‹ Back to *Objective-C Data Types*

NSNumber

The `NSNumber` class is a lightweight, object-oriented wrapper around C's numeric primitives. It's main job is to store and retrieve primitive values, and it comes with dedicated methods for each data type:

```
NSNumber *aBool = [NSNumber numberWithBool:NO];
NSNumber *aChar = [NSNumber numberWithChar:'z'];
NSNumber *aUChar = [NSNumber numberWithUnsignedChar:255];
NSNumber *aShort = [NSNumber numberWithShort:32767];
NSNumber *aUShort = [NSNumber numberWithUnsignedShort:65535];
NSNumber *anInt = [NSNumber numberWithInt:2147483647];
NSNumber *aUInt = [NSNumber numberWithUnsignedInt:4294967295];
NSNumber *aLong = [NSNumber numberWithLong:9223372036854775807];
NSNumber *aULong = [NSNumber numberWithUnsignedLong:18446744073709551615];
NSNumber *aFloat = [NSNumber numberWithFloat:26.99f];
NSNumber *aDouble = [NSNumber numberWithDouble:26.99];

NSLog(@"%@", [aBool boolValue] ? @"YES" : @"NO");
NSLog(@"%c", [aChar charValue]);
NSLog(@"%hhu", [aUChar unsignedCharValue]);
NSLog(@"%hi", [aShort shortValue]);
NSLog(@"%hu", [aUShort unsignedShortValue]);
NSLog(@"%i", [anInt intValue]);
NSLog(@"%u", [aUInt unsignedIntValue]);
NSLog(@"%li", [aLong longValue]);
NSLog(@"%lu", [aULong unsignedLongValue]);
NSLog(@"%f", [aFloat floatValue]);
NSLog(@"%f", [aDouble doubleValue]);
```

It may seem redundant to have an object-oriented version of all the C primitives, but it's necessary if you want to store numeric values in an `NSArray`, `NSDictionary`, or any of the other Foundation Framework collections. These classes require all of their elements to be objects—they do not know how to interact with primitive values. In addition, `NSNumber` makes it possible to pass numbers to methods like `performSelector:withObject:`, as discussed in Selectors.

But, aside from the object-oriented interface, there are a few perks to using `NSNumber`. For one, it provides a straightforward way to convert between primitive data types or get an `NSString` representation of the value:

```
NSNumber *anInt = [NSNumber numberWithInt:42];
float asFloat = [anInt floatValue];
NSLog(@"%.2f", asFloat);
NSString *asString = [anInt stringValue];
NSLog(@"%@", asString);
```

And, like all Objective-C objects, `NSNumber` can be displayed with the `%@` format specifier, which means that you can forget about all of the C-style specifiers introduced in the Primitives module. This makes life a tiny bit easier when trying to debug values:

```
NSNumber *aUChar = [NSNumber numberWithUnsignedChar:255];
NSNumber *anInt = [NSNumber numberWithInt:2147483647];
NSNumber *aFloat = [NSNumber numberWithFloat:26.99f];
NSLog(@"%@", aUChar);
NSLog(@"%@", anInt);
NSLog(@"%@", aFloat);
```

Numeric Literals

Xcode 4.4 introduced numeric literals, which offer a much more convenient alternative to the above factory methods. The `NSNumber` version of `BOOL`'s, `char`'s, `int`'s and `double`'s can all be created by simply prefixing the corresponding primitive type with the `@` symbol; however, `unsigned int`'s, `long`'s, and `float`'s must be appended with the `U`, `L`, or `F` modifiers, as shown below.

```
NSNumber *aBool = @NO;
NSNumber *aChar = @'z';
NSNumber *anInt = @2147483647;
NSNumber *aUInt = @4294967295U;
NSNumber *aLong = @9223372036854775807L;
NSNumber *aFloat = @26.99F;
NSNumber *aDouble = @26.99;
```

In addition to literal values, it's possible to box arbitrary C expressions using the `@()` syntax. This makes it trivial to turn basic arithmetic calculations into `NSNumber` objects:

```
double x = 24.0;
NSNumber *result = @(x * .15);
NSLog(@"%.2f", [result doubleValue]);
```

Immutability

It's important to understand that `NSNumber` objects are immutable—it's not possible to change its associated value after you create it. In this sense, an `NSNumber` instance acts exactly like a primitive value: When you need a new `double` value, you create a new literal—you don't change an existing one.

From a practical standpoint, this means you need to create a new `NSNumber` object every time you change its value. For example, the following loop increments a `counter` object by adding to its primitive value, then re-boxing it into a new `NSNumber` and assigning it to the same variable.

```
NSNumber *counter = @0;
for (int i=0; i<10; i++) {
    counter = @([counter intValue] + 1);
    NSLog(@"%@", counter);
}
```

As you could probably imagine, this isn't the most efficient way to work with numbers. In real applications, you should limit yourself to primitive numeric types for computationally intensive algorithms and wait as long as possible to store the result in an `NSNumber` container.

Comparing Numbers

While it's possible to directly compare `NSNumber` pointers, the `isEqualToNumber:` method is a much more robust way to check for equality. It guarantees that two *values* will compare equal, even if they are stored in different *objects*. For example, the following snippet shows a common case of pointer comparison failure.

```
NSNumber *anInt = @27;
NSNumber *sameInt = @27U;
// Pointer comparison (fails)
if (anInt == sameInt) {
    NSLog(@"They are the same object");
}
// Value comparison (succeeds)
if ([anInt isEqualToNumber:sameInt]) {
    NSLog(@"They are the same value");
}
```

If you need to check for inequalities, you can use the related `compare:` method. Instead of a Boolean value, it returns an `NSComparisonResult`, which is an `enum` that defines the relationship between the operands:

Return Value	Description
`NSOrderedAscending`	receiver < argument
`NSOrderedSame`	receiver == argument
`NSOrderedDescending`	receiver > argument

The following example shows these enumerators in action.

```
NSNumber *anInt = @27;
NSNumber *anotherInt = @42;
NSComparisonResult result = [anInt compare:anotherInt];
if (result == NSOrderedAscending) {
    NSLog(@"27 < 42");
} else if (result == NSOrderedSame) {
    NSLog(@"27 == 42");
} else if (result == NSOrderedDescending) {
    NSLog(@"27 > 42");
}
```

This kind of object-oriented comparison may not be as convenient as the familiar <, ==, and > operators, but abstracting them into methods affords a lot more flexibility to the Foundation Framework classes.

Continue to *NSDecimalNumber* ›