Representing Non-Object Values

On This Page

The default implementation of the key-value coding protocol methods provided by `NSObject` work with both object and non-object properties. The default implementation automatically translates between object parameters or return values, and non-object properties. This allows the signatures of the key-based getters and setters to remain consistent even when the stored property is a scalar or a structure.

> NOTE
>
> Because all properties in Swift are objects, this section only apples to Objective-C properties.

When you invoke one of the protocol's getters, such as `valueForKey:`, the default implementation determines the particular accessor method or instance variable that supplies the value for the specified key according to the rules described in Accessor Search Patterns. If the return value is not an object, the getter uses this value to initialize an `NSNumber` object (for scalars) or `NSValue` object (for structures) and returns that instead.

Similarly, by default, setters like `setValue:forKey:` determine the data type required by a property's accessor or instance variable, given a particular key. If the data type is not an object, the setter first sends an appropriate `<type>Value` message to the incoming value object to extract the underlying data, and stores that instead.

> NOTE
>
> When you invoke one of the key-value coding protocol setters with a `nil` value for a non-object property, the setter has no obvious, general course of action to take. Therefore, it sends a `setNilValueForKey:` message to the object receiving the setter call. The default implementation of this method raises an `NSInvalidArgumentException` exception, but subclasses may override this behavior, as described in Handling Non-Object Values, for example to set a marker value, or provide a meaningful default.

Wrapping and Unwrapping Scalar Types

Table 5-1 lists the scalar types that the default key-value coding implementation wraps using an `NSNumber` instance. For each data type, the table shows the creation method used to initialize an `NSNumber` from the underlying property value to supply a getter return value. It then shows the accessor method used to extract the value from the setter input parameter during a set operation.

Table 5-1 Scalar types as wrapped in `NSNumber` objects

Data type	Creation method	Accessor method
`BOOL`	`numberWithBool:`	`boolValue` (in iOS) `charValue` (in macOS)*
`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`
`unsigned long`	`numberWithUnsignedLong:`	`unsignedLong`
`unsigned long long`	`numberWithUnsignedLongLong:`	`unsignedLongLong`
`unsigned short`	`numberWithUnsignedShort:`	`unsignedShort`

> NOTE

*In macOS, for historical reasons, `BOOL` is type defined as `signed char`, and KVC does not distinguish between these. As a result, you should not pass a string value such as `@"true"` or `@"YES"` to `setValue:forKey:` when the key is a `BOOL`. KVC will attempt to invoke `charValue` (because the `BOOL` is inherently a `char`), but `NSString` does not implement this method, which results in a runtime error. Instead, pass only the key is type `bool` and KVC invokes `boolValue`, which works for either an `NSNumber` object or a properly formatted `NSString` object.

On This Page

Wrapping and Unwrapping Structures

Table 5-2 shows the creation and accessor methods that the default accessors use for wrapping and unwrapping the common `NSPoint`, `NSRange`, `NSRect`, and `NSSize` structures.

Table 5-2 Common struct types as wrapped using `NSValue`.

Data type	Creation method	Accessor method
`NSPoint`	`valueWithPoint:`	`pointValue`
`NSRange`	`valueWithRange:`	`rangeValue`
`NSRect`	`valueWithRect:` (macOS 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, consider the structure and class interface declared in Listing 5-1.

Listing 5-1 A sample class using a custom structure

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

@interface MyClass
@property (nonatomic) ThreeFloats threeFloats;
@end
```

Using an instance of this class called `myClass`, you obtain the `threeFloats` value with key-value coding:

```
NSValue* result = [myClass valueForKey:@"threeFloats"];
```

The default implementation of `valueForKey:` invokes the `threeFloats` getter, and then returns the result wrapped in an `NSValue` object.

Similarly, you can set the `threeFloats` value using key-value coding:

```
ThreeFloats floats = {1., 2., 3.};
NSValue* value = [NSValue valueWithBytes:&floats objCType:@encode(ThreeFloats)];
[myClass setValue:value forKey:@"threeFloats"];
```

The default implementation unwraps the value with a `getValue:` message, and then invokes `setThreeFloats:` with the resulting structure.