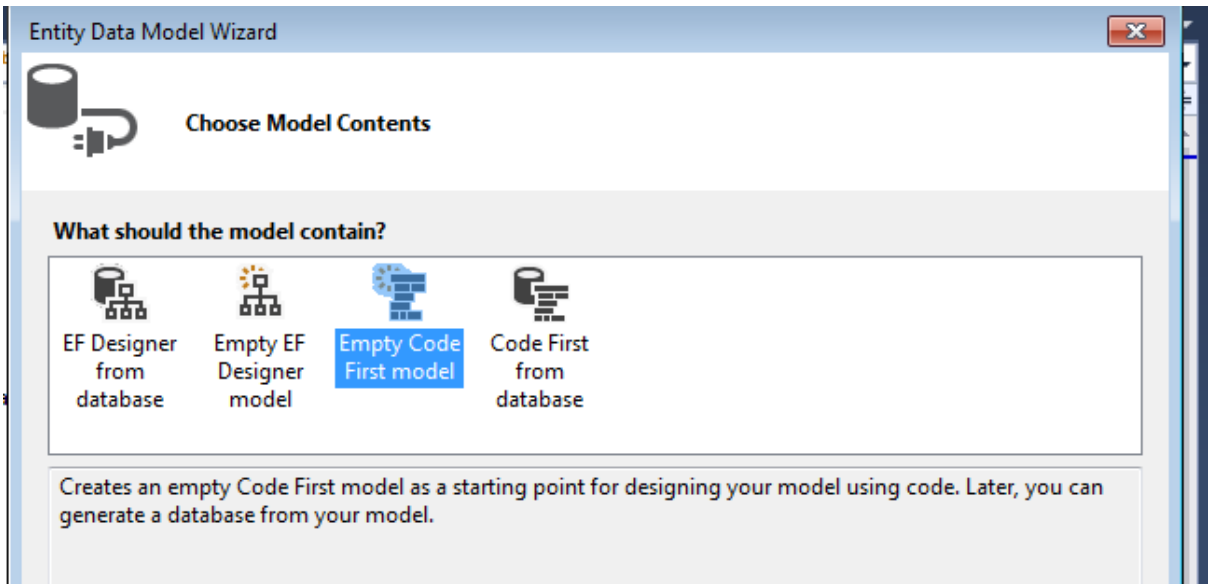
Utilizare EF – (pentru laborator)

Intr-un proiect deja creat (CUI /GUI .NET Framework) adaugam un template *ADO.NET Entity Data Model* din template Data. Name = ModelSelfReferences. Folosim *Empty Code First model.*

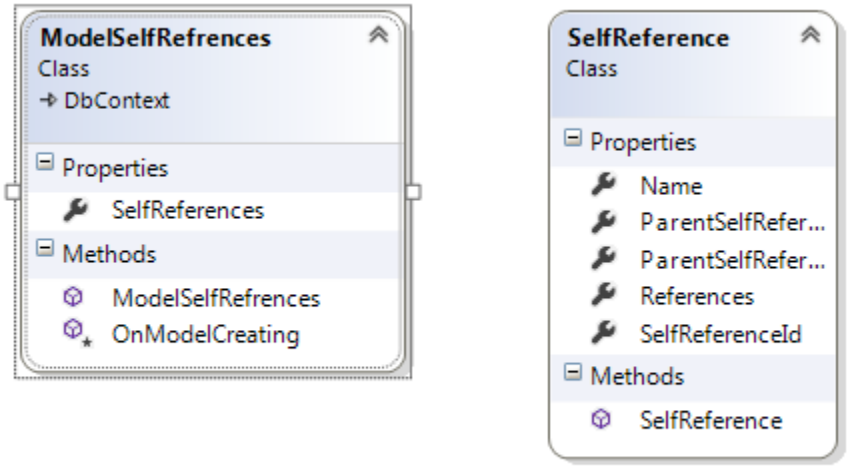


Scenariul 1 : O tabela ce se autoreferentiaza.

Cream o tabela numita *SelfReference* ce are urmatoarea structura:
SelfReferenceId – cheie primara cu auto increment.
Name – string.
ParentSelfReferenceId – poate fi null si este folosita pentru cheia straina.

Scriem clasele corespunzatoare – vedeti fisierul generat.

Diagrama de clase arata astfel:



Clasa POCO corespunzatoare acestei tabele va fi:

```
public class SelfReference
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int SelfReferenceId { get; private set; }
        public string Name { get; set; }
        public int? ParentSelfReferenceId { get; private set; }

        [ForeignKey("ParentSelfReferenceId")]
        public SelfReference ParentSelfReference { get; set; }

        public virtual ICollection<SelfReference> References { get; set; }
        public SelfReference()
        {
            References = new HashSet<SelfReference>();
        }
    }
```

iar contextul:

```
using System;
    using System.Data.Entity;
    using System.Linq;
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;
    using System.ComponentModel.DataAnnotations.Schema;
    public class ModelSelfRefrences : DbContext
    {
        public ModelSelfRefrences()
            : base("name=ModelSelfRefrences")
        {
        }

        public virtual DbSet<SelfReference> SelfReferences { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<SelfReference>()
```

```
            .HasMany(m => m.References)
            .WithOptional(m => m.ParentSelfReference);
        }
    }
```

Observati codul din OnModelCreating(). Descoperiti si alte metode acceptate de DbModelBuilder, adica Fluent API.

Scrieti cod pentru incarcare si afisare date in aceasta baza de date.

Observatie:
Pentru a intelege mai bine ce se intampla, utilizati modelul de programare "Database First" – deci creati o tabela ce se autorefentiaza si apoi generati modelul.

Observatii

Relatiile din baza de date sunt caracterizate de *grad*, *multiplicitate* si *directie*.

Gradul reprezinta numarul tipurilor de entitati ce participa intr-o relatie. Relatiile unare si binare sunt cele mai des utilizate.
Multiplicitatea este data de numarul de tipuri de entitati ce se gasesc la sfarsitul fiecarei relatii (Un client are mai multe comenzi – tabele implicate Client cu PK si Comanda cu FK). Exista multiplicitate 0...1, 1 si * (mai multe).
Directia poate fi unidirectionala sau bidirectionala.

EDM suporta un tip particular de relatie numit « *association type* ».
O relatie **« *association type* »** are gradul unar sau binar, multiplicitate 0...1, 1, * si o directie bidirectionala.

In acest exemplu, gradul este unar (este implicat numai tipul de entitate SelfReference), multiplicitatea este 0...1 si directia este bidirectionala.
In acest caz e posibil sa existe inregistrari « orfane ».

Scenariul 2 : Doua tabele ce partajeaza aceeasi cheie primara.

Dorim sa mapam o singura entitate la aceste doua tabele. Model de programare ca la Scenariul 1.

Observatie : Putem folosi si modelul de programare *Database First* pentru a vedea modelul generat.

Pentru a crea un model cu o singura entitate ce reprezinta cele doua tabele putem proceda astfel :

1. Cream o clasa noua in proiect, clasa derivata din **`DbContext`**.
2. Cream entitatea *Product* (POCO) folosind codul de mai jos (cele doua tabele sunt descrise intr-o singura entitate) :

```
public class Product
{
       [Key]
       [DatabaseGenerated(DatabaseGeneratedOption.None)]
       public int SKU { get; set; }
       public string Description { get; set; }
```

```
        public decimal Price { get; set; }
        public string ImageURL { get; set; }
}
```

3. Adaugam o proprietate **DbSet<Product>** la clasa derivata din **DbContext**.
4. Suprasriem metoda **OnModelCreating() din DbContext**

```
public class EF6RecipesContext : DbContext
{
        public DbSet<Product> Products { get; set; }
        public ProductContext() : base("name=ProductContext")
        {}
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
                base.OnModelCreating(modelBuilder);
                modelBuilder.Entity<Product>()
                        .Map(m =>
                        {
                        m.Properties(p => new {p.SKU, p.Description, p.Price});
                        m.ToTable("Product", "BazaDeDate");
                        })
                        .Map(m =>
                        {
                        m.Properties(p => new {p.SKU, p.ImageURL});
                        m.ToTable("ProductWebInfo", "BazaDeDate");
        });
}
}
```

Exemplu de utilizare

Prin contopirea a doua sau mai multe tabele intr-o singura entitate sau altfel spus prin divizarea unei entitati la doua sau mai multe tabele, putem trata toate partile ca o singura entitate logica. Acest proces este cunoscut si sub numele de ***vertical splitting***.
Efectul este ca avem nevoie de informatii aditionale in cazul cand dorim sa regasim o instanta a entitatii create.

Join-ul cerut de *Vertical Splitting*

```
SELECT
[Extent1].[SKU] AS [SKU],
[Extent2].[Description] AS [Description],
[Extent2].[Price] AS [Price],
[Extent1].[ImageURL] AS [ImageURL]
FROM [dbo].[ProductWebInfo] AS [Extent1]
INNER JOIN [dbo].[Product] AS [Extent2] ON [Extent1].[SKU] =
[Extent2].[SKU]
```

Inserare si regasire informatii folosind acest model.

```
using (var context = new ...Context())
{
        var product = new Product
        {
                SKU = 147,
                Description = "Expandable Hydration Pack",
                Price = 19.97M,
                ImageURL = "/pack147.jpg"
```

```
        };
        context.Products.Add(product);
        product = new Product
        {
                SKU = 178,
                Description = "Rugged Ranger Duffel Bag",
                Price = 39.97M,
                ImageURL = "/pack178.jpg"
        };
        context.Products.Add(product);
        product = new Product
        {
                SKU = 186,
                Description = "Range Field Pack",
                Price = 98.97M,
                ImageURL = "/noimage.jp"
        };
        context.Products.Add(product);
        product = new Product
        {
                SKU = 202,
                Description = "Small Deployment Back Pack",
                Price = 29.97M,
                ImageURL = "/pack202.jpg"
        };
        context.Products.Add(product);
        context.SaveChanges();
}

using (var context = new ...Context())
{
        foreach (var p in context.Products)
        {
        Console.WriteLine("{0} {1} {2} {3}", p.SKU, p.Description,
                p.Price.ToString("C"), p.ImageURL);
        }
}
```

Scenariul 3 : Mai multe entitati pentru aceeasi tabela

Avem o tabela cu multe coloane. Tabela este folosita des, dar nu toate coloanele sunt necesare intr-o anumita cerere. Trebuie sa cream mai multe entitati pentru aceasta tabela.

Indicatie

1. Cream o noua clasa in proiect, clasa derivata din **`DbContext`**.
2. Cream entitatea POCO *Photograph* folosind urmatorul cod.

```
public class Photograph
{
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int PhotoId { get; set; }
        public string Title { get; set; }
        public byte[] ThumbnailBits { get; set; }

        [ForeignKey("PhotoId")]
        public virtual PhotographFullImage PhotographFullImage { get; set; }
```

```
}
```

3. Cream o clasa POCO *PhotographFullImage* folosind urmatorul cod:

```
public class PhotographFullImage
{
      [Key]
      public int PhotoId { get; set; }
      public byte[] HighResolutionBits { get; set; }
      [ForeignKey("PhotoId")]
      public virtual Photograph Photograph { get; set; }
}
```

4. Adaugam o proprietate de tip **DbSet<Photograph>** la clasa derivata din **DbContext**.
5. Adaugam o proprietate de tip **DbSet<PhotographFullImage>** la clasa derivata din **DbContext**.
6. Suprascriem metoda **OnModelCreating**() din clasa **DbContext** astfel:

```
potected override void OnModelCreating(DbModelBuilder modelBuilder)
{
      base.OnModelCreating(modelBuilder);
      modelBuilder.Entity<Photograph>()
            .HasRequired(p => p.PhotographFullImage)
            .WithRequiredPrincipal(p => p.Photograph);
      modelBuilder.Entity<Photograph>()
            .ToTable("Photograph", "BazaDeDate");
      modelBuilder.Entity<PhotographFullImage>()
            .ToTable("Photograph", "BazaDeDate");
}
```

Exemplu de utilizare

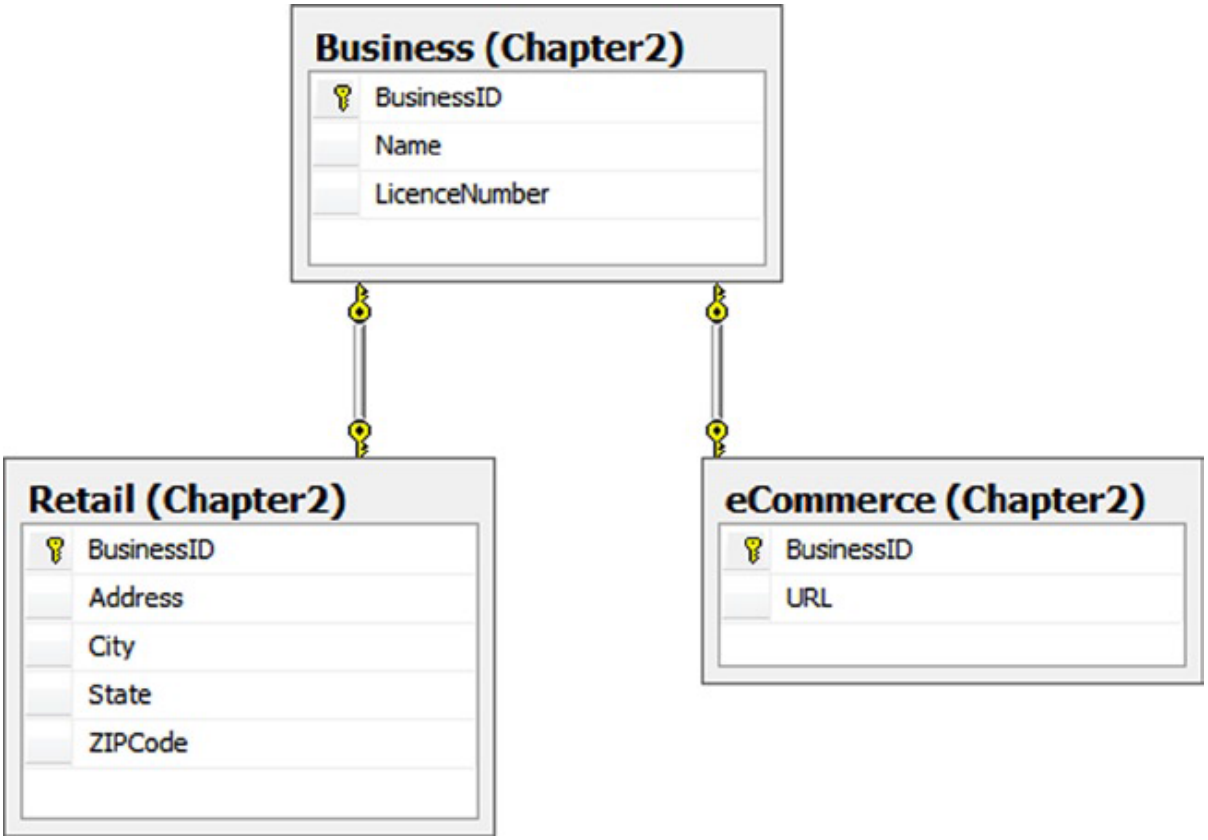
EF nu suporta incarcarea intarziata a proprietatilor unei entitati individuale. Pentru a obtine acest lucru ne folosim de faptul ca EF are suport pentru incarcarea intarziata a entitatilor asociate.
Am creat doua entitati, in una avem cheie primara iar in cealalta cheie secundara.
Am simulat in fapt divizarea tabelei din baza de date in doua tabele.
Observatie: Nu se permit inregistrari orfane in acest caz.

```
byte[] thumbBits = new byte[100];
byte[] fullBits = new byte[2000];
using (var context = new ...Context())
{
      var photo = new Photograph { Title = "My Dog",
      ThumbnailBits = thumbBits };
      var fullImage = new PhotographFullImage { HighResolutionBits =
      fullBits };
      photo.PhotographFullImage = fullImage;
      context.Photographs.Add(photo);
      context.SaveChanges();
}

using (var context = new ...Context())
{
      foreach (var photo in context.Photographs)
      {
            Console.WriteLine("Photo: {0}, ThumbnailSize {1} bytes",
```

```
                photo.Title, photo.ThumbnailBits.Length);
                // explicitly load the "expensive" entity,
                context.Entry(photo)
                      .Reference(p => p.PhotographFullImage).Load();
                Console.WriteLine("Full Image Size: {0} bytes",
                photo.PhotographFullImage.HighResolutionBits.Length);
          }
}
```

Scenariul 4: Modelare tabela prin mostenire tip.



Tabelele *Retail* si *eCommerce* sunt in relatie cu tabela *Business* si au aceeasi cheie primara.
Relatia dintre *Business* si *Retail* respectiv *eCommerce* este 0...1.
Acest caz poate fi tratat si ca o singura entitate.
Etape:

1. Cream o noua clasa derivata din **DbContext**.
2. Cream clasa *Business* (POCO):

```
[Table("Business", Schema = "BazaDeDate")]
public class Business
{
	[Key]
	[DatabaseGenerated(DatabaseGeneratedOption.Identity)]
	public int BusinessId { get; protected set; }
	public string Name { get; set; }
	public string LicenseNumber { get; set; }
}
```

3. Cream clasele *eCommerce* si *Retail* derivate din *Business*:

```
[Table("eCommerce", Schema = "BazaDeDate")]
public class eCommerce : Business
{
	public string URL { get; set; }
}

[Table("Retail", Schema = "BazaDeDate")]
public class Retail : Business
```

```
{
        public string Address { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string ZIPCode { get; set; }
}
```

4. Adaugam proprietatea de tip *DbSet<Business>* la clasa derivata din *DbContext*.

Exemplu de utilizare

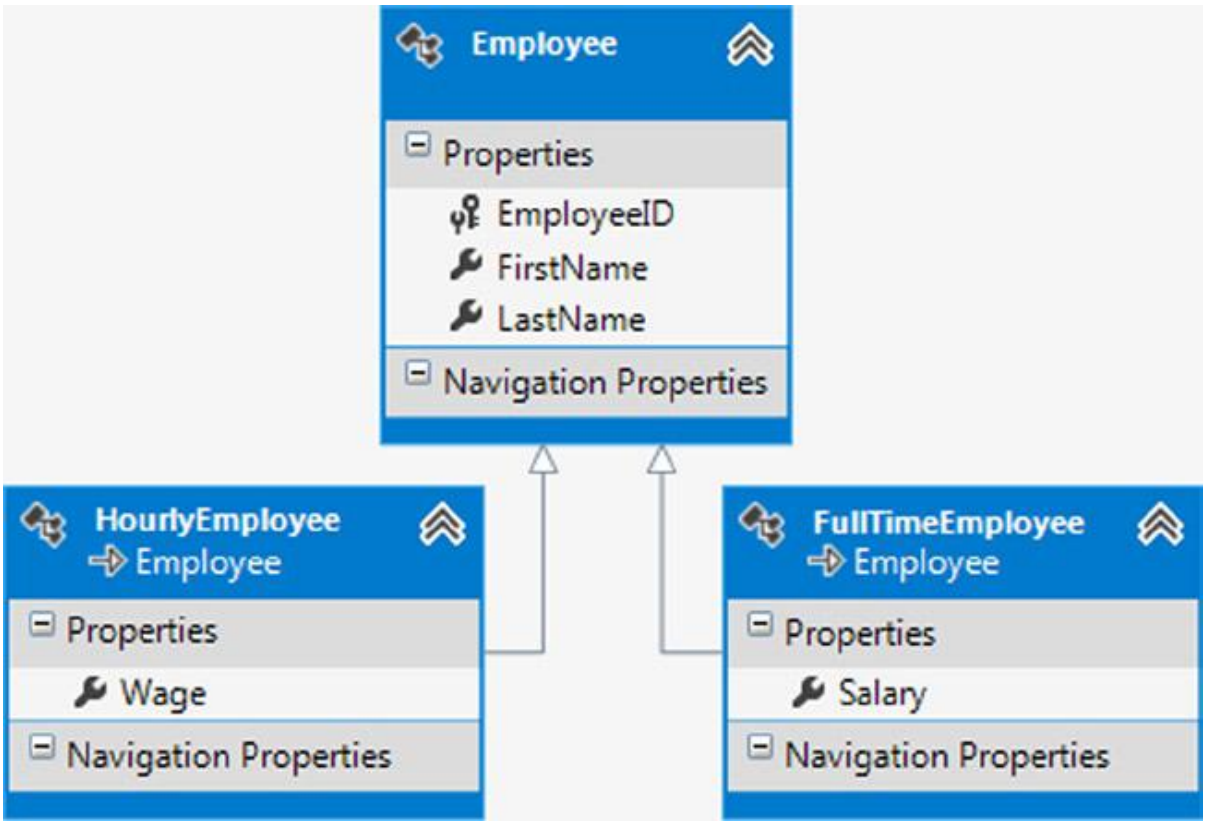
Tipul Business trebuie sa existe in orice situatie. Retail si/sau eCommerce nu sunt absolut necesare pentru ca relatiile intre Business si Retail, eCommerce sunt 1:0...1.
Fiecare tip derivat din Business, in cazul de fata, trebuie sa fie memorat intr-o tabela separata.

```
using (var context = new ...Context())
{
      var business = new Business { Name = "Corner Dry Cleaning",
      LicenseNumber = "100x1" };
      context.Businesses.Add(business);
      var retail = new Retail { Name = "Shop and Save", LicenseNumber =
      "200C",
      Address = "101 Main", City = "Anytown",
      State = "TX", ZIPCode = "76106" };
      context.Businesses.Add(retail);
      var web = new eCommerce { Name = "BuyNow.com", LicenseNumber =
      "300AB",
      URL = "www.buynow.com" };
      context.Businesses.Add(web);
      context.SaveChanges();
}

using (var context = new ...Context())
{
      Console.WriteLine("\n--- All Businesses ---");
      foreach (var b in context.Businesses)
      {
            Console.WriteLine("{0} (#{1})", b.Name, b.LicenseNumber);
      }
      Console.WriteLine("\n--- Retail Businesses ---");
      foreach (var r in context.Businesses.OfType<Retail>())
      {
            Console.WriteLine("{0} (#{1})", r.Name, r.LicenseNumber);
            Console.WriteLine("{0}", r.Address);
            Console.WriteLine("{0}, {1} {2}", r.City, r.State, r.ZIPCode);
      }
      Console.WriteLine("\n--- eCommerce Businesses ---");
      foreach (var e in context.Businesses.OfType<eCommerce>())
      {
            Console.WriteLine("{0} (#{1})", e.Name, e.LicenseNumber);
            Console.WriteLine("Online address is: {0}", e.URL);
      }
}
```

Scenariul 5: Modelare tabela folosind mostenire ierarhie.

Tabela *Employee* contine inregistrari pentru salariati platiti cu ora sau cei ce lucreaza tot timpul. Diferentierea este data de coloana *EmployeeType* ce are valoarea 1 pentru salariat ce lucreaza tot timpul si 2 pentru salariat angajat la plata cu ora. Din cauza ca aceasta coloana este folosita intr-o conditie, coloana nu mai apare in model.



Crearea modelului se poate face astfel:

1. Se creaza o clasa derivata din **`DbContext`**.
2. Se creaza clasa abstracta *Employee* (POCO):

```
public abstract class Employee
{
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int EmployeeId { get; protected set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
}
```

3. Din *Employee* se deriveaza clasele corespunzatoare tabelelor din figura de mai sus.

```
public class FullTimeEmployee : Employee
{
        public decimal? Salary { get; set; }
}

public class HourlyEmployee : Employee
{
        public decimal? Wage { get; set; }
```

```
}
```

4. Se aduaga o proprietate de tip **`DbSet<Employee>`** la clasa derivata din DbContext.
5. Se suprascrie metoda **`OnModelCreating()`** din **`DbContext`**:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
      base.OnModelCreating(modelBuilder);
      modelBuilder.Entity<Employee>()
            .Map<FullTimeEmployee>(m => m.Requires("EmployeeType")
            .HasValue(1))
            .Map<HourlyEmployee>(m => m.Requires("EmployeeType")
            .HasValue(2));
}
```

Exemplu de utilizare

In metoda **OnModelCreating**() ce configureaza entitatea, se stabileste ce inregistrari vor fi mapate (filtrate) la model.

```
using (var context = new ...Context())
{
      var fte = new FullTimeEmployee { FirstName = "Jane", LastName =
      "Doe",Salary = 71500M};
      context.Employees.Add(fte);
      fte = new FullTimeEmployee { FirstName = "John", LastName = "Smith",
      Salary = 62500M };
      context.Employees.Add(fte);
      var hourly = new HourlyEmployee { FirstName = "Tom", LastName =
      "Jones", Wage = 8.75M };
      context.Employees.Add(hourly);
      context.SaveChanges();
}
using (var context = new ...Context())
{
      Console.WriteLine("--- All Employees ---");
      foreach (var emp in context.Employees)
      {
            bool fullTime = emp is HourlyEmployee ? false : true;
            Console.WriteLine("{0} {1} ({2})", emp.FirstName, emp.LastName,
            fullTime ? "Full Time" : "Hourly");
      }
      Console.WriteLine("--- Full Time ---");
      foreach (var fte in context.Employees.OfType<FullTimeEmployee>())
      {
            Console.WriteLine("{0} {1}", fte.FirstName, fte.LastName);
      }
      Console.WriteLine("--- Hourly ---");
      foreach (var hourly in context.Employees.OfType<HourlyEmployee>())
      {
            Console.WriteLine("{0} {1}", hourly.FirstName,
                              hourly.LastName);
      }
}
```