‹ Back to *Objective-C Data Types*

NSDecimalNumber

The `NSDecimalNumber` class provides fixed-point arithmetic capabilities to Objective-C programs. They're designed to perform base-10 calculations without loss of precision and with predictable rounding behavior. This makes it a better choice for representing currency than floating-point data types like `double`. However, the trade-off is that they are more complicated to work with.

Internally, a fixed-point number is expressed as `sign mantissa x 10^exponent`. The sign defines whether it's positive or negative, the mantissa is an unsigned integer representing the significant digits, and the exponent determines where the decimal point falls in the mantissa.

It's possible to manually assemble an `NSDecimalNumber` from a mantissa, exponent, and sign, but it's often easier to convert it from a string representation. The following snippet creates the value `15.99` using both methods.

```
NSDecimalNumber *price;
price = [NSDecimalNumber decimalNumberWithMantissa:1599
                                           exponent:-2
                                           isNegative:NO];
price = [NSDecimalNumber decimalNumberWithString:@"15.99"];
```

Like `NSNumber`, all `NSDecimalNumber` objects are immutable, which means you cannot

change their value after they've been created.

Arithmetic

The main job of `NSDecimalNumber` is to provide fixed-point alternatives to C's native arithmetic operations. All five of `NSDecimalNumber`'s arithmetic methods are demonstrated below.

```
NSDecimalNumber *price1 = [NSDecimalNumber decimalNumberWithString:@"15.99"];
NSDecimalNumber *price2 = [NSDecimalNumber decimalNumberWithString:@"29.99"];
NSDecimalNumber *coupon = [NSDecimalNumber decimalNumberWithString:@"5.00"];
NSDecimalNumber *discount = [NSDecimalNumber decimalNumberWithString:@".90"];
NSDecimalNumber *numProducts = [NSDecimalNumber decimalNumberWithString:@"2.0"];

NSDecimalNumber *subtotal = [price1 decimalNumberByAdding:price2];
NSDecimalNumber *afterCoupon = [subtotal decimalNumberBySubtracting:coupon];
NSDecimalNumber *afterDiscount = [afterCoupon decimalNumberByMultiplyingBy:discount];
NSDecimalNumber *average = [afterDiscount decimalNumberByDividingBy:numProducts];
NSDecimalNumber *averageSquared = [average decimalNumberByRaisingToPower:2];

NSLog(@"Subtotal: %@", subtotal);                    // 45.98
NSLog(@"After coupon: %@", afterCoupon);            // 40.98
NSLog((@"After discount: %@"), afterDiscount);       // 36.882
NSLog(@"Average price per product: %@", average);    // 18.441
NSLog(@"Average price squared: %@", averageSquared); // 340.070481
```

Unlike their floating-point counterparts, these operations are guaranteed to be accurate. However, you'll notice that many of the above calculations result in extra decimal places. Depending on the application, this may or may not be desirable (e.g., you might want to constrain currency values to 2 decimal places). This is where custom rounding behavior comes in.

Rounding Behavior

Each of the above arithmetic methods have an alternate `withBehavior:` form that let you define how the operation rounds the resulting value. The `NSDecimalNumberHandler` class encapsulates a particular rounding behavior and can

be instantiated as follows:

```
NSDecimalNumberHandler *roundUp = [NSDecimalNumberHandler
                                   decimalNumberHandlerWithRoundingMode:NSRoundUp
                                   scale:2
                                   raiseOnExactness:NO
                                   raiseOnOverflow:NO
                                   raiseOnUnderflow:NO
                                   raiseOnDivideByZero:YES];
```

The `NSRoundUp` argument makes all operations round up to the nearest place. Other rounding options are `NSRoundPlain`, `NSRoundDown`, and `NSRoundBankers`, all of which are defined by `NSRoundingMode`. The `scale:` parameter defines the number of decimal places the resulting value should have, and the rest of the parameters define the exception-handling behavior of any operations. In this case, `NSDecimalNumber` will only raise an exception if you try to divide by zero.

This rounding behavior can then be passed to the `decimalNumberByMultiplyingBy:withBehavior:` method (or any of the other arithmetic methods), as shown below.

```
NSDecimalNumber *subtotal = [NSDecimalNumber decimalNumberWithString:@"40.98"];
NSDecimalNumber *discount = [NSDecimalNumber decimalNumberWithString:@".90"];

NSDecimalNumber *total = [subtotal decimalNumberByMultiplyingBy:discount
                                                   withBehavior:roundUp];
NSLog(@"Rounded total: %@", total);
```

Now, instead of `36.882`, the `total` gets rounded up to two decimal points, resulting in `36.89`.

Comparing NSDecimalNumbers

Like `NSNumber`, `NSDecimalNumber` objects should use the `compare:` method instead of the native inequality operators. Again, this ensures that *values* are compared, even if they are stored in different *instances*. For example:

```
NSDecimalNumber *discount1 = [NSDecimalNumber decimalNumberWithString:@".85"];
NSDecimalNumber *discount2 = [NSDecimalNumber decimalNumberWithString:@".9"];
NSComparisonResult result = [discount1 compare:discount2];
if (result == NSOrderedAscending) {
    NSLog(@"85%% < 90%%");
} else if (result == NSOrderedSame) {
    NSLog(@"85%% == 90%%");
} else if (result == NSOrderedDescending) {
    NSLog(@"85%% > 90%%");
}
```

`NSDecimalNumber` also inherits the `isEqualToNumber:` method from `NSNumber`.

Decimal Numbers in C

For most practical purposes, the `NSDecimalNumber` class should satisfy your fixed-point needs; however, it's worth noting that there is also a function-based alternative available in pure C. This provides increased efficiency over the OOP interface discussed above and is thus preferred for high-performance applications dealing with a large number of calculations.

NSDecimal

Instead of an `NSDecimalNumber` object, the C interface is built around the `NSDecimal` `struct`. Unfortunately, the Foundation Framework doesn't make it easy to create an `NSDecimal` from scratch. You need to generate one from a full-fledged `NSDecimalNumber` using its `decimalValue` method. There is a corresponding factory method, also shown below.

```
NSDecimalNumber *price = [NSDecimalNumber decimalNumberWithString:@"15.99"];
NSDecimal asStruct = [price decimalValue];
NSDecimalNumber *asNewObject = [NSDecimalNumber decimalNumberWithDecimal:asStruct];
```

This isn't exactly an ideal way to create `NSDecimal`'s, but once you have a `struct` representation of your initial values, you can stick to the functional API presented

below. All of these functions use `struct`'s as inputs and outputs.

Arithmetic Functions

In lieu of the arithmetic methods of `NSDecimalNumber`, the C interface uses functions like `NSDecimalAdd()`, `NSDecimalSubtract()`, etc. Instead of returning the result, these functions populate the first argument with the calculated value. This makes it possible to reuse an existing `NSDecimal` in several operations and avoid allocating unnecessary structs just to hold intermediary values.

For example, the following snippet uses a single `result` variable across 5 function calls. Compare this to the Arithmetic section, which created a new `NSDecimalNumber` object for each calculation.

```
NSDecimal price1 = [[NSDecimalNumber decimalNumberWithString:@"15.99"] decimalValue];
NSDecimal price2 = [[NSDecimalNumber decimalNumberWithString:@"29.99"] decimalValue];
NSDecimal coupon = [[NSDecimalNumber decimalNumberWithString:@"5.00"] decimalValue];
NSDecimal discount = [[NSDecimalNumber decimalNumberWithString:@".90"] decimalValue];
NSDecimal numProducts = [[NSDecimalNumber decimalNumberWithString:@"2.0"] decimalValu
NSLocale *locale = [NSLocale currentLocale];
NSDecimal result;

NSDecimalAdd(&result, &price1, &price2, NSRoundUp);
NSLog(@"Subtotal: %@", NSDecimalString(&result, locale));
NSDecimalSubtract(&result, &result, &coupon, NSRoundUp);
NSLog(@"After coupon: %@", NSDecimalString(&result, locale));
NSDecimalMultiply(&result, &result, &discount, NSRoundUp);
NSLog(@"After discount: %@", NSDecimalString(&result, locale));
NSDecimalDivide(&result, &result, &numProducts, NSRoundUp);
NSLog(@"Average price per product: %@", NSDecimalString(&result, locale));
NSDecimalPower(&result, &result, 2, NSRoundUp);
NSLog(@"Average price squared: %@", NSDecimalString(&result, locale));
```

Notice that these functions accept *references* to `NSDecimal` structs, which is why we need to use the reference operator (`&`) instead of passing them directly. Also note that rounding is an inherent part of each operation—it's not encapsulated in a separate entity like `NSDecimalNumberHandler`.

The `NSLocale` instance defines the formatting of `NSDecimalString()`, and is discussed more thoroughly in the Dates module.

Error Checking

Unlike their OOP counterparts, the arithmetic functions don't raise exceptions when a calculation error occurs. Instead, they follow the common C pattern of using the return value to indicate success or failure. All of the above functions return an `NSCalculationError`, which defines what kind of error occurred. The potential scenarios are demonstrated below.

```
NSDecimal a = [[NSDecimalNumber decimalNumberWithString:@"1.0"] decimalValue];
NSDecimal b = [[NSDecimalNumber decimalNumberWithString:@"0.0"] decimalValue];
NSDecimal result;
NSCalculationError success = NSDecimalDivide(&result, &a, &b, NSRoundPlain);
switch (success) {
    case NSCalculationNoError:
        NSLog(@"Operation successful");
        break;
    case NSCalculationLossOfPrecision:
        NSLog(@"Error: Operation resulted in loss of precision");
        break;
    case NSCalculationUnderflow:
        NSLog(@"Error: Operation resulted in underflow");
        break;
    case NSCalculationOverflow:
        NSLog(@"Error: Operation resulted in overflow");
        break;
    case NSCalculationDivideByZero:
        NSLog(@"Error: Tried to divide by zero");
        break;
    default:
        break;
}
```

Comparing NSDecimals

Comparing `NSDecimal`'s works exactly like the OOP interface, except you use the `NSDecimalCompare()` function:

```
NSDecimal discount1 = [[NSDecimalNumber decimalNumberWithString:@".85"] decimalValue];
NSDecimal discount2 = [[NSDecimalNumber decimalNumberWithString:@".9"] decimalValue];
NSComparisonResult result = NSDecimalCompare(&discount1, &discount2);
if (result == NSOrderedAscending) {
    NSLog(@"85%% < 90%%");
} else if (result == NSOrderedSame) {
    NSLog(@"85%% == 90%%");
} else if (result == NSOrderedDescending) {
    NSLog(@"85%% > 90%%");
}
```

Continue to *NSString* ›