‹ Back to *Objective-C Data Types*

NSString

As we've already seen several times throughout this tutorial, the NSString class is the basic tool for representing text in an Objective-C application. Aside from providing an object-oriented wrapper for strings, NSString provides many powerful methods for searching and manipulating its contents. It also comes with native Unicode support.

Like NSNumber and NSDecimalNumber, NSString is an immutable type, so you cannot change it after it's been instantiated. It does, however, have a mutable counterpart called NSMutableString, which will be discussed at the end of this module.

Creating Strings

The most common way to create strings is using the literal @"Some String" syntax, but the stringWithFormat: class method is also useful for generating strings that are composed of variable values. It takes the same kind of format string as NSLog():

```
NSString *make = @"Porsche";
NSString *model = @"911";
int year = 1968;
NSString *message = [NSString stringWithFormat:@"That's a %@ %@ from %d!",
                     make, model, year];
NSLog(@"%@", message);
```

Notice that we used the @"%@" format specifier in the NSLog() call instead of passing the string directly with NSLog(message). Using a literal for the first argument of NSLog() is a best practice, as it sidesteps potential bugs when the string you want to display contains % signs. Think about what would happen when message = @"The tank is 50% full".

NSString provides built-in support for Unicode, which means that you can include UTF-8 characters directly in string literals. For example, you can paste the following into Xcode and use it the same way as any other NSString object.

```
NSString *make = @"Côte d'Ivoire";
```

Enumerating Strings

The two most basic NSString methods are length and characterAtIndex:, which return the number of characters in the string and the character at a given index, respectively. You probably won't have to use these methods unless you're doing low-level string manipulation, but they're still good to know:

```
NSString *make = @"Porsche";
for (int i=0; i<[make length]; i++) {
    unichar letter = [make characterAtIndex:i];
    NSLog(@"%d: %hu", i, letter);
}
```

As you can see, characterAtIndex: has a return type of unichar, which is a typedef for unsigned short. This value represents the Unicode decimal number for the character.

Comparing Strings

String comparisons present the same issues as NSNumber comparisons. Instead of comparing *pointers* with the == operator, you should always use the isEqualToString: method for a more robust *value* comparison. The following example shows you how this works, along with the useful hasPrefix: and hasSuffix: methods for partial comparisons.

```
NSString *car = @"Porsche Boxster";
if ([car isEqualToString:@"Porsche Boxster"]) {
    NSLog(@"That car is a Porsche Boxster");
}
if ([car hasPrefix:@"Porsche"]) {
    NSLog(@"That car is a Porsche of some sort");
}
if ([car hasSuffix:@"Carrera"]) {
    // This won't execute
    NSLog(@"That car is a Carrera");
}
```

And, just like `NSNumber`, `NSString` has a `compare:` method that can be useful for alphabetically sorting strings:

```
NSString *otherCar = @"Ferrari";
NSComparisonResult result = [car compare:otherCar];
if (result == NSOrderedAscending) {
    NSLog(@"The letter 'P' comes before 'F'");
} else if (result == NSOrderedSame) {
    NSLog(@"We're comparing the same string");
} else if (result == NSOrderedDescending) {
    NSLog(@"The letter 'P' comes after 'F'");
}
```

Note that this is a case-sensitive comparison, so uppercase letters will always be *before* their lowercase counterparts. If you want to ignore case, you can use the related `caseInsensitiveCompare:` method.

Combining Strings

The two methods presented below are a way to concatenate `NSString` objects. But, remember that `NSString` is an immutable type, so these methods actually return a *new* string and leave the original arguments unchanged.

```
NSString *make = @"Ferrari";
NSString *model = @"458 Spider";
NSString *car = [make stringByAppendingString:model];
NSLog(@"%@", car);         // Ferrari458 Spider
car = [make stringByAppendingFormat:@" %@", model];
```

```
NSLog(@"%@", car);        // Ferrari 458 Spider (note the space)
```

Searching Strings

NSString's search methods all return an NSRange struct, which defines a location and a length field. The location is the index of the beginning of the match, and the length is the number of characters in the match. If no match was found, location will contain NSNotFound. For example, the following snippet searches for the Cabrio substring.

```
NSString *car = @"Maserati GranCabrio";
NSRange searchResult = [car rangeOfString:@"Cabrio"];
if (searchResult.location == NSNotFound) {
    NSLog(@"Search string was not found");
} else {
    NSLog(@"'Cabrio' starts at index %lu and is %lu characters long",
          searchResult.location,        // 13
          searchResult.length);         // 6
}
```

The next section shows you how to create NSRange structs from scratch.

Subdividing Strings

You can divide an existing string by specifying the first/last index of the desired substring. Again, since NSString is immutable, the following methods return a new object, leaving the original intact.

```
NSString *car = @"Maserati GranTurismo";
NSLog(@"%@", [car substringToIndex:8]);               // Maserati
NSLog(@"%@", [car substringFromIndex:9]);             // GranTurismo
NSRange range = NSMakeRange(9, 4);
NSLog(@"%@", [car substringWithRange:range]);         // Gran
```

The global NSMakeRange() method creates an NSRange struct. The first argument specifies the location field, and the second defines the length field. The `substringWithRange:` method interprets these as the first index of the substring and the number of characters to include, respectively.

It's also possible to split a string into an `NSArray` using the `componentsSeparatedByString:` method, as shown below.

```
NSString *models = @"Porsche,Ferrari,Maserati";
NSArray *modelsAsArray = [models componentsSeparatedByString:@","];
NSLog(@"%@", [modelsAsArray objectAtIndex:1]);        // Ferrari
```

Replacing Substrings

Replacing part of a string is just like subdividing a string, except you provide a replacement along with the substring you're looking for. The following snippet demonstrates the two most common substring replacement methods.

```
NSString *elise = @"Lotus Elise";
NSRange range = NSMakeRange(6, 5);
NSString *exige = [elise stringByReplacingCharactersInRange:range
                                                 withString:@"Exige"];
NSLog(@"%@", exige);          // Lotus Exige
NSString *evora = [exige stringByReplacingOccurrencesOfString:@"Exige"
                                                   withString:@"Evora"];
NSLog(@"%@", evora);          // Lotus Evora
```

Changing Case

The `NSString` class also provides a few convenient methods for changing the case of a string. This can be used to normalize user-submitted values. As with all `NSString` manipulation methods, these return new strings instead of changing the existing instance.

```
NSString *car = @"lotUs beSpoKE";
NSLog(@"%@", [car lowercaseString]);      // lotus bespoke
NSLog(@"%@", [car uppercaseString]);      // LOTUS BESPOKE
NSLog(@"%@", [car capitalizedString]);    // Lotus Bespoke
```

Numerical Conversions

NSString defines several conversion methods for interpreting strings as primitive values. These are occasionally useful for very simple string processing, but you should really consider NSScanner or NSNumberFormatter if you need a robust string-to-number conversion tool.

```
NSString *year = @"2012";
BOOL asBool = [year boolValue];
int asInt = [year intValue];
NSInteger asInteger = [year integerValue];
long long asLongLong = [year longLongValue];
float asFloat = [year floatValue];
double asDouble = [year doubleValue];
```

NSMutableString

The NSMutableString class is a mutable version of NSString. Unlike immutable strings, it's possible to alter individual characters of a mutable string without creating a brand new object. This makes NSMutableString the preferred data structure when you're performing several small edits on the same string.

NSMutableString inherits from NSString, so aside from the ability to manipulate it in place, you can use a mutable string just like you would an immutable string. That is to say, the API discussed above will still work with an NSMutableString instance, although methods like stringByAppendingString: will still return a NSString object—not an NSMutableString.

The remaining sections present several methods defined by the NSMutableString class. You'll notice that the fundamental workflow for mutable strings is different than that of immutable ones. Instead of creating a new object and replacing the old value, NSMutableString methods operate directly on the existing instance.

Creating Mutable Strings

Mutable strings can be created through the `stringWithString:` class method, which turns a literal string or an existing `NSString` object into a mutable one:

```
NSMutableString *car = [NSMutableString stringWithString:@"Porsche 911"];
```

After you've created a mutable string, the `setString:` method lets you assign a new value to the instance:

```
[car setString:@"Porsche Boxster"];
```

Compare this to `NSString`, where you re-assign a new value to the variable. With mutable strings, we don't change the instance reference, but rather manipulate its contents through the mutable API.

Expanding Mutable Strings

`NSMutableString` provides mutable alternatives to many of the `NSString` manipulation methods discussed above. Again, the mutable versions don't need to copy the resulting string into a new memory location and return a new reference to it. Instead, they directly change the existing object's underlying value.

```
NSMutableString *car = [NSMutableString stringWithCapacity:20];
NSString *model = @"458 Spider";

[car setString:@"Ferrari"];
[car appendString:model];
NSLog(@"%@", car);                    // Ferrari458 Spider

[car setString:@"Ferrari"];
[car appendFormat:@" %@", model];
NSLog(@"%@", car);                    // Ferrari 458 Spider

[car setString:@"Ferrari Spider"];
[car insertString:@"458 " atIndex:8];
```

```
NSLog(@"%@", car);                  // Ferrari 458 Spider
```

Also note that, like any well-designed Objective-C class, the method names of `NSString` and `NSMutableString` reflect exactly what they do. The former creates and returns a brand new string, so it uses names like `stringByAppendingString:`. On the other hand, the latter operates on the object itself, so it uses verbs like `appendString:`.

Replacing/Deleting Substrings

It's possible to replace or delete substrings via the `replaceCharactersInRange:withString:` and `deleteCharactersInRange:` methods, as shown below.

```
NSMutableString *car = [NSMutableString stringWithCapacity:20];
[car setString:@"Lotus Elise"];
[car replaceCharactersInRange:NSMakeRange(6, 5)
                   withString:@"Exige"];
NSLog(@"%@", car);                              // Lotus Exige
[car deleteCharactersInRange:NSMakeRange(5, 6)];
NSLog(@"%@", car);                              // Lotus
```

When To Use Mutable Strings

Since `NSString` and `NSMutableString` provide such similar functionality, it can be hard to know when to use one over the other. In general, the static nature of `NSString` makes it more efficient for most tasks; however, the fact that an immutable string can't be changed without generating a new object makes it less than ideal when you're trying to perform several small edits.

The two examples presented in this section demonstrate the advantages of mutable strings. First, let's take a look at an anti-pattern for immutable strings. The following loop generates a string containing all of the numbers between 0 and 999 using `NSString`.

```
// DO NOT DO THIS. EVER.
```

```
NSString *indices = @"";
for (int i=0; i<1000; i++) {
    indices = [indices stringByAppendingFormat:@"%d", i];
}
```

Remember that `stringByAppendingFormat:` creates a new `NSString` instance, which means that in each iteration, the entire string gets copied to a new block of memory. The above code allocates 999 string objects that serve only as intermediary values, resulting in an application that requires a whopping 1.76 MB of memory. Needless to say, this is incredibly inefficient.

Now, let's take a look at the mutable version of this snippet:

```
NSMutableString *indices = [NSMutableString stringWithCapacity:1];
for (int i=0; i<1000; i++) {
    [indices appendFormat:@"%d", i];
}
```

Since mutable strings manipulate their contents in place, no copying is involved, and we completely avoid the 999 unnecessary allocations. Internally, the mutable string's storage dynamically expands to accommodate longer values. This reduces the memory footprint to around 19 KB, which is much more reasonable.

So, a good rule of thumb is to use a mutable string whenever you're running any kind of algorithm that edits or assembles a string in several passes and to use an immutable string for everything else. This also applies to sets, arrays, and dictionaries.

Continue to *NSSet* ›