Predicate Programming Guide

Developer

Contents

Introduction

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.

Predicates provide a general means of specifying queries in Cocoa. The predicate system is capable of handling a large number of domains, including Core Data and Spotlight. This document describes predicates in general, their use, their syntax, and their limitations.

At a Glance

In Cocoa, a *predicate* is a logical statement that evaluates to a Boolean value (true or false). There are two types of predicate, known as *comparison* and *compound*:

- A *comparison predicate* compares two *expressions* using an *operator*. The expressions are referred to as the left hand side and the right hand side of the predicate (with the operator in the middle). A comparison predicate returns the result of invoking the operator with the results of evaluating the expressions.
- A *compound predicate* compares the results of evaluating two or more other predicates, or negates another predicate.

Cocoa supports a wide range of types of predicate, including the following:

- Simple comparisons, such as `grade == 7` or `firstName like 'Mark'`
- Case or diacritic insensitive lookups, such as `name contains[cd] 'citroen'`
- Logical operations, such as `(firstName beginswith 'M') AND (lastName like 'Adderley')`

You can also create predicates for relationships—such as `group.name matches 'work.*'`, `ALL children.age > 12`, and `ANY children.age > 12`—and for operations such as `@sum.items.price < 1000`.

Cocoa predicates provide a means of encoding queries in a manner that is independent of the store used to hold the data being searched. You use predicates to represent logical conditions used for constraining the set of objects retrieved by Spotlight and Core Data, and for in-memory filtering of objects.

You can use predicates with any class of object, but a class must be key-value coding compliant for the keys you want to use in a predicate.

Predicate Classes

Cocoa provides `NSPredicate` and its two subclasses, `NSComparisonPredicate` and `NSCompoundPredicate`.

The `NSPredicate` class provides methods to evaluate a predicate and to create a predicate from a string (such as `firstName like 'Mark'`). When you create a predicate from a string, `NSPredicate` creates the appropriate predicate and expression instances for you. In some situations, you want to create comparison or compound predicates yourself, in which case you can use the `NSComparisonPredicate` and `NSCompoundPredicate` classes.

Predicate expressions in Cocoa are represented by instances of the `NSExpression` class. The simplest predicate expression represents a constant value. Frequently, though, you use expressions that retrieve the value for a key path of the object currently being evaluated in the predicate. You can also create an expression to represent the object currently being evaluated in the predicate, to serve as a placeholder for a variable, or to return the result of performing an operation on an array.

For more on creating predicates and expressions, see Creating Predicates (page 7).

Constraints and Limitations

If you use predicates with Core Data or Spotlight, take care that they work with the corresponding data store. There is no specific implementation language for predicate queries; a predicate query may be translated into SQL, XML, or another format, depending on the requirements of the backing store (if indeed there is one).

The Cocoa predicate system is intended to support a useful range of operators, so provides neither the set union nor the set intersection of all operators supported by all backing stores. Therefore, not all possible predicate queries are supported by all backing stores, and not all operations supported by all backing stores can be expressed with `NSPredicate` and `NSExpression` objects. The back end might downgrade a predicate (for example it might make a case-sensitive comparison case-insensitive) or raise an exception if you try to use an unsupported operator. For example:

- The `matches` operator uses `regex`, so is not supported by Core Data's SQL store — although it does work with in-memory filtering.

- The Core Data SQL store supports only one to-many operation per query; therefore in any predicate sent to the SQL store, there may be only one operator (and one instance of that operator) from `ALL`, `ANY`, and `IN`.
- You cannot necessarily translate arbitrary SQL queries into predicates.
- The `ANYKEY` operator can only be used with Spotlight.
- Spotlight does not support relationships.

How to Use This Document

The following articles explain the basics of predicates in Cocoa, explain how to create and use predicate objects, and define the predicate syntax:

- Creating Predicates (page 7) describes how to correctly instantiate predicates programmatically and how to retrieve them from a managed object model.
- Using Predicates (page 14) explains how to use predicates and discusses some issues related to performance.
- Comparison of NSPredicate and Spotlight Query Strings (page 22) compares NSPredicate and Spotlight queries.
- Predicate Format String Syntax (page 24) describes the syntax of the predicate format string.
- BNF Definition of Cocoa Predicates (page 31) provides a definition of Cocoa predicates in Backus-Naur Form notation.

Creating Predicates

There are three ways to create a predicate in Cocoa: using a format string, directly in code, and from a predicate template.

Creating a Predicate Using a Format String

You can use `NSPredicate` class methods of the form `predicateWithFormat...` to create a predicate directly from a string. You define the predicate as a string, optionally using variable substitution. At runtime, variable substitution—if any—is performed, and the resulting string is parsed to create corresponding predicate and expression objects. The following example creates a compound predicate with two comparison predicates.

```
NSPredicate *predicate = [NSPredicate
    predicateWithFormat:@"(lastName like[cd] %@) AND (birthday > %@)",
            lastNameSearchString, birthdaySearchDate];
```

(In the example, `like[cd]` is a modified "like" operator that is case-insensitive and diacritic-insensitive.) For a complete description of the string syntax and a list of all the operators available, see Predicate Format String Syntax (page 24).

Important: The predicate format string syntax is different from the regular expression syntax. The regular expression syntax is defined by the ICU—see http://www.icu-project.org/userguide/regexp.html.

The predicate string parser is whitespace insensitive, case insensitive with respect to keywords, and supports nested parenthetical expressions. It also supports `printf`-style format specifiers (like `%x` and `%@`)—see Formatting String Objects. Variables are denoted with a `$` (for example `$VARIABLE_NAME`)—see Creating Predicates Using Predicate Templates (page 11) for more details.

The parser does not perform any semantic type checking. It makes a best-guess effort to create suitable expressions, but—particularly in the case of substitution variables—it is possible that a runtime error will be generated.

This approach is typically best used for predefined query terms, although variable substitution allows for considerable flexibility. The disadvantage of this technique is that you must take care to avoid introducing errors into the string—you will not discover mistakes until runtime.

String Constants, Variables, and Wildcards

String constants must be quoted within the expression—single and double quotes are both acceptable, but must be paired appropriately (that is, a double quote (") does not match a single quote (')). If you use variable substitution using `%@` (such as `firstName like %@`), the quotation marks are added for you automatically. If you use string constants within your format string, you must quote them yourself, as in the following example.

```
NSPredicate *predicate = [NSPredicate
    predicateWithFormat:@"lastName like[c] \"S*\""];
```

You must take automatic quotation into account when using wildcards—you must add wildcards to a variable prior to substitution, as shown in the following example.

```
NSString *prefix = @"prefix";
NSString *suffix = @"suffix";
NSPredicate *predicate = [NSPredicate
    predicateWithFormat:@"SELF like[c] %@",
    [[prefix stringByAppendingString:@"*"] stringByAppendingString:suffix]];
BOOL ok = [predicate evaluateWithObject:@"prefixxxxxsuffix"];
```

In this example, variable substitution produces the predicate string `SELF LIKE[c] "prefix*suffix"`, and the value of `ok` is `YES`. The following fragment, by contrast, yields the predicate string `SELF LIKE[c] "prefix" * "suffix"`, and the predicate evaluation yields a runtime error:

```
predicate = [NSPredicate
    predicateWithFormat:@"SELF like[c] %@*%@", prefix, suffix];
ok = [predicate evaluateWithObject:@"prefixxxxxsuffix"];
```

Finally, the following fragment results in a runtime parse error (`Unable to parse the format string "SELF like[c] %@*"`).

```
predicate = [NSPredicate
    predicateWithFormat:@"SELF like[c] %@*", prefix];
```

You should also note the difference between variable substitution in the format string and variable expressions. The following code fragment creates a predicate with a right-hand side that is a variable expression.

```
predicate = [NSPredicate
    predicateWithFormat:@"lastName like[c] $LAST_NAME"];
```

For more about variable expressions, see Creating Predicates Using Predicate Templates (page 11).

Boolean Values

You specify and test for equality of Boolean values as illustrated in the following examples:

```
NSPredicate *newPredicate =
    [NSPredicate predicateWithFormat:@"anAttribute == %@", [NSNumber
numberWithBool:aBool]];
NSPredicate *testForTrue =
    [NSPredicate predicateWithFormat:@"anAttribute == YES"];
```

Dynamic Property Names

Because string variables are surrounded by quotation marks when they are substituted into a format string using `%@`, you *cannot* use `%@` to specify a dynamic property name—as illustrated in the following example.

```
NSString *attributeName = @"firstName";
NSString *attributeValue = @"Adam";
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"%@ like %@",
        attributeName, attributeValue];
```

The predicate format string in this case evaluates to `"firstName" like "Adam"`.

If you want to specify a dynamic property name, you use `%K` in the format string, as shown in the following fragment.

```
predicate = [NSPredicate predicateWithFormat:@"%K like %@",
        attributeName, attributeValue];
```

The predicate format string in this case evaluates to `firstName like "Adam"` (note that there are no quotation marks around `firstName`).

Creating Predicates Directly in Code

You can create predicate and expression instances directly in code. `NSComparisonPredicate` and `NSCompoundPredicate` provide convenience methods that allow you to easily create compound and comparison predicates respectively. `NSComparisonPredicate` provides a number of operators ranging from simple equality tests to custom functions.

The following example shows how to create a predicate to represent `(revenue >= 1000000) and (revenue < 100000000)`. Note that the same left-hand side expression is used for both comparison predicates.

```
NSExpression *lhs = [NSExpression expressionForKeyPath:@"revenue"];


NSExpression *greaterThanRhs = [NSExpression expressionForConstantValue:[NSNumber
 numberWithInt:1000000]];
NSPredicate *greaterThanPredicate = [NSComparisonPredicate
    predicateWithLeftExpression:lhs
    rightExpression:greaterThanRhs
    modifier:NSDirectPredicateModifier
    type:NSGreaterThanOrEqualToPredicateOperatorType
    options:0];


NSExpression *lessThanRhs = [NSExpression expressionForConstantValue:[NSNumber
numberWithInt:100000000]];
NSPredicate *lessThanPredicate = [NSComparisonPredicate
    predicateWithLeftExpression:lhs
    rightExpression:lessThanRhs
    modifier:NSDirectPredicateModifier
    type:NSLessThanPredicateOperatorType
    options:0];


NSCompoundPredicate *predicate = [NSCompoundPredicate andPredicateWithSubpredicates:
    @[greaterThanPredicate, lessThanPredicate]];
```

The disadvantage of this technique should be immediately apparent—you may have to write a lot of code. The advantages are that it is less prone to spelling and other typographical errors that may only be discovered at runtime and it may be faster than depending on string parsing.

This technique is most likely to be useful when the creation of the predicate is itself dynamic, such as in a predicate builder.

Creating Predicates Using Predicate Templates

Predicate templates offer a good compromise between the easy-to-use but potentially error-prone format string-based technique and the code-intensive pure coding approach. A predicate template is simply a predicate that includes a variable expression. (If you are using the Core Data framework, you can use the Xcode design tools to add predicate templates for fetch requests to your model—see Managed Object Models.) The following example uses a format string to create a predicate with a right-hand side that is a variable expression.

```
NSPredicate *predicateTemplate = [NSPredicate
    predicateWithFormat:@"lastName like[c] $LAST_NAME"];
```

This is equivalent to creating the variable expression directly as shown in the following example.

```
NSExpression *lhs = [NSExpression expressionForKeyPath:@"lastName"];

NSExpression *rhs = [NSExpression expressionForVariable:@"LAST_NAME"];

NSPredicate *predicateTemplate = [NSComparisonPredicate
    predicateWithLeftExpression:lhs
    rightExpression:rhs
    modifier:NSDirectPredicateModifier
    type:NSLikePredicateOperatorType
    options:NSCaseInsensitivePredicateOption];
```

To create a valid predicate to evaluate against an object, you use the `NSPredicate` method `predicateWithSubstitutionVariables:` to pass in a dictionary that contains the variables to be substituted. (Note that the dictionary must contain key-value pairs for all the variables specified in the predicate.)

```
NSPredicate *predicate = [predicateTemplate predicateWithSubstitutionVariables:
    [NSDictionary dictionaryWithObject:@"Turner" forKey:@"LAST_NAME"]];
```

The new predicate returned by this example is `lastName LIKE[c] "Turner"`.

Because the substitution dictionary must contain key-value pairs for all the variables specified in the predicate, if you want to match a null value, you must provide a null value in the dictionary, as illustrated in the following example.

```
NSPredicate *predicate = [NSPredicate
    predicateWithFormat:@"date = $DATE"];
predicate = [predicate predicateWithSubstitutionVariables:
    [NSDictionary dictionaryWithObject:[NSNull null] forKey:@"DATE"]];
```

The predicate formed by this example is `date == <null>`.

Format String Summary

It is important to distinguish between the different types of value in a format string. Note also that single or double quoting variables (or substitution variable strings) will cause `%@`, `%K`, or `$variable` to be interpreted as a literal in the format string and so will prevent any substitution.

`@"attributeName == %@"`
: This predicate checks whether the value of the key `attributeName` is the same as the value of the object `%@` that is supplied at runtime as an argument to `predicateWithFormat:`. Note that `%@` can be a placeholder for any object whose description is valid in the predicate, such as an instance of `NSDate`, `NSNumber`, `NSDecimalNumber`, or `NSString`.

`@"%K == %@"`
: This predicate checks whether the value of the key `%K` is the same as the value of the object `%@`. The variables are supplied at runtime as arguments to `predicateWithFormat:`.

`@"name IN $NAME_LIST"`
: This is a template for a predicate that checks whether the value of the key `name` is in the variable `$NAME_LIST` (no quotes) that is supplied at runtime using `predicateWithSubstitutionVariables:`.

`@"'name' IN $NAME_LIST"`
: This is a template for a predicate that checks whether the constant value `'name'` (note the quotes around the string) is in the variable `$NAME_LIST` that is supplied at runtime using `predicateWithSubstitutionVariables:`.

`@"$name IN $NAME_LIST"`
: This is a template for a predicate that expects values to be substituted (using `predicateWithSubstitutionVariables:`) for both `$NAME` and `$NAME_LIST`.

`@"%K == '%@'"`

This predicate checks whether the value of the key `%K` is equal to the string literal "`%@`" (note the single quotes around `%@`). The key name `%K` is supplied at runtime as an argument to `predicateWithFormat:`.

Using Predicates

This document describes in general how you use predicates, and how the use of predicates may influence the structure of your application data.

Evaluating Predicates

To evaluate a predicate, you use the `NSPredicate` method `evaluateWithObject:` and pass in the object against which the predicate will be evaluated. The method returns a Boolean value—in the following example, the result is `YES`.

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"SELF IN %@", @[@"Stig",
 @"Shaffiq", @"Chris"]];
BOOL result = [predicate evaluateWithObject:@"Shaffiq"];
```

You can use predicates with any class of object, but the class must support key-value coding for the keys you want to use in a predicate.

Using Predicates with Arrays

`NSArray` and `NSMutableArray` provide methods to filter array contents. `NSArray` provides `filteredArrayUsingPredicate:` which returns a new array containing objects in the receiver that match the specified predicate. `NSMutableArray` provides `filterUsingPredicate:` which evaluates the receiver's content against the specified predicate and leaves only objects that match.

```
NSMutableArray *array =
    [NSMutableArray arrayWithObjects:@"Nick", @"Ben", @"Adam", @"Melissa", nil];


NSPredicate *bPredicate = [NSPredicate predicateWithFormat:@"SELF beginswith[c]
'a'"];
NSArray *beginWithB = [array filteredArrayUsingPredicate:bPredicate];
// beginWithB contains { @"Adam" }.
```

```
NSPredicate *sPredicate = [NSPredicate predicateWithFormat:@"SELF contains[c]
'e'"];
[array filterUsingPredicate:sPredicate];
// array now contains { @"Nick", @"Ben", @"Melissa" }
```

If you use the Core Data framework, the array methods provide an efficient means of filtering an existing array of objects without—as a fetch does—requiring a round trip to a persistent data store.

Using Predicates with Key-Paths

Recall that you can follow relationships in a predicate using a key path. The following example illustrates the creation of a predicate to find employees that belong to a department with a given name (but see also Performance (page 20)).

```
NSString *departmentName = ... ;
NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"department.name like %@", departmentName];
```

If you use a to-many relationship, the construction of a predicate is slightly different. If you want to fetch Departments in which at least one of the employees has the first name "Matthew," for instance, you use an `ANY` operator as shown in the following example:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"ANY employees.firstName like 'Matthew'"];
```

If you want to find Departments in which at least one of the employees is paid more than a certain amount, you use an `ANY` operator as shown in the following example:

```
float salary = ... ;
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"ANY employees.salary
> %f", salary];
```

Using Null Values

A comparison predicate does not match any value with null except null (`nil`) or the `NSNull` null value (that is, (`$value == nil`) returns `YES` if `$value` is `nil`). Consider the following example.

```
NSString *firstName = @"Ben";

NSArray *array = @[ @{ @"lastName" : "Turner" }];
                    @{ @"firstName" : @"Ben", @"lastName" : @"Ballard",
                       @"birthday", [NSDate dateWithString:@"1972-03-24 10:45:32
+0600"] } ];

NSPredicate *predicate =
    [NSPredicate predicateWithFormat:@"firstName like %@", firstName];
NSArray *filteredArray = [array filteredArrayUsingPredicate:predicate];

NSLog(@"filteredArray: %@", filteredArray);
// Output:
// filteredArray ({birthday = 1972-03-24 10:45:32 +0600; \\
                      firstName = Ben; lastName = Ballard;})
```

The predicate does match the dictionary that contains a value `Ben` for the key `firstName`, but does not match the dictionary with no value for the key `firstName`. The following code fragment illustrates the same point using a date and a greater-than comparator.

```
NSDate *referenceDate = [NSDate dateWithTimeIntervalSince1970:0];

predicate = [NSPredicate predicateWithFormat:@"birthday > %@", referenceDate];
filteredArray = [array filteredArrayUsingPredicate:predicate];

NSLog(@"filteredArray: %@", filteredArray);
// Output:
// filteredArray: ({birthday = 1972-03-24 10:45:32 +0600; \\
                       firstName = Ben; lastName = Ballard;})
```

Testing for Null

If you want to match null values, you must include a specific test in addition to other comparisons, as illustrated in the following fragment.

```
predicate = [NSPredicate predicateWithFormat:@"(firstName == %@) || (firstName =
nil)", firstName];

filteredArray = [array filteredArrayUsingPredicate:predicate];

NSLog(@"filteredArray: %@", filteredArray);


// Output:

// filteredArray: ( { lastName = Turner; }, { birthday = 1972-03-23 20:45:32 -0800;
 firstName = Ben; lastName = Ballard; }
```

By implication, a test for null that matches a null value returns true. In the following code fragment, `ok` is set to `YES` for both predicate evaluations.

```
predicate = [NSPredicate predicateWithFormat:@"firstName = nil"];

BOOL ok = [predicate evaluateWithObject:[NSDictionary dictionary]];


ok = [predicate evaluateWithObject:

    [NSDictionary dictionaryWithObject:[NSNull null] forKey:@"firstName"]];
```

Using Predicates with Core Data

If you are using the Core Data framework, you can use predicates in the same way as you would if you were not using Core Data (for example, to filter arrays or with an array controller). In addition, however, you can also use predicates as constraints on a fetch request and you can store fetch request templates in the managed object model (see Managed Object Models).

Limitations: You cannot necessarily translate "arbitrary" SQL queries into predicates or fetch requests. There is no way, for example, to convert a SQL statement such as

```
SELECT t1.name, V1, V2
    FROM table1 t1 JOIN (SELECT t2.name AS V1, count(*) AS V2
        FROM table2 t2 GROUP BY t2.name as V) on t1.name = V.V1
```

into a fetch request. You must fetch the objects of interest, then either perform a calculation directly using the results, or use an array operator.

Fetch Requests

You create a predicate to match properties of the target entity (note that you can follow relationships using key paths) and associate the predicate with a fetch request. When the request is executed, an array is returned that contains the objects (if any) that match the criteria specified by the predicate. The following example illustrates the use of a predicate to find employees that earn more than a specified amount.

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Employee"
        inManagedObjectContext:managedObjectContext];
[request setEntity:entity];

NSNumber *salaryLimit = <#A number representing the limit#>;
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"salary > %@",
salaryLimit];
[request setPredicate:predicate];
NSError *error;
NSArray *array = [managedObjectContext executeFetchRequest:request error:&error];
```

Object Controllers

If you are using Cocoa bindings, you can specify a fetch predicate for an object controller (such as an instance of `NSObjectController` or `NSArrayController`). You can type a predicate directly into the predicate editor text field in the Attributes Inspector in Xcode or you can set it programmatically using `setFetchPredicate:`. The predicate is used to constrain the results returned when the controller executes

a fetch. If you are using an `NSObjectController` object, you specify a fetch that uniquely identifies the object you want to be the controller's content—for example, if the controller's entity is Department, the predicate might be `name like "Engineering"`.

Using Regular Expressions

The `MATCHES` operator uses ICU's Regular Expressions package, as illustrated in the following example:

```
NSArray *array = @[@"TATACCATGGGCCATCATCATCATCATCATCATCATCATCACAG",
                   @"CGGGATCCCTATCAAGGCACCTCTTCG", @"CATGCCATGGATACCAACGAGTCCGAAC",
                     @"CAT", @"CATCATCATGTCT", @"DOG"];

// find strings that contain a repetition of at least 3 'CAT' sequences,
// but not followed by a further 'CA'
NSPredicate *catPredicate =
    [NSPredicate predicateWithFormat:@"SELF MATCHES '.*(CAT){3,}(?!CA).*'"];

NSArray *filteredArray = [array filteredArrayUsingPredicate:catPredicate];
// filteredArray contains just 'CATCATCATGTCT'
```

According to the ICU specification, regular expression metacharacters are not valid inside a pattern set. For example, the regular expression `\d{9}[\dxX]` does *not* match valid ISBN numbers (any ten digit number, or a string with nine digits and the letter 'X') since the pattern set (`[\dxX]`) contains a metacharacter (`\d`). Instead you could write an `OR` expression, as shown in the following code sample:

```
NSArray *isbnTestArray = @[@"123456789X", @"987654321x", @"1234567890", @"12345X",
 @"1234567890X"];
NSPredicate *isbnPredicate =
    [NSPredicate predicateWithFormat:@"SELF MATCHES '\\\\d{10}|\\\\d{9}[Xx]'"];

NSArray *isbnArray = [isbnTestArray filteredArrayUsingPredicate:isbnPredicate];
// isbnArray contains (123456789X, 987654321x, 1234567890)
```

Performance Considerations

You should structure compound predicates to minimize the amount of work done. Regular expression matching in particular is an expensive operation. In a compound predicate, you should therefore perform simple tests before a regular expression; thus instead of using a predicate shown in the following example:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"( title matches .*mar[1-10] ) OR ( type = 1 )"];
```

you should write

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"( type = 1 ) OR ( title matches .*mar[1-10] )"];
```

In the second example, the regular expression is evaluated only if the first clause is false.

Using Joins

In general, joins (queries that cross relationships) are also expensive operations, and you should avoid them if you can. When testing to-one relationships, if you already have—or can easily retrieve—the relationship source object (or its object ID), it is more efficient to test for object equality than to test for a property of the source object. Instead of writing the following:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"department.name like %@", [department name]];
```

it is more efficient to write:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"department == %@", department];
```

If a predicate contains more than one expression, it is also typically more efficient to structure it to avoid joins. For example, `@"firstName beginswith[cd] 'Matt' AND (ANY directreports.paygrade <= 7)"` is likely to be more efficient than `@"(ANY directreports.paygrade <= 7) AND (firstName beginswith[cd] 'Matt')"` because the former avoids making a join unless the first test succeeds.

Structuring Your Data

In some situations, there may be tension between the representation of your data and the use of predicates. If you intend to use predicates in your application, the pattern of typical query operations may influence how you structure your data. In Core Data, although you specify entities and entity-class mapping, the levels that create the underlying structures in the persistent store are opaque. Nevertheless, you still have control over your entities and the properties they have.

In addition to tending to be expensive, joins may also restrict flexibility. It may be appropriate, therefore, to de-normalize your data. In general—assuming that queries are issued often—it may be a good trade-off to have larger objects, but for it to be easier to find the right ones (and so have fewer in memory).

Using Predicates with Cocoa Bindings

In OS X, you can set a predicate for an array controller to filter the content array. You can set the predicate in code (using `setFilterPredicate:`). You can also bind the array controller's `filterPredicate` binding to a method that returns an `NSPredicate` object. The object that implements the method may be the File's Owner or another controller object. If you change the predicate, remember that you must do so in a key-value observing compliant way (see *Key-Value Observing Programming Guide*) so that the array controller updates itself accordingly.

You can also bind the `predicate` binding of an `NSSearchField` object to the `filterPredicate` of an array controller. A search field's `predicate` binding is a multi-value binding, described in Binding Types.

Comparison of NSPredicate and Spotlight Query Strings

Both Spotlight and the `NSPredicate` class implement a query string syntax, and though they are similar, they differ in several respects. One query string can be converted into the other string form, as long as the common subset of functionality is used.

Spotlight and NSPredicate

The `NSMetadataQuery` class, which is the Cocoa interface to Spotlight, uses `NSPredicate` in its API. Apart from this, there is no relationship between Spotlight and `NSPredicate`. The Spotlight query string syntax is similar to, but different from, the `NSPredicate` query string syntax. You can convert one query string into the other string form, as long as you use syntax that both APIs understand. Spotlight's query syntax is a subset of that of `NSPredicate`. For a complete description of the Spotlight query expression syntax, see File Metadata Query Expression Syntax, and for a complete description of the `NSPredicate` string syntax, see Predicate Format String Syntax (page 24).

Spotlight requires comparison clauses to take the form `"KEY operator VALUE"` and does not accept `"VALUE operator KEY"`. Moreover, the kinds of attribute Spotlight accepts for `VALUE` are more limited than those accepted by `NSPredicate`. As a partial consequence of this limitation, you do not always have to quote literal strings in Spotlight queries. You can omit the quotes when `VALUE` is a string and no special operators need to be applied to it. You cannot do this with an `NSPredicate` query string, as the result would be ambiguous.

The syntax for denoting case- and diacritic-insensitivity for queries in Spotlight is different from the `NSPredicate` version. In Spotlight, you append markers to the end of the comparison string (for example, `"myAttribute == 'foo'cd"`). In `NSPredicate` strings, you use the `like` operator and prefix the markers within "[]"s (for example, `"myAttribute like[cd] 'foo'"`). In both cases, `'cd'` means case-insensitive and diacritic-insensitive. Spotlight puts the modifiers on the value, `NSPredicate` puts the modifiers on the operator.

You cannot use an `MDQuery` operator as the `VALUE` of an `NSPredicate` object `"KEY operator VALUE"` string. For example, you write an “is-substring-of” expression in Spotlight like this: `"myAttribute = '*foo*'"`; in `NSPredicate` strings you use the `contains` operator, like this: `"myAttribute contains 'foo'"`. Spotlight takes glob-like expressions, `NSPredicate` uses a different operator.

If you use "*" as left-hand-side key in a comparison expression, in Spotlight it means "any key in the item" and can only be used with ==. You could only use this expression in an `NSPredicate` object in conjunction with an `NSMetadataQuery` object.

Creating a Predicate Format String From a Spotlight Search in Finder

You can create a predicate format string from a search in Finder. Perform a search, save it, then select the folder where you saved it and choose Show Info—the Info panel shows the query that is used by Spotlight. Note however, that there are slight differences between the `NSPredicate` format string and the one stored in Finder. The Finder string might look like the following example.

```
(((* = "FooBar*"wcd) || (kMDItemTextContent = "FooBar*"cd))
    && (kMDItemContentType != com.apple.mail.emlx)
    && (kMDItemContentType != public.vcard))
```

Typically all you have to do to convert the Spotlight query into a predicate format string is make sure the predicate does not start with * (this is not supported by `NSMetadataQuery` when parsing a predicate). In addition, when you want to use a wildcard, you should use `LIKE`, as shown in the following example.

```
((kMDItemTextContent LIKE[cd] "FooBar")
    && (kMDItemContentType != "com.apple.mail.emlx")
    && (kMDItemContentType != "public.vcard"))
```

Predicate Format String Syntax

This article describes the syntax of the predicate string and some aspects of the predicate parser.

The parser string is different from a string expression passed to the regex engine. This article describes the parser text, not the syntax for the regex engine.

Parser Basics

The predicate string parser is whitespace insensitive, case insensitive with respect to keywords, and supports nested parenthetical expressions. The parser does not perform semantic type checking.

Variables are denoted with a dollar-sign (`$`) character (for example, `$VARIABLE_NAME`). The question mark (`?`) character is not a valid parser token.

The format string supports `printf`-style format specifiers such as `%x` (see Formatting String Objects). Two important format specifiers are `%@` and `%K`.

- `%@` is a var arg substitution for an object value—often a string, number, or date.
- `%K` is a var arg substitution for a key path.

When string variables are substituted into a string using the `%@` format specifier, they are surrounded by quotation marks. If you want to specify a dynamic property name, use `%K` in the format string, as shown in the following example.

```
NSString *attributeName  = @"firstName";
NSString *attributeValue = @"Adam";
NSPredicate *predicate   = [NSPredicate predicateWithFormat:@"%K like %@",
        attributeName, attributeValue];
```

The predicate format string in this case evaluates to `firstName like "Adam"`.

Single or double quoting variables (or substitution variable strings) cause `%@`, `%K`, or `$variable` to be interpreted as a literal in the format string and so prevent any substitution. In the following example, the predicate format string evaluates to `firstName like "%@"` (note the single quotes around `%@`).

```
NSString *attributeName = @"firstName";
NSString *attributeValue = @"Adam";
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"%K like '%@'",
        attributeName, attributeValue];
```

Important: Use a `%@` format specifier only to represent an expression. Do not use it to represent an entire predicate.

If you attempt to use a format specifier to represent an entire predicate, the system raises an exception.

Basic Comparisons

`=, ==`

The left-hand expression is equal to the right-hand expression.

`>=, =>`

The left-hand expression is greater than or equal to the right-hand expression.

`<=, =<`

The left-hand expression is less than or equal to the right-hand expression.

`>`

The left-hand expression is greater than the right-hand expression.

`<`

The left-hand expression is less than the right-hand expression.

`!=, <>`

The left-hand expression is not equal to the right-hand expression.

`BETWEEN`

The left-hand expression is between, or equal to either of, the values specified in the right-hand side.

The right-hand side is a two value array (an array is required to specify order) giving upper and lower bounds. For example, `1 BETWEEN { 0 , 33 }`, or `$INPUT BETWEEN { $LOWER, $UPPER }`.

In Objective-C, you could create a BETWEEN predicate as shown in the following example:

```
NSPredicate *betweenPredicate =
    [NSPredicate predicateWithFormat: @"attributeName BETWEEN %@", @[@1, @10]];
```

This creates a predicate that matches `( ( 1 <= attributeValue ) && ( attributeValue <= 10 ) )`, as illustrated in the following example:

```
NSPredicate *betweenPredicate =
    [NSPredicate predicateWithFormat: @"attributeName BETWEEN %@", @[@1, @10]];

NSDictionary *dictionary = @{ @"attributeName" : @5 };

BOOL between = [betweenPredicate evaluateWithObject:dictionary];
if (between) {
    NSLog(@"between");
}
```

Boolean Value Predicates

`TRUEPREDICATE`
: A predicate that always evaluates to `TRUE`.

`FALSEPREDICATE`
: A predicate that always evaluates to `FALSE`.

Basic Compound Predicates

`AND`, `&&`
: Logical AND.

`OR`, `||`
: Logical OR.

`NOT`, `!`
: Logical NOT.

String Comparisons

String comparisons are, by default, case and diacritic sensitive. You can modify an operator using the key characters `c` and `d` within square braces to specify case and diacritic insensitivity respectively, for example `firstName BEGINSWITH[cd] $FIRST_NAME`.

`BEGINSWITH`
: The left-hand expression begins with the right-hand expression.

`CONTAINS`
: The left-hand expression contains the right-hand expression.

`ENDSWITH`
: The left-hand expression ends with the right-hand expression.

`LIKE`
: The left hand expression equals the right-hand expression: `?` and `*` are allowed as wildcard characters, where `?` matches `1` character and `*` matches `0` or more characters.

`MATCHES`
: The left hand expression equals the right hand expression using a `regex`-style comparison according to ICU v3 (for more details see the ICU User Guide for Regular Expressions).

`UTI-CONFORMS-TO`
: The left hand argument to this operator is an expression that evaluates to a universal type identifier (UTI) you want to match. The right hand argument is an expression that evaluates to a UTI. The comparison evaluates to `TRUE` if the UTI returned by the left hand expression conforms to the UTI returned by the right hand expression. For information on which types conform to a given type, see System-Declared Uniform Type Identifiers in *Uniform Type Identifiers Reference*. To learn how to declare conformance to a custom UTI, see Declaring New Uniform Type Identifiers in *Uniform Type Identifiers Overview*.

  The clause `A UTI-CONFORMS-TO B` provides the same result as employing the `UTTypeConformsTo` method as follows:

```
UTTypeConformsTo (A, B)
```

  When evaluating attachments in an app extension item (of type `NSExtensionItem`), you could use a statement similar to the following:

```
SUBQUERY (
    extensionItems,
    $extensionItem,
    SUBQUERY (
        $extensionItem.attachments,
        $attachment,
        ANY $attachment.registeredTypeIdentifiers UTI-CONFORMS-TO "com.adobe.pdf"
    ).@count == $extensionItem.attachments.@count
).@count == 1
```

`UTI-EQUALS`

The left hand argument to this operator is an expression that evaluates to a universal type identifier (UTI) you want to match. The right hand argument is an expression that evaluates to a UTI. The comparison evaluates to `TRUE` if the UTI returned by the left hand expression equals the UTI returned by the right hand expression.

The clause `A UTI-EQUALS B` provides the same result as employing the `UTTypeEqual` method as follows:

```
UTTypeEqual (A, B)
```

See the code example in the `UTI-CONFORMS-TO` entry, which applies as well to the `UTI-EQUALS` operator by replacing the operator.

Aggregate Operations

`ANY`, `SOME`

Specifies any of the elements in the following expression. For example `ANY children.age < 18`.

`ALL`

Specifies all of the elements in the following expression. For example `ALL children.age < 18`.

`NONE`

Specifies none of the elements in the following expression. For example, `NONE children.age < 18`. This is logically equivalent to `NOT (ANY ...)`.

`IN`

Equivalent to an SQL IN operation, the left-hand side must appear in the collection specified by the right-hand side.

For example, `name IN { 'Ben', 'Melissa', 'Nick' }`. The collection may be an array, a set, or a dictionary—in the case of a dictionary, its values are used.

In Objective-C, you could create a IN predicate as shown in the following example:

```
NSPredicate *inPredicate =
            [NSPredicate predicateWithFormat: @"attribute IN %@", aCollection];
```

where `aCollection` may be an instance of `NSArray`, `NSSet`, `NSDictionary`, or of any of the corresponding mutable classes.

`array[index]`
: Specifies the element at the specified index in the array `array`.

`array[FIRST]`
: Specifies the first element in the array `array`.

`array[LAST]`
: Specifies the last element in the array `array`.

`array[SIZE]`
: Specifies the size of the array `array`.

Identifiers

C style identifier
: Any C style identifier that is not a reserved word.

#symbol
: Used to escape a reserved word into a user identifier.

[\]{octaldigit}{3}
: Used to escape an octal number (`\` followed by 3 octal digits).

[\][xX]{hexdigit}{2}
: Used to escape a hex number (`\x` or `\X` followed by 2 hex digits).

[\][uU]{hexdigit}{4}
: Used to escape a Unicode number (`\u` or `\U` followed by 4 hex digits).

Literals

Single and double quotes produce the same result, but they do not terminate each other. For example, `"abc"` and `'abc'` are identical, whereas `"a'b'c"` is equivalent to a space-separated concatenation of `a`, `'b'`, `c`.

`FALSE`, `NO`
: Logical false.

`TRUE`, `YES`
: Logical true.

`NULL`, `NIL`
: A null value.

`SELF`
: Represents the object being evaluated.

`"text"`
: A character string.

`'text'`
: A character string.

Comma-separated literal array
: For example, `{ 'comma', 'separated', 'literal', 'array' }`.

Standard integer and fixed-point notations
: For example, `1`, `27`, `2.71828`, `19.75`.

Floating-point notation with exponentiation
: For example, `9.2e-5`.

`0x`
: Prefix used to denote a hexadecimal digit sequence.

`0o`
: Prefix used to denote an octal digit sequence.

`0b`
: Prefix used to denote a binary digit sequence.

Reserved Words

The following words are reserved:

`AND`, `OR`, `IN`, `NOT`, `ALL`, `ANY`, `SOME`, `NONE`, `LIKE`, `CASEINSENSITIVE`, `CI`, `MATCHES`, `CONTAINS`, `BEGINSWITH`, `ENDSWITH`, `BETWEEN`, `NULL`, `NIL`, `SELF`, `TRUE`, `YES`, `FALSE`, `NO`, `FIRST`, `LAST`, `SIZE`, `ANYKEY`, `SUBQUERY`, `CAST`, `TRUEPREDICATE`, `FALSEPREDICATE`, `UTI-CONFORMS-TO`, `UTI-EQUALS`

BNF Definition of Cocoa Predicates

This article defines Cocoa predicates in Backus-Naur Form notation.

NSPredicate

```
NSPredicate ::= NSComparisonPredicate | NSCompoundPredicate
    | "(" NSPredicate ")" | TRUEPREDICATE | FALSEPREDICATE
```

NSCompoundPredicate

```
NSCompoundPredicate ::= NSPredicate "AND" NSPredicate
    | NSPredicate "OR" NSPredicate
    | "NOT" NSPredicate
```

NSComparisonPredicate

```
NSComparisonPredicate ::= expression operation expression
    | aggregate_qualifier NSComparisonPredicate
```

Operations

`CONTAINS` and `IN` serve both as aggregate operators and string operators, depending on the types of their arguments.

```
operation ::= "=" | "!=" | "<" | ">" | "<=" | ">="
    | BETWEEN
    | aggregate_operations [ "[" string_options "]" ]
```

```
aggregate_operations ::= CONTAINS | IN | string_operations

string_operations ::= BEGINSWITH | ENDSWITH | LIKE | MATCHES

string_options ::= c | d | cd
```

Aggregate Qualifier

```
aggregate_qualifier ::= ANY | ALL | NONE | SOME
```

Expression

```
expression ::= "(" expression ")"
    | binary_expression
    | function_expression
    | assignment_expression
    | index_expression
    | keypath_expression
    | value_expression
```

Value Expression

```
value_expression ::= literal_value | literal_aggregate
```

Literal Value

```
literal_value ::= string_value
```

```
    | numeric_value
    | predicate_argument
    | predicate_variable
    | NULL
    | TRUE
    | FALSE
    | SELF
```

String Value

```
string_value ::= "text" | 'text'
```

Predicate Argument

```
predicate_argument ::= "%" format_argument
```

Format Argument

```
format_argument ::= "@" | "%" | "K"
    printf style conversion character
```

Predicate Variable

```
predicate_variable ::= "$" identifier
```

Keypath Expression

```
keypath_expression ::= identifier | "@" identifier
```

```
    | expression "." expression
```

Literal Aggregate

```
literal_aggregate ::= "{" [ expression [ "," expression ... ] ] "}"
```

Index Expression

```
index_expression ::= array_expression "[" integer_expression "]"
    | dictionary_expression   "[" expression "]"
    | aggregate_expression "[" FIRST "]"
    | aggregate_expression "[" LAST "]"
    | aggregate_expression "[" SIZE "]"
```

Aggregate Expression

```
aggregate_expression ::= array_expression | dictionary_expression
```

Assignment Expression

```
assignment_expression ::= predicate_variable ":=" expression
```

Binary Expression

```
binary_expression ::= expression binary_operator expression
    | "-" expression
```

Binary Operator

```
binary_operator ::= "+" | "-" | "*" | "/" | "**"
```

Function Expression

```
function_expression ::= function_name "(" [ expression [ "," expression ... ] ]
")"
```

Function Name

```
function_name ::= "sum" | "count" | "min" | "max"
    | "average" | "median" | "mode" | "stddev"
    | "sqrt" | "log" | "ln" | "exp"
    | "floor" | "ceiling" | "abs" | "trunc"
    | "random" | "randomn" | "now"
```

Array Expression

```
array_expression ::= any expression that evaluates to an NSArray object
```

Dictionary Expression

```
dictionary_expression ::= any expression that evaluates to an NSDictionary object
```

Integer Expression

```
integer_expression ::= any expression that evaluates to an integral value
```

Numeric Value

```
numeric_value ::= C style numeric constant
```

Identifier

```
identifier ::= C style identifier | "#" reserved_word
```

Document Revision History

This table describes the changes to *Predicate Programming Guide*.

Date	Notes
2014-09-17	Added descriptions for the new UTI-CONFORMS-TO and UTI-EQUALS predicate operators. Performed a few corrections for clarity.
2010-06-14	Revised introductory article for clarity; corrected minor error in Predicate Format String From a Spotlight Search in Finder.
2009-11-17	Corrected description of operations supported by Core Data SQLite store.
2009-02-03	Updated for iOS 3.0.
2008-10-15	Clarified example of BETWEEN operator.
2006-10-03	Clarified issues with variable substitution in Predicate Format String Syntax and added an example for Booleans.
2006-05-23	Clarified the use of wildcards in the LIKE operator.
2006-04-04	Moved discussion of Core Data model-based fetch requests to "Core Data Programming Guide."
2006-03-08	Clarified need for key-value coding compliance for keys to be used in a predicate.

Date	Notes
2006-01-10	Corrected typographical errors.
2005-11-09	Corrected minor typographical errors.
2005-10-04	Clarified relationships examples; added section on regular expressions.
2005-08-11	Added information comparing NSPredicate with Spotlight queries, and described how to use predicates with Cocoa bindings and Core Data.
2005-07-07	Corrected minor typographical errors.
2005-04-29	New document that describes how to specify queries in Cocoa.

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