Achieving Basic Key-Value Coding Compliance

On This Page

When adopting key-value coding for an object, you rely on the default implementation of the `NSKeyValueCoding` protocol by having your object inherit from `NSObject` or one of its many subclasses. The default implementation, in turn, relies on you to define your object's instance variables (or *ivars*) and accessor methods following certain well-defined patterns, so that it can associate key strings with properties when it receives key-value coded messages, such as `valueForKey:` and `setValue:forKey:`.

You often adhere to the standard patterns in Objective-C by simply declaring a property using a `@property` statement, and allowing the compiler to automatically synthesize the ivar and accessors. The compiler follows the expected patterns by default.

> NOTE
>
> In Swift, simply declaring a property in the usual way produces the appropriate accessors automatically, and you never interact directly with ivars. For more information about properties in Swift, read Properties in the *The Swift Programming Language (Swift 3.0.1)*. For information specific to interacting with Objective-C properties from Swift, read Accessing Properties in *Using Swift with Cocoa and Objective-C (Swift 3.0.1)*.

If you do need to implement accessors or ivars manually in Objective-C, follow the guidelines in this section to maintain basic compliance. To provide additional functionality that enhances interaction with your object's collection properties in any language, implement the methods described in Defining Collection Methods. To further enhance your object with key-value validation, implement the methods described in Adding Validation.

> NOTE
>
> The default implementation of key-value coding works with a broader range of ivars and accessors than are described here. If you have legacy code that uses other variable or accessor conventions, examine the search patterns in Accessor Search Patterns to determine if the default implementation can locate your object's properties.

Basic Getters

To implement a getter that returns the value of a property, while perhaps doing additional custom work, use a method named like the property, such as for the `title` string property:

```
- (NSString*)title
{
   // Extra getter logic…

   return _title;
}
```

For a property holding a Boolean value, you can alternatively use a method prefixed with `is`, such as for the `hidden` Boolean property:

```
- (BOOL)isHidden
{
   // Extra getter logic…

   return _hidden;
}
```

When the property is a scalar or a structure, the default implementation of key-value coding wraps the value in an object for use on the protocol method's interface, as described in Representing Non-Object Values. You do not need to do anything special to support this behavior.

Basic Setters

To implement a setter that stores the value of a property, use a method with the capitalized name of the property prefixed by the word `set`. For the `hidden` property:

```
- (voic
{
    // Extra setter logic…

   _hidden = hidden;
}
```

On This Page

> WARNING
> Never call the validation methods described in Validating Properties from inside a `set<Key>:` method.

When a property is a non-object type, such as the Boolean `hidden`, the protocol's default implementation detects the underlying data type, and unwraps the object value (an `NSNumber` instance in this case) that comes from `setValue:forKey:` before applying it to your setter, as described in Representing Non-Object Values. You do not need to handle this in the setter itself. However, if there is a possibility that a `nil` value might be written to your non-object property, you override `setNilValueForKey:` to handle this situation, as described in Handling Non-Object Values. An appropriate behavior for the `hidden` property might simply be to interpret `nil` as `NO`:

```
- (void)setNilValueForKey:(NSString *)key
{
    if ([key isEqualToString:@"hidden"]) {
        [self setValue:@(NO) forKey:@”hidden”];
    } else {
        [super setNilValueForKey:key];
    }
}
```

You provide the above method override, if appropriate, even when you allow the compiler to synthesize the setter.

Instance Variables

When the default implementation of one of the key-value coding accessor methods can't find a property's accessor, it queries its class's `accessInstanceVariablesDirectly` method to see if the class allows direct use of instance variables. By default, this class method returns `YES`, although you can override this method to return `NO`.

If you do allow use of ivars, ensure that they are named in the usual way, using the property name prefixed by an underscore (_). Normally, the compiler does this for you when automatically synthesizing properties, but if you use an explicit `@synthesize` directive, you can enforce this naming yourself:

```
@synthesize title = _title;
```

In some cases, instead of using a `@synthesize` directive or allowing the compiler to automatically synthesize a property, you use a `@dynamic` directive to inform the compiler that you will provide getters and setters at runtime. You might do this to avoid automatically synthesizing a getter, so that you can provide collection accessors instead, as described in Defining Collection Methods. In this, case you declare the ivar yourself as part of the interface declaration:

```
@interface MyObject : NSObject {
    NSString* _title;
}

@property (nonatomic) NSString* title;

@end
```