Number and Value Programming Topics

Developer

Contents

Introduction to Numbers and Other Values

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

This topic describes object wrappers for primitive C data types, which are implemented by `NSValue` and its subclasses `NSNumber` and `NSDecimalNumber`, and the `NSNull` instance used to represent a null value.

Organization of This Document

This document contains the following articles:

- Using Values (page 4) describes the generic value type.
- Using Numbers (page 6) describes scalars.
- Using Decimal Numbers (page 8) describes the objects used for base-10 arithmetic.
- Using NSNull (page 10) describes using the `NSNull` instance.

Using Values

An `NSValue` object is a simple container for a single C or Objective-C data item. It can hold any of the scalar types such as `int`, `float`, and `char`, as well as pointers, structures, and object `id`s. The purpose of this class is to allow items of such data types to be added to collection objects such as instances of `NSArray` or `NSSet`, which require their elements to be objects. `NSValue` objects are always immutable.

To create an `NSValue` object with a particular data item, you provide a pointer to the item along with a C string describing the item's type in Objective-C type encoding. You get this string using the `@encode()` compiler directive, which returns the platform-specific encoding for the given type (see Type Encodings for more information about `@encode()` and a list of type codes). For example, this code excerpt creates `theValue` containing an `NSRange`:

```
NSRange myRange = {4, 10};
NSValue *theValue = [NSValue valueWithBytes:&myRange objCType:@encode(NSRange)];
```

The following example illustrates encoding a custom C structure.

```
// assume ImaginaryNumber defined:
typedef struct {
    float real;
    float imaginary;
} ImaginaryNumber;


ImaginaryNumber miNumber;
miNumber.real = 1.1;
miNumber.imaginary = 1.41;

NSValue *miValue = [NSValue valueWithBytes: &miNumber
                                withObjCType:@encode(ImaginaryNumber)];


ImaginaryNumber miNumber2;
```

```
[miValue getValue:&miNumber2];
```

The type you specify must be of constant length. You cannot store C strings, variable-length arrays and structures, and other data types of indeterminate length in an `NSValue`—you should use `NSString` or `NSData` objects for these types. You can store a pointer to variable-length item in an `NSValue` object. The following code excerpt incorrectly attempts to place a C string directly into an `NSValue` object:

```
/* INCORRECT! */
char *myCString = "This is a string.";
NSValue *theValue = [NSValue valueWithBytes:myCString withObjCType:@encode(char
*)];
```

In this code excerpt the contents of `myCString` are interpreted as a pointer to a `char`, so the first four bytes contained in the string are treated as a pointer (the actual number of bytes used may vary with the hardware architecture). That is, the sequence "This" is interpreted as a pointer value, which is unlikely to be a legal address. The correct way to store such a data item is to use an `NSString` object (if you need to contain the characters in an object), or to pass the address of its pointer, not the pointer itself:

```
/* Correct. */
char *myCString = "This is a string.";
NSValue *theValue = [NSValue valueWithBytes:&myCString withObjCType:@encode(char
**)];
```

Here the *address* of `myCString` is passed (`&myCString`), so the address of the first character of the string is stored in `theValue`.

Important: The `NSValue` object doesn't copy the contents of the string, but the pointer itself. If you create an `NSValue` object with an allocated data item, don't free the data's memory while the `NSValue` object exists.

Using Numbers

`NSNumber` is a subclass of `NSValue` that offers a value as any C scalar (numeric) type. It defines a set of methods specifically for creating number objects and accessing the value as a signed or unsigned `char`, `short int`, `int`, `NSInteger`, `long int`, `long long int`, `float`, or `double`, or as a `BOOL`.

```
NSInteger nine = 9;
float ten = 10.0;

NSNumber *nineFromInteger = [NSNumber alloc] initWithInteger:nine];
NSNumber *tenFromFloat = [NSNumber numberWithFloat:ten];
```

You can also create number object directly as literals using `@`:

```
NSNumber *nineFromInteger = @9;
NSNumber *tenFromFloat = @10.0;
NSNumber *nineteenFromExpression = @(nine + ten);
```

`NSNumber` defines a `compare:` method to determine the ordering of two `NSNumber` objects. :

```
NSComparisonResult comparison = [nineFromInteger compare:tenFromFloat];
// comparison = NSOrderedAscending

float aFloat = [nineFromInteger floatValue];
// aFloat = 9.0
BOOL ok = [tenFromFloat boolValue];
// ok = YES
```

An `NSNumber` object records the numeric type with which it is created, and uses the C rules for numeric conversion when comparing `NSNumber` objects of different numeric types and when returning values as C numeric types. See any standard C reference for information on type conversion. (If you ask a number for its `objCType`, however, the returned type does not necessarily match the method the receiver was created with.)

If you ask an `NSNumber` object for its value using a type that cannot hold the value, you get back an erroneous result—for example, if you ask for the `float` value of a number created with a `double` that is greater than `FLT_MAX`, or the `integer` value of a number created with a `float` that is greater than the maximum value of `NSInteger`.

```
NSNumber *bigNumber = @(FLT_MAX);
NSInteger badInteger = [bigNumber integerValue];
NSLog(@"bigNumber: %@; badInteger: %d", bigNumber, badInteger);
// output: "bigNumber: 3.402823e+38; badInteger: 0"
```

Using Decimal Numbers

`NSDecimalNumber` is an immutable subclass of `NSNumber` that provides an object-oriented wrapper for doing base-10 arithmetic. An instance can represent any number that can be expressed as `mantissa x 10 exponent` where *mantissa* is a decimal integer up to 38 digits long, and *exponent* is an integer between -128 and 127.

In the course of doing arithmetic, a method may produce calculation errors, such as division by zero. It may also meet circumstances where it has a choice of ways to round a number off. The way the method acts on such occasions is called its "behavior."

Behavior is set by methods in the `NSDecimalNumberBehaviors` protocol. Every `NSDecimalNumber` argument called `behavior` requires an object that conforms to this protocol. For more on behaviors, see the specifications for the `NSDecimalNumberBehaviors` protocol and the `NSDecimalNumberHandler` class. Also see the `defaultBehavior` method description.

C Interface to Decimal Numbers

You can access the arithmetic and rounding methods of `NSDecimalNumber` through group of C functions:

`NSDecimalAdd`	Adds two decimal values.
`NSDecimalCompact`	Compacts the decimal structure for efficiency.
`NSDecimalCompare`	Compares two decimal values.
`NSDecimalCopy`	Copies the value of a decimal number.
`NSDecimalDivide`	Divides one decimal value by another.
`NSDecimalIsNotANumber`	Returns a Boolean that indicates whether a given decimal contains a valid number.
`NSDecimalMultiply`	Multiplies two decimal numbers together.
`NSDecimalMultiplyByPowerOf10`	Multiplies a decimal by the specified power of 10.
`NSDecimalNormalize`	Normalizes the internal format of two decimal numbers to simplify later operations.

`NSDecimalPower`	Raises the decimal value to the specified power.
`NSDecimalRound`	Rounds off the decimal value.
`NSDecimalString`	Returns a string representation of the decimal value.
`NSDecimalSubtract`	Subtracts one decimal value from another.

You might consider the C interface if you don't need to treat decimal numbers as objects—that is, if you don't need to store them in an object-oriented collection like an instance of `NSArray` or `NSDictionary`. You might also consider the C interface if you need maximum efficiency. The C interface is faster and uses less memory than the `NSDecimalNumber` class.

If you need mutability, you can combine the two interfaces. Use functions from the C interface and convert their results to instances of `NSDecimalNumber`.

Using NSNull

The `NSNull` class defines a singleton object you use to represent null values in situations where `nil` is prohibited as a value (typically in a collection object such as an array or a dictionary).

```
NSNull *nullValue = [NSNull null];
NSArray *arrayWithNull = @[nullValue];
NSLog(@"arrayWithNull: %@", arrayWithNull);
// Output: "arrayWithNull: (<null>)"
```

It is important to appreciate that the `NSNull` instance is semantically different from `NO` or `false`—these both represent a logical value; the `NSNull` instance represents the absence of a value. The `NSNull` instance is semantically equivalent to `nil`, however it is also important to appreciate that it is not equal to `nil`. To test for a null object value, you must therefore make a direct object comparison.

```
id aValue = [arrayWithNull objectAtIndex:0];
if (aValue == nil) {
    NSLog(@"equals nil");
}
else if (aValue == [NSNull null]) {
    NSLog(@"equals NSNull instance");
    if ([aValue isEqual:nil]) {
        NSLog(@"isEqual:nil");
    }
}
// Output: "equals NSNull instance"
```

Document Revision History

This table describes the changes to *Number and Value Programming Topics*.

Date	Notes
2008-02-08	Updated for OS X v10.5.
2007-10-31	Corrected a typographical error.
2007-01-08	Added discussion of NSNumber's out-of-range behaviors; added article describing use of NSNull.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.

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