Contents

This topic explains how to use the LINQ ***Join()*** operator ... and why you should rarely use it.

LINQ has a ***Join()*** operator. Developers who are new to Entity Framework (EF) are quick to use it because "JOIN" is such a common SQL operation. Experienced EF developers prefer to use subqueries and **almost never need to use *Join()***.

Why? Primarily because the entity data model (EDM) represents relationships between entities as associations. You build that model so you can escape the mechanical details and think in object terms rather than database terms. You want to write *anOrder.OrderDetails* without getting into the nitty-gritty of using an outer-left-join of *Order.OrderID* and *OrderDetail.OrderID*. You'd prefer not to think about foreign keys at all; they're a concept that is alien to object thinking.

The *Join()* operation breaks that abstraction. It necessarily forces you to think about precisely how you get the *OrderDetails* related to an order. You have to express the *Join()* in terms of *Order.OrderID* and *OrderDetail.OrderID*. Why do that if you don't have to?

All abstractions "leak" eventually; it's unavoidable in real world programming. But we strive to retain the benefits of our abstractions while it is easy and prudent to do so.

Life without joins

Suppose we want to query for just those *Customer*s that have placed an *Order* with our company. In SQL you'd write a "JOIN". In EF LINQ you'd write a subquery instead and test to see if there were any orders. That query written in comprehension syntax could look like this:

```
query   = from c in manager.Customers
          where c.Orders.Any()
          select c
results = query.ToList()
```

The *manager* variable in these examples is an instance of the *NorthwindIBEntityManager* generated from a model that accesses the "NorthwindIB" tutorial database. If you prefer method chaining (lambda) syntax you could write it this way:

```
query   =  manager.Customers.Where(c => c.Orders.Any())
results = query.ToList();
```

```
query   =  manager.Customers.Where(Function (c) c.Orders.Any())
results = query.ToList()
```

We're taking advantage of the fact that we've modeled the association between Customer and Order. We never need to JOIN when we have an association.

You could have joined

You could have achieved the same effect with a LINQ *Join()*. First with comprehension syntax.

```
query   = from c in manager.Customers
          join o in manager.Orders
          on c.CustomerID equals o.CustomerID
          select c
results = query.Distinct().ToList()
```

That's a lot of messy detail. We're fortunate that *Customer* has a single value key; it gets very messy if *Customer* has a composite key. Don't forget the *Distinct()* method or you'll get 800+ *Customer*s, one for every *Order* in the tutorial database, when you only want the ~80 distinct *Customer*s.

Here it is again in method chaining (lambda) syntax:

```
query = manager
        .Customers              // Customers
        .Join(manager.Orders,   // Orders
          c => c.CustomerID,    // Customers key
          o => o.CustomerID,    // Order key
          (c, o) => c);         // projected value (the "select")
results = query.Distinct().ToList();
```

```
query = manager
```

```
        .Customers                    ' Customers
       .Join(manager.Orders,          ' Orders
          Function(c) c.CustomerID,   ' Customers key
          Function(o) o.CustomerID,   ' Order key
          Function(c, o) c)           ' projected value (the "select")
results = query.Distinct().ToList()
```

When you have to join

Occasionally you know that two entities are related even though there is no association in the model.

Imagine that the *Employee* entity has a deprecated key (*EmployeeNumber*) which remains the only basis for linking to historical data (*OldEmployeeData*). Entity Framework won't let you associate a dependent entity's key (*OldEmployeeData.EmployeeNumber*) with a non-primary key of the parent (*Employee.EmployeeNumber*). You'll have to join them.

We don't have data like that in the DevForce "NorthwindIB" tutorial database so we'll make up an absurd example in which you query for the *Employee*s whose IDs happen also to be *Product* IDs. You can join these entities yourself; *Employee* and *Product* have integer IDs with values that overlap.

The values that you join must be comparable with equality. A common mistake is to join on two properties that have different data types. You'll learn of your error in a runtime exception.

Here's the query in comprehension syntax:

```
query   = from e in manager.Employees
          join p in manager.Products
          on e.EmployeeID equals p.ProductID
          select e
results = query.ToList()
```

The results include all nine *Employee*s in the "NorthwindIB" tutorial database; they happen to have ids in the range {1..9} … as do the first nine *Product*s.

Distinct() wasn't necessary because the query compares primary key values which are necessarily unique. Here's the same query in method chaining (lambda) syntax:

```
query   = manager
          .Employees
          .Join(
              manager.Products,
              e => e.EmployeeID,
              p => p.ProductID,
             (e, p) => e);
results = query.ToList();
```

```
query   = manager
          .Employees
          .Join(
              manager.Products,
              Function(e) e.EmployeeID,
              Function(p) p.ProductID,
              Function(e, p) e)
results = query.ToList()
```