Chapter 8. LINQ Queries

LINQ, or Language Integrated Query, is a set of language and runtime features for writing structured type-safe queries over local object collections and remote data sources.

LINQ enables you to query any collection implementing `IEnumerable<T>`, whether an array, list, or XML Document Object Model (DOM), as well as remote data sources, such as tables in an SQL Server database. LINQ offers the benefits of both compile-time type checking and dynamic query composition.

This chapter describes the LINQ architecture and the fundamentals of writing queries. All core types are defined in the `System.Linq` and `System.Linq.Expressions` namespaces.

NOTE

The examples in this and the following two chapters are preloaded into an interactive querying tool called LINQPad. You can download LINQPad from *http://www.linqpad.net*.

Getting Started

The basic units of data in LINQ are *sequences* and *elements*. A sequence is any object that implements `IEnumerable<T>`, and an element is each item in the sequence. In the following example, `names` is a sequence, and `"Tom"`, `"Dick"`, and `"Harry"` are elements:

```
string[] names = { "Tom", "Dick", "Harry" };
```

We call this a *local sequence* because it represents a local collection of objects in memory.

A *query operator* is a method that transforms a sequence. A typical query operator accepts an *input sequence* and emits a transformed *output sequence*. In the `Enumerable` class in `System.Linq`, there are around 40 query operators—all implemented as static extension methods. These are called *standard query operators*.

NOTE

Queries that operate over local sequences are called *local* queries or *LINQ-to-objects* queries.

LINQ also supports sequences that can be dynamically fed from a remote data source such as an SQL Server database. These sequences additionally implement the `IQueryable<T>` interface and are supported through a matching set of standard query operators in the `Queryable` class. We discuss this further in "Interpreted Queries".

A query is an expression that, when enumerated, transforms sequences with query operators. The simplest query comprises one input sequence and one operator. For instance, we can apply the `Where` operator on a simple array to extract those strings whose length is at least four characters, as follows:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = System.Linq.Enumerable.Where
                                    (names, n => n.Length >= 4);
foreach (string n in filteredNames)
  Console.WriteLine (n);

Dick
Harry
```

Because the standard query operators are implemented as extension methods, we can call `Where` directly on `names`, as though it were an instance method:

```
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

For this to compile, you must import the `System.Linq` namespace. Here’s a complete example:

```
using System;
using System.Collections.Generic;
using System.Linq;

string[] names = { "Tom", "Dick", "Harry" };

IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
foreach (string name in filteredNames) Console.WriteLine (name);

Dick
Harry
```

NOTE

We could further shorten our code by implicitly typing `filteredNames`:

```
var filteredNames = names.Where (n => n.Length >= 4);
```

This can hinder readability, however, outside of an IDE, where there are no tool tips to help. For this reason, we make less use of implicit typing in this chapter than you might in your own projects.

Most query operators accept a lambda expression as an argument. The lambda expression helps guide and shape the query. In our example, the lambda expression is as follows:

```
n => n.Length >= 4
```

The input argument corresponds to an input element. In this case, the input argument `n` represents each name in the array and is of type `string`. The

Where operator requires that the lambda expression return a bool value, which if true, indicates that the element should be included in the output sequence. Here's its signature:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

The following query extracts all names that contain the letter "a":

```
IEnumerable<string> filteredNames = names.Where (n => n.Contains ("a"));

foreach (string name in filteredNames)
  Console.WriteLine (name);             // Harry
```

So far, we've built queries using extension methods and lambda expressions. As you'll see shortly, this strategy is highly composable in that it allows the chaining of query operators. In this book, we refer to this as *fluent syntax*.[1] C# also provides another syntax for writing queries, called *query expression* syntax. Here's our preceding query written as a query expression:

```
IEnumerable<string> filteredNames = from n in names
                                    where n.Contains ("a")
                                    select n;
```

Fluent syntax and query syntax are complementary. In the following two sections, we explore each in more detail.

Fluent Syntax

Fluent syntax is the most flexible and fundamental. In this section, we describe how to chain query operators to form more complex queries—and show why extension methods are important to this process. We also describe how to formulate lambda expressions for a query operator and introduce several new query operators.

Chaining Query Operators

In the preceding section, we showed two simple queries, each comprising a single query operator. To build more complex queries, you append additional query operators to the expression, creating a chain. To illustrate, the following query extracts all strings containing the letter “a,” sorts them by length, and then converts the results to uppercase:

```
using System;
using System.Collections.Generic;
using System.Linq;

string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query = names
  .Where   (n => n.Contains ("a"))
  .OrderBy (n => n.Length)
  .Select  (n => n.ToUpper());

foreach (string name in query) Console.WriteLine (name);

JAY
MARY
HARRY
```

NOTE

The variable, `n`, in our example, is privately scoped to each of the lambda expressions. We can reuse the identifier `n` for the same reason that we can reuse the identifier `c` in the following method:

```
void Test()
{
  foreach (char c in "string1") Console.Write (c);
  foreach (char c in "string2") Console.Write (c);
  foreach (char c in "string3") Console.Write (c);
}
```

`Where`, `OrderBy`, and `Select` are standard query operators that resolve to extension methods in the `Enumerable` class (if you import the `System.Linq` namespace).

We already introduced the `Where` operator, which emits a filtered version of the input sequence. The `OrderBy` operator emits a sorted version of its input sequence; the `Select` method emits a sequence in which each input element is transformed or *projected* with a given lambda expression (`n.ToUpper()`, in this case). Data flows from left to right through the chain of operators, so the data is first filtered, then sorted, and then projected.

NOTE

A query operator never alters the input sequence; instead, it returns a new sequence. This is consistent with the *functional programming* paradigm from which LINQ was inspired.

Here are the signatures of each of these extension methods (with the `OrderBy` signature slightly simplified):

```
public static IEnumerable<TSource> Where<TSource>
```

```
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)

public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)

public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

When query operators are chained as in this example, the output sequence of one operator is the input sequence of the next. The complete query resembles a production line of conveyor belts, as illustrated in Figure 8-1.

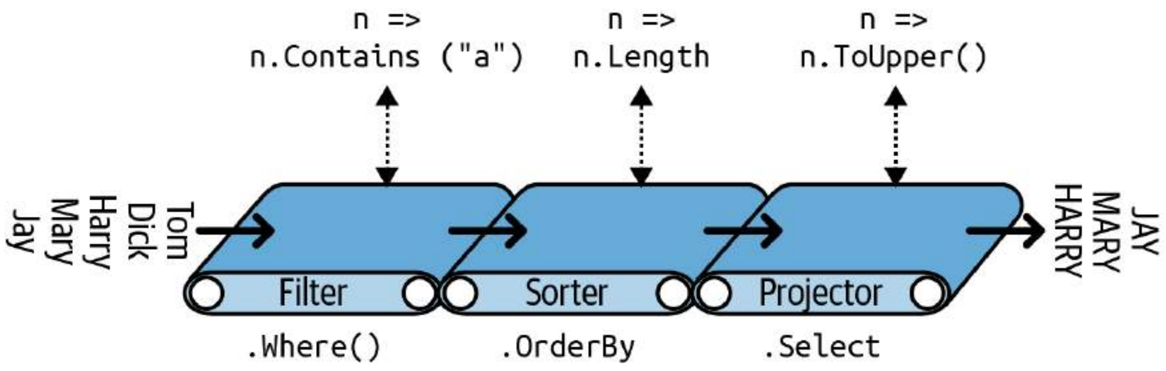

Figure 8-1. Chaining query operators

We can construct the identical query *progressively*, as follows:

```
// You must import the System.Linq namespace for this to compile:

IEnumerable<string> filtered   = names   .Where   (n => n.Contains ("a"));
IEnumerable<string> sorted     = filtered.OrderBy (n => n.Length);
IEnumerable<string> finalQuery = sorted  .Select  (n => n.ToUpper());
```

`finalQuery` is compositionally identical to the `query` we constructed previously. Further, each intermediate step also comprises a valid query that we can execute:

```
foreach (string name in filtered)
  Console.Write (name + "|");        // Harry|Mary|Jay|

Console.WriteLine();
foreach (string name in sorted)
```

```
    Console.Write (name + "|");        // Jay|Mary|Harry|

  Console.WriteLine();
  foreach (string name in finalQuery)
    Console.Write (name + "|");        // JAY|MARY|HARRY|
```

Why extension methods are important

Instead of using extension method syntax, you can use conventional static method syntax to call the query operators:

```
  IEnumerable<string> filtered = Enumerable.Where (names,
                                                   n => n.Contains ("a"));
  IEnumerable<string> sorted = Enumerable.OrderBy (filtered, n => n.Length);
  IEnumerable<string> finalQuery = Enumerable.Select (sorted,
                                                      n => n.ToUpper());
```

This is, in fact, how the compiler translates extension method calls. Shunning extension methods comes at a cost, however, if you want to write a query in a single statement as we did earlier. Let's revisit the single-statement query—first in extension method syntax:

```
  IEnumerable<string> query = names.Where   (n => n.Contains ("a"))
                                   .OrderBy (n => n.Length)
                                   .Select  (n => n.ToUpper());
```

Its natural linear shape reflects the left-to-right flow of data and also keeps lambda expressions alongside their query operators (*infix* notation). Without extension methods, the query loses its *fluency*:

```
  IEnumerable<string> query =
    Enumerable.Select (
      Enumerable.OrderBy (
        Enumerable.Where (
          names, n => n.Contains ("a")
        ), n => n.Length
      ), n => n.ToUpper()
    );
```

Composing Lambda Expressions

In previous examples, we fed the following lambda expression to the `Where` operator:

```
n => n.Contains ("a")      // Input type = string, return type = bool.
```

> **NOTE**
>
> A lambda expression that takes a value and returns a `bool` is called a *predicate*.

The purpose of the lambda expression depends on the particular query operator. With the `Where` operator, it indicates whether an element should be included in the output sequence. In the case of the `OrderBy` operator, the lambda expression maps each element in the input sequence to its sorting key. With the `Select` operator, the lambda expression determines how each element in the input sequence is transformed before being fed to the output sequence.

> **NOTE**
>
> A lambda expression in a query operator always works on individual elements in the input sequence—not the sequence as a whole.

The query operator evaluates your lambda expression upon demand, typically once per element in the input sequence. Lambda expressions allow you to feed your own logic into the query operators. This makes the query operators versatile as well as being simple under the hood. Here's a complete implementation of `Enumerable.Where`, exception handling aside:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

```
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

Lambda expressions and Func signatures

The standard query operators utilize generic `Func` delegates. `Func` is a family of general-purpose generic delegates in the `System` namespace, defined with the following intent:

> *The type arguments in `Func` appear in the same order as they do in lambda expressions.*

Hence, `Func<TSource,bool>` matches a `TSource=>bool` lambda expression: one that accepts a `TSource` argument and returns a `bool` value.

Similarly, `Func<TSource,TResult>` matches a `TSource=>TResult` lambda expression.

The `Func` delegates are listed in "Lambda Expressions".

Lambda expressions and element typing

The standard query operators use the following type parameter names:

Generic type letter	Meaning
`TSource`	Element type for the input sequence
`TResult`	Element type for the output sequence (if different from `TSource`)
`TKey`	Element type for the *key* used in sorting, grouping, or joining

`TSource` is determined by the input sequence. `TResult` and `TKey` are typically *inferred from your lambda expression*.

For example, consider the signature of the `Select` query operator:

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

`Func<TSource,TResult>` matches a `TSource=>TResult` lambda expression: one that maps an *input element* to an *output element*. `TSource` and `TResult` can be different types, so the lambda expression can change the type of each element. Further, the lambda expression *determines the output sequence type*. The following query uses `Select` to transform string type elements to integer type elements:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<int> query = names.Select (n => n.Length);

foreach (int length in query)
  Console.Write (length + "|");    // 3|4|5|4|3|
```

The compiler can *infer* the type of `TResult` from the return value of the lambda expression. In this case, `n.Length` returns an `int` value, so `TResult` is inferred to be `int`.

The `Where` query operator is simpler and requires no type inference for the output because input and output elements are of the same type. This makes sense because the operator merely filters elements; it does not *transform* them:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

Finally, consider the signature of the `OrderBy` operator:

```
// Slightly simplified:
public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)
```

`Func<TSource,TKey>` maps an input element to a *sorting key*. `TKey` is inferred from your lambda expression and is separate from the input and output element types. For instance, we could choose to sort a list of names by length (`int` key) or alphabetically (`string` key):

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> sortedByLength, sortedAlphabetically;
sortedByLength       = names.OrderBy (n => n.Length);   // int key
sortedAlphabetically = names.OrderBy (n => n);          // string key
```

NOTE

You can call the query operators in `Enumerable` with traditional delegates that refer to methods instead of lambda expressions. This approach is effective in simplifying certain kinds of local queries—particularly with LINQ to XML—and is demonstrated in Chapter 10. It doesn't work with `IQueryable<T>`-based sequences, however (e.g., when querying a database), because the operators in `Queryable` require lambda expressions in order to emit expression trees. We discuss this later in "Interpreted Queries".

Natural Ordering

The original ordering of elements within an input sequence is significant in LINQ. Some query operators rely on this ordering, such as `Take`, `Skip`, and `Reverse`.

The `Take` operator outputs the first `x` elements, discarding the rest:

```
int[] numbers  = { 10, 9, 8, 7, 6 };
IEnumerable<int> firstThree = numbers.Take (3);     // { 10, 9, 8 }
```

The `Skip` operator ignores the first `x` elements and outputs the rest:

```
IEnumerable<int> lastTwo    = numbers.Skip (3);     // { 7, 6 }
```

`Reverse` does exactly as it says:

```
IEnumerable<int> reversed   = numbers.Reverse();    // { 6, 7, 8, 9, 10 }
```

With local queries (LINQ-to-objects), operators such as `Where` and `Select` preserve the original ordering of the input sequence (as do all other query operators, except for those that specifically change the ordering).

Other Operators

Not all query operators return a sequence. The *element* operators extract one element from the input sequence; examples are `First`, `Last`, and `ElementAt`:

```
int[] numbers    = { 10, 9, 8, 7, 6 };
int firstNumber  = numbers.First();                        // 10
int lastNumber   = numbers.Last();                         // 6
int secondNumber = numbers.ElementAt(1);                   // 9
int secondLowest = numbers.OrderBy(n=>n).Skip(1).First();  // 7
```

Because these operators return a single element, you don't usually call further query operators on their result unless that element itself is a collection.

The *aggregation* operators return a scalar value, usually of numeric type:

```
int count = numbers.Count();          // 5;
int min = numbers.Min();              // 6;
```

The *quantifiers* return a `bool` value:

```
bool hasTheNumberNine = numbers.Contains (9);          // true
bool hasMoreThanZeroElements = numbers.Any();          // true
bool hasAnOddElement = numbers.Any (n => n % 2 != 0);  // true
```

Some query operators accept two input sequences. Examples are `Concat`, which appends one sequence to another, and `Union`, which does the same but with duplicates removed:

```
int[] seq1 = { 1, 2, 3 };
int[] seq2 = { 3, 4, 5 };
IEnumerable<int> concat = seq1.Concat (seq2);    //  { 1, 2, 3, 3, 4, 5 }
IEnumerable<int> union  = seq1.Union (seq2);     //  { 1, 2, 3, 4, 5 }
```

The joining operators also fall into this category. Chapter 9 covers all of the query operators in detail.

Query Expressions

C# provides a syntactic shortcut for writing LINQ queries, called *query expressions*. Contrary to popular belief, a query expression is not a means of embedding SQL into C#. In fact, the design of query expressions was inspired primarily by *list comprehensions* from functional programming languages such as LISP and Haskell, although SQL had a cosmetic influence.

NOTE

In this book, we refer to query expression syntax simply as *query syntax*.

In the preceding section, we wrote a fluent-syntax query to extract strings containing the letter “a,” sorted by length and converted to uppercase. Here’s the same thing in query syntax:

```
using System;
using System.Collections.Generic;
using System.Linq;

string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query =
  from    n in names
  where   n.Contains ("a")     // Filter elements
  orderby n.Length             // Sort elements
  select  n.ToUpper();         // Translate each element (project)

foreach (string name in query) Console.WriteLine (name);

JAY
MARY
HARRY
```

Query expressions always start with a `from` clause and end with either a `select` or `group` clause. The `from` clause declares a *range variable* (in this case, `n`), which you can think of as traversing the input sequence—rather like `foreach`. Figure 8-2 illustrates the complete syntax as a railroad diagram.

NOTE

To read this diagram, start at the left and then proceed along the track as if you were a train. For instance, after the mandatory `from` clause, you can optionally include an `orderby`, `where`, `let`, or `join` clause. After that, you can either continue with a `select` or `group` clause, or go back and include another `from`, `orderby`, `where`, `let`, or `join` clause.

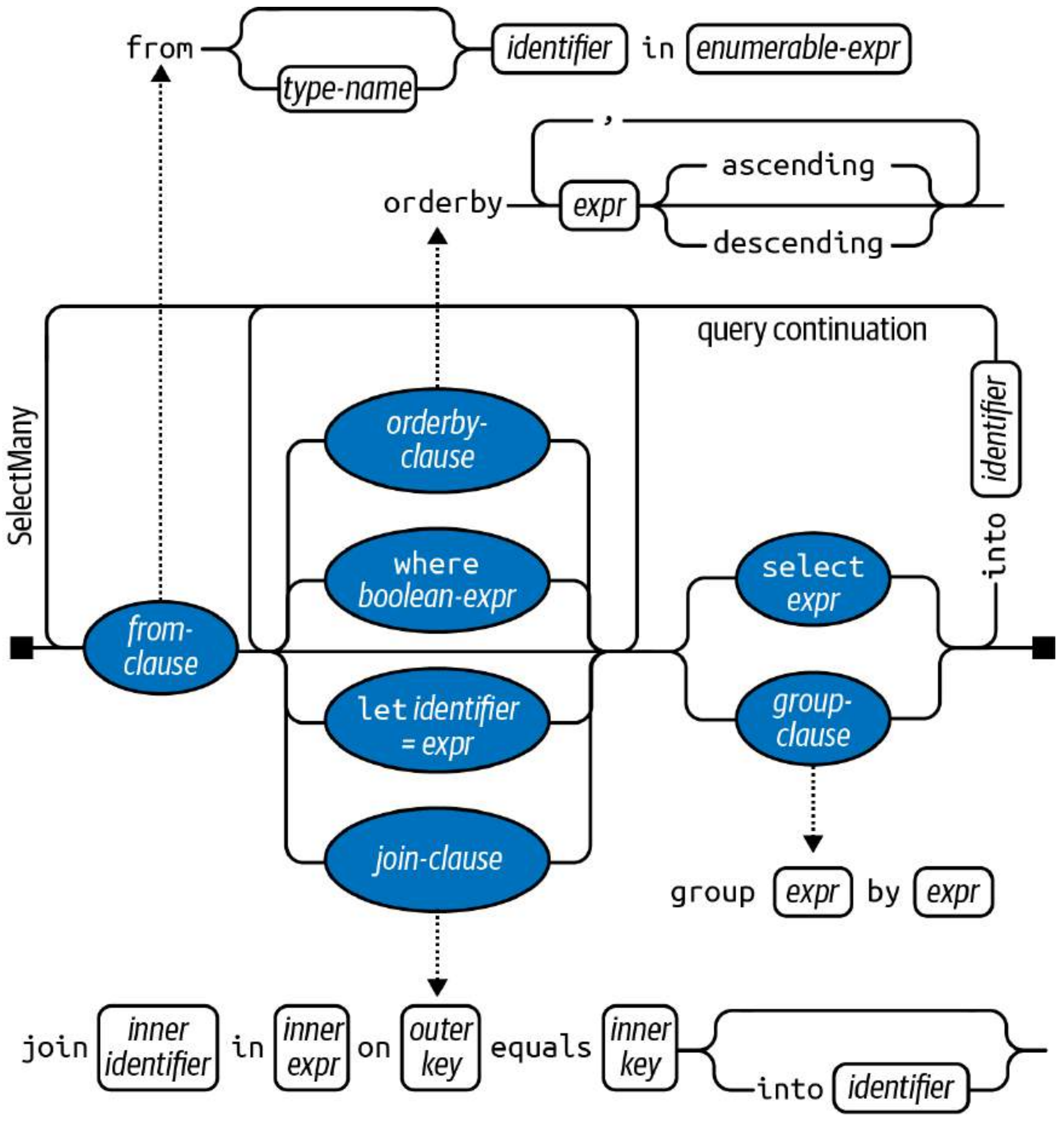

Figure 8-2. Query syntax

The compiler processes a query expression by translating it into fluent syntax. It does this in a fairly mechanical fashion—much like it translates `foreach` statements into calls to `GetEnumerator` and `MoveNext`. This means that anything you can write in query syntax you can also write in fluent syntax. The compiler (initially) translates our example query into the following:

```
IEnumerable<string> query = names.Where   (n => n.Contains ("a"))
                                 .OrderBy (n => n.Length)
                                 .Select  (n => n.ToUpper());
```

The `Where`, `OrderBy`, and `Select` operators then resolve using the same rules that would apply if the query were written in fluent syntax. In this case, they bind to extension methods in the `Enumerable` class because the `System.Linq` namespace is imported and `names` implements `IEnumerable<string>`. The compiler doesn't specifically favor the `Enumerable` class, however, when translating query expressions. You can think of the compiler as mechanically injecting the words "Where," "OrderBy," and "Select" into the statement and then compiling it as though you had typed the method names yourself. This offers flexibility in how they resolve. The operators in the database queries that we write in later sections, for instance, will bind instead to extension methods in `Queryable`.

NOTE

If we remove the `using System.Linq` directive from our program, the query would not compile, since the `Where`, `OrderBy`, and `Select` methods would have nowhere to bind. Query expressions cannot compile unless you import `System.Linq` or another namespace with an implementation of these query methods.

Range Variables

The identifier immediately following the `from` keyword syntax is called the *range variable*. A range variable refers to the current element in the sequence on which the operation is to be performed.

In our examples, the range variable `n` appears in every clause in the query. And yet, the variable actually enumerates over a *different* sequence with each clause:

```
from    n in names           // n is our range variable
```

```
where   n.Contains ("a")     // n = directly from the array
orderby n.Length             // n = subsequent to being filtered
select  n.ToUpper()          // n = subsequent to being sorted
```

This becomes clear when we examine the compiler's mechanical translation to fluent syntax:

```
names.Where   (n => n.Contains ("a"))      // Locally scoped n
     .OrderBy (n => n.Length)              // Locally scoped n
     .Select  (n => n.ToUpper())           // Locally scoped n
```

As you can see, each instance of `n` is scoped privately to its own lambda expression.

Query expressions also let you introduce new range variables via the following clauses:

- `let`
- `into`
- An additional `from` clause
- `join`

We cover these later in this chapter in "Composition Strategies" as well as in Chapter 9, in "Projecting" and "Joining".

Query Syntax Versus SQL Syntax

Query expressions look superficially like SQL, yet the two are very different. A LINQ query boils down to a C# expression, and so follows standard C# rules. For example, with LINQ, you cannot use a variable before you declare it. In SQL, you can reference a table alias in the `SELECT` clause before defining it in a `FROM` clause.

A subquery in LINQ is just another C# expression and so requires no special syntax. Subqueries in SQL are subject to special rules.

With LINQ, data logically flows from left to right through the query. With SQL, the order is less well structured with regard to data flow.

A LINQ query comprises a conveyor belt or *pipeline* of operators that accept and emit sequences whose element order can matter. An SQL query comprises a *network* of clauses that work mostly with *unordered sets*.

Query Syntax Versus Fluent Syntax

Query and fluent syntax each have advantages.

Query syntax is simpler for queries that involve any of the following:

- A `let` clause for introducing a new variable alongside the range variable
- `SelectMany`, `Join`, or `GroupJoin`, followed by an outer range variable reference

(We describe the `let` clause in "Composition Strategies"; we describe `SelectMany`, `Join`, and `GroupJoin` in Chapter 9.)

The middle ground is queries that involve the simple use of `Where`, `OrderBy`, and `Select`. Either syntax works well; the choice here is largely personal.

For queries that comprise a single operator, fluent syntax is shorter and less cluttered.

Finally, there are many operators that have no keyword in query syntax. These require that you use fluent syntax—at least in part. This means any operator outside of the following:

```
Where, Select, SelectMany
OrderBy, ThenBy, OrderByDescending, ThenByDescending
GroupBy, Join, GroupJoin
```

Mixed-Syntax Queries

If a query operator has no query-syntax support, you can mix query syntax and fluent syntax. The only restriction is that each query-syntax component must be complete (i.e., start with a `from` clause and end with a `select` or `group` clause).

Assuming this array declaration

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

the following example counts the number of names containing the letter "a":

```
int matches = (from n in names where n.Contains ("a") select n).Count();
// 3
```

The next query obtains the first name in alphabetical order:

```
string first = (from n in names orderby n select n).First();   // Dick
```

The mixed-syntax approach is sometimes beneficial in more complex queries. With these simple examples, however, we could stick to fluent syntax throughout without penalty:

```
int matches = names.Where (n => n.Contains ("a")).Count();   // 3
string first = names.OrderBy (n => n).First();               // Dick
```

NOTE

There are times when mixed-syntax queries offer by far the highest "bang for the buck" in terms of function and simplicity. It's important not to unilaterally favor either query or fluent syntax; otherwise, you'll be unable to write mixed-syntax queries when they are the best option.

Where applicable, the remainder of this chapter shows key concepts in both fluent and query syntax.

Deferred Execution

An important feature of most query operators is that they execute not when constructed but when *enumerated* (in other words, when `MoveNext` is called on its enumerator). Consider the following query:

```
var numbers = new List<int> { 1 };

IEnumerable<int> query = numbers.Select (n => n * 10);    // Build query

numbers.Add (2);                    // Sneak in an extra element

foreach (int n in query)
  Console.Write (n + "|");          // 10|20|
```

The extra number that we sneaked into the list *after* constructing the query is included in the result because it's not until the `foreach` statement runs that any filtering or sorting takes place. This is called *deferred* or *lazy* execution and is the same as what happens with delegates:

```
Action a = () => Console.WriteLine ("Foo");
// We've not written anything to the Console yet. Now let's run it:
a();  // Deferred execution!
```

All standard query operators provide deferred execution, with the following exceptions:

- Operators that return a single element or scalar value, such as `First` or `Count`
- The following *conversion operators*:

```
ToArray, ToList, ToDictionary, ToLookup, ToHashSet
```

These operators cause immediate query execution because their result types have no mechanism to provide deferred execution. The `Count` method, for instance, returns a simple integer, which doesn't then get enumerated. The following query is executed immediately:

```
int matches = numbers.Where (n => n <= 2).Count();    // 1
```

Deferred execution is important because it decouples query *construction* from query *execution*. This allows you to construct a query in several steps and also makes database queries possible.

> **NOTE**
>
> Subqueries provide another level of indirection. Everything in a subquery is subject to deferred execution, including aggregation and conversion methods. We describe this in "Subqueries".

Reevaluation

Deferred execution has another consequence: a deferred execution query is reevaluated when you reenumerate:

```
var numbers = new List<int>() { 1, 2 };

IEnumerable<int> query = numbers.Select (n => n * 10);
foreach (int n in query) Console.Write (n + "|");   // 10|20|

numbers.Clear();
foreach (int n in query) Console.Write (n + "|");   // <nothing>
```

There are a couple of reasons why reevaluation is sometimes disadvantageous:

- Sometimes, you want to “freeze” or cache the results at a certain point in time.
- Some queries are computationally intensive (or rely on querying a remote database), so you don’t want to unnecessarily repeat them.

You can defeat reevaluation by calling a conversion operator such as `ToArray` or `ToList`. `ToArray` copies the output of a query to an array; `ToList` copies to a generic `List<T>`:

```
var numbers = new List<int>() { 1, 2 };

List<int> timesTen = numbers
  .Select (n => n * 10)

  .ToList();                // Executes immediately into a List<int>

numbers.Clear();
Console.WriteLine (timesTen.Count);      // Still 2
```

Captured Variables

If your query’s lambda expressions *capture* outer variables, the query will honor the value of those variables at the time the query *runs*:

```
int[] numbers = { 1, 2 };

int factor = 10;
IEnumerable<int> query = numbers.Select (n => n * factor);
factor = 20;
foreach (int n in query) Console.Write (n + "|");   // 20|40|
```

This can be a trap when building up a query within a `for` loop. For example, suppose that we want to remove all vowels from a string. The following, although inefficient, gives the correct result:

```
IEnumerable<char> query = "Not what you might expect";
```

```
query = query.Where (c => c != 'a');
query = query.Where (c => c != 'e');
query = query.Where (c => c != 'i');
query = query.Where (c => c != 'o');
query = query.Where (c => c != 'u');

foreach (char c in query) Console.Write (c);  // Nt wht y mght xpct
```

Now watch what happens when we refactor this with a `for` loop:

```
IEnumerable<char> query = "Not what you might expect";
string vowels = "aeiou";

for (int i = 0; i < vowels.Length; i++)
  query = query.Where (c => c != vowels[i]);

foreach (char c in query) Console.Write (c);
```

An `IndexOutOfRangeException` is thrown upon enumerating the query because, as we saw in Chapter 4 (see "Capturing Outer Variables"), the compiler scopes the iteration variable in the `for` loop as if it were declared *outside* the loop. Hence, each closure captures the *same* variable (`i`) whose value is 5 when the query is actually enumerated. To solve this, you must assign the loop variable to another variable declared *inside* the statement block:

```
for (int i = 0; i < vowels.Length; i++)
{
  char vowel = vowels[i];
  query = query.Where (c => c != vowel);
}
```

This forces a fresh local variable to be captured on each loop iteration.

NOTE

Another way to solve the problem is to replace the `for` loop with a `foreach` loop:

```
foreach (char vowel in vowels)
  query = query.Where (c => c != vowel);
```

How Deferred Execution Works

Query operators provide deferred execution by returning *decorator* sequences.

Unlike a traditional collection class such as an array or linked list, a decorator sequence (in general) has no backing structure of its own to store elements. Instead, it wraps another sequence that you supply at runtime, to which it maintains a permanent dependency. Whenever you request data from a decorator, it in turn must request data from the wrapped input sequence.

NOTE

The query operator's transformation constitutes the "decoration." If the output sequence performed no transformation, it would be a *proxy* rather than a decorator.

Calling `Where` merely constructs the decorator wrapper sequence, which holds a reference to the input sequence, the lambda expression, and any other arguments supplied. The input sequence is enumerated only when the decorator is enumerated.

Figure 8-3 illustrates the composition of the following query:

```
IEnumerable<int> lessThanTen = new int[] { 5, 12, 3 }.Where (n => n < 10);
```

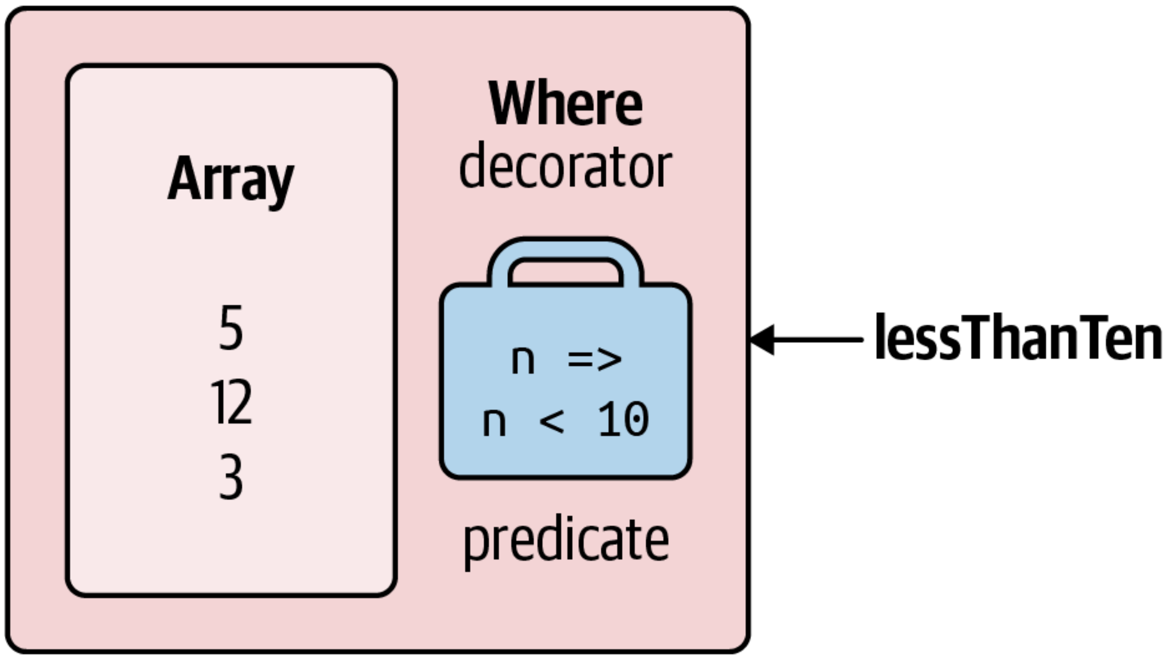

Figure 8-3. Decorator sequence

When you enumerate `lessThanTen`, you are, in effect, querying the array through the `Where` decorator.

The good news—should you ever want to write your own query operator—is that implementing a decorator sequence is easy with a C# iterator. Here's how you can write your own `Select` method:

```
public static IEnumerable<TResult> MySelect<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

This method is an iterator by virtue of the `yield return` statement. Functionally, it's a shortcut for the following:

```
public static IEnumerable<TResult> MySelect<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  return new SelectSequence (source, selector);
}
```

where *SelectSequence* is a (compiler-written) class whose enumerator encapsulates the logic in the iterator method.

Hence, when you call an operator such as `Select` or `Where`, you're doing nothing more than instantiating an enumerable class that decorates the input sequence.

Chaining Decorators

Chaining query operators creates a layering of decorators. Consider the following query:

```
IEnumerable<int> query = new int[] { 5, 12, 3 }.Where   (n => n < 10)
                                                .OrderBy (n => n)
                                                .Select  (n => n * 10);
```

Each query operator instantiates a new decorator that wraps the previous sequence (rather like a Russian nesting doll). Figure 8-4 illustrates the object model of this query. Note that this object model is fully constructed prior to any enumeration.

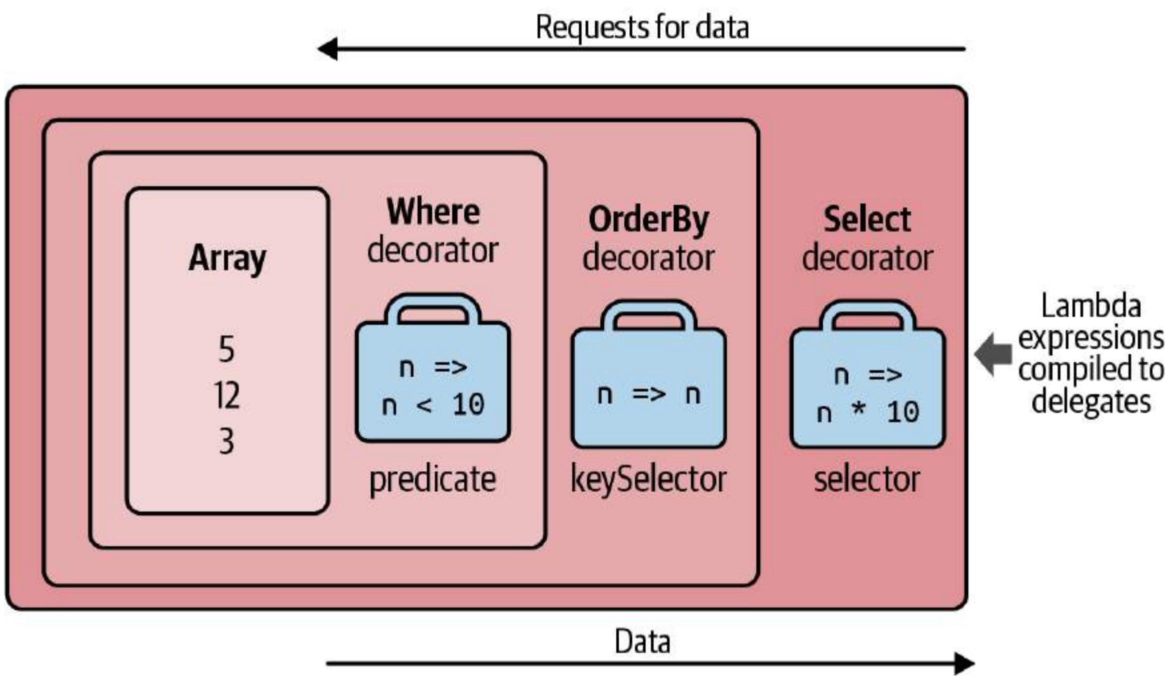

Figure 8-4. Layered decorator sequences

When you enumerate `query`, you're querying the original array, transformed through a layering or chain of decorators.

> **NOTE**
>
> Adding `ToList` onto the end of this query would cause the preceding operators to execute immediately, collapsing the whole object model into a single list.

Figure 8-5 shows the same object composition in Unified Modeling Language (UML) syntax. `Select`'s decorator references the `OrderBy` decorator, which references `Where`'s decorator, which references the array. A feature of deferred execution is that you build the identical object model if you compose the query progressively:

```
IEnumerable<int>
  source    = new int[] { 5, 12, 3 },
  filtered  = source   .Where   (n => n < 10),
  sorted    = filtered .OrderBy (n => n),
  query     = sorted   .Select  (n => n * 10);
```

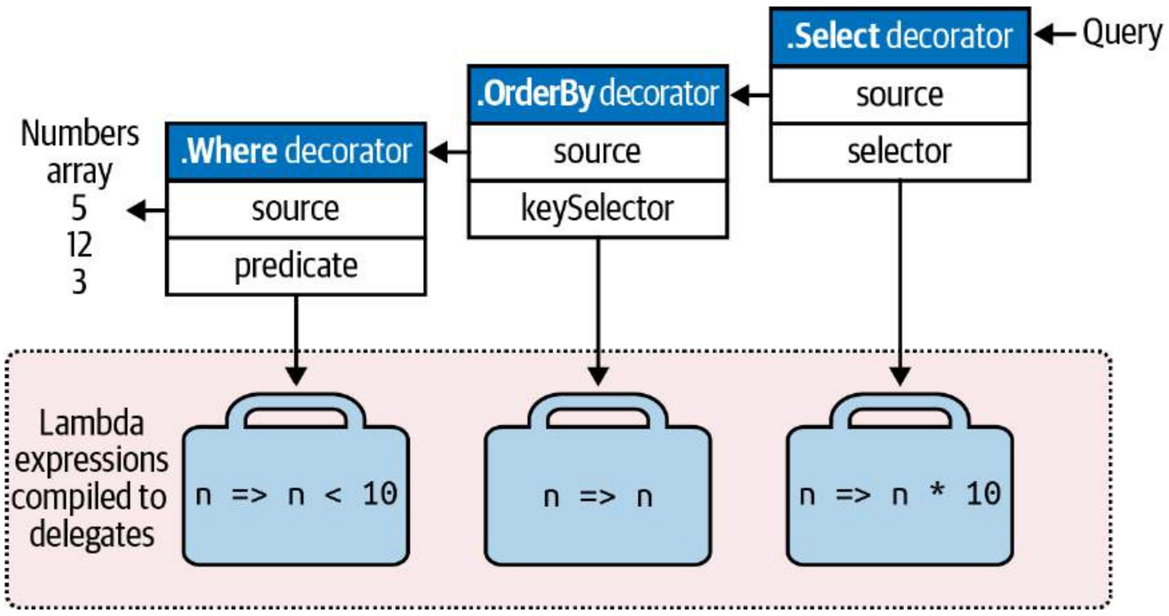

Figure 8-5. UML decorator composition

How Queries Are Executed

Here are the results of enumerating the preceding query:

```
foreach (int n in query) Console.WriteLine (n);

30
50
```

Behind the scenes, the `foreach` calls `GetEnumerator` on `Select`'s decorator (the last or outermost operator), which kicks off everything. The result is a chain of enumerators that structurally mirrors the chain of decorator sequences. Figure 8-6 illustrates the flow of execution as enumeration proceeds.

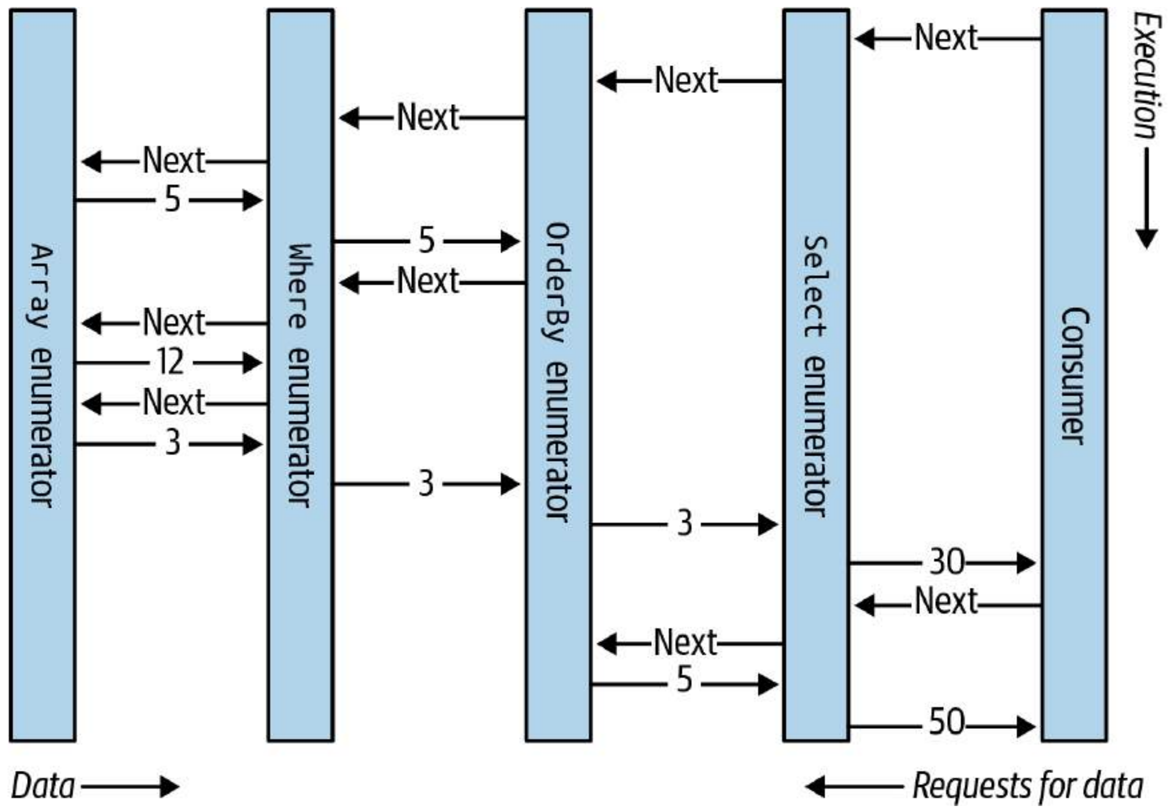

Figure 8-6. Execution of a local query

In the first section of this chapter, we depicted a query as a production line of conveyor belts. Extending this analogy, we can say a LINQ query is a lazy production line, where the conveyor belts roll elements only upon *demand*. Constructing a query constructs a production line—with everything in place—but with nothing rolling. Then, when the consumer requests an element (enumerates over the query), the rightmost conveyor belt activates; this in turn triggers the others to roll—as and when input sequence elements are needed. LINQ follows a demand-driven *pull* model, rather than a supply-driven *push* model. This is important—as you'll see later—in allowing LINQ to scale to querying SQL databases.

Subqueries

A *subquery* is a query contained within another query's lambda expression. The following example uses a subquery to sort musicians by their last name:

```
string[] musos =
  { "David Gilmour", "Roger Waters", "Rick Wright", "Nick Mason" };

IEnumerable<string> query = musos.OrderBy (m => m.Split().Last());
```

`m.Split` converts each string into a collection of words, upon which we then call the `Last` query operator. `m.Split().Last` is the subquery; `query` references the *outer query*.

Subqueries are permitted because you can put any valid C# expression on the righthand side of a lambda. A subquery is simply another C# expression. This means that the rules for subqueries are a consequence of the rules for lambda expressions (and the behavior of query operators in general).

NOTE

The term *subquery*, in the general sense, has a broader meaning. For the purpose of describing LINQ, we use the term only for a query referenced from within the lambda expression of another query. In a query expression, a subquery amounts to a query referenced from an expression in any clause except the `from` clause.

A subquery is privately scoped to the enclosing expression and can reference parameters in the outer lambda expression (or range variables in a query expression).

`m.Split().Last` is a very simple subquery. The next query retrieves all strings in an array whose length matches that of the shortest string:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> outerQuery = names
  .Where (n => n.Length == names.OrderBy (n2 => n2.Length)
                                .Select  (n2 => n2.Length).First());

// Tom, Jay
```

Here’s the same thing as a query expression:

```
IEnumerable<string> outerQuery =
  from   n in names
  where  n.Length ==
           (from n2 in names orderby n2.Length select n2.Length).First()
  select n;
```

Because the outer range variable (`n`) is in scope for a subquery, we cannot reuse `n` as the subquery’s range variable.

A subquery is executed whenever the enclosing lambda expression is evaluated. This means that a subquery is executed upon demand, at the discretion of the outer query. You could say that execution proceeds from the *outside in*. Local queries follow this model literally; interpreted queries (e.g., database queries) follow this model *conceptually*.

The subquery executes as and when required, to feed the outer query. As Figures 8-7 and 8-8 illustrate, the subquery in our example (the top conveyor belt in Figure 8-7) executes once for every outer loop iteration.

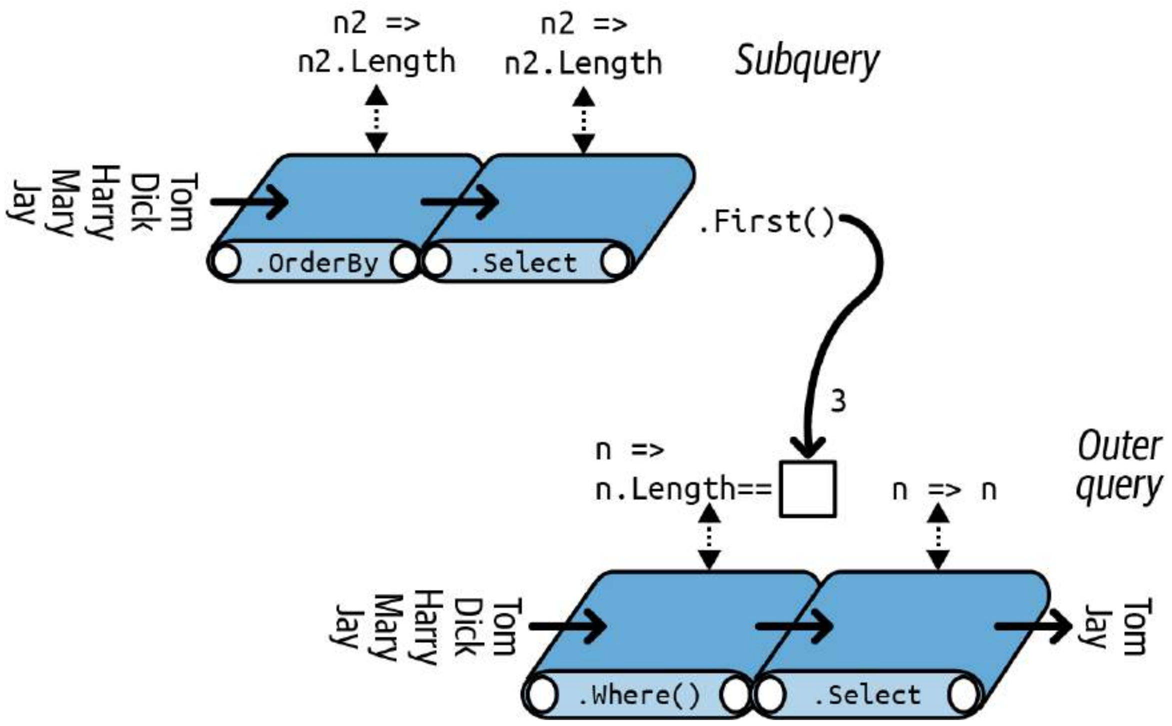

Figure 8-7. Subquery composition

We can express our preceding subquery more succinctly as follows:

```
IEnumerable<string> query =
  from   n in names
  where  n.Length == names.OrderBy (n2 => n2.Length).First().Length
  select n;
```

With the `Min` aggregation function, we can simplify the query further:

```
IEnumerable<string> query =
  from   n in names
  where  n.Length == names.Min (n2 => n2.Length)
  select n;
```

In "Interpreted Queries", we describe how remote sources such as SQL tables can be queried. Our example makes an ideal database query because it would be processed as a unit, requiring only one round trip to the database server. This query, however, is inefficient for a local collection because the subquery is recalculated on each outer loop iteration. We can avoid this inefficiency by running the subquery separately (so that it's no longer a subquery):

```
int shortest = names.Min (n => n.Length);

IEnumerable<string> query = from   n in names
                            where  n.Length == shortest
                            select n;
```

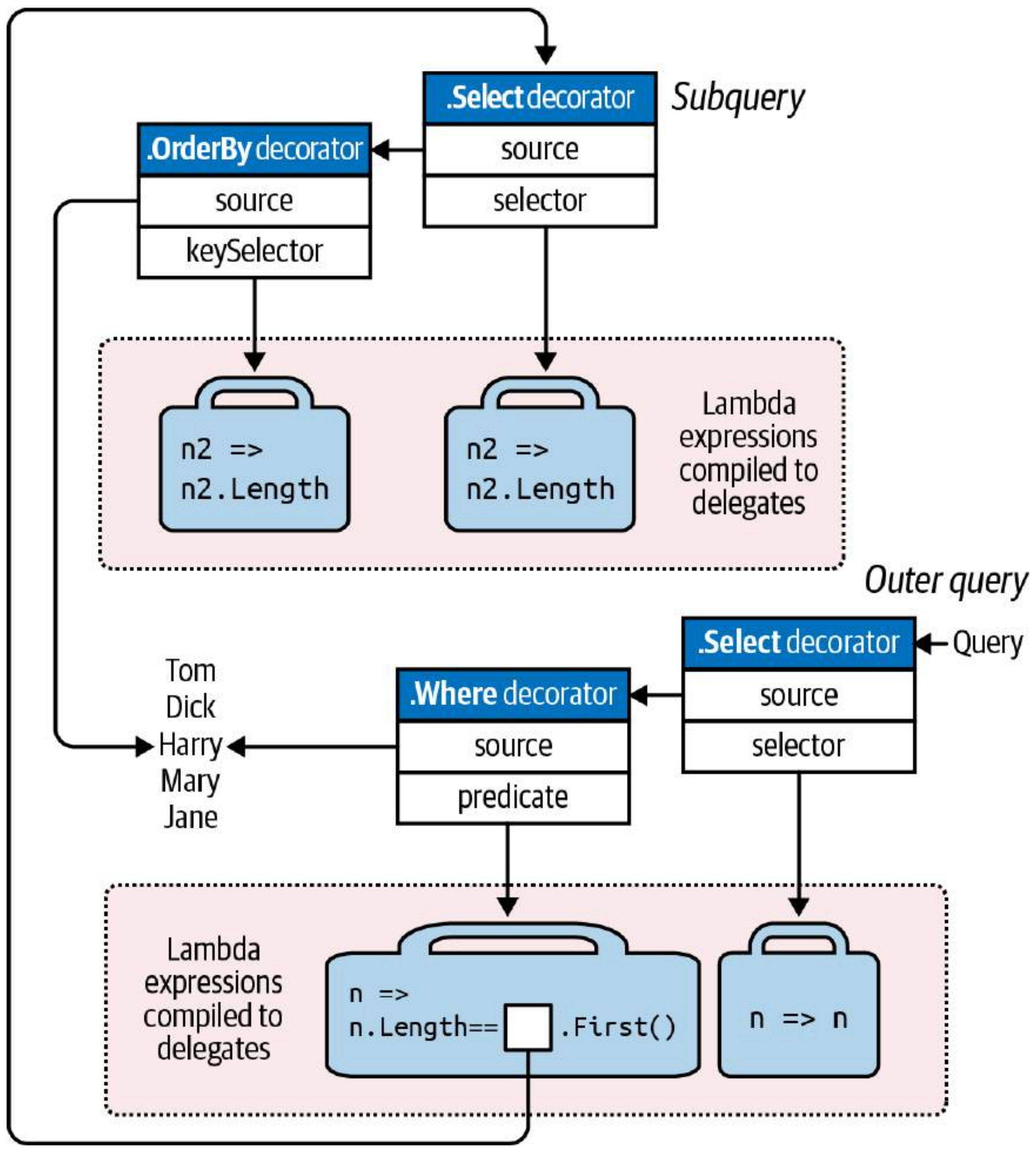

Figure 8-8. UML subquery composition

NOTE

Factoring out subqueries in this manner is nearly always desirable when querying local collections. An exception is when the subquery is *correlated*, meaning that it references the outer range variable. We explore correlated subqueries in "Projecting".

Subqueries and Deferred Execution

An element or aggregation operator such as `First` or `Count` in a subquery doesn't force the *outer* query into immediate execution—deferred execution still holds for the outer query. This is because subqueries are called *indirectly*—through a delegate in the case of a local query, or through an expression tree in the case of an interpreted query.

An interesting case arises when you include a subquery within a `Select` expression. In the case of a local query, you're actually *projecting a sequence of queries*—each itself subject to deferred execution. The effect is generally transparent, and it serves to further improve efficiency. We revisit `Select` subqueries in some detail in Chapter 9.

Composition Strategies

In this section, we describe three strategies for building more complex queries:

- Progressive query construction
- Using the `into` keyword
- Wrapping queries

All are *chaining* strategies and produce identical runtime queries.

Progressive Query Building

At the start of the chapter, we demonstrated how you could build a fluent query progressively:

```
var filtered   = names    .Where   (n => n.Contains ("a"));
var sorted     = filtered .OrderBy (n => n);
var query      = sorted   .Select  (n => n.ToUpper());
```

Because each of the participating query operators returns a decorator sequence, the resultant query is the same chain or layering of decorators that you would get from a single-expression query. There are a couple of potential benefits, however, to building queries progressively:

- It can make queries easier to write.
- You can add query operators *conditionally*. For example,

  ```
  if (includeFilter) query = query.Where (...)
  ```

  is more efficient than

  ```
  query = query.Where (n => !includeFilter || <expression>)
  ```

  because it avoids adding an extra query operator if `includeFilter` is false.

A progressive approach is often useful in query comprehensions. Imagine that we want to remove all vowels from a list of names and then present in alphabetical order those whose length is still more than two characters. In fluent syntax, we could write this query as a single expression—by projecting *before* we filter:

```
IEnumerable<string> query = names
  .Select  (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", ""))
  .Where   (n => n.Length > 2)
  .OrderBy (n => n);

// Dck
// Hrry
// Mry
```

NOTE

Rather than calling `string`'s `Replace` method five times, we could remove vowels from a string more efficiently with a regular expression:

```
n => Regex.Replace (n, "[aeiou]", "")
```

`string`'s `Replace` method has the advantage, though, of also working in database queries.

Translating this directly into a query expression is troublesome because the `select` clause must come after the `where` and `orderby` clauses. And if we rearrange the query so as to project last, the result would be different:

```
IEnumerable<string> query =
  from    n in names
  where   n.Length > 2
  orderby n
  select  n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
           .Replace ("o", "").Replace ("u", "");

// Dck
// Hrry
// Jy
// Mry
// Tm
```

Fortunately, there are a number of ways to get the original result in query syntax. The first is by querying progressively:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "");

query = from n in query where n.Length > 2 orderby n select n;

// Dck
// Hrry
// Mry
```

The into Keyword

> **NOTE**
>
> The `into` keyword is interpreted in two very different ways by query expressions, depending on context. The meaning we're describing now is for signaling *query continuation* (the other is for signaling a `GroupJoin`).

The `into` keyword lets you "continue" a query after a projection and is a shortcut for progressively querying. With `into`, we can rewrite the preceding query as follows:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "")
  into noVowel
    where noVowel.Length > 2 orderby noVowel select noVowel;
```

The only place you can use `into` is after a `select` or `group` clause. `into` "restarts" a query, allowing you to introduce fresh `where`, `orderby`, and `select` clauses.

> **NOTE**
>
> Although it's easiest to think of `into` as restarting a query from the perspective of a query expression, it's *all one query* when translated to its final fluent form. Hence, there's no intrinsic performance hit with `into`. Nor do you lose any points for its use!

The equivalent of `into` in fluent syntax is simply a longer chain of operators.

Scoping rules

All range variables are out of scope following an `into` keyword. The following will not compile:

```
var query =
  from n1 in names
  select n1.ToUpper()
  into n2                             // Only n2 is visible from here on.
    where n1.Contains ("x")           // Illegal: n1 is not in scope.
    select n2;
```

To see why, consider how this maps to fluent syntax:

```
var query = names
  .Select (n1 => n1.ToUpper())
  .Where  (n2 => n1.Contains ("x"));     // Error: n1 no longer in scope
```

The original name (`n1`) is lost by the time the `Where` filter runs. `Where`'s input sequence contains only uppercase names, so it cannot filter based on `n1`.

Wrapping Queries

A query built progressively can be formulated into a single statement by wrapping one query around another. In general terms,

```
var tempQuery = tempQueryExpr
var finalQuery = from ... in tempQuery ...
```

can be reformulated as:

```
var finalQuery = from ... in (tempQueryExpr)
```

Wrapping is semantically identical to progressive query building or using the `into` keyword (without the intermediate variable). The end result in all cases is a linear chain of query operators. For example, consider the following query:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "");

query = from n in query where n.Length > 2 orderby n select n;
```

Reformulated in wrapped form, it's the following:

```
IEnumerable<string> query =
  from n1 in
  (
    from   n2 in names
    select n2.Replace ("a", "").Replace ("e", "").Replace ("i", "")
             .Replace ("o", "").Replace ("u", "")
  )
  where n1.Length > 2 orderby n1 select n1;
```

When converted to fluent syntax, the result is the same linear chain of operators as in previous examples:

```
IEnumerable<string> query = names
  .Select  (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", ""))
  .Where   (n => n.Length > 2)
  .OrderBy (n => n);
```

(The compiler does not emit the final `.Select (n => n)`, because it's redundant.)

Wrapped queries can be confusing because they resemble the *subqueries* we wrote earlier. Both have the concept of an inner and outer query. When converted to fluent syntax, however, you can see that wrapping is simply a strategy for sequentially chaining operators. The end result bears no resemblance to a subquery, which embeds an inner query within the *lambda expression* of another.

Returning to a previous analogy: when wrapping, the “inner” query amounts to the *preceding conveyor belts*. In contrast, a subquery rides above a conveyor belt and is activated upon demand through the conveyor belt’s lambda worker (as illustrated in Figure 8-7).

Projection Strategies

Object Initializers

So far, all of our `select` clauses have projected scalar element types. With C# object initializers, you can project into more complex types. For example, suppose, as a first step in a query, we want to strip vowels from a list of names while still retaining the original versions alongside, for the benefit of subsequent queries. We can write the following class to assist:

```
class TempProjectionItem
{
  public string Original;    // Original name
  public string Vowelless;   // Vowel-stripped name
}
```

We then can project into it with object initializers:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<TempProjectionItem> temp =
  from n in names
  select new TempProjectionItem
  {
    Original  = n,
    Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
  };
```

The result is of type `IEnumerable<TempProjectionItem>`, which we can subsequently query:

```
IEnumerable<string> query = from   item in temp
                            where  item.Vowelless.Length > 2
                            select item.Original;
// Dick
// Harry
// Mary
```

Anonymous Types

Anonymous types allow you to structure your intermediate results without writing special classes. We can eliminate the `TempProjectionItem` class in our previous example with anonymous types:

```
var intermediate = from n in names

  select new
  {
    Original = n,
    Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
  };

IEnumerable<string> query = from   item in intermediate
                            where  item.Vowelless.Length > 2
                            select item.Original;
```

This gives the same result as the previous example, but without needing to write a one-off class. The compiler does the job instead, generating a temporary class with fields that match the structure of our projection. This means, however, that the `intermediate` query has the following type:

```
IEnumerable <random-compiler-generated-name>
```

The only way we can declare a variable of this type is with the `var` keyword. In this case, `var` is more than just a clutter reduction device; it's a necessity.

We can write the entire query more succinctly with the `into` keyword:

```
var query = from n in names
  select new
  {
     Original = n,
     Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
  }
  into temp
  where temp.Vowelless.Length > 2
  select temp.Original;
```

Query expressions provide a shortcut for writing this kind of query: the `let` keyword.

The let Keyword

The `let` keyword introduces a new variable alongside the range variable.

With `let`, we can write a query extracting strings whose length, excluding vowels, exceeds two characters, as follows:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query =
  from n in names
  let vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", "")
  where vowelless.Length > 2
  orderby vowelless
  select n;       // Thanks to let, n is still in scope.
```

The compiler resolves a `let` clause by projecting into a temporary anonymous type that contains both the range variable and the new expression variable. In other words, the compiler translates this query into the preceding example.

`let` accomplishes two things:

- It projects new elements alongside existing elements.
- It allows an expression to be used repeatedly in a query without being rewritten.

The `let` approach is particularly advantageous in this example because it allows the `select` clause to project either the original name (`n`) or its vowel-removed version (`vowelless`).

You can have any number of `let` statements before or after a `where` statement (see Figure 8-2). A `let` statement can reference variables introduced in earlier `let` statements (subject to the boundaries imposed by an `into` clause). `let` *reprojects* all existing variables transparently.

A `let` expression need not evaluate to a scalar type: sometimes it's useful to have it evaluate to a subsequence, for instance.

Interpreted Queries

LINQ provides two parallel architectures: *local* queries for local object collections and *interpreted* queries for remote data sources. So far, we've examined the architecture of local queries, which operate over collections implementing `IEnumerable<T>`. Local queries resolve to query operators in the `Enumerable` class (by default), which in turn resolve to chains of decorator sequences. The delegates that they accept—whether expressed in query syntax, fluent syntax, or traditional delegates—are fully local to Intermediate Language (IL) code, just like any other C# method.

By contrast, interpreted queries are *descriptive*. They operate over sequences that implement `IQueryable<T>`, and they resolve to the query operators in the `Queryable` class, which emit *expression trees* that are interpreted at runtime. These expression trees can be translated, for instance, to SQL queries, allowing you to use LINQ to query a database.

NOTE

The query operators in `Enumerable` can actually work with `IQueryable<T>` sequences. The difficulty is that the resultant queries always execute locally on the client. This is why a second set of query operators is provided in the `Queryable` class.

To write interpreted queries, you need to start with an API that exposes sequences of type `IQueryable<T>`. An example is Microsoft's *Entity Framework Core* (EF Core), which allows you to query a variety of databases, including SQL Server, Oracle, MySQL, PostgreSQL, and SQLite.

It's also possible to generate an `IQueryable<T>` wrapper around an ordinary enumerable collection by calling the `AsQueryable` method. We describe `AsQueryable` in "Building Query Expressions".

NOTE

`IQueryable<T>` is an extension of `IEnumerable<T>` with additional methods for constructing expression trees. Most of the time you can ignore the details of these methods; they're called indirectly by the runtime. "Building Query Expressions" covers `IQueryable<T>` in more detail.

To illustrate, let's create a simple customer table in SQL Server and populate it with a few names using the following SQL script:

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30)
)
insert Customer values (1, 'Tom')
insert Customer values (2, 'Dick')
insert Customer values (3, 'Harry')
insert Customer values (4, 'Mary')
insert Customer values (5, 'Jay')
```

With this table in place, we can write an interpreted LINQ query in C# that uses EF Core to retrieve customers whose name contains the letter “a,” as follows:

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;

using var dbContext = new NutshellContext();

IQueryable<string> query = from c in dbContext.Customers
  where   c.Name.Contains ("a")
  orderby c.Name.Length
  select  c.Name.ToUpper();

foreach (string name in query) Console.WriteLine (name);

public class Customer
{
  public int ID { get; set; }
  public string Name { get; set; }
}

// We’ll explain the following class in more detail in the next section.
public class NutshellContext : DbContext
{
  public virtual DbSet<Customer> Customers { get; set; }

  protected override void OnConfiguring (DbContextOptionsBuilder builder)
    => builder.UseSqlServer ("...connection string...");

  protected override void OnModelCreating (ModelBuilder modelBuilder)
    => modelBuilder.Entity<Customer>().ToTable ("Customer")
                                      .HasKey (c => c.ID);
}
```

EF Core translates this query into the following SQL:

```
SELECT UPPER([c].[Name])
FROM [Customers] AS [c]
WHERE CHARINDEX(N'a', [c].[Name]) > 0
ORDER BY CAST(LEN([c].[Name]) AS int)
```

Here's the end result:

```
// JAY
// MARY
// HARRY
```

How Interpreted Queries Work

Let's examine how the preceding query is processed.

First, the compiler converts query syntax to fluent syntax. This is done exactly as with local queries:

```
IQueryable<string> query = dbContext.customers
                                    .Where   (n => n.Name.Contains ("a"))
                                    .OrderBy (n => n.Name.Length)
                                    .Select  (n => n.Name.ToUpper());
```

Next, the compiler resolves the query operator methods. Here's where local and interpreted queries differ—interpreted queries resolve to query operators in the `Queryable` class instead of the `Enumerable` class.

To see why, we need to look at the `dbContext.Customers` variable, the source upon which the entire query builds. `dbContext.Customers` is of type `DbSet<T>`, which implements `IQueryable<T>` (a subtype of `IEnumerable<T>`). This means that the compiler has a choice in resolving `Where`: it could call the extension method in `Enumerable` or the following extension method in `Queryable`:

```
public static IQueryable<TSource> Where<TSource> (this
  IQueryable<TSource> source, Expression <Func<TSource,bool>> predicate)
```

The compiler chooses `Queryable.Where` because its signature is a *more specific match*.

`Queryable.Where` accepts a predicate wrapped in an `Expression<TDelegate>` type. This instructs the compiler to translate the supplied lambda expression—in other words, `n=>n.Name.Contains("a")`—to an *expression tree* rather than a compiled delegate. An expression tree is an object model based on the types in `System.Linq.Expressions` that can be inspected at runtime (so that EF Core can later translate it to an SQL statement).

Because `Queryable.Where` also returns `IQueryable<T>`, the same process follows with the `OrderBy` and `Select` operators. Figure 8-9 illustrates the end result. In the shaded box, there is an *expression tree* describing the entire query, which can be traversed at runtime.

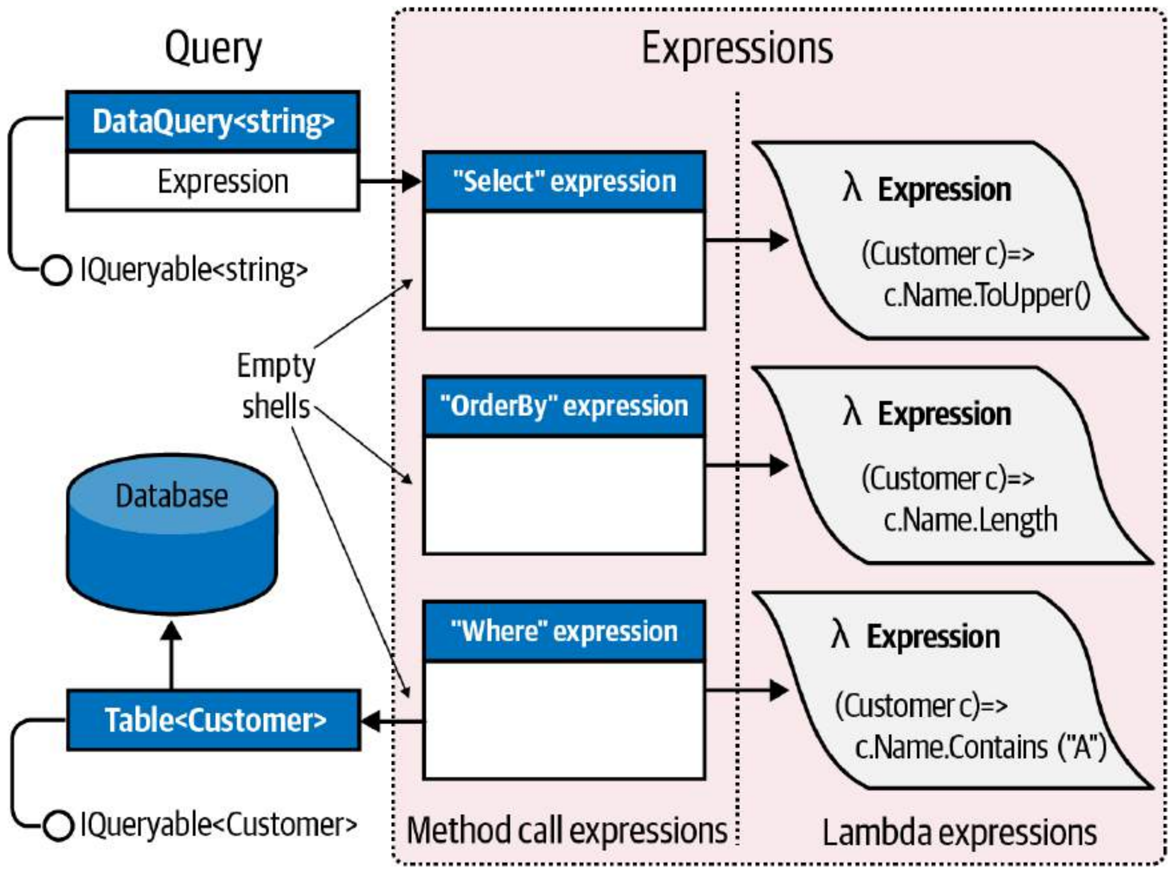

Figure 8-9. Interpreted query composition

Execution

Interpreted queries follow a deferred execution model—just like local queries. This means that the SQL statement is not generated until you start enumerating the query. Further, enumerating the same query twice results in the database being queried twice.

Under the hood, interpreted queries differ from local queries in how they execute. When you enumerate over an interpreted query, the outermost sequence runs a program that traverses the entire expression tree, processing it as a unit. In our example, EF Core translates the expression tree to an SQL statement, which it then executes, yielding the results as a sequence.

> **NOTE**
>
> To work, EF Core needs to understand the schema of the database. It does this by leveraging conventions, code attributes, and a fluent configuration API. We'll explore this in detail later in the chapter.

We said previously that a LINQ query is like a production line. However, when you enumerate an `IQueryable` conveyor belt, it doesn't start up the whole production line, like with a local query. Instead, just the `IQueryable` belt starts up, with a special enumerator that calls upon a production manager. The manager reviews the entire production line—which consists not of compiled code but of *dummies* (method call expressions) with instructions pasted to their *foreheads* (expression trees). The manager then traverses all the expressions, in this case transcribing them to a single piece of paper (an SQL statement), which it then executes, feeding the results back to the consumer. Only one belt turns; the rest of the production line is a network of empty shells, existing just to describe what needs to be done.

This has some practical implications. For instance, with local queries, you can write your own query methods (fairly easily, with iterators) and then use them to supplement the predefined set. With remote queries, this is difficult and even undesirable. If you wrote a `MyWhere` extension method accepting `IQueryable<T>`, it would be like putting your own dummy into the production line. The production manager wouldn't know what to do with your dummy. Even if you intervened at this stage, your solution would be hardwired to a particular provider, such as EF Core, and would not work with other `IQueryable` implementations. Part of the benefit of having a standard set of methods in `Queryable` is that they define a *standard vocabulary* for querying *any* remote collection. As soon as you try to extend the vocabulary, you're no longer interoperable.

Another consequence of this model is that an `IQueryable` provider might be unable to cope with some queries—even if you stick to the standard methods. EF Core is limited by the capabilities of the database server; some LINQ queries have no SQL translation. If you're familiar with SQL, you'll have a good intuition for what these are, although at times you'll need to experiment to see what causes a runtime error; it can be surprising what *does* work!

Combining Interpreted and Local Queries

A query can include both interpreted and local operators. A typical pattern is to have the local operators on the *outside* and the interpreted components on the *inside*; in other words, the interpreted queries feed the local queries. This pattern works well when querying a database.

For instance, suppose that we write a custom extension method to pair up strings in a collection:

```
public static IEnumerable<string> Pair (this IEnumerable<string> source)
{
  string firstHalf = null;
  foreach (string element in source)
    if (firstHalf == null)
      firstHalf = element;
    else
    {
      yield return firstHalf + ", " + element;
```

```
        firstHalf = null;
      }
  }
```

We can use this extension method in a query that mixes EF Core and local operators:

```
using var dbContext = new NutshellContext ();
IEnumerable<string> q = dbContext.Customers
  .Select (c => c.Name.ToUpper())
  .OrderBy (n => n)
  .Pair()                         // Local from this point on.
  .Select ((n, i) => "Pair " + i.ToString() + " = " + n);

foreach (string element in q) Console.WriteLine (element);

// Pair 0 = DICK, HARRY
// Pair 1 = JAY, MARY
```

Because `dbContext.Customers` is of a type implementing `IQueryable<T>`, the `Select` operator resolves to `Queryable.Select`. This returns an output sequence also of type `IQueryable<T>`, so the `OrderBy` operator similarly resolves to `Queryable.OrderBy`. But the next query operator, `Pair`, has no overload accepting `IQueryable<T>`—only the less specific `IEnumerable<T>`. So, it resolves to our local `Pair` method—wrapping the interpreted query in a local query. `Pair` also returns `IEnumerable`, so the `Select` that follows resolves to another local operator.

On the EF Core side, the resulting SQL statement is equivalent to this:

```
SELECT UPPER([c].[Name]) FROM [Customers] AS [c] ORDER BY UPPER([c].[Name])
```

The remaining work is done locally. In effect, we end up with a local query (on the outside) whose source is an interpreted query (the inside).

AsEnumerable

`Enumerable.AsEnumerable` is the simplest of all query operators. Here's its complete definition:

```
public static IEnumerable<TSource> AsEnumerable<TSource>
              (this IEnumerable<TSource> source)
{
    return source;
}
```

Its purpose is to cast an `IQueryable<T>` sequence to `IEnumerable<T>`, forcing subsequent query operators to bind to `Enumerable` operators instead of `Queryable` operators. This causes the remainder of the query to execute locally.

To illustrate, suppose that we had a `MedicalArticles` table in SQL Server and wanted to use EF Core to retrieve all articles on influenza whose abstract contained fewer than 100 words. For the latter predicate, we need a regular expression:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

using var dbContext = new NutshellContext ();

var query = dbContext.MedicalArticles
  .Where (article => article.Topic == "influenza" &&
                     wordCounter.Matches (article.Abstract).Count < 100);
```

The problem is that SQL Server doesn't support regular expressions, so EF Core will throw an exception, complaining that the query cannot be translated to SQL. We can solve this by querying in two steps: first retrieving all articles on influenza through an EF Core query, and then filtering *locally* for abstracts of fewer than 100 words:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

using var dbContext = new NutshellContext ();
```

```
IEnumerable<MedicalArticle> efQuery = dbContext.MedicalArticles
  .Where (article => article.Topic == "influenza");

IEnumerable<MedicalArticle> localQuery = efQuery
  .Where (article => wordCounter.Matches (article.Abstract).Count < 100);
```

Because `efQuery` is of type `IEnumerable<MedicalArticle>`, the second query binds to the local query operators, forcing that part of the filtering to run on the client.

With `AsEnumerable`, we can do the same in a single query:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

using var dbContext = new NutshellContext ();

var query = dbContext.MedicalArticles
  .Where (article => article.Topic == "influenza")

  .AsEnumerable()
  .Where (article => wordCounter.Matches (article.Abstract).Count < 100);
```

An alternative to calling `AsEnumerable` is to call `ToArray` or `ToList`. The advantage of `AsEnumerable` is that it doesn't force immediate query execution, nor does it create any storage structure.

NOTE

Moving query processing from the database server to the client can hurt performance, especially if it means retrieving more rows. A more efficient (though more complex) way to solve our example would be to use SQL CLR integration to expose a function on the database that implemented the regular expression.

We further demonstrate combined interpreted and local queries in Chapter 10.

EF Core

Throughout this and Chapter 9, we use EF Core to demonstrate interpreted queries. Let's now examine the key features of this technology.

EF Core Entity Classes

EF Core lets you use any class to represent data, as long as it contains a public property for each column that you want to query.

For instance, we could define the following entity class to query and update a *Customers* table in the database:

```
public class Customer
{
  public int ID { get; set; }
  public string Name { get; set; }
}
```

DbContext

After defining entity classes, the next step is to subclass `DbContext`. An instance of that class represents your sessions working with the database. Typically, your `DbContext` subclass will contain one `DbSet<T>` property for each entity in your model:

```
public class NutshellContext : DbContext
{
  public DbSet<Customer> Customers { get; set; }
  ... properties for other tables ...

}
```

A `DbContext` object does three things:

- It acts as a factory for generating `DbSet<>` objects that you can query.

- It keeps track of any changes that you make to your entities so that you can write them back (see "Change Tracking").
- It provides virtual methods that you can override to configure the connection and model.

Configuring the connection

By overriding the `OnConfiguring` method, you can specify the database provider and connection string:

```
public class NutshellContext : DbContext
{
  ...
  protected override void OnConfiguring (DbContextOptionsBuilder
                                          optionsBuilder) =>
    optionsBuilder.UseSqlServer
      (@"Server=(local);Database=Nutshell;Trusted_Connection=True");
}
```

In this example, the connection string is specified as a string literal. Production applications would typically retrieve it from a configuration file such as *appsettings.json*.

`UseSqlServer` is an extension method defined in an assembly that's part of the *Microsoft.EntityFramework.SqlServer* NuGet package. Packages are available for other database providers, including Oracle, MySQL, PostgreSQL, and SQLite.

NOTE

If you’re using ASP.NET, you can allow its dependency injection framework to preconfigure `optionsBuilder`; in most cases, this lets you avoid overriding `OnConfiguring` altogether. To enable this, define a constructor on `DbContext` as follows:

```
public NutshellContext (DbContextOptions<NutshellContext>
                        options)
  : base(options) { }
```

If you do choose to override `OnConfiguring` (perhaps to provide a configuration if your `DbContext` is used in another scenario), you can check whether options have already been configured as follows:

```
protected override void OnConfiguring (
  DbContextOptionsBuilder optionsBuilder)
{
  if (!optionsBuilder.IsConfigured)
  {
    ...
  }
}
```

In the `OnConfiguring` method, you can enable other options, including lazy loading (see “Lazy loading”).

Configuring the model

By default, EF Core is *convention based*, meaning that it infers the database schema from your class and property names.

You can override the defaults using the *fluent api* by overriding `OnModelCreating` and calling extension methods on the `ModelBuilder` parameter. For example, we can explicitly specify the database table name for our `Customer` entity as follows:

```
protected override void OnModelCreating (ModelBuilder modelBuilder) =>
  modelBuilder.Entity<Customer>()
    .ToTable ("Customer");   // Table is called 'Customer'
```

Without this code, EF Core would map this entity to a table named "Customers" rather than "Customer", because we have a `DbSet<Customer>` property in our `DbContext` called `Customers`:

```
public DbSet<Customer> Customers { get; set; }
```

NOTE

The following code maps all of your entities to table names that match the entity *class name* (which is typically singular) rather than the `DbSet<T>` *property name* (which is typically plural):

```
protected override void OnModelCreating (ModelBuilder
                                         modelBuilder)
{
  foreach (IMutableEntityType entityType in
           modelBuilder.Model.GetEntityTypes())
  {
    modelBuilder.Entity (entityType.Name)
      .ToTable (entityType.ClrType.Name);
  }
}
```

The fluent API offers an expanded syntax for configuring columns. In the next example, we use two popular methods:

- `HasColumnName`, which maps a property to a differently named column
- `IsRequired`, which indicates that a column is not nullable

```
protected override void OnModelCreating (ModelBuilder modelBuilder) =>
```

```
modelBuilder.Entity<Customer> (entity =>
{
    entity.ToTable ("Customer");
    entity.Property (e => e.Name)
          .HasColumnName ("Full Name")  // Column name is 'Full Name'
          .IsRequired();                // Column is not nullable
});
```

Table 8-1 lists some of the most important methods in the fluent API.

NOTE

Instead of using the fluent API, you can configure your model by applying special attributes to your entity classes and properties (“data annotations”). This approach is less flexible in that the configuration must be fixed at compile-time, and is less powerful in that there are some options that can be configured only via the fluent API.

Table 8-1. Fluent API model configuration methods

Method	Purpose	Example

`ToTable`	Specify the database table name for a given entity	`builder`
		`  .Entity<Customer>()`
		`  .ToTable("Customer");`

`HasColumnName`	Specify the column name for a given property	`builder.Entity<Customer>()`
		`  .Property(c => c.Name)`
		`  .HasColumnName("Full Name");`

`HasKey`	Specify a key (usually that deviates from convention)	`builder.Entity<Customer>()`
		`  .HasKey(c => c.CustomerNr);`

`IsRequired`	Specify that the property requires a value (is not nullable)	`builder.Entity<Customer>()`
		`  .Property(c => c.Name)`
		`  .IsRequired();`

`HasMaxLength`	Specify the maximum length of a variable-length type (usually a string) whose width can vary	`builder.Entity<Customer>()`
		`  .Property(c => c.Name)`
		`  .HasMaxLength(60);`

`HasColumnType`	Specify the database data type for a column	`builder.Entity<Purchase>()`
		`  .Property(p =>`
		`p.Description)`
		
		`.HasColumnType("varchar(80)");`

`Ignore`	Ignore a type	`builder.Ignore<Products>();`

`Ignore`	Ignore a property of a type	`builder.Entity<Customer>()`
		`  .Ignore(c => c.ChatName);`

Method	Purpose	Example

`HasIndex`	Specify a property (or combination of properties) should serve in the database as an index	`// Compound index:`
		`builder.Entity<Purchase>()`
		`  .HasIndex(p =>`
		`     new { p.Date, p.Price });`
		
		`// Unique index on one`
		`property`
		`builder`
		`  .Entity<MedicalArticle>()`
		`  .HasIndex(a => a.Topic)`
		`  .IsUnique();`

`HasOne`	See "Navigation Properties"	`builder.Entity<Purchase>()`
		`  .HasOne(p => p.Customer)`
		`  .WithMany(c => c.Purchases);`

`HasMany`	See "Navigation Properties"	`builder.Entity<Customer>()`
		`  .HasMany(c => c.Purchases)`
		`  .WithOne(p => p.Customer);`

Creating the database

EF Core supports a *code-first* approach, which means that you can start by defining entity classes and then ask EF Core to create the database. The easiest way to do the latter is to call the following method on a `DbContext` instance:

```
dbContext.Database.EnsureCreated();
```

A better approach, however, is to use EF Core's *migrations* feature, which not only creates the database but configures it such that EF Core can automatically update the schema in the future when your entity classes change. You can enable migrations in Visual Studio's Package Manager Console and ask it to create the database with the following commands:

```
Install-Package Microsoft.EntityFrameworkCore.Tools
Add-Migration InitialCreate
Update-Database
```

The first command installs tools to manage EF Core from within Visual Studio. The second command generates a special C# class known as a code migration that contains instructions to create the database. The final command runs those instructions against the database connection string specified in the project's application configuration file.

Using DbContext

After you've defined Entity classes and subclassed `DbContext`, you can instantiate your `DbContext` and query the database, as follows:

```
using var dbContext = new NutshellContext();
Console.WriteLine (dbContext.Customers.Count());
// Executes "SELECT COUNT(*) FROM [Customer] AS [c]"
```

You can also use your `DbContext` instance to write to the database. The following code inserts a row into the Customer table:

```
using var dbContext = new NutshellContext();
Customer cust = new Customer()
{
  Name = "Sara Wells"
};
dbContext.Customers.Add (cust);
dbContext.SaveChanges();    // Writes changes back to database
```

The following queries the database for the customer that was just inserted:

```
using var dbContext = new NutshellContext();
Customer cust = dbContext.Customers
  .Single (c => c.Name == "Sara Wells")
```

The following updates that customer's name and writes the change to the database:

```
cust.Name = "Dr. Sara Wells";
dbContext.SaveChanges();
```

NOTE

The `Single` operator is ideal for retrieving a row by primary key. Unlike `First`, it throws an exception if more than one element is returned.

Object Tracking

A `DbContext` instance keeps track of all the entities it instantiates, so it can feed the same ones back to you whenever you request the same rows in a table. In other words, a context in its lifetime will never emit two separate entities that refer to the same row in a table (where a row is identified by primary key). This capability is called *object tracking*.

To illustrate, suppose the customer whose name is alphabetically first also has the lowest ID. In the following example, `a` and `b` will reference the same object:

```
using var dbContext = new NutshellContext ();

Customer a = dbContext.Customers.OrderBy (c => c.Name).First();
Customer b = dbContext.Customers.OrderBy (c => c.ID).First();
```

DISPOSING DBCONTEXT

Although `DbContext` implements `IDisposable`, you can (in general) get away with not disposing instances. Disposing forces the context's connection to dispose—but this is usually unnecessary because EF Core closes connections automatically whenever you finish retrieving results from a query.

Disposing a context prematurely can actually be problematic because of lazy evaluation. Consider the following:

```
IQueryable<Customer> GetCustomers (string prefix)
{
  using (var dbContext = new NutshellContext ())
    return dbContext.Customers
                    .Where (c => c.Name.StartsWith (prefix));
}
...
foreach (Customer c in GetCustomers ("a"))
  Console.WriteLine (c.Name);
```

This will fail because the query is evaluated when we enumerate it—which is *after* disposing its `DbContext`.

There are some caveats, though, on not disposing contexts:

- It relies on the connection object releasing all unmanaged resources on the `Close` method. Even though this holds true with `SqlConnection`, it's theoretically possible for a third-party connection to keep resources open if you call `Close` but not `Dispose` (though this would arguably violate the contract defined by `IDbConnection.Close`).

- If you manually call `GetEnumerator` on a query (instead of using `foreach`) and then fail to either dispose the enumerator or consume the sequence, the connection will remain open. Disposing the `DbContext` provides a backup in such scenarios.

- Some people feel that it's tidier to dispose contexts (and all objects that implement `IDisposable`).

If you want to explicitly dispose contexts, you must pass a `DbContext` instance into methods such as `GetCustomers` to avoid the problem described. In scenarios such as ASP.NET Core MVC where the context instance is provided via dependency injection (DI), the DI infrastructure will manage the context lifetime. It will be created when a unit of work (such as an HTTP request processed in the controller) begins and disposed when that unit of work ends.

Consider what happens when EF Core encounters the second query. It starts by querying the database—and obtaining a single row. It then reads the primary key of this row and performs a lookup in the context's entity cache. Seeing a match, it returns the existing object *without updating any values*. So, if another user had just updated that customer's `Name` in the database, the new value would be ignored. This is essential for avoiding unexpected side effects (the `Customer` object could be in use elsewhere) and also for managing concurrency. If you had altered properties on the `Customer` object and not yet called `SaveChanges`, you wouldn't want your properties automatically overwritten.

NOTE

You can disable object tracking by chaining the `AsNoTracking` extension method to your query or by setting `ChangeTracker.QueryTrackingBehavior` on the context to `QueryTrackingBehavior.NoTracking`. No-tracking queries are useful when data is used read-only as it improves performance and reduces memory use.

To get fresh information from the database, you must either instantiate a new context or call the `Reload` method, as follows:

```
dbContext.Entry (myCustomer).Reload();
```

The best practice is to use a fresh `DbContext` instance per unit of work so that the need to manually reload an entity is rare.

Change Tracking

When you change a property value in an entity loaded via `DbContext`, EF Core recognizes the change and updates the database accordingly upon calling `SaveChanges`. To do that, it creates a snapshot of the state of entities loaded through your `DbContext` subclass and compares the current state to the original one when `SaveChanges` is called (or when you manually query change tracking, as you'll see in a moment). You can enumerate the tracked changes in a `DbContext` as follows:

```
foreach (var e in dbContext.ChangeTracker.Entries())
{
  Console.WriteLine ($"{e.Entity.GetType().FullName} is {e.State}");
  foreach (var m in e.Members)
    Console.WriteLine (
      $"  {m.Metadata.Name}: '{m.CurrentValue}' modified: {m.IsModified}");
}
```

When you call `SaveChanges`, EF Core uses the information in the `ChangeTracker` to construct SQL statements that will update the database to match the changes in your objects, issuing insert statements to add new rows, update statements to modify data, and delete statements to remove rows that were removed from the object graph in your `DbContext` subclass. Any `TransactionScope` is honored; if none is present, it wraps all statements in a new transaction.

You can optimize change tracking by implementing `INotifyPropertyChanged` and, optionally, `INotifyPropertyChanging` in your entities. The former allows EF Core to avoid the overhead of comparing modified with original entities; the latter allows EF Core to avoid storing the original values altogether. After implementing these interfaces, call the `HasChangeTrackingStrategy` method on the

`ModelBuilder` when configuring the model in order to activate the optimized change tracking.

Navigation Properties

Navigation properties allow you to do the following:

- Query related tables without having to manually join
- Insert, remove, and update related rows without explicitly updating foreign keys

For example, suppose that a customer can have a number of purchases. We can represent a one-to-many relationship between *Customer* and *Purchase* with the following entities:

```
public class Customer
{
  public int ID { get; set; }
  public string Name { get; set; }

  // Child navigation property, which must be of type ICollection<T>:
  public virtual List<Purchase> Purchases {get;set;} = new List<Purchase>();
}

public class Purchase
{
  public int ID { get; set; }
  public DateTime Date { get; set; }
  public string Description { get; set; }
  public decimal Price { get; set; }
  public int CustomerID? { get; set; }     // Foreign key field

  public Customer Customer { get; set; }   // Parent navigation property
}
```

EF Core is able to infer from these entities that `CustomerID` is a foreign key to the *Customer* table, because the name “CustomerID” follows a popular naming convention. If we were to ask EF Core to create a database from these entities, it would create a foreign key constraint between `Purchase.CustomerID` and `Customer.ID`.

NOTE

If EF Core is unable to infer the relationship, you can configure it explicitly in the `OnModelCreating` method as follows:

```
modelBuilder.Entity<Purchase>()
  .HasOne (e => e.Customer)
  .WithMany (e => e.Purchases)
  .HasForeignKey (e => e.CustomerID);
```

With these navigation properties set up, we can write queries such as this:

```
var customersWithPurchases = Customers.Where (c => c.Purchases.Any());
```

We cover how to write such queries in detail in Chapter 9.

Adding and removing entities from navigation collections

When you add new entities to a collection navigation property, EF Core automatically populates the foreign keys upon calling `SaveChanges`:

```
Customer cust = dbContext.Customers.Single (c => c.ID == 1);

Purchase p1 = new Purchase { Description="Bike",  Price=500 };
Purchase p2 = new Purchase { Description="Tools", Price=100 };

cust.Purchases.Add (p1);
cust.Purchases.Add (p2);

dbContext.SaveChanges();
```

In this example, EF Core automatically writes 1 into the `CustomerID` column of each of the new purchases and writes the database-generated ID for each purchase to `Purchase.ID`.

When you remove an entity from a collection navigation property and call `SaveChanges`, EF Core will either clear the foreign key field or delete the corresponding row from the database, depending on how the relationship has been configured or inferred. In this case, we've defined `Purchase.CustomerID` as a nullable integer (so that we can represent purchases without a customer, or cash transactions), so removing a purchase from a customer would clear its foreign key field rather than deleting it from the database.

Loading navigation properties

When EF Core populates an entity, it does not (by default) populate its navigation properties:

```
using var dbContext = new NutshellContext();
var cust = dbContext.Customers.First();
Console.WriteLine (cust.Purchases.Count);    // Always 0
```

One solution is to use the `Include` extension method, which instructs EF Core to *eagerly* load navigation properties:

```
var cust = dbContext.Customers
  .Include (c => c.Purchases)
  .Where (c => c.ID == 2).First();
```

Another solution is to use a projection. This technique is particularly useful when you need to work with only some of the entity properties, because it reduces data transfer:

```
var custInfo = dbContext.Customers
  .Where (c => c.ID == 2)
  .Select (c => new
    {
      Name = c.Name,
      Purchases = c.Purchases.Select (p => new { p.Description, p.Price })
    })
  .First();
```

Both of these techniques inform EF Core what data you require so that it can be fetched in a single database query. It’s also possible to manually instruct EF Core to populate a navigation property as needed:

```
dbContext.Entry (cust).Collection (b => b.Purchases).Load();
// cust.Purchases is now populated.
```

This is called *explicit loading*. Unlike the preceding approaches, this generates an extra round trip to the database.

Lazy loading

Another approach for loading navigation properties is called *lazy loading*. When enabled, EF Core populates navigation properties on demand by generating a proxy class for each of your entity classes that intercepts attempts to access unloaded navigation properties. For this to work, each navigation property must be virtual, and the class it’s defined in must be inheritable (not sealed). Also, the context must not have been disposed when the lazy load occurs, so that an additional database request can be performed.

You can enable lazy loading in the `OnConfiguring` method of your `DbContext` subclass, as follows:

```
protected override void OnConfiguring (DbContextOptionsBuilder
                                       optionsBuilder)
{
  optionsBuilder
    .UseLazyLoadingProxies()
    ...
}
```

(You will also need to add a reference to the `Microsoft.EntityFrameworkCore.Proxies` NuGet package.)

The cost of lazy loading is that EF Core must make an additional request to the database each time you access an unloaded navigation property. If you make many such requests, performance can suffer as a result of excessive round-tripping.

NOTE

With lazy loading enabled, the runtime type of your classes is a proxy derived from your entity class. For example:

```
using var dbContext = new NutshellContext();
var cust = dbContext.Customers.First();
Console.WriteLine (cust.GetType());
// Castle.Proxies.CustomerProxy
```

Deferred Execution

EF Core queries are subject to deferred execution, just like local queries. This allows you to build queries progressively. There is one aspect, however, in which EF Core has special deferred execution semantics, and that is when a subquery appears within a `Select` expression.

With local queries, you get double-deferred execution, because from a functional perspective, you're selecting a sequence of *queries*. So, if you enumerate the outer result sequence but never enumerate the inner sequences, the subquery will never execute.

With EF Core, the subquery is executed at the same time as the main outer query. This prevents excessive round-tripping.

For example, the following query executes in a single round trip upon reaching the first `foreach` statement:

```
using var dbContext = new NutshellContext ();

var query = from c in dbContext.Customers
            select
               from p in c.Purchases
               select new { c.Name, p.Price };

foreach (var customerPurchaseResults in query)
  foreach (var namePrice in customerPurchaseResults)
    Console.WriteLine ($"{ namePrice.Name} spent { namePrice.Price}");
```

Any navigation properties that you explicitly project are fully populated in a single round trip:

```
var query = from c in dbContext.Customers
            select new { c.Name, c.Purchases };

foreach (var row in query)
  foreach (Purchase p in row.Purchases)   // No extra round-tripping
    Console.WriteLine (row.Name + " spent " + p.Price);
```

But if we enumerate a navigation property without first having either eagerly loaded or projected, deferred execution rules apply. In the following example, EF Core executes another `Purchases` query on each loop iteration (assuming lazy loading is enabled):

```
foreach (Customer c in dbContext.Customers.ToArray())
  foreach (Purchase p in c.Purchases)    // Another SQL round-trip
    Console.WriteLine (c.Name + " spent " + p.Price);
```

This model is advantageous when you want to *selectively* execute the inner loop, based on a test that can be performed only on the client:

```
foreach (Customer c in dbContext.Customers.ToArray())
  if (myWebService.HasBadCreditHistory (c.ID))
    foreach (Purchase p in c.Purchases)   // Another SQL round trip
      Console.WriteLine (c.Name + " spent " + p.Price);
```

NOTE

Note the use of ToArray in the previous two queries. By default, SQL Server cannot initiate a new query while the results of the current query are still being processed. Calling ToArray materializes the customers so that additional queries can be issued to retrieve purchases per customer. It is possible to configure SQL Server to allow multiple active result sets (MARS) by appending ;MultipleActiveResultSets=True to the database connection string. Use MARS with caution as it can mask a chatty database design that could be improved by eager loading and/or projecting the required data.

(In Chapter 9, we explore Select subqueries in more detail, in "Projecting".)

Building Query Expressions

So far in this chapter, when we've needed to dynamically compose queries, we've done so by conditionally chaining query operators. Although this is adequate in many scenarios, sometimes you need to work at a more granular level and dynamically compose the lambda expressions that feed the operators.

In this section, we assume the following Product class:

```
public class Product
{
  public int ID { get; set; }
  public string Description { get; set; }
  public bool Discontinued { get; set; }
  public DateTime LastSale { get; set; }
}
```

Delegates Versus Expression Trees

Recall that:

- Local queries, which use Enumerable operators, take delegates.

- Interpreted queries, which use `Queryable` operators, take expression trees.

We can see this by comparing the signature of the `Where` operator in `Enumerable` and `Queryable`:

```
public static IEnumerable<TSource> Where<TSource> (this
  IEnumerable<TSource> source, Func<TSource,bool> predicate)

public static IQueryable<TSource> Where<TSource> (this
  IQueryable<TSource> source, Expression<Func<TSource,bool>> predicate)
```

When embedded within a query, a lambda expression looks identical whether it binds to `Enumerable`'s operators or `Queryable`'s operators:

```
IEnumerable<Product> q1 = localProducts.Where (p => !p.Discontinued);
IQueryable<Product>  q2 = sqlProducts.Where   (p => !p.Discontinued);
```

When you assign a lambda expression to an intermediate variable, however, you must be explicit about whether to resolve to a delegate (i.e., `Func<>`) or an expression tree (i.e., `Expression<Func<>>`). In the following example, `predicate1` and `predicate2` are not interchangeable:

```
Func <Product, bool> predicate1 = p => !p.Discontinued;
IEnumerable<Product> q1 = localProducts.Where (predicate1);

Expression <Func <Product, bool>> predicate2 = p => !p.Discontinued;
IQueryable<Product> q2 = sqlProducts.Where (predicate2);
```

Compiling expression trees

You can convert an expression tree to a delegate by calling `Compile`. This is of particular value when writing methods that return reusable expressions. To illustrate, let's add a static method to the `Product` class that returns a predicate evaluating to `true` if a product is not discontinued and has sold in the past 30 days:

```
public class Product
{
  public static Expression<Func<Product, bool>> IsSelling()
  {
    return p => !p.Discontinued && p.LastSale > DateTime.Now.AddDays (-30);
  }
}
```

The method just written can be used both in interpreted and local queries, as follows:

```
void Test()
{
  var dbContext = new NutshellContext();
  Product[] localProducts = dbContext.Products.ToArray();

  IQueryable<Product> sqlQuery =
    dbContext.Products.Where (Product.IsSelling());

  IEnumerable<Product> localQuery =
    localProducts.Where (Product.IsSelling().Compile());
}
```

NOTE

.NET does not provide an API to convert in the reverse direction, from a delegate to an expression tree. This makes expression trees more versatile.

AsQueryable

The `AsQueryable` operator lets you write whole *queries* that can run over either local or remote sequences:

```
IQueryable<Product> FilterSortProducts (IQueryable<Product> input)
{
  return from p in input
         where ...
         orderby ...
```

```
          select p;
}

void Test()
{
  var dbContext = new NutshellContext();
  Product[] localProducts = dbContext.Products.ToArray();

  var sqlQuery   = FilterSortProducts (dbContext.Products);
  var localQuery = FilterSortProducts (localProducts.AsQueryable());
  ...
}
```

`AsQueryable` wraps `IQueryable<T>` clothing around a local sequence so that subsequent query operators resolve to expression trees. When you later enumerate over the result, the expression trees are implicitly compiled (at a small performance cost), and the local sequence enumerates as it would ordinarily.

Expression Trees

We said previously that an implicit conversion from a lambda expression to `Expression<TDelegate>` causes the C# compiler to emit code that builds an expression tree. With some programming effort, you can do the same thing manually at runtime—in other words, dynamically build an expression tree from scratch. The result can be cast to an `Expression<TDelegate>` and used in EF Core queries or compiled into an ordinary delegate by calling `Compile`.

The Expression DOM

An expression tree is a miniature code DOM. Each node in the tree is represented by a type in the `System.Linq.Expressions` namespace. Figure 8-10 illustrates these types.

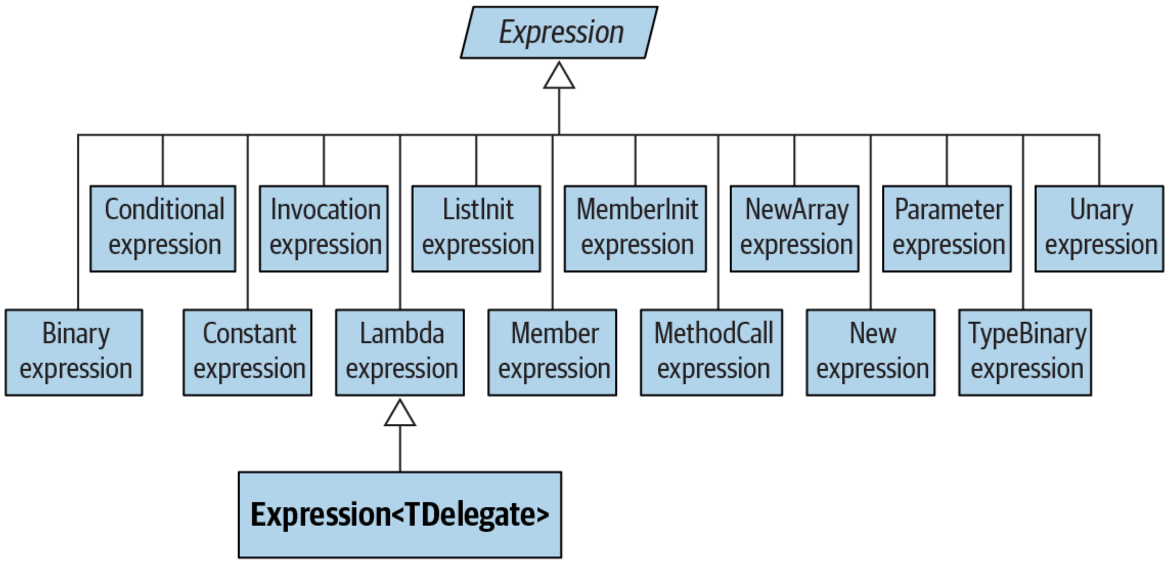

Figure 8-10. Expression types

The base class for all nodes is the (nongeneric) `Expression` class. The generic `Expression<TDelegate>` class actually means "typed lambda expression" and might have been named `LambdaExpression<TDelegate>` if it wasn't for the clumsiness of this:

```
LambdaExpression<Func<Customer,bool>> f = ...
```

`Expression<T>`'s base type is the (nongeneric) `LambdaExpression` class. `LamdbaExpression` provides type unification for lambda expression trees: any typed `Expression<T>` can be cast to a `LambdaExpression`.

The thing that distinguishes `LambdaExpressions` from ordinary `Expressions` is that lambda expressions have *parameters*.

To create an expression tree, don't instantiate node types directly; rather, call static methods provided on the `Expression` class, such as `Add`, `And`, `Call`, `Constant`, `LessThan`, and so on.

Figure 8-11 shows the expression tree that the following assignment creates:

```
Expression<Func<string, bool>> f = s => s.Length < 5;
```

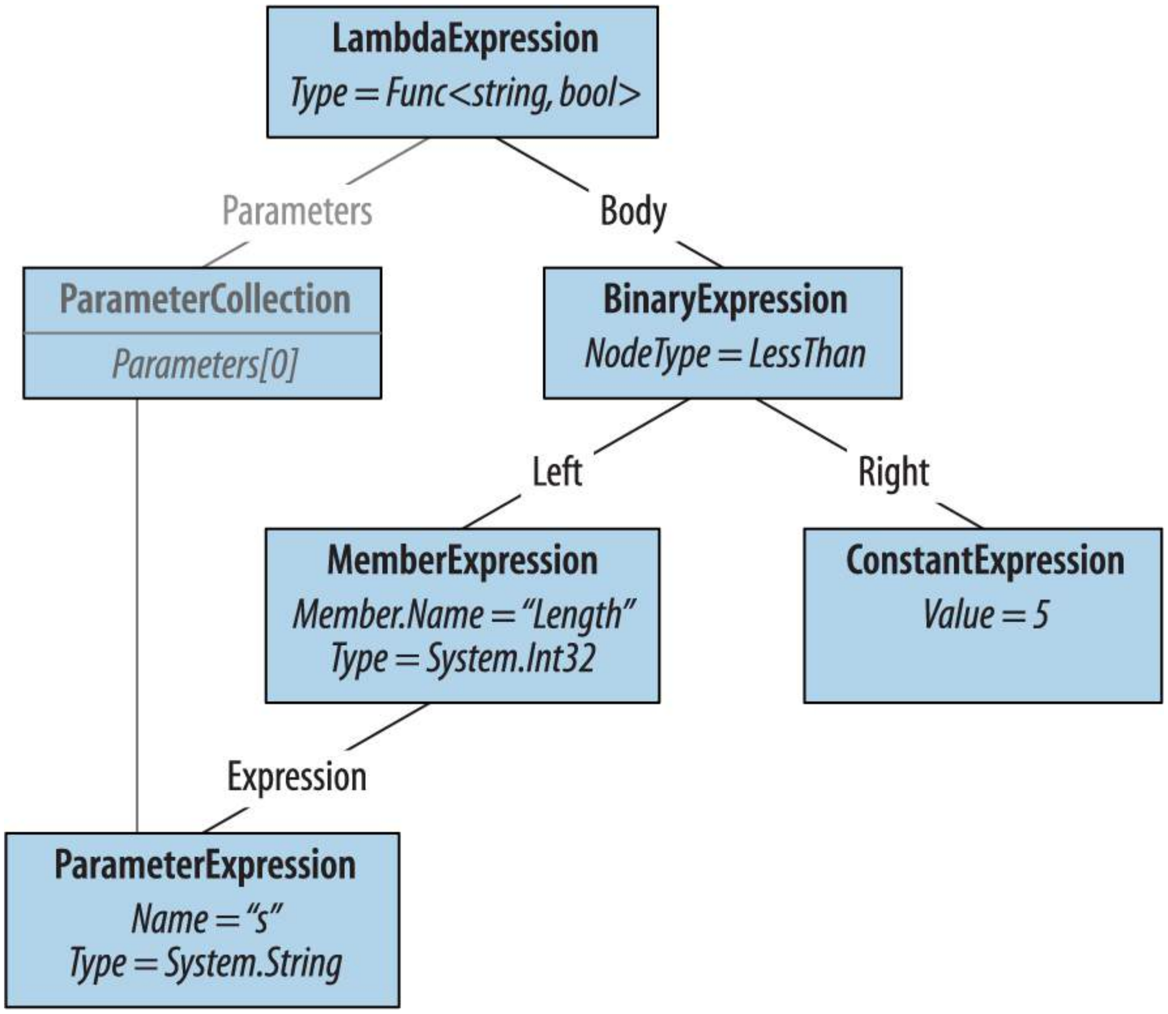

Figure 8-11. Expression tree

We can demonstrate this as follows:

```
Console.WriteLine (f.Body.NodeType);                        // LessThan
Console.WriteLine (((BinaryExpression) f.Body).Right);   // 5
```

Let's now build this expression from scratch. The principle is that you start from the bottom of the tree and work your way up. The bottommost thing in our tree is a `ParameterExpression`, the lambda expression parameter called "s" of type `string`:

```
ParameterExpression p = Expression.Parameter (typeof (string), "s");
```

The next step is to build the `MemberExpression` and `ConstantExpression`. In the former case, we need to access the `Length` *property* of our parameter, “s”:

```
MemberExpression stringLength = Expression.Property (p, "Length");
ConstantExpression five = Expression.Constant (5);
```

Next is the `LessThan` comparison:

```
BinaryExpression comparison = Expression.LessThan (stringLength, five);
```

The final step is to construct the lambda expression, which links an expression `Body` to a collection of parameters:

```
Expression<Func<string, bool>> lambda
  = Expression.Lambda<Func<string, bool>> (comparison, p);
```

A convenient way to test our lambda is by compiling it to a delegate:

```
Func<string, bool> runnable = lambda.Compile();

Console.WriteLine (runnable ("kangaroo"));          // False
Console.WriteLine (runnable ("dog"));               // True
```

NOTE

The easiest way to determine which expression type to use is to examine an existing lambda expression in the Visual Studio debugger.

We continue this discussion online, at *http://www.albahari.com/expressions*.

[1] The term is based on Eric Evans and Martin Fowler’s work on fluent interfaces.