when it's mandatory (as we'll see later, in the section "Projection Strategies" on page 337), or when a query's type is irrelevant to an example.

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 retrieves all names that contain the letter "a":

```
IEnumerable<string> filteredNames = names.Where (n => n.Contains ("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 we'll see shortly, this strategy is highly composable in that it allows the chaining of query operators. In the book, we refer to this as *fluent syntax*.* 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.

we explore each in more detail.

LINQ Queries

* The term is based on Eric Evans and Martin Fowler's work on fluent interfaces.



# Fluent Syntax

# 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;
```

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

class LinqDemo
{
  static void Main()
  {
    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
```

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

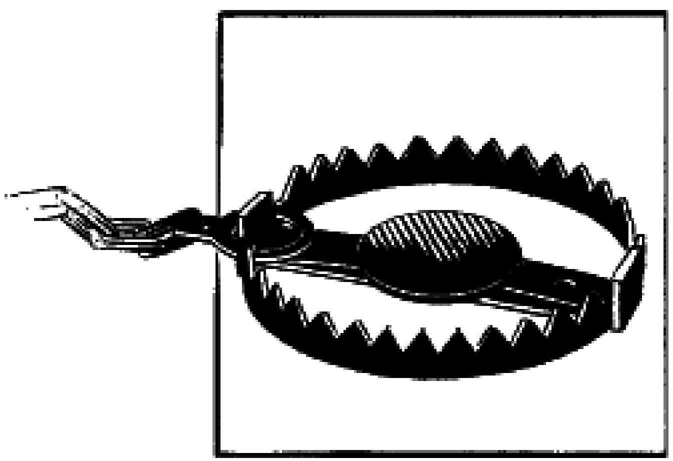
```
void Test()
{
  foreach (char c in "string1") Console.Write (c);
  foreach (char c in "string2") Console.Write (c);
  foreach (char c in "string3") Console.Write (c);
}
```

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

`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 where 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, then projected.



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

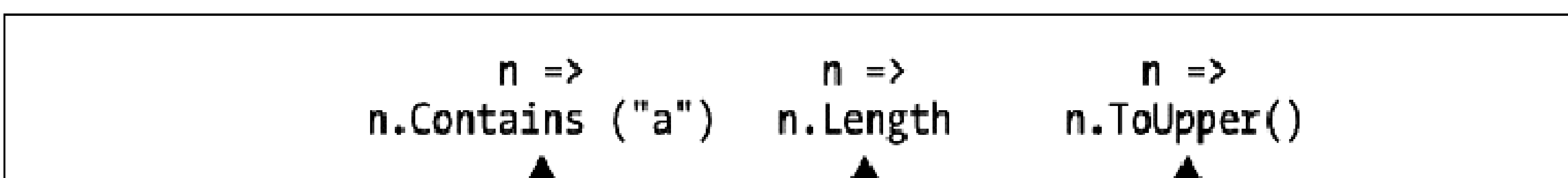
Here are the signatures of each of these extension methods (with the `OrderBy` signature simplified slightly):

```
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 end result resembles a production line of conveyor belts, as illustrated in Figure 8-1.



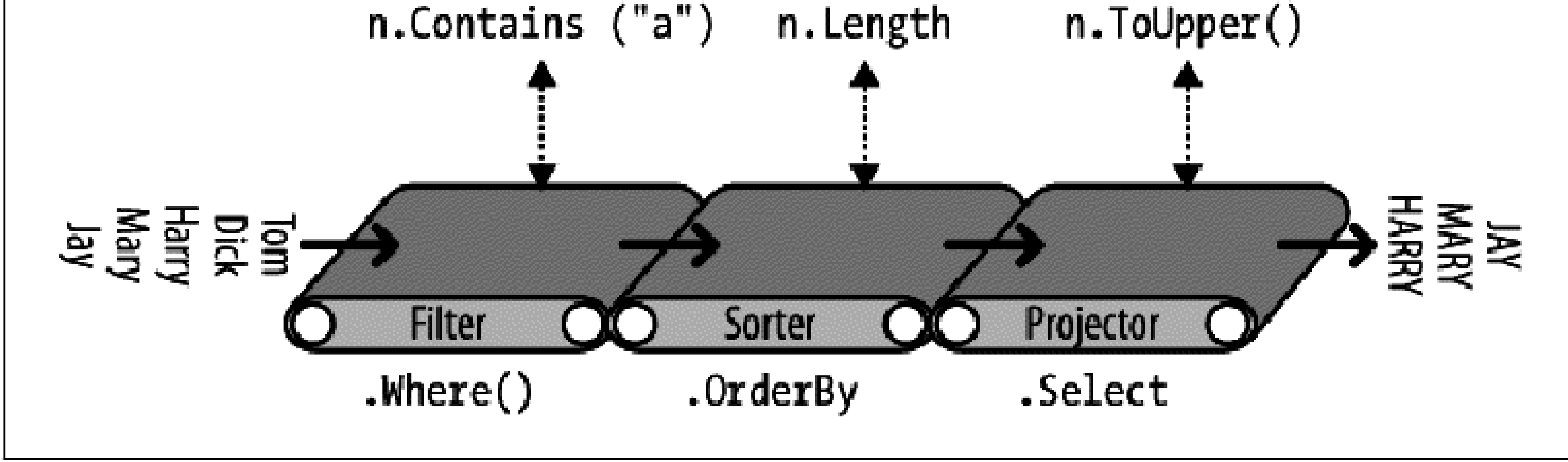



*Figure 8-1. Chaining query operators*

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 had 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|
```

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

LINQ Queries



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

```
foreach (string name in sorted)
  Console.Write (name + "|");
Console.WriteLine();
foreach (string name in finalQuery)
  Console.Write (name + "|");
```

```
// Jay|Mary|Harry|
// 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. For example:

```
IEnumerable<string> filtered = Enumerable.Where (names,
```

```
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, as well as keeping lambda expressions alongside their query operators (*infix* notation). Without extension methods, the query loses its *fluency*:

```
IEnumerable<string> query =
  Enumerable.Select (
    Enumerable.OrderBy (
```

```
    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.
```

An expression returning a `bool` value is called a *predicate*.

An expression returning a `bool` value 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.

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

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 `System.Linq`, defined with the following intent:

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

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 the section "Lambda Expressions" on page 130 in Chapter 4.

# Lambda expressions and element typing

The standard query operators use the following generic type 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 |

| TKey | Element type for the *key* used in sorting, grouping, or joining |
|---|---|



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



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

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` are 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 *infers* the type of `TResult` from the return value of the lambda expres-

The compiler *infers* the type of `TResult` from the return value of the lambda expression. In this case, `TResult` is inferred to be of type `int`.

The `Where` query operator is simpler and requires no type inference for the output, since 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
```

```
IEnumerable<string> sortedByLength, sortedAlphabetically;
sortedByLength       = names.OrderBy (n => n.Length);   // int key
sortedAlphabetically = names.OrderBy (n => n);          // string key
```

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 the section "Interpreted Queries" on page 339.



# Natural Ordering

The original ordering of elements within an input sequence is significant in LINQ.

The original ordering of elements within an input sequence is significant in LINQ. Some query operators rely on this behavior, 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 }
```

Operators such as Where and Select preserve the original ordering of the input se-

Operators such as `Where` and `Select` preserve the original ordering of the input sequence. LINQ preserves the ordering of elements in the input sequence wherever possible.

# 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 lowestNumber = numbers.OrderBy (n => n).First();   // 6
```

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:

The *quantifiers* return a bool value:

```
bool hasTheNumberNine = numbers.Contains (9);
bool hasMoreThanZeroElements = numbers.Any();
bool hasAnOddElement = numbers.Any (n => n % 2 == 1);
```

```
// true
// true
// true
```

Because these operators don't return a collection, you can't call further query operators on their result. In other words, they must appear as the last operator in a query.

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

```
//
//
```

```
{ 1, 2, 3, 3, 4, 5 }
{ 1, 2, 3, 4, 5 }
```

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



# Query Expressions

C# provides a syntactic shortcut for writing LINQ queries, called *query expressions*. Contrary to popular belief, query expressions are based not on SQL, but on *list comprehensions* from functional programming languages such as LISP and *sions*. Contrary to popular belief, query expressions are based not on SQL, but on *list comprehensions* from functional programming languages such as LISP and Haskell.

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;
```

```
class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

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

IEnumerable<string> query =
  from    n in names
  where   n.Contains ("a")
  orderby n.Length
  select  n.ToUpper();

// Filter elements
// Sort elements
// Translate each element (project)

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

```
    }
}
```

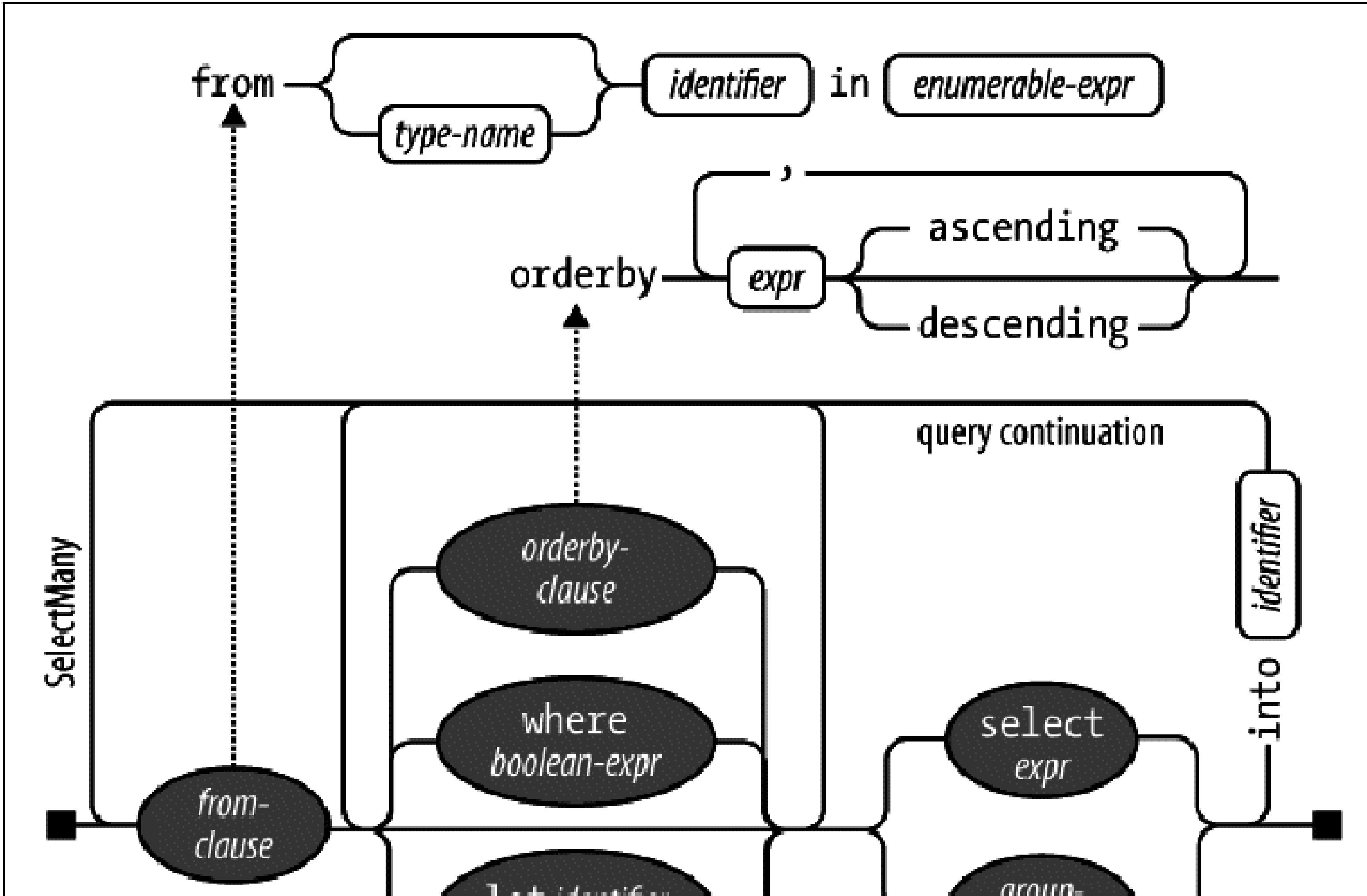
```
JAY
MARY
HARRY
```

Query expressions always start with a `from` clause and ends with either a `select` or `group` clause. The `from` clause declares an *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.

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.

orderby, where, let or join clause.





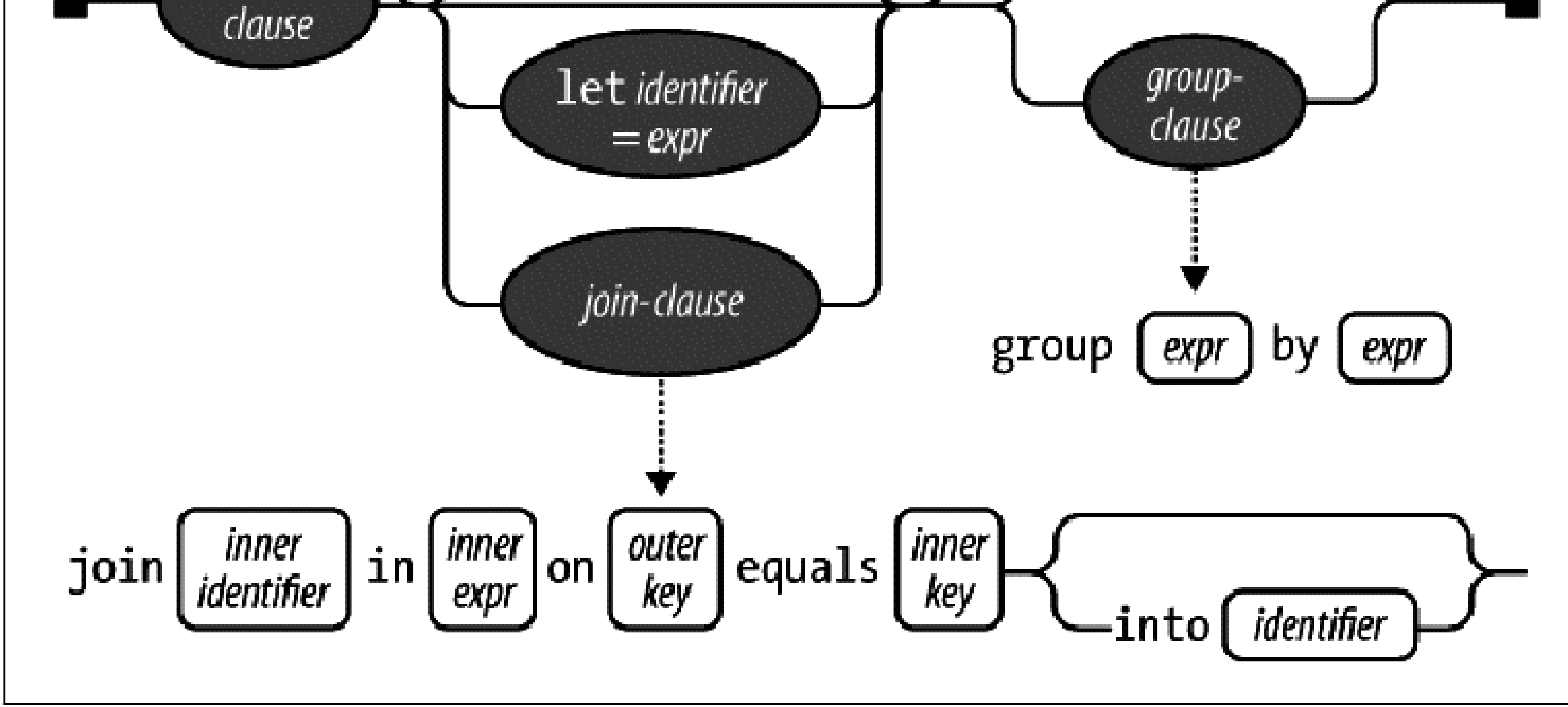



*Figure 8-2. Query syntax*

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)
```

```
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'd typed the method names yourself. This offers flexibility in how they resolve. The operators in the database queries that we'll write in later sections, for instance, will bind instead to extension methods in `Queryable`.

> If we remove the `using System.Linq` directive from our program, the query would not compile, because the `Where`, `OrderBy`, and `Select` methods would have nowhere to bind. Query expressions *cannot compile* unless you import a namespace (or write an instance method for every query operator!).

# Range Variables

The identifier immediately following the `from` keyword syntax is called the *range*

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

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"))      // Privately scoped n
     .OrderBy (n => n.Length)              // Privately scoped n
     .Select  (n => n.ToUpper())           // Privately 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

We will cover these later in this chapter in “Composition Strategies” on page 333, and also in Chapter 9, in “Projecting” on page 369 and “Joining” on page 370.

## 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 reference a table alias in the SELECT clause before defining it in a FROM clause.

ence 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 data flow.

A LINQ query comprises a conveyor belt or *pipeline* of operators that accept and emit *ordered sequences*. A 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 the later section, “Composition Strategies” on page 333; 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
```

```
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();
```

```
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();
string first = names.OrderBy (n => n).First();

// 3
// Dick
```

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 without feeling a sense of failure!

The remainder of this chapter will show key concepts in both fluent and query syntax, where applicable.

syntax, where applicable.

# 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>();
numbers.Add (1);
```

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

```
// Build query
```

```
numbers.Add (2);
foreach (int n in query)
  Console.Write (n + "|");
```

```
Console.Write (n + "|");
```

```
// Sneak in an extra element
// 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. 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
```

`ToArray, ToList, ToDictionary, ToLookup`

These operators cause immediate query execution because their result types have no mechanism for providing deferred execution. The `Count` method, for instance, returns a simple integer, which then doesn't 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, as well as making database queries possible.



Subqueries provide another level of indirection. Everything in a subquery is subject to deferred execution—including aggregation and conversion methods. We describe this in the section "Subqueries" on page 330 later in this chapter.

## Reevaluation

# Reevaluation

Deferred execution has another consequence: reevaluated when you re-enumerate:

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

a deferred execution query is

```
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<>`:

```
var numbers = new List<int>() { 1, 2 };
List<int> timesTen = numbers
  .Select (n => n * 10)
  .ToList();
```

```
.ToList();
// Executes immediately into a List<int>
numbers.Clear();
Console.WriteLine (timesTen.Count);
// Still 2
```

## Captured Variables

Deferred execution also has a sinister effect. If your query's lambda expressions reference local variables, these variables are subject to *captured variable* semantics. This means that if you later change their value, the query changes as well:

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

```
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|
```

LINQ Queries

This can be a real trap when building up a query within a `foreach` loop. For example, suppose we wanted 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 `foreach` loop:

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

foreach (char vowel in "aeiou")
  query = query.Where (c => c != vowel);

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

// Not what yo might expect
```

Only the `'u'` is stripped! This is because, as we saw in Chapter 4 (see "Capturing Outer Variables" on page 132 in "Lambda Expressions" on page 130), the compiler scopes the iteration variable in the `foreach` loop as if it was declared *outside* the loop:

```
IEnumerable<char> vowels = "aeiou";
using (IEnumerator<char> rator = vowels.GetEnumerator())
```

```
IEnumerable<char> vowels = "aeiou";
using (IEnumerator<char> rator = vowels.GetEnumerator())
{
  char vowel;
  while (rator.MoveNext())
  {
    vowel = rator.Current;
    query = query.Where (c => c != vowel);
  }
}
```

Because `vowel` is declared outside the loop, the *same* variable is repeatedly updated, so each lambda expression captures the same `vowel`. When we later enumerate the query, all lambda expressions reference that single variable's current value, which is `'u'`. To solve this, you must assign the loop variable to another variable declared *inside* the statement block:

```
foreach (char vowel in "aeiou")
{
  char temp = vowel;
```

```
{
  char temp = vowel;
  query = query.Where (c => c != temp);
}
```

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



# 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.

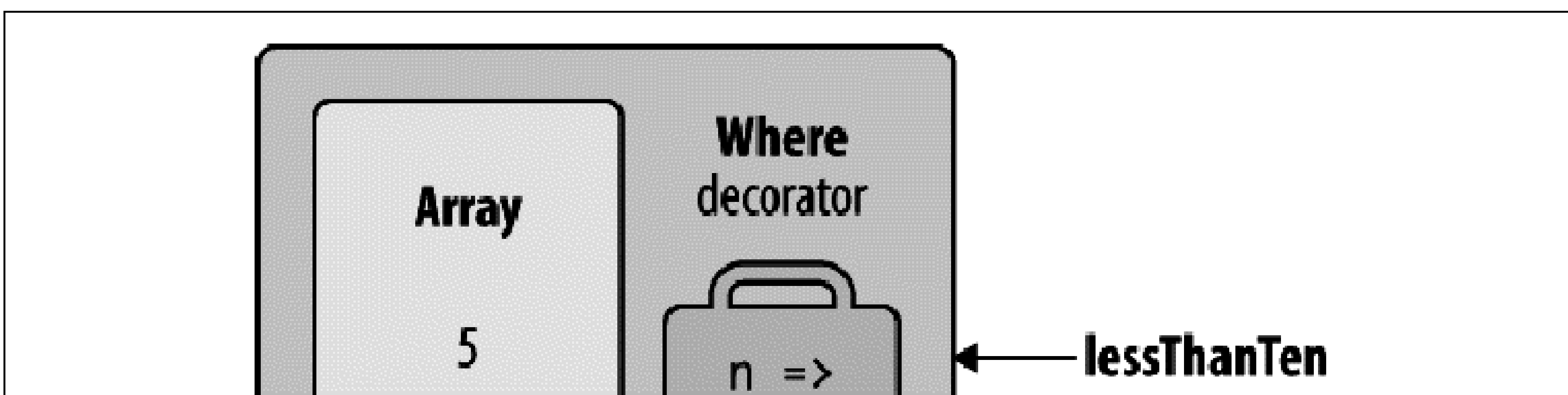
request data from the wrapped input sequence.

> 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, holding 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);
```



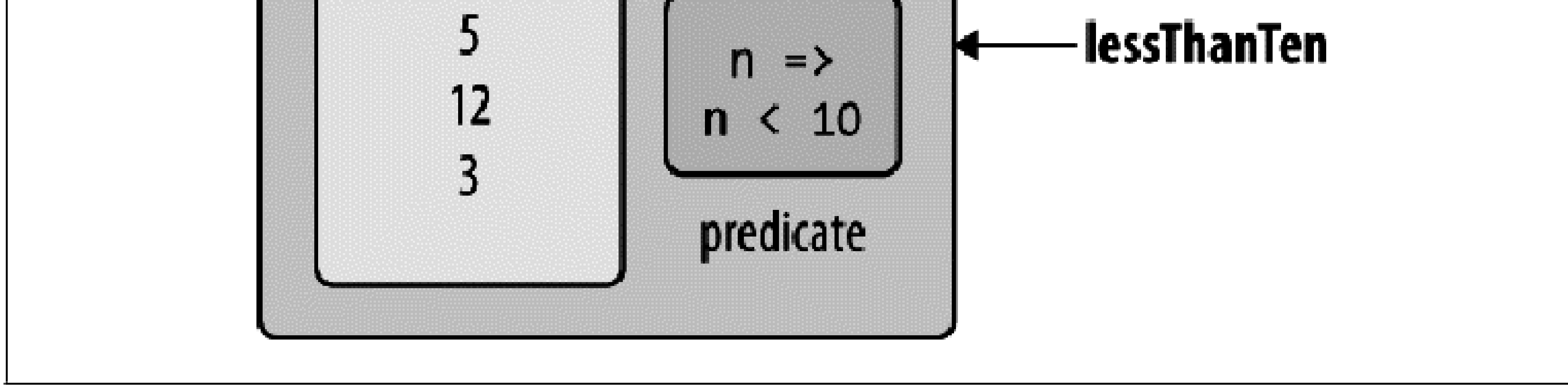



*Figure 8-3. Decorator sequence*

When you enumerate lessThanTen, you're, in effect, querying the array through the Where decorator.

The good news—if 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> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

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

```
public static IEnumerable<TResult> Select<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 doll). The object model of this query is illustrated in Fig-

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

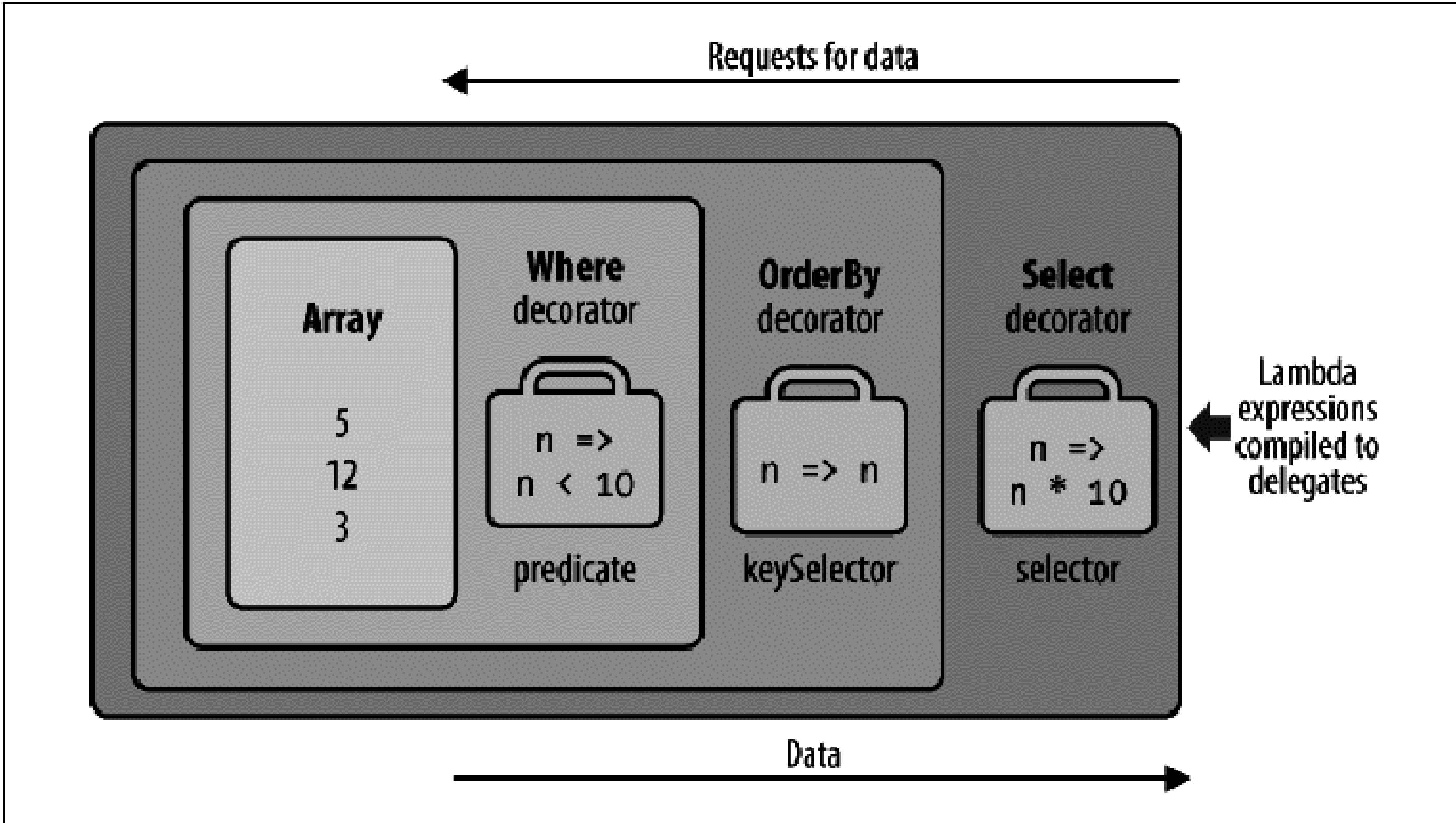



Figure 8-4. Layered decorator sequences

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



Adding ToList onto the end of this query would cause the preceding operators to execute right away, collapsing the whole object model into a single list.

Figure 8-5 shows the same object composition in 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),
```

```
filtered  = source   .Where   (n => n < 10),
sorted    = filtered .OrderBy (n => n),
query     = sorted   .Select  (n => n * 10);
```

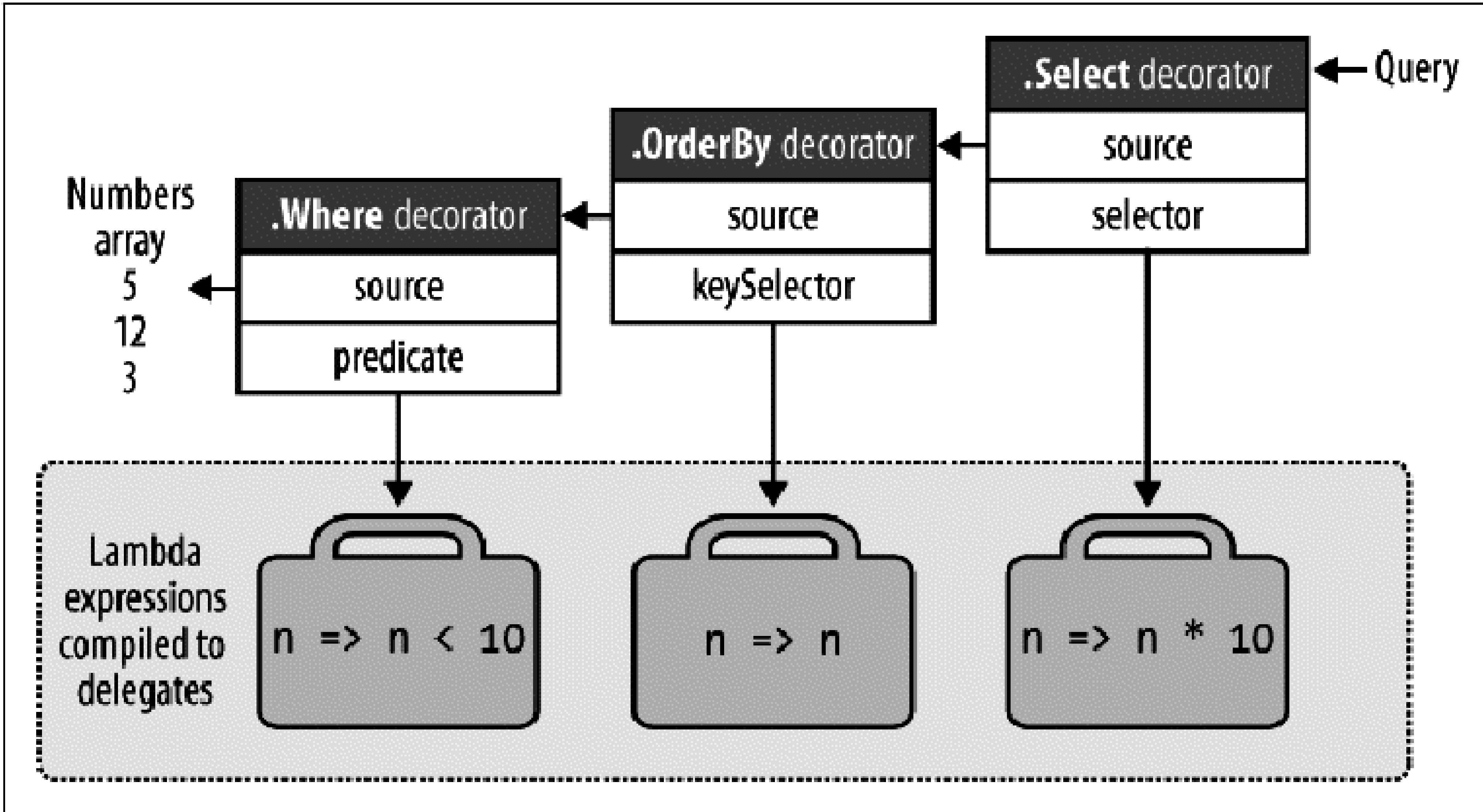



Figure 8-5. UML decorator composition

*Figure 8-5. UML decorator composition*

# How Queries Are Executed

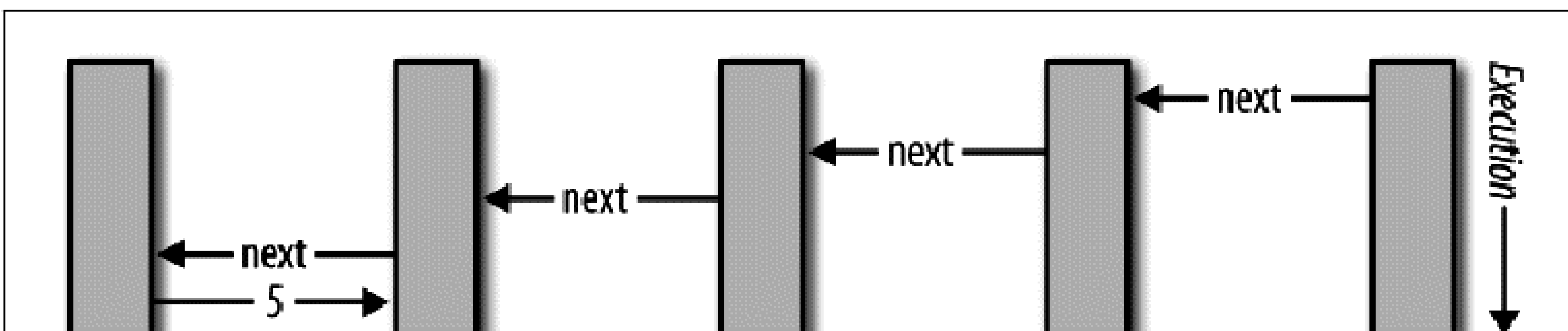
Here are the results of enumerating the preceding query:

```
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 everything off. 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.

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


Execution
next
next
next
next
5

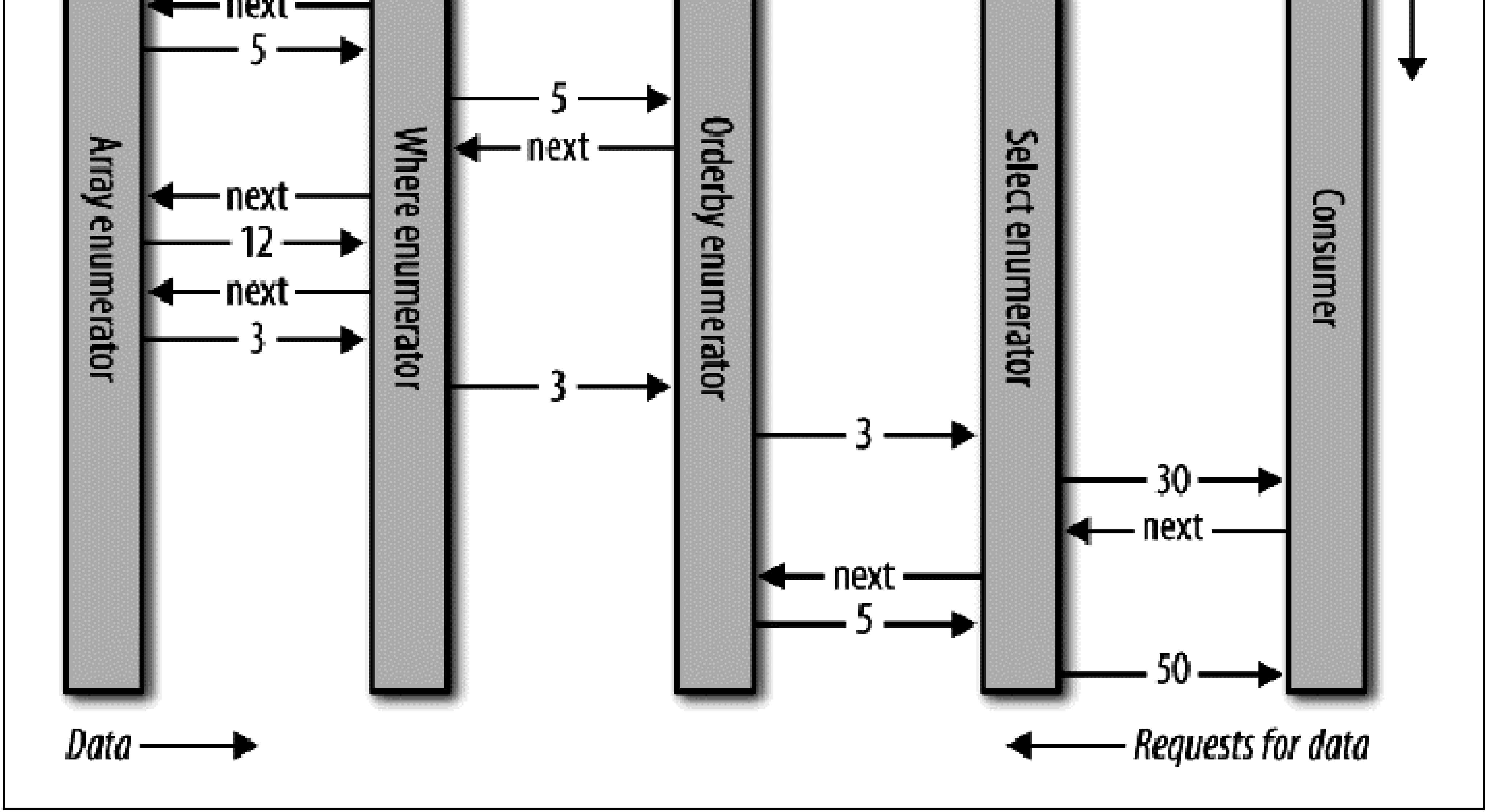



*Figure 8-6. Execution of a local query*

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 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 we'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" };
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 right-hand 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).

that the rules for subqueries are a consequence of the rules for lambda expressions (and the behavior of query operators in general).



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 is able to reference the outer lambda argument (or range variable 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)
```

```
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 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

model literally; interpreted queries (e.g., database queries) follow this model *conceptually*.

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

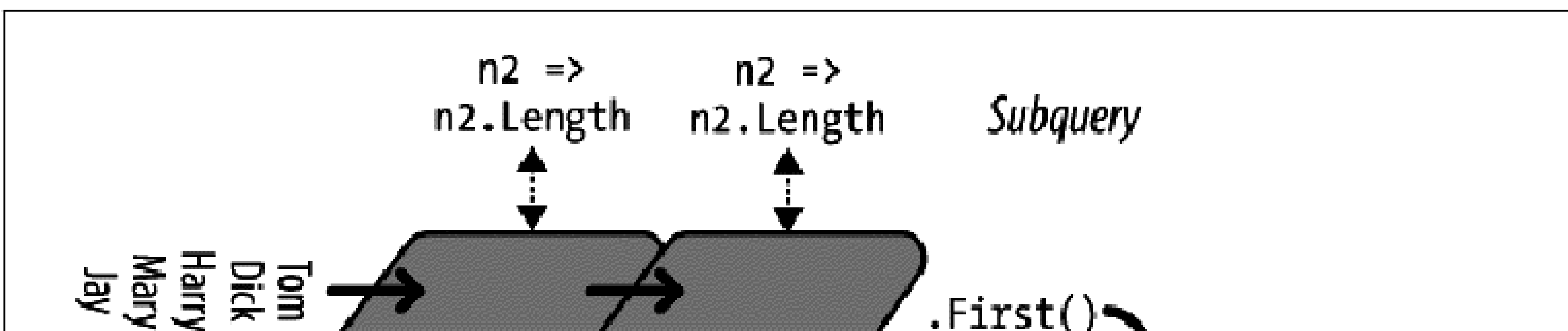
We can express our preceding subquery more succinctly as follows:

```
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;
```

[redacted]


n2 =>
n2.Length
n2 =>
n2.Length
Subquery
Tom
Dick
Harry
Mary
Jay
.First()

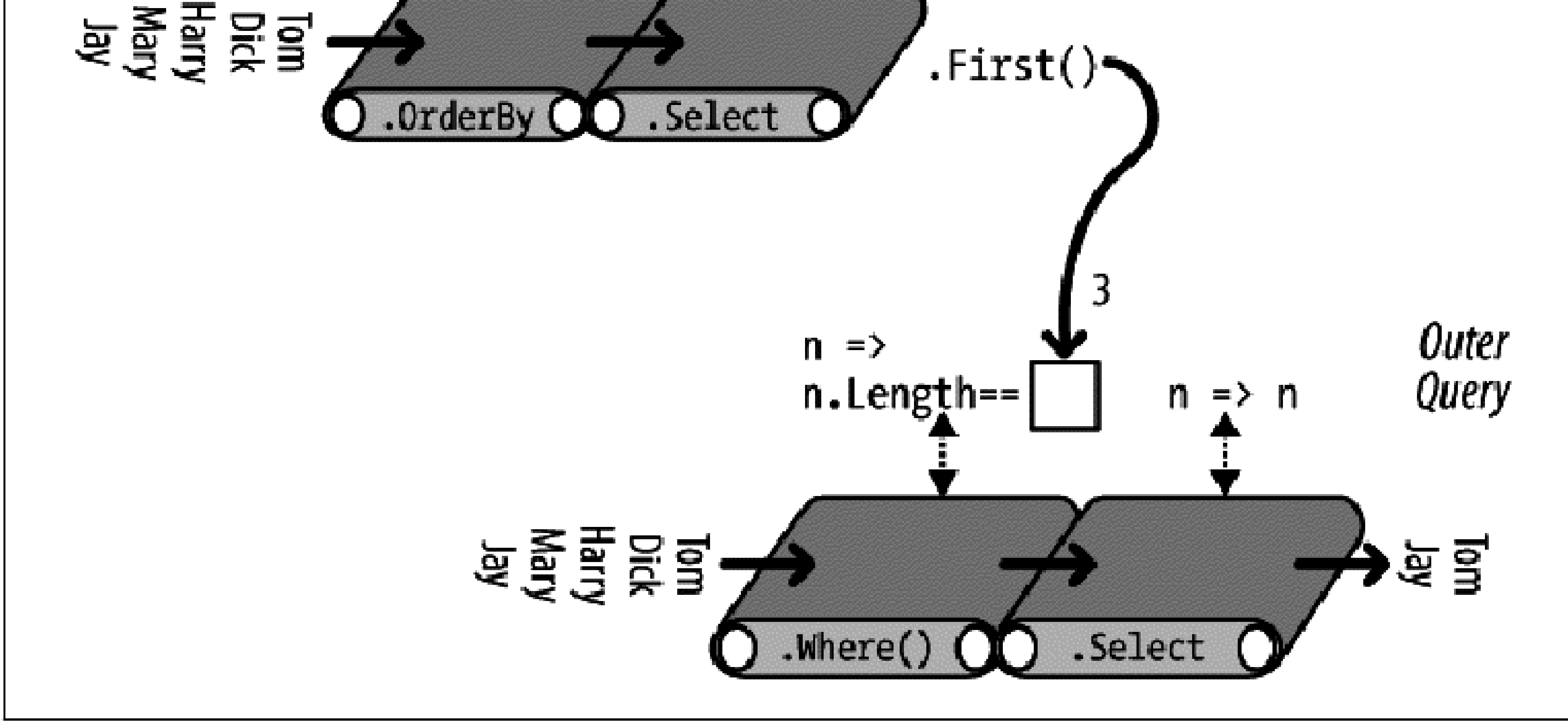



*Figure 8-7. Subquery composition*

In the later section "Interpreted Queries" on page 339, we'll 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);
```

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;
```

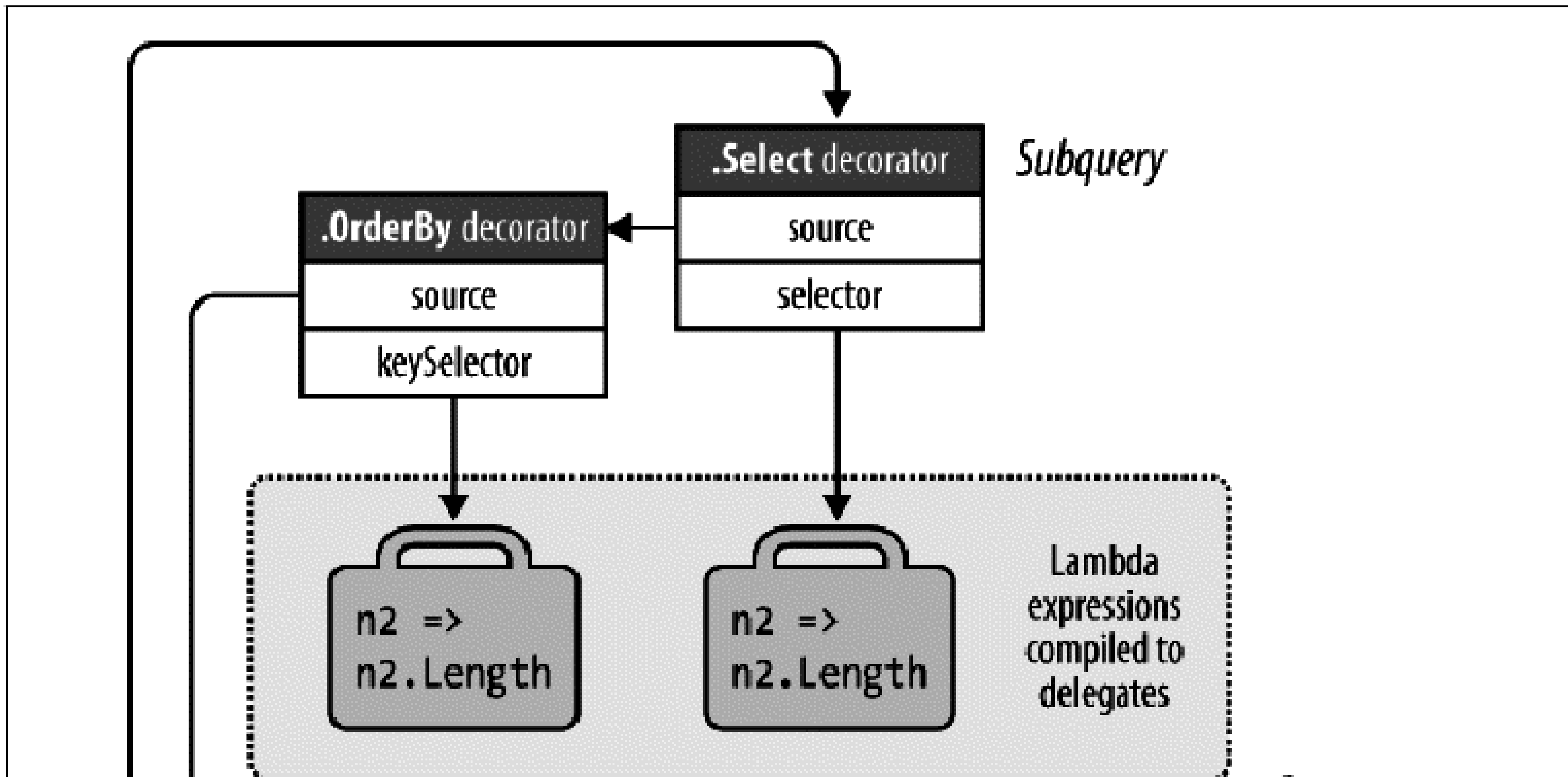
Factoring out subqueries in this manner is nearly always desirable when querying local collections. An exception is when the subquery is *correlated*, meaning that it references the outer range variable. We explore correlated subqueries in the following chapter, in the section "Projecting" on page 369.

# 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 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





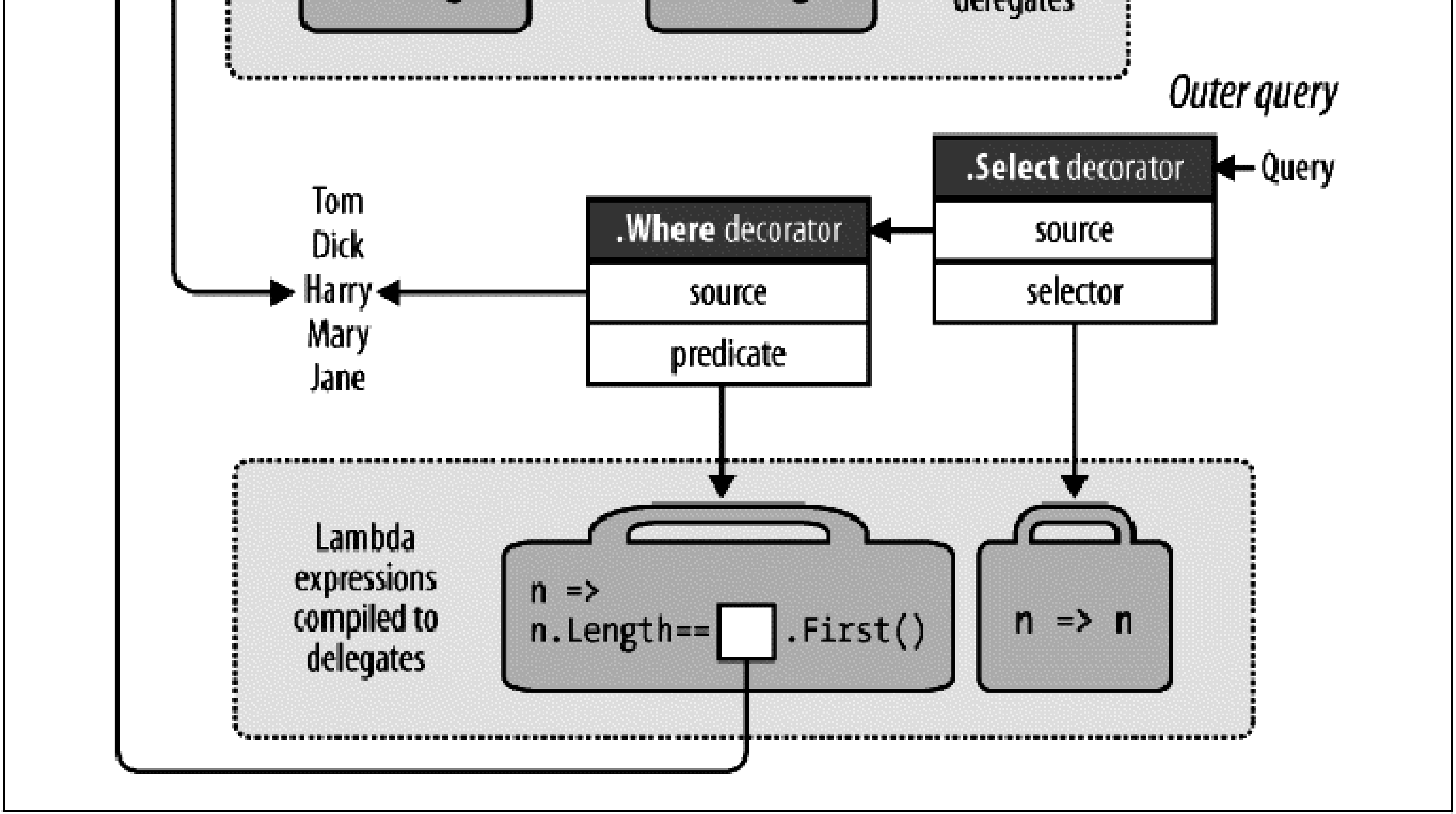



*Figure 8-8. UML subquery composition*

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.

ter 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.

LINQ Queries



# Progressive Query Building

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

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 (...)
```

This is more efficient than:

# This 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. To illustrate, imagine 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);
```

RESULT: { "Dck", "Hrry", "Mry" }

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

```
select
```

```
n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
 .Replace ("o", "").Replace ("u", "");
```

```
RESULT: { "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;
```

```
RESULT: { "Dck", "Hrry", "Mry" }
```

# The into Keyword

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:

```
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.

> 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 query 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
```

```
  into n2
    where n1.Contains ("x")
    select n2;
```

```
// Only n2 is visible from here on.
// Illegal: n1 is not in scope.
```

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
```

```
// 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:

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.

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) activated upon demand through the conveyor belt's lambda worker (as illustrated in Figure 8-7).

# Projection Strategies

## Object Initializers

So far, all 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;
```

```
    public string Original;
    public string Vowelless;
}
```

```
// Original name
// Vowel-stripped name
```

and then 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", "")
```

```
    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
```

```
{
  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, writing 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-produced-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 whole query more succinctly with the `into` keyword:

We can write the whole 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:

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:

`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 (`v`).

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

# 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<>`. Local queries resolve to query operators in the `Enumerable` class, 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.

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.

There are two `IQueryable<T>` implementations in the .NET Framework:

There are two `IQueryable<T>` implementations in the .NET Framework:

- LINQ to SQL
- Entity Framework (EF)

These *LINQ-to-db* technologies are very similar in their LINQ support: the LINQ-to-db queries in this book will work with both LINQ to SQL and EF unless otherwise specified.

It's also possible to generate an `IQueryable<T>` wrapper around an ordinary enumerable collection by calling the `AsQueryable` method. We describe `AsQueryable` in the section "Building Query Expressions" on page 361 later in this chapter.

In this section, we'll use LINQ to SQL to illustrate interpreted query architecture because LINQ to SQL lets us query without having to first write an Entity Data Model. The queries that we write, however, work equally well with Entity Framework (and also many third-party products).

`IQueryable<T>` is an extension of `IEnumerable<>` with additional methods for constructing expression trees. Most of the time you methods for constructing expression trees. Most of the time you can ignore the details of these methods; they're called indirectly by the Framework. The section "Building Query Expressions" on page 361 covers `IQueryable<T>` in more detail.

Suppose we 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 (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# to retrieve customers whose name contains the letter "a" as follows:

```
using System;
using System.Linq;
using System.Data.Linq;
using System.Data.Linq.Mapping;
```

```
// in System.Data.Linq.dll
```

```
[Table] public class Customer
{
  [Column(IsPrimaryKey=true)] public int ID;
  [Column]                    public string Name;
}
```

```
class Test
{
  static void Main()
  {
    DataContext dataContext = new DataContext ("connection string");
    Table<Customer> customers = dataContext.GetTable <Customer>();
```



```
IQueryable<string> query = from c in customers
  where   c.Name.Contains ("a")
  orderby c.Name.Length
```

```
  where   c.Name.Contains ("a")
  orderby c.Name.Length
  select  c.Name.ToUpper();

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

LINQ to SQL translates this query into the following SQL:

```
SELECT UPPER([t0].[Name]) AS [value]
FROM [Customer] AS [t0]
WHERE [t0].[Name] LIKE @p0
ORDER BY LEN([t0].[Name])
```

with the following end result:

```
JAY
MARY
```

```
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 = 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 `customers` variable, the source upon which the whole query builds. `customers` is of type `Table<>`, which implements `IQueryable<T>` (a subtype of `IEnumerable<>`). This means the compiler has a choice in resolving `Where`: it could call the extension method in `Enumerable` or the following

IQueryable<T> (a subtype of IEnumerable<>). This means 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 LINQ to SQL or EF can later translate it to a SQL statement).

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