Accessing Object Properties

An object typically specifies *properties* in its interface declaration, and these properties belong to one of several categories:

- **Attributes.** These are simple values, such as a scalars, strings, or Boolean values. Value objects such as `NSNumber` and other immutable types such as as `NSColor` are also considered attributes.
- **To-one relationships.** These are mutable objects with properties of their own. An object's properties can change without the object itself changing. For example, a bank account object might have an owner property that is an instance of a `Person` object, which itself has an address property. The owner's address may change without changing the owner reference held by the bank account. The bank account's owner did not change. Only her address did.
- **To-many relationships.** These are collection objects. You commonly use an instance of `NSArray` or `NSSet` to hold such a collection, although custom collection classes are also possible.

The `BankAccount` object declared in Listing 2-1 demonstrates one of each type of property.

Listing 2-1 Properties of the `BankAccount` object

```
@interface BankAccount : NSObject

@property (nonatomic) NSNumber* currentBalance;              // An attribute
@property (nonatomic) Person* owner;                         // A to-one relation
@property (nonatomic) NSArray< Transaction* >* transactions; // A to-many relation

@end
```

In order to maintain encapsulation, an object typically provides accessor methods for the properties on its interface. The object's author may write these methods explicitly or may rely on the compiler to synthesize them automatically. Either way, the author of code using one of these accessors must write the property name into the code before compiling it. The name of the accessor method becomes a static part of the code that is using it. For example, given the bank account object declared in Listing 2-1, the compiler synthesizes a setter that you can invoke for the `myAccount` instance:

```
[myAccount setCurrentBalance:@(100.0)];
```

This is direct, but lacks flexibility. A key-value coding compliant object, on the other hand, offers a more general mechan

On This Page

Identifying an Object's Properties with Keys and Key Paths

A *key* is a string that identifies a specific property. Typically, by convention, the key representing a property is the name of the property itself as it appears in code. Keys must use ASCII encoding, may not contain whitespace, and usually begin with a lowercase letter (though there are exceptions, such as the `URL` property found in many classes).

Because the `BankAccount` class in Listing 2-1 is key-value coding compliant, it recognizes the keys `owner`, `currentBalance`, and `transactions`, which are the names of its properties. Instead of calling the `setCurrentBalance:` method, you can set the value by its key:

```
[myAccount setValue:@(100.0) forKey:@"currentBalance"];
```

In fact, you can set all the properties of the `myAccount` object with the same method, using different key parameters. Because the parameter is a string, it can be a variable that is manipulated at run-time.

A *key path* is a string of dot-separated keys 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. Key paths are useful for drilling down into a hierarchy of objects with a single method call.

For example, the key path `owner.address.street` applied to a bank account instance refers to the value of the street string that is stored in the address of the bank account's owner, assuming the `Person` and `Address` classes are also key-value coding compliant.

> NOTE
>
> In Swift, instead of using a string to indicate a key or key path, you can use a `#keyPath` expression. This offers the advantage of a compile time check, as described in the Keys and Key Paths section of the *Using Swift with Cocoa and Objective-C (Swift 3.0.1)* guide.

Getting Attribute Values Using Keys

An object is key-value coding compliant when it adopts the `NSKeyValueCoding` protocol. An object that inherits from `NSObject`, which provides a default implementation of the protocol's essential methods, automatically adopts this protocol with certain default behaviors. Such an object implements at least the following basic key-based getters:

- `valueForKey:` - Returns the value of a property named by the key parameter. If the property named by the key cannot be found according to the rules described in Accessor Search Patterns, then the object sends itself a `valueForUndefinedKey:` message. The default implementation of `valueForUndefinedKey:` raises an `NSUndefinedKeyException`, but subclasses may override this behavior and handle the situation more gracefully.
- `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 a particular key—that is, for which the default implementation of `valueForKey:` cannot find an accessor method—receives a `valueForUndefinedKey:` message.
- `dictionaryWithValuesForKeys:` - Returns the values for an array of keys relative to the receiver. The method calls `valueForKey:` for each key in the array. 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 the `NSNull` object. `NSNull` provides a single instance that represents the `nil` value for object properties. The default implementations of `dictionaryWithValuesForKeys:` and the related `setValuesForKeysWithDictionary:` translate between `NSNull` (in the dictionary parameter) and `nil` (in the stored property) automatically.

When you use a key path to address a property, if any but the final key in the key path is a to-many relationship (that is, it references a collection), 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.p`

On This Page

for multiple arrays in the key path. The key path `accounts.transactions.payee` returns an array with all the payee objects for all the transactions in all the accounts.

Setting Attribute Values Using Keys

As with getters, key-value coding compliant objects also provide a small group of generalized setters with default behavior based upon the implementation of the `NSKeyValueCoding` protocol found in `NSObject`:

- `setValue:forKey:` - Sets the value of the specified key relative to the object receiving the message to the given value. The default implementation of `setValue:forKey:` automatically unwraps `NSNumber` and `NSValue` objects that represent scalars and structs and assigns them to the property. See Representing Non-Object Values for details on the wrapping and unwrapping semantics.

  If the specified key corresponds to a property that the object receiving the setter call does not have, the object sends itself a `setValue:forUndefinedKey:` message. The default implementation of `setValue:forUndefinedKey:` raises an `NSUndefinedKeyException`. However, subclasses may override this method to handle the request in a custom manner.
- `setValue:forKeyPath:` - Sets the given value at the specified key path relative to the receiver. Any object in the key path sequence that is not key-value coding compliant for a particular key receives a `setValue:forUndefinedKey:` message.
- `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.

In the default implementation, when you attempt to set a non-object property to a `nil` value, the key-value coding compliant object sends itself a `setNilValueForKey:` message. The default implementation of `setNilValueForKey:` raises an `NSInvalidArgumentException`, but an object may override this behavior to substitute a default value or a marker value instead, as described in Handling Non-Object Values.

Using Keys to Simplify Object Access

To see how key-based getters and setters can simplify your code, consider the following example. In macOS, `NSTableView` and `NSOutlineView` objects associate an identifier string with each of their columns. If the model object backing the table is not key-value coding compliant, the table's data source method is forced to examine each column identifier in turn to find the correct property to return, as shown in Listing 2-2. Further, in the future, when you add another property to your model, in this case the `Person` object, you must also revisit the data source method, adding another condition to test for the new property and return the relevant value.

Listing 2-2 Implementation of data source method without key-value coding

```
- (id)tableView:(NSTableView *)tableview objectValueForTableColumn:(id)column row:(NSInteger)row
{
    id result = nil;
    Person *person = [self.people objectAtIndex:row];

    if ([[column identifier] isEqualToString:@"name"]) {
        result = [person name];
    } else if ([[column identifier] isEqualToString:@"age"]) {
        result = @([person age]);  // Wrap age, a scalar, as an NSNumber
    } else if ([[column identifier] isEqualToString:@"favoriteColor"]) {
        result = [person favoriteColor];
    } // And so on...

    return result;
}
```

On the other hand, Listing 2-3 shows a much more compact implementation of the same data source method that takes advantage of a key-value coding compliant `Person` object. Using only the `valueForKey:` getter, the data source method returns the appropriate value using the column identifier as a key. In addition to being shorter, it is also more general, because it continues to work unchanged when new columns are added later, as long as the column identifiers always match the model object's property names.

Listing 2-3 Implementation of data source method with key-value coding

On This Page

```
- (id)tableView:(NSTableView *)tableview objectValueForTableColumn:(id)column row:(NSInteger)row
{
    return [[self.people objectAtIndex:row] valueForKey:[column identifier]];
}
```