustrates the operator key path format.

Figure 1 Operator key path format



All the collection operators, with the exception of `@count`, require a key path to the right of the collection operator.

> **Note:** It is not currently possible to define your own collection operators.

The type of object value returned depends on the operator:

- Simple collection operators return strings, numbers, or dates depending on the value right hand key (see Simple Collection Operators (page 37)).
- Object operators return an `NSArray` instance, as do the object operators (see Object Operators (page 38)).
- Array and set operators return an array or set object, depending on the operator (see Array and Set Operators (page 39)).

Data Used In Examples

Each of the operators in Simple Collection Operators (page 37), Object Operators (page 38), and Array and Set Operators (page 39) includes an example code snippet using a key path containing an operator, as well as the results that would be generated. To illustrate this it is necessary to use a theoretical class, in this case the `Transaction` class. The `Transaction` class encapsulates the simplest form of checkbook entires. It contains three properties: `payee` (an `NSString`), `amount` (an `NSNumber`), and `date` (an `NSDate`).

The `Transaction` instances are stored in a collection object that is assigned to the hypothetical variable `transactions`, which is an instance of `NSArray`. For the operator that operates on an `NSSet` collection, it can be assumed that the `transactions` collection is an instance of `NSSet`. This is discussed in the relevant operators.

In order to provide results data for the operator action on the `transactions` collection sample data is required. Each row in Table 1 contains the data for an instance of `Transaction`. The values in the table have been formatted (where applicable) to make them easier to understand. The results displayed by the examples are also formatted.

Table 1 Example data for the `Transactions` objects

payee values	amount values (formatted as currency)	date values (formatted as month day, year)
Green Power	$120.00	Dec 1, 2009
Green Power	$150.00	Jan 1, 2010
Green Power	$170.00	Feb 1, 2010
Car Loan	$250.00	Jan 15, 2010
Car Loan	$250.00	Feb 15, 2010
Car Loan	$250.00	Mar 15, 2010
General Cable	$120.00	Dec 1, 2009
General Cable	$155.00	Jan 1, 2010
General Cable	$120.00	Feb 1, 2010
Mortgage	$1,250.00	Jan 15, 2010
Mortgage	$1,250.00	Feb 15, 2010
Mortgage	$1,250.00	Mar 15, 2010

payee values	amount values (formatted as currency)	date values (formatted as month day, year)
Animal Hospital	$600.00	Jul 15, 2010

Simple Collection Operators

Simple collection operators operate on the properties to the right of the operator in either an array or set.

@avg

The `@avg` operator uses `valueForKeyPath:` to get the values specified by the property specified by the key path to the right of the operator, converts each to a `double`, and returns the average value as an instance of `NSNumber`. In the case where the value is `nil`, `0` is assumed instead.

The following example returns the average value of the transaction amount for the objects in `transactions`:

```
NSNumber *transactionAverage = [transactions valueForKeyPath:@"@avg.amount"];
```

The formatted result of `transactionAverage` is $456.54.

@count

The `@count` operator returns the number of objects in the left key path collection as an instance of `NSNumber`, the key path to the right of the operator is ignored.

The following example returns the number of `Transaction` objects in `transactions`:

```
NSNumber *numberOfTransactions = [transactions valueForKeyPath:@"@count"];
```

The value of `numberOfTransactions` is 13.

@max

The `@max` operator compares the values of the property specified by the key path to the right of the operator and returns the maximum value found. The maximum value is determined using the `compare:` method of the objects at the specified key path. The compared property objects must support comparison with each other. If the value of the right side of the key path is `nil`, it is ignored.

The following example returns the maximum value of the date values (date of the latest transaction) for the `Transaction` objects in `transactions`:

```
NSDate *latestDate = [transactions valueForKeyPath:@"@max.date"];
```

The `latestDate` value (formatted) is Jul 15, 2010.

@min

The `@min` operator compares the values of the property specified by the key path to the right of the operator and returns the minimum value found. The minimum value is determined using the `compare:` method of the objects at the specified key path. The compared property objects must support comparison with each other. If the value of the right side of the key path is `nil`, it is ignored.

The following example returns the minimum value (date of the earliest transaction) of the `date` property for the `Transaction` objects in `transactions`:

```
NSDate *earliestDate = [transactions valueForKeyPath:@"@min.date"];
```

The `earliestDate` value (formatted) is Dec 1, 2009.

@sum

The `@sum` operator returns the sum of the values of the property specified by the key path to the right of the operator. Each number is converted to a `double`, the sum of the values is computed, and the total is wrapped as an instance of `NSNumber` and returned. If the value of the right side of the key path is `nil`, it is ignored.

The following example returns the sum of the `amounts` property for the transactions in `transactions`:

```
NSNumber *amountSum = [transactions valueForKeyPath:@"@sum.amount"];
```

The resulting `amountSum` value (formatted) is $5,935.00.

Object Operators

The object operators provide results when they are applied to a single collection instance.

@distinctUnionOfObjects

The `@distinctUnionOfObjects` operator returns an array containing the distinct objects in the property specified by the key path to the right of the operator.

The following example returns the `payee` property values for the transactions in `transactions` with any duplicate values removed:

```
NSArray *payees = [transactions valueForKeyPath:@"@distinctUnionOfObjects.payee"];
```

The resulting `payees` array contains the following strings: Car Loan, General Cable, Animal Hospital, Green Power, Mortgage.

The `@unionOfObjects` operator is similar, but does not remove duplicate objects.

Important: This operator raises an exception if any of the leaf objects is `nil`.

@unionOfObjects

The `@unionOfObjects` operator returns an array containing the distinct objects in the property specified by the key path to the right of the operator. Unlike @distinctUnionOfObjects, duplicate objects are not removed.

The following example returns the `payee` property values for the transactions in `transactions`:

```
NSArray *payees = [transactions valueForKeyPath:@"@unionOfObjects.payee"];
```

The resulting `payees` array contains the following strings: Green Power, Green Power, Green Power, Car Loan, Car Loan, Car Loan, General Cable, General Cable, General Cable, Mortgage, Mortgage, Mortgage, Animal Hospital.

The `@distinctUnionOfArrays` operator is similar, but removes duplicate objects.

Important: This operator raises an exception if any of the leaf objects is `nil`.

Array and Set Operators

The array and set operators operate on nested collections, that is, a collection where each entry contains a collection.

The variable `arrayOfTransactions` is used in the example for each operator. It is an array containing two arrays of `Transaction` objects.

The following code snippet shows how the nested collection would be created:

```
// Create the array that contains additional arrays.
self.arrayOfTransactionsArray = [NSMutableArray array];

// Add the array of objects used in the above examples.
[arrayOfTransactionsArray addObject:transactions];

// Add a second array of objects; this array contains alternate values.
[arrayOfTransactionsArrays addObject:moreTransactions];
```

The first array of `Transaction` objects contains the data listed in Table 1 (page 36) and the second array (`moreTransactions`) contains `Transaction` objects with the hypothetical data in Table 2.

Table 2 Hypothetical `Transaction` data in the `moreTransactions` array

payee values	`amount` values (formatted as currency)	`date` values (formatted as month day, year)
General Cable – Cottage	$120.00	Dec 18, 2009
General Cable – Cottage	$155.00	Jan 9, 2010
General Cable – Cottage	$120.00	Dec 1, 2010
Second Mortgage	$1,250.00	Nov 15, 2010
Second Mortgage	$1,250.00	Sep 20, 2010
Second Mortgage	$1,250.00	Feb 12, 2010
Hobby Shop	$600.00	Jun 14, 2010

@distinctUnionOfArrays

The `@distinctUnionOfArrays` operator returns an array containing the distinct objects in the property specified by the key path to the right of the operator.

The following code example will return the distinct values of the `payee` property in all the arrays with `arrayOfTransactionsArrays`:

```
NSArray *payees = [arrayOfTransactionsArrays
valueForKeyPath:@"@distinctUnionOfArrays.payee"];
```

The resulting `payees` array contains the following values: Hobby Shop, Mortgage, Animal Hospital, Second Mortgage, Car Loan, General Cable - Cottage, General Cable, Green Power.

The `@unionOfArrays` operator is similar, but does not remove duplicate objects.

Important: This operator raises an exception if any of the leaf objects is `nil`.

@unionOfArrays

The `@unionOfArrays` operator returns an array containing the objects in the property specified by the key path to the right of the operator. Unlike `@distinctUnionOfArrays`, duplicate objects are not removed.

The following code example will return the values of the `payee` property in all the arrays with `arrayOfTransactionsArrays`:

```
NSArray *payees = [arrayOfTransactionsArrays
valueForKeyPath:@"@unionOfArrays.payee"];
```

The resulting `payees` array contains the following values: Green Power, Green Power, Green Power, Car Loan, Car Loan, Car Loan, General Cable, General Cable, General Cable, Mortgage, Mortgage, Mortgage, Animal Hospital, General Cable - Cottage, General Cable - Cottage, General Cable - Cottage, Second Mortgage, Second Mortgage, Second Mortgage, Hobby Shop.

Important: This operator raises an exception if any of the leaf objects is `nil`.

@distinctUnionOfSets

The `@distinctUnionOfSets` operator returns a set containing the distinct objects in the property specified by the key path to the right of the operator.

This operator works the same as `@distinctUnionOfArrays`, except that it expects an `NSSet` instance containing `NSSet` instances of `Transaction` objects rather than arrays. It returns an `NSSet` instance. Using the example data set, the returned set would contain the results as those shown in @distinctUnionOfArrays (page 40).

Important: This operator raises an exception if any of the leaf objects is `nil`.

Accessor Search Implementation Details

Key-value coding attempts to use accessor methods to get and set values, before resorting to directly accessing the instance variable. This article describes how the key-value coding methods determine how the value is accessed.

Accessor Search Patterns for Simple Attributes

Default Search Pattern for setValue:forKey:

When the default implementation of `setValue:forKey:` is invoked for a property the following search pattern is used:

1. The receiver's class is searched for an accessor method whose name matches the pattern `set<Key>:`.
2. If no accessor is found, and the receiver's class method `accessInstanceVariablesDirectly` returns `YES`, the receiver is searched for an instance variable whose name matches the pattern `_<key>`, `_is<Key>`, `<key>`, or `is<Key>`, in that order.
3. If a matching accessor or instance variable is located, it is used to set the value. If necessary, the value is extracted from the object as described in Representing Non-Object Values (page 31).
4. If no appropriate accessor or instance variable is found, `setValue:forUndefinedKey:` is invoked for the receiver.

Default Search Pattern for valueForKey:

When the default implementation of `valueForKey:` is invoked on a receiver, the following search pattern is used:

1. Searches the class of the receiver for an accessor method whose name matches the pattern `get<Key>`, `<key>`, or `is<Key>`, in that order. If such a method is found it is invoked. If the type of the method's result is an object pointer type the result is simply returned. If the type of the result is one of the scalar types supported by `NSNumber` conversion is done and an `NSNumber` is returned. Otherwise, conversion is done and an `NSValue` is returned. Results of arbitrary types are converted to `NSValue` objects, not just `NSPoint`, `NSRange`, `NSRect`, and `NSSize` types).

2. Otherwise (no simple accessor method is found), searches the class of the receiver for methods whose names match the patterns `countOf<Key>` and `objectIn<Key>AtIndex:` (corresponding to the primitive methods defined by the `NSArray` class) and `<key>AtIndexes:` (corresponding to the `NSArray` method `objectsAtIndexes:`).

   If the `countOf<Key>` method and at least one of the other two possible methods are found, a collection proxy object that responds to all `NSArray` methods is returned. Each `NSArray` message sent to the collection proxy object will result in some combination of `countOf<Key>`, `objectIn<Key>AtIndex:`, and `<key>AtIndexes:` messages being sent to the original receiver of `valueForKey:`. If the class of the receiver also implements an optional method whose name matches the pattern `get<Key>:range:` that method will be used when appropriate for best performance.

3. Otherwise (no simple accessor method or set of array access methods is found), searches the class of the receiver for a threesome of methods whose names match the patterns `countOf<Key>`, `enumeratorOf<Key>`, and `memberOf<Key>:` (corresponding to the primitive methods defined by the `NSSet` class).

   If all three methods are found, a collection proxy object that responds to all `NSSet` methods is returned. Each `NSSet` message sent to the collection proxy object will result in some combination of `countOf<Key>`, `enumeratorOf<Key>`, and `memberOf<Key>:` messages being sent to the original receiver of `valueForKey:`.

4. Otherwise (no simple accessor method or set of collection access methods is found), if the receiver's class method `accessInstanceVariablesDirectly` returns `YES`, the class of the receiver is searched for an instance variable whose name matches the pattern `_<key>`, `_is<Key>`, `<key>`, or `is<Key>`, in that order. If such an instance variable is found, the value of the instance variable in the receiver is returned. If the type of the result is one of the scalar types supported by `NSNumber` conversion is done and an `NSNumber` is returned. Otherwise, conversion is done and an `NSValue` is returned. Results of arbitrary types are converted to `NSValue` objects, not just `NSPoint`, `NSRange`, `NSRect`, and `NSSize` types.

5. If none of the above situations occurs, returns a result the default implementation invokes `valueForUndefinedKey:`.

Accessor Search Pattern for Ordered Collections

The default search pattern for `mutableArrayValueForKey:` is as follows:

1. The receiver's class is searched for a pair of methods whose names match the patterns `insertObject:in<Key>AtIndex:` and `removeObjectFrom<Key>AtIndex:` (corresponding to the `NSMutableArray` primitive methods `insertObject:atIndex:` and `removeObjectAtIndex:`

respectively), or methods matching the pattern `insert<Key>:atIndexes:` and `remove<Key>AtIndexes:` (corresponding to the `NSMutableArrayinsertObjects:atIndexes:` and `removeObjectsAtIndexes:` methods).

If at least one insertion method and at least one removal method are found each `NSMutableArray` message sent to the collection proxy object will result in some combination of `insertObject:in<Key>AtIndex:`, `removeObjectFrom<Key>AtIndex:`, `insert<Key>:atIndexes:`, and `remove<Key>AtIndexes:` messages being sent to the original receiver of `mutableArrayValueForKey:`.

If receiver's class also implements the optional replace object method matching the pattern `replaceObjectIn<Key>AtIndex:withObject:` or `replace<Key>AtIndexes:with<Key>:` that method will be used when appropriate for best performance.

2. Otherwise, the receiver's class is searched for an accessor method whose name matches the pattern `set<Key>:`. If such a method is found each `NSMutableArray` message sent to the collection proxy object will result in a `set<Key>:` message being sent to the original receiver of `mutableArrayValueForKey:`. It is much more efficient to implement the indexed accessor methods discussed in the previous step.

3. Otherwise, if the receiver's class responds `YES` to `accessInstanceVariablesDirectly`, the receiver's class is searched for an instance variable whose name matches the pattern `_<key>` or `<key>`, in that order.

   If such an instance variable is found, each `NSMutableArray` message sent to the collection proxy object will be forwarded to the instance variable's value, which typically will be an instance of `NSMutableArray` or a subclass of `NSMutableArray`.

4. Otherwise, returns a mutable collection proxy object that results in a `setValue:forUndefinedKey:` message being sent to the original receiver of the `mutableArrayValueForKey:` message whenever the proxy receives an `NSMutableArray` message.

   The default implementation of setValue:forUndefinedKey: raises an `NSUndefinedKeyException`, but you can override it in your application.

Note: The repetitive `set<Key>:` messages implied by the description in step 2 are a potential performance problem. For better performance, implement methods that fulfill the requirements for Step 1 in your key-value coding-compliant class.

Accessor Search Pattern for Uniquing Ordered Collections

The default implementation of `mutableOrderedSetValueForKey:` recognizes the same simple accessor methods and ordered set accessor methods as `valueForKey` (see Default Search Pattern for valueForKey: (page 43)), and follows the same direct instance variable access policies, but always returns a mutable collection proxy object instead of the immutable collection that `valueForKey:` would return. It also:

1. Searches the class of the receiver for methods whose names match the patterns `insertObject:in<Key>AtIndex:` and `removeObjectFrom<Key>AtIndex:` (corresponding to the two most primitive methods defined by the `NSMutableOrderedSet` class), and also `insert<Key>:atIndexes:` and `remove<Key>AtIndexes:` (corresponding to `insertObjects:atIndexes:`] and `removeObjectsAtIndexes:`).

   If at least one insertion method and at least one removal method are found each `NSMutableOrderedSet` message sent to the collection proxy object will result in some combination of `insertObject:in<Key>AtIndex:`, `removeObjectFrom<Key>AtIndex:`, `insert<Key>:atIndexes:`, and `remove<Key>AtIndexes:` messages being sent to the original receiver of `mutableOrderedSetValueForKey:`.

   If the class of the receiver also implements an optional method whose name matches the pattern `replaceObjectIn<Key>AtIndex:withObject:` or `replace<Key>AtIndexes:with<Key>:` that method will be used when appropriate for best performance.

2. Otherwise, searches the class of the receiver for an accessor method whose name matches the pattern `set<Key>:`. If such a method is found each NSMutableOrderedSet message sent to the collection proxy object will result in a `set<Key>:` message being sent to the original receiver of `mutableOrderedSetValueForKey:`.

3. Otherwise, if the receiver's class's `accessInstanceVariablesDirectly` method returns `YES`, searches the class of the receiver for an instance variable whose name matches the pattern _<key> or <key>, in that order. If such an instance variable is found, each `NSMutableOrderedSet` message sent to the collection proxy object will be forwarded to the instance variable's value, which therefore must typically be an instance of `NSMutableOrderedSet` or of a subclass of `NSMutableOrderedSet`.

4. Otherwise, returns a mutable collection proxy object anyway. Each `NSMutableOrderedSet` message sent to the collection proxy object will result in a `setValue:forUndefinedKey:` message being sent to the original receiver of `mutableOrderedSetValueForKey:`. The default implementation of `setValue:forUndefinedKey:` raises an `NSUndefinedKeyException`, but you can override it in your application.

Note: The repetitive `set<Key>:` messages implied by Step 2's description are a potential performance problem. For better performance, implement insertion and removal methods that fulfill the requirements for Step 1 in your KVC-compliant class. For best performance implement a replacement method as well.

Accessor Search Pattern for Unordered Collections

The default search pattern for `mutableSetValueForKey:` is as follows:

1. Searches the receiver's class for methods whose names match the patterns `add<Key>Object:` and `remove<Key>Object:` (corresponding to the `NSMutableSet` primitive methods `addObject:` and `removeObject:` respectively) and also `add<Key>:` and `remove<Key>:` (corresponding to `NSMutableSet` methods `unionSet:` and `minusSet:`). If at least one addition method and at least one removal method are found each `NSMutableSet` message sent to the collection proxy object will result in some combination of `add<Key>Object:`, `remove<Key>Object:`, `add<Key>:`, and `remove<Key>:` messages being sent to the original receiver of `mutableSetValueForKey:`.

   If the class of the receiver also implements an optional method whose name matches the pattern `intersect<Key>:` or `set<Key>:`, that method will be used when appropriate for best performance.

2. If the receiver is a managed object, the search pattern does not continue as it would for non-managed objects. See Managed Object Accessor Methods in *Core Data Programming Guide* for more information.

3. Otherwise, the receiver's class is searched for an accessor method whose name matches the pattern `set<Key>:`. If such a method is found, each `NSMutableSet` message sent to the collection proxy object will result in a `set<Key>:` message being sent to the original receiver of `mutableSetValueForKey:`.

4. Otherwise, if the receiver's class method `accessInstanceVariablesDirectly` returns `YES`, the class is searched for an instance variable whose name matches the pattern `_<key>` or `<key>`, in that order. If such an instance variable is found, each `NSMutableSet` message sent to the collection proxy object will be forwarded to the instance variable's value, which therefore must typically be an instance of `NSMutableSet` or a subclass of `NSMutableSet`.

5. Otherwise, returns a mutable collection proxy object anyway. Each `NSMutableSet` message sent to the collection proxy object will result in a `setValue:forUndefinedKey:` message being sent to the original receiver of `mutableSetValueForKey:`.

Note: The repetitive `set<Key>:` messages implied by the description in step 3 are a potential performance problem. For better performance implement methods that fulfill the requirements for Step 1 in your key-value coding-compliant class.

Describing Property Relationships

Class descriptions provide a method of describing the to-one and to-many properties in a class. Defining these relationships between class properties allows for more intelligent and flexible manipulation of these properties with key-value coding.

Class Descriptions

`NSClassDescription` is a base class that provides the interface for obtaining meta-data about classes. A class description object records the available attributes of objects of a particular class and the relationships (one-to-one, one-to-many, and inverse) between objects of that class and other objects. For example the `attributes` method returns the list of all attributes defined for a class; the methods `toManyRelationshipKeys` and `toOneRelationshipKeys` return arrays of keys that define to-many and to-one relationships; and `inverseRelationshipKey:` returns the name of the relationship pointing back to the receiver from the destination of the relationship for the provided key.

`NSClassDescription` does not define methods for defining the relationships. Concrete subclasses must define these methods. Once created, you register a class description with the NSClassDescription `registerClassDescription:forClass:` class method.

`NSScriptClassDescription` is the only concrete subclass of NSClassDescription provided in Cocoa. It encapsulates an application's scripting information.

Performance Considerations

Though key-value coding is efficient, it adds a level of indirection that is slightly slower than direct method invocations. You should use key-value coding only when you can benefit from the flexibility that it provides.

Additional optimization opportunities may be added in the future, but these will not change basic methods for key-value coding compliance.

Overriding Key-Value Coding Methods

The default implementations of the key-value coding methods, such as `valueForKey:`, cache Objective-C runtime information to increase efficiency. You should take care when overriding these implementations to ensure that you do not adversely affect application performance.

Optimizing To-Many Relationships

To-many relationships that are implemented using the indexed form of the accessors will provide significant performance gains in many cases.

It's recommended that you implement at least the minimum indexed accessors for your to-many collections. See Collection Accessor Patterns for To-Many Properties (page 17) for further information.

Document Revision History

This table describes the changes to *Key-Value Coding Programming Guide*.

Date	Notes
2012-07-17	Updated for OS X v10.8 to include new Objective-C features.
2011-06-06	Updated for OS X v10.7 to include ordered uniquing relationships.
2010-09-01	Clarified that "valueWithRect:" is OS X only. Revised "Collection Operators" chapter.
2010-04-28	Corrected typo.
2010-01-20	Clarified "Registering Dependent Keys" and Core Data managed objects interaction. Corrected "Mutable Unordered Accessors" description.
2009-02-04	Added substantial task information and sample code.
2007-06-06	Added a warning that array and set operators raise on nil values.
2007-01-08	Added caution about checking the error parameter in validateName:error: and making sure a valid NSError object is returned.
2006-06-28	Added Key-Value Observing Programming Guide to the list of related documents.
2006-04-04	Added example method signature for the -get<Key>:range: accessor pattern

Date	Notes
2006-03-08	Corrected variable names in the @distinctUnionOfArrays example.
2005-08-11	Changed title of "Why Use Key-Value Coding" article.
2005-07-07	Added descriptions for the collection operators @unionOfSets and @distinctUnionOfSets. Clarified the return policy for validateValue:forKey:.
2005-04-29	Corrected minor typos.
2004-08-31	Updated table of contents. Clarified that the `@sum` array operator returns an NSNumber in Collection Operators (page 35). Corrected minor typos.
2004-06-28	Corrected typos.
2004-04-19	Clarified that indexed accessor methods make a property appear as an array in Collection Accessor Patterns for To-Many Properties (page 17).
2003-10-15	*Key-Value Coding* has been rewritten for OS X v10.3.
2003-07-19	Accessor Search Implementation Details (page 43) was updated with information on the methods deprecated in OS X v10.3.
2002-11-12	Revision history was added to *Key-Value Coding*.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010