مروری سریع بر Entity Framework

سید کاوه احمدی

آنچه از OOP باید بدانید

شی و کلاس

- **شی**
  - نمایش «چیزها/اشیا» در دنیای واقعی یا در دامنه مسئله
    - ماشین قرمز که آن پایین در پارکینگ پارک شده

- **کلاس**
  - تمام اشیا از یک نوع را مشخص می‌کند
    - ماشین

متدها و پارامترها

- اشیا دارای عملیاتی هستند که می‌توانند فراخوانی شوند. (به آنها متد گفته می‌شود)
- متدها ممکن است دارای پارامترهایی برای انتقال اطلاعات اضافی که برای اجرا نیاز دارند باشند.

موارد دیگر

- نمونه‌های متعددی می‌تواند از روی یک کلاس ایجاد شود.
- یک شی دارای خصوصیات متعددی است: مقادیر در فیلدها ذخیره می‌شوند.
- کلاس مشخص می‌کند که یک شی دارای چه فیلدهایی است اما هر شی مجموعه مقادیر ویژه خود را ذخیره می‌کند. این مقادیر وضعیت شی را مشخص می‌کنند.

ساختار پایه کلاس

```
public class ClassName
{
    Fields
    Constructors
    Methods
}
```

فیلدها

- فیلدها مقادیر شی‌ها را ذخیره می‌کند.
- به عنوان متغیرهای یک نمونه نیز شناخته می‌شوند.
- فیلدها وضعیت شی را مشخص می‌کند.

```
public class Point
{
    private int x;
    private int y;

    Further details omitted.
}
```

visibility modifier → private
type → int
variable name → x

```
private int x;
```

- نام کلاس‌ها را با حروف بزرگ شروع کنید. نام فیلدها را با حروف کوچک.

معرف دسترسی (visibility modifier) (بیان غیر دقیق)

- Public: فیلد یا متد می‌تواند از همه جا فراخوانی شود.
- Private: فراخوانی فقط از همان کلاس.
  - به ما کمک می‌کند پیاده‌سازی‌ها را از بیرون پنهان کند.
- فیلدها معمولا private هستند. سازنده‌ها معمولا public و متدها فقط وقتی که ضروری است باید public تعریف شوند.
- مشخص کننده سطح دسترسی
- سطوح دسترسی دیگری نیز وجود دارد. (در آینده)

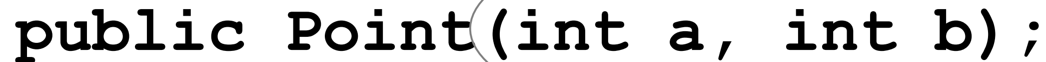
سازنده‌ها

- یک شی را مقداردهی اولیه می‌کند.
- همنام با کلاس هستند.
- ارتباط نزدیکی با فیلدها دارد.
  - برای ذخیره‌سازی مقادیر اولیه فیلدها مورد استفاده قرار می‌گیرند.
  - برای این کار از پارامترها استفاده می‌کنند.

```
public class Point
{
    private int x;
    private int y;

    public Point(int a, int b)
    {
        x = a;
        y = b;
    }
    ...
}
```

ایجاد شی

in class Point:



formal parameter

in class Point:

actual parameter

متدها (Methods)

- متدها رفتارهای یک شی را پیاده‌سازی می‌کنند.
- متدها دارای ساختاری مشابه و دارای یک هدر (header) و یک بدنه (body) است.
- متدهای دسترسی (Accessor methods) اطلاعاتی در مورد شی را ارائه می‌کنند.
- متدهای تغییر (Mutator methods) وضعیت یک شی را تغییر می‌دهند.
- انواع دیگر متدها برای انجام کارهای دیگر مورد استفاده قرار می‌گیرد.

متدها (Methods)

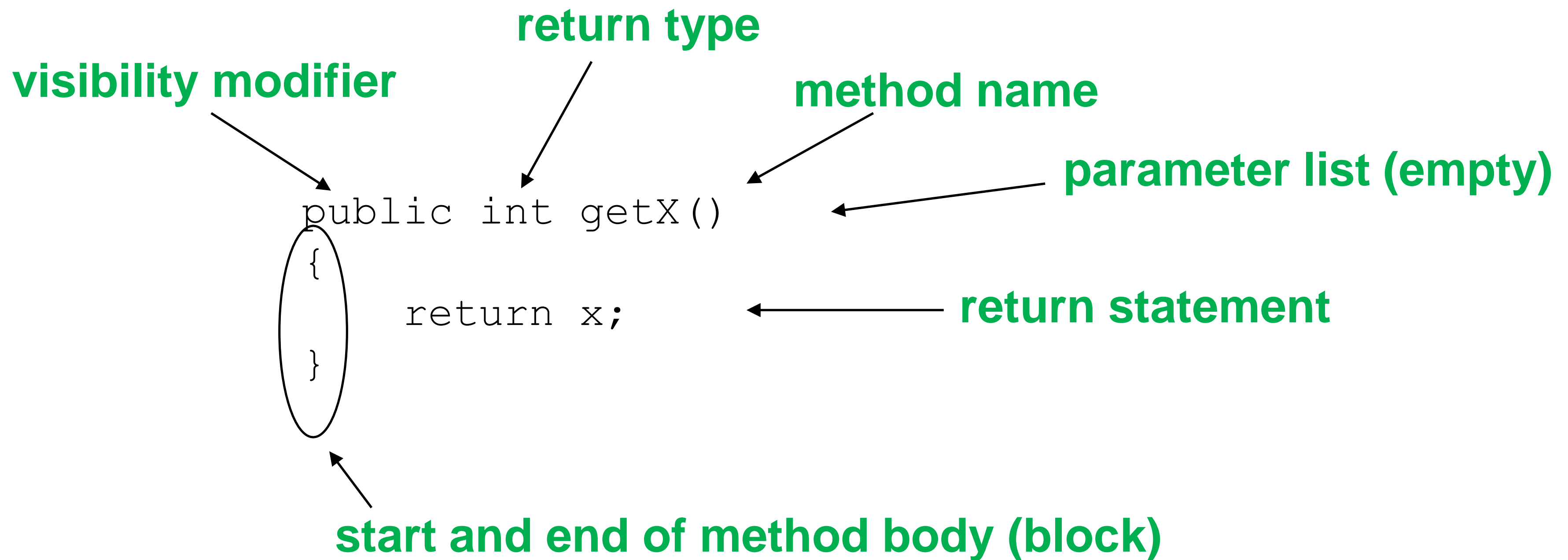
- سرآیند (header)، شناسنامه متد است!

```
public int getX()
```

- سرآیند به ما می‌گوید:
  - نام متد
  - چه پارامترهایی می‌گیرد
  - چه نتیجه‌ای بازمی‌گرداند
  - دسترسی سایر اشیا و کلاس‌ها به آن

- بدنه پیاده‌سازی متد را در بر می‌گیرد.

(Methods) متدها



متدهای دسترسی (Accessor methods)

- یک متد دسترسی همیشه نوع بازگشتی‌ای غیر از `void` دارند.
- یک متد دسترسی یک مقدار (نتیجه) که در سرآیند مشخص شده را باز می‌گرداند.
- متد دارای یک عبارت `return` برای بازگرداندن مقدار است.
- بازگرداندن به معنی چاپ کردن نیست!

متدهای دسترسی (Accessor methods)

- دستیابی به اطلاعاتی پیرامون شی.
- به آنها متدهای getter نیز گفته می‌شود.

```
public class Point
{
    private int x;
    private int y;

    public Point(int a, int b)
    {
        x = a;
        y = b;
    }

    public int getX()
    {
        return x;
    }
}
```

تعریف فیلد (خصوصیت)

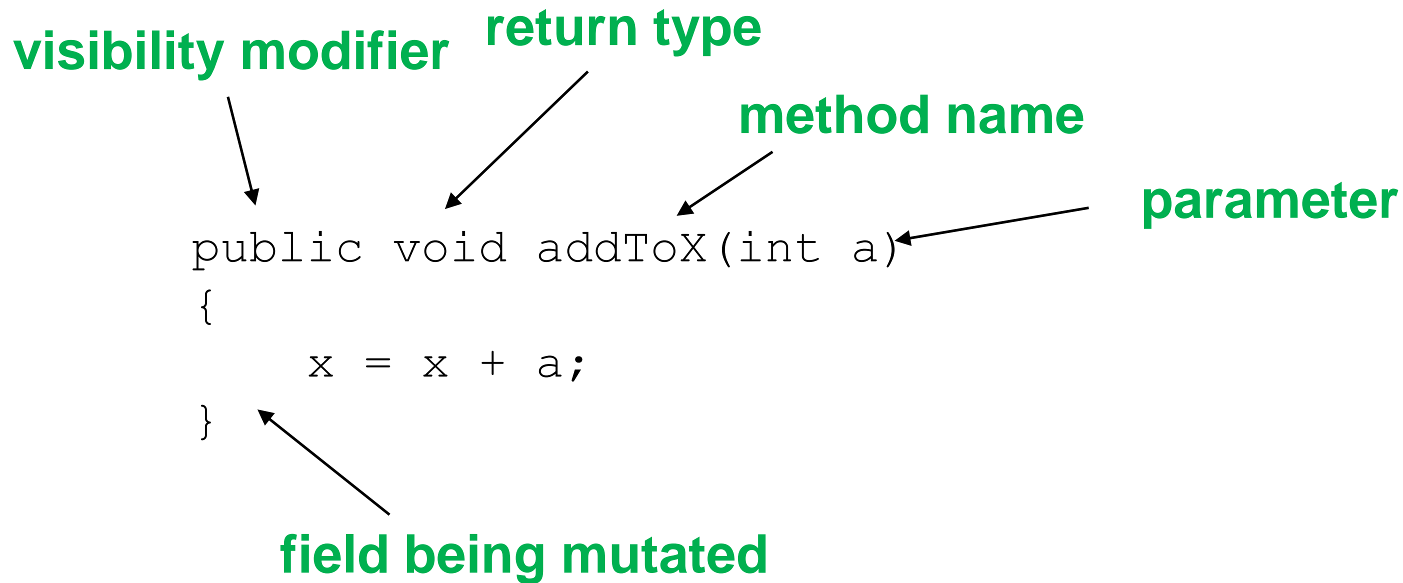
مقداردهی اولیه به خصوصیت
(سازنده با پارامتر ورودی)

دسترسی به مقدار خصوصیت

متدهای تغییر (Mutator methods)

- ساختار متد مشخص: سرآیند و بدنه.
- برای تغییر دادن وضعیت شی مورد استفاده قرار می‌گیرد.
  - به وسیله تغییر دادن مقدار یک یا چند فیلد.
    - به طور معمول شامل عملگر انتساب.
  - معمولا پارامتر دریافت می‌کنند.

(Mutator methods) متدهای تغییر



متدهای تغییر set

- فیلدها معمولا متدهای تغییر (set) ویژه خود را دارند.
- شکل مشخص و ساده‌ای دارند:
  - نوع بازگشتی `void`
  - نام متد مرتبط به نام فیلد
  - دارای یک پارامتر هم نوع با فیلد
  - یک عملگر انتساب

یک متد set مرسوم

```
public void setX(int a)
{
    x = a;
}
```

می‌توان نتیجه گرفت x یک فیلد با نوع int است:

```
private int x;
```

تغییرات محافظت شده

- یک متد set لزوما نباید پارامتر را به فیلد منتسب کند.
- پارامتر ممکن است به لحاظ معتبر بودن مورد بررسی قرار گیرد و در صورت نامناسب بودن رد شود.
- در نتیجه متدهای تغییر از فیلدها محافظت می‌کنند.
- متدهای تغییر از محصورسازی پشتیبانی می‌کنند.

پنهان سازی اطلاعات در زبان‌های شی‌گرا

- کلاس‌ها امکان پنهان سازی اطلاعات را فراهم می‌آورند.
  - با تعریف به صورت private
    - به نحوی جزئیات پیاده‌سازی کلاس را پنهان می‌کند.
- ایجاد متدهای getter و setter برای دسترسی به اطلاعات.
- اما توسعه دهندگان C# معتقدند این روش خسته کننده است!
  - مقایسه کنید x = foo.getX() را با x = foo.x;
  - ایده ادغام مزایای استفاده از متدهای دسترسی است در حالی که از سینتکس دسترسی مستقیم استفاده می‌کنیم است!

C# Properties

- فراهم کردن دسترسی محافظت شده به داده‌ها
  - همانند متدهای دسترسی
- اما دارای نحوی همانند دسترسی مستقیم به داده‌ها
- قابلیتی ساده که قرار است کارها را ساده کند.

C# Properties

```
class Person
{
    private string _name;
    
    public string name
    {
        get
        {
            return _name != null ? _name : "NA";
        }
        set
        {
            _name = value;
        }
    }
}
```

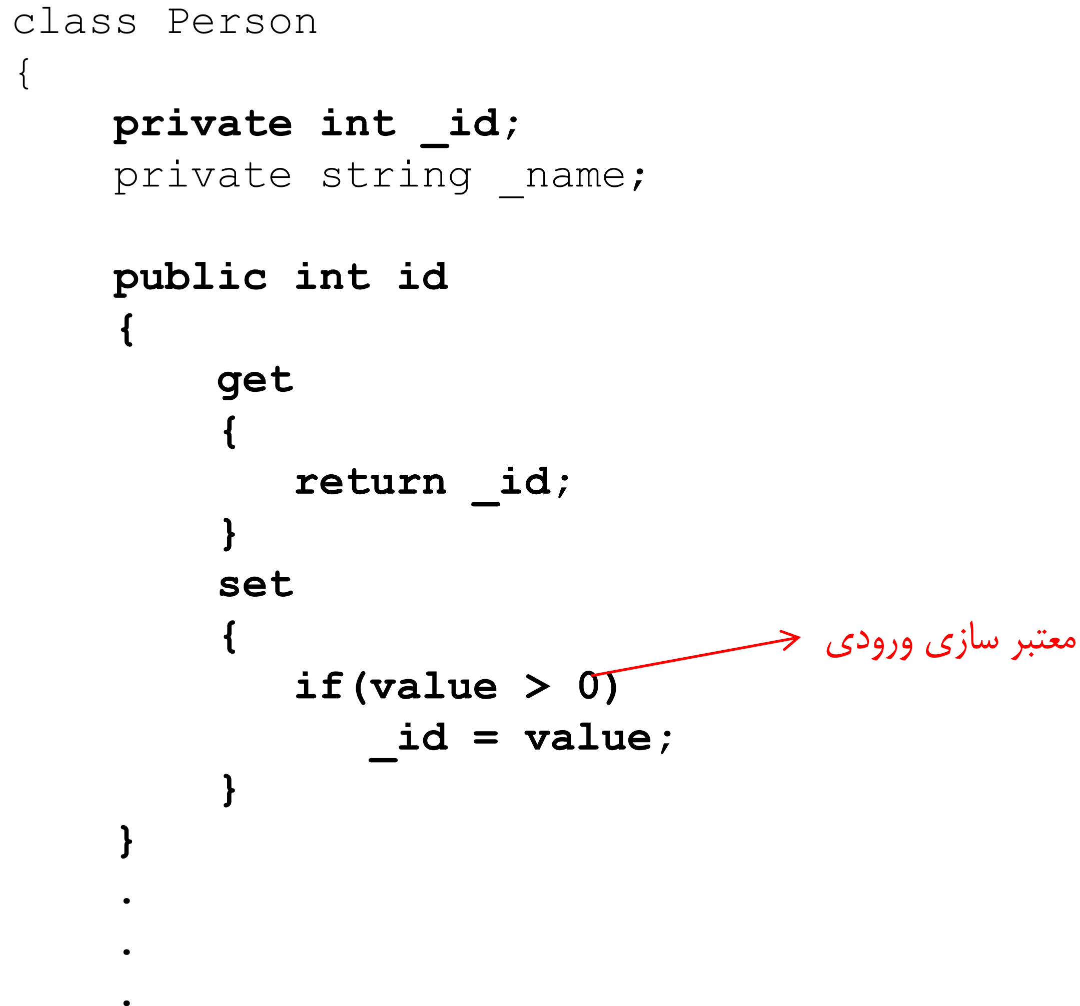
Field

Property

مقدار ارسال شده اینجا ذخیره شده!

```
// Test code
Person p = new Person();

// Call set accessor
p.name = "Kaveh";

// Call get accessor
System.Console.Write(p.name);
```

C# Properties



C# Properties

- فیلد و property همزمان
  - چه اشکالی دارد؟
  - چه کاربردی دارد؟

```
class Person
{
    public int ID
    {
        get; set;
    }
}
```

```
// Test code
Person p = new Person();

// Call set accessor
p.ID = 123;

// Call get accessor
System.Console.Write(p.ID);
```

```
class Person
{
    public int ID
    {
        get; private set;
    }
}
```

```
class Person
{
    private int _id;
    private string _name;

    public int id
    {
        get
        {
            return _id;
        }
        set
        {
            if (value > 0)
                _id = value;
        }
    }

    public string name
    {
        get
        {
            return _name != null ? _name : "NA";
        }
        set
        {
            _name = value;
        }
    }
}
```

```
class Person
{
    public int _id
    {
        get;

        set
        {
            if (value > 0)
                _id = value;
        }
    }

    public string _name
    {
        get
        {
            return _name != null ? _name : "NA";
        }
        set;
}
```

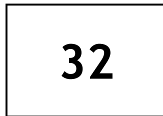
C# Properties

- ایجاد شی بدون داشتن سازنده
  - نیازی به سازنده‌های مختلف نخواهیم داشت

```
Person p1 = new Person() {
   id = 25
};


Person p2 = new Person() {
   name = "Kaveh"
};


Person p3 = new Person() {
   id = 25,
   name = "Kaveh"
};
```

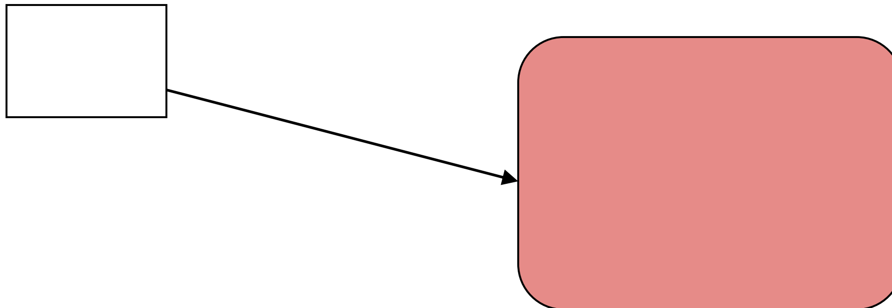
متغیرهای مقداری و ارجاعی

`int i;`



نوع مقداری

`SomeObject obj;`

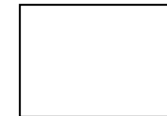


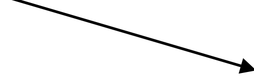
نوع ارجاعی

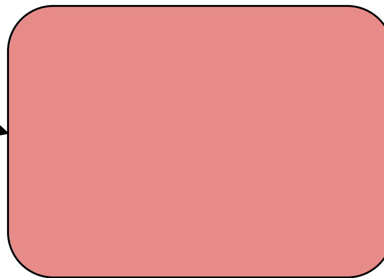
متغیرهای مقداری در برابر ارجاعی

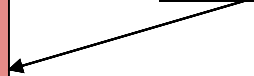
`ObjectType a;`

`ObjectType b;`









`b = a;`

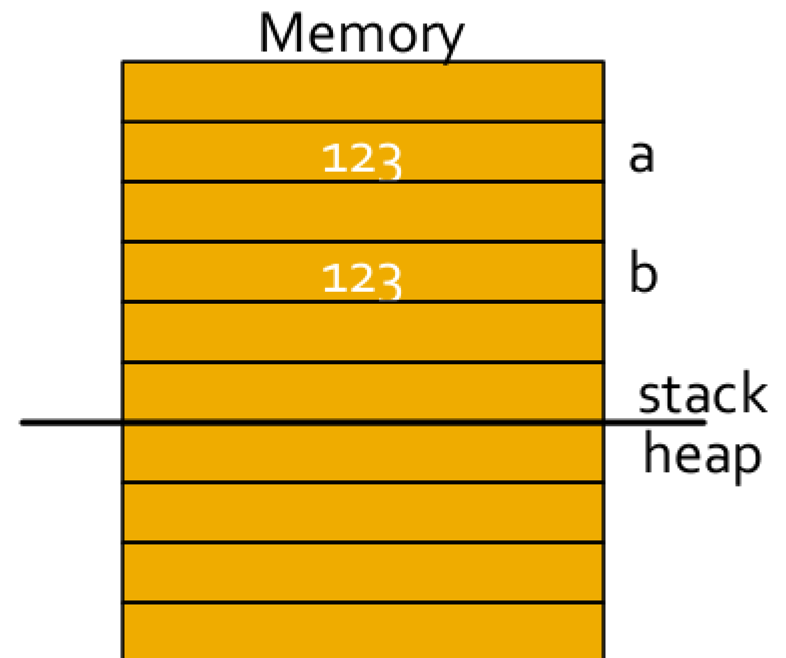
`int a;`

`int b;`

قالب حافظه: نوع‌های مقداری

- مقدار واقعی را در خود ذخیره می‌کند به مکان حافظه را.
- کپی کردن متغیرهای مقداری، یک کپی واقعی ایجاد می‌کند.

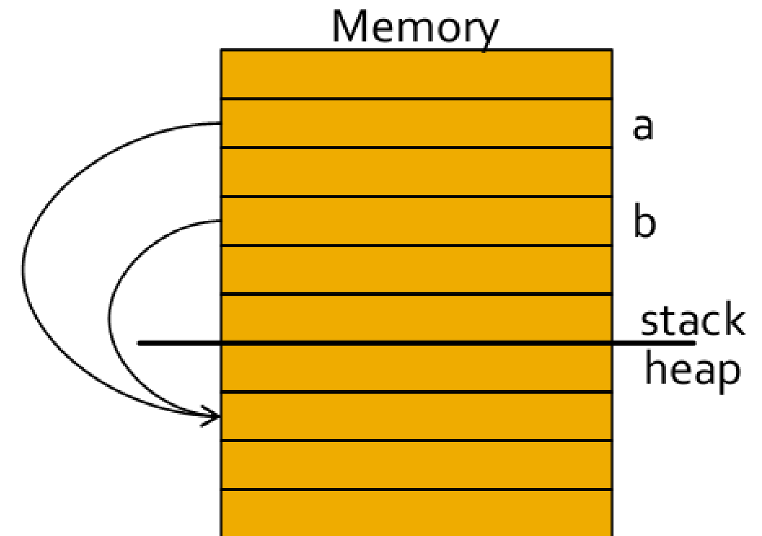
```
int a = 123;
int b = a;
```



قالب حافظه: نوع‌های ارجاعی

- ارجاع به یک مکان از حافظه
  - بسیار شبیه به اشاره‌گرها در زبان‌هایی همانند C/C++.
- می‌تواند با null تنظیم شود:

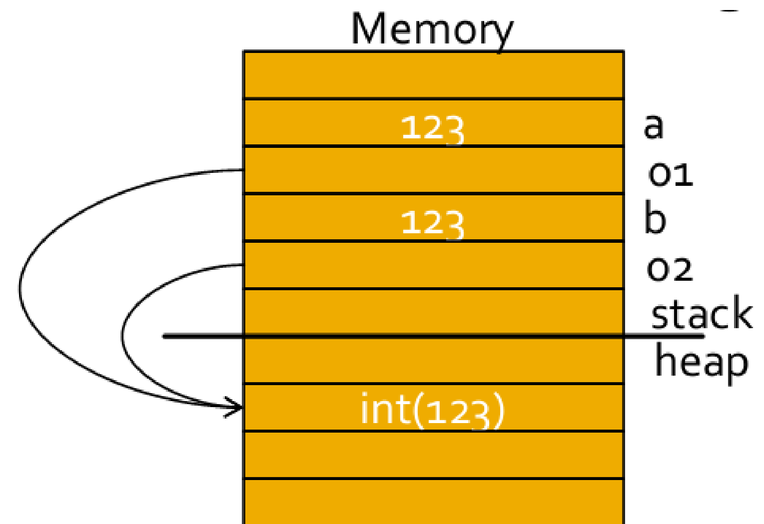
```
TypeA a = new TypeA();
TypeA b = a;
```



Boxing و Unboxing

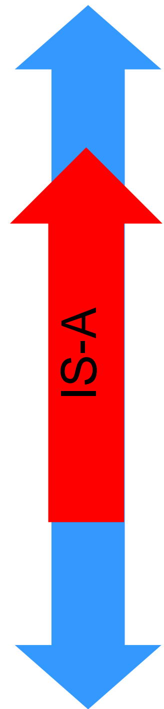
- نوع‌های مقداری شی نیستند.
  - این امر در اکثر موارد کارآیی را افزایش می‌دهد.
  - اما متغیرهای مقداری می‌توانند هنگام نیاز به شی تبدیل شوند.
  - به این امر Boxing گفته می‌شود. به عمل عکس Unboxing گفته می‌شود.

```
int a = 123;
object o1 = a;
object o2 = o1;
int b = (int) o2;
```



سلسله مراتب وراثت

افزایش تجرید



کاهش تجرید

Object Class

Animal

Mammal

Bird

Dog

Cat

Chicken

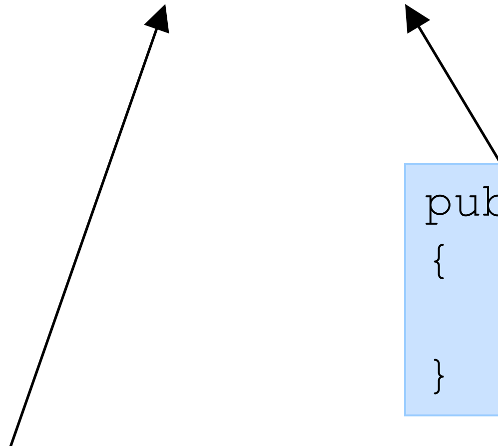
Sparrow

Poodle

Dalmatian

وراثت در C#

```
public class Animal
{
      ...
}
```



```
public class Mammal : Animal
{
      ...
}
```

```
public class Bird : Animal
{
      ...
}
```

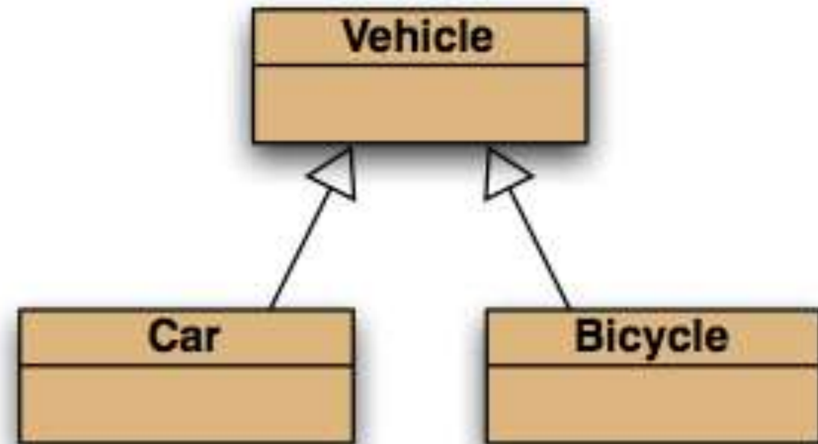
زیرکلاس‌ها و زیرنمونه‌ها

- کلاس‌ها، نوع‌هایی را تعریف می‌کنند.
- زیرکلاس‌ها، زیرنوع‌هایی را تعریف می‌کنند.
- اشیای زیرکلاس‌ها زمانی که اشیایی که از نوع ابرکلاس نیاز است می‌تواند مورد استفاده قرار گیرد.

  (این امر جایگزینی – **substitution** خوانده می‌شود.)

زیرنوع‌ها و انتساب

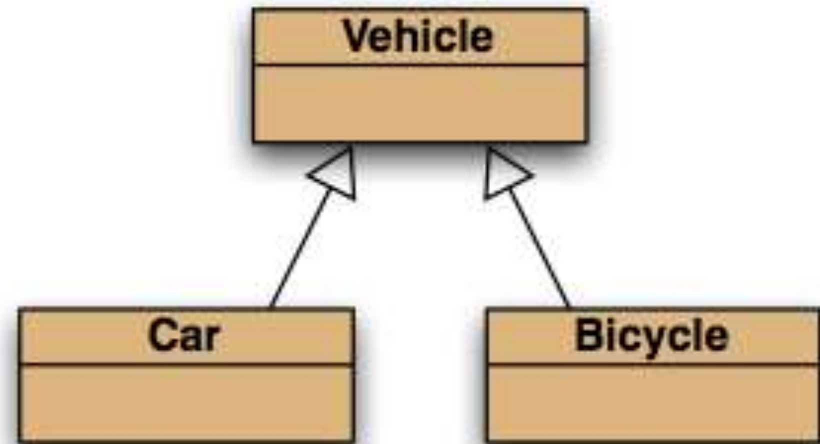
اشیای یک زیرکلاس
می‌تواند به متغیرهای از نوع
ابرکلاس منتسب شود.




```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```

زیرنوع‌ها و انتساب

اشیای از نوع ابرکلاس می‌تواند به متغیرهای از نوع زیرکلاس منتسب شود.
مشروط بر اینکه متغیر به شی‌ای از نوع زیرکلاس اشاره کند.
برای بررسی این امر نیاز به تغییر نوع (Casting) از نوع صریح است.




```
Vehicle v1 = new Car();
Car c1 = (Car) v1;
Bicycle b1 = (Bicycle)v1; //Run Time Error
```

نیاز به گروه‌بندی اشیا

- بسیاری از برنامه‌های کاربردی شامل یک مجموعه (collection) از اشیا هستند.
  - Personal organizers.
  - Library catalogs.
  - Student-record system.

- تعداد اقلامی که باید ذخیره شوند مشخص نیست.
  - اقلامی اضافه می‌شوند.
  - اقلامی حذف می‌شوند.

استفاده از آرایه به عنوان مجموعه

- عیب: طول ثابت
- مزیت: از نوع داده مطمئن هستیم!

```
string[] names = new string[2];
names[0] = "ali";
names[1] = "ahmad";


string[] names = {"ali", "ahmad"};
```

کتابخانه‌های کلاس (Class libraries)

- یک مجموعه از کلاس‌های سودمند.
  - لازم نیست همه چیز را از اول بنویسیم.
    - قرار نیست چرخ را دوباره اختراع کنیم، از آن استفاده می‌کنیم.
- گروه‌بندی اشیا:
  - `System.Collections` یک بسته در C# است شامل کلاس‌هایی برای انجام این کار.
  - نیاز به الحاق (using) این بسته‌ها برای استفاده است.
  - الحاق باید پیش از تعریف کلاس باشد.

ArrayList

- یک کلاس از کتابخانه کلاس C#
- ArrayList یک کلاس عمومی برای پیاده‌سازی یک مجموعه از اشیا است.
- عیب: از نوع داده مطمئن نیستیم / نوع داده ذخیره شده برای کامپایلر مشخص نیست و داده‌ها به عنوان object ذخیره می‌شوند بنابراین نیاز به casting خواهیم داشت.
- مزیت: محدودیت در طول ندارد.

ArrayList

```
ArrayList names = new ArrayList();
names.Add(“ali”);
names.Add(“ahmad”);
```

```
ArrayList names = new ArrayList() {‘ali’, ‘ahmad’};
```

- دسترسی

```
string n = names[0];
```

ArrayList

- در یک ArrayList می‌توان اشیای از نوع‌های مختلف را قرار داد و این امر می‌تواند ما را اجرای برنامه دچار مشکل کند.

```
ArrayList names = new ArrayList();
Person p = new Person("Ali");
string s = "ahmad";

names.Add(p);
names.Add(s);
```

پیمایش ArrayList

```
for(int i = 0; i < persons.Count; i++)
{
        Console.WriteLine(((Person)persons[i]).Name);
}
```

- مناسب برای زمانی که اشیا از نوع‌های مختلفی در ArrayList قرار دارند:

```
persons.Add(new Person { ID=1, Name="Ali"});
persons.Add("A Simple String Object");

for(int i = 0; i < persons.Count; i++)
{
       if(persons[i] is Person)
             Console.WriteLine(((Person)persons[i]).Name);
}
```

پیمایش ArrayList

- استفاده از foreach

```
foreach (Person p in persons)
{
        Console.WriteLine(p.Name);
}
```

Generic Lists

- مزیت: محدودیت در طول ندارد.
- مزیت: از نوع داده مطمئن هستیم!

```
List<string> names = new List<string>();

names.Add(“ali”);

names.Add(“ahmad”);


List<string> names = new List<string>() {“ali”,
   “ahmad”};
```

نوع بندی ضمنی – Type Inference

- `var`
  - به کامپایلر می‌گوید خودش در مورد نوع داده تصمیم بگیرد.

```
var i = 1;
```

کامپایلر نوع متغیر را `int` در نظر می‌گیرد. بنابراین دستور انتساب زیر منجر به خطای کامپایل می‌شود:

```
i = "Hello World";
```

```
var names = new List<string>() {'ali', 'ahmad'};
```

نوع بندی ضمنی – Type Inference

- از نوع بندی ضمنی نمی‌توان برای فیلدها، مقادیر بازگشتی متدها، پارامترها و... استفاده کرد. (فقط متغیرهای محلی می‌تواند باشد)
- این متغیرها باید در زمان تعریف مقدار دهی شوند تا نوع آنها مشخص شود.
- به معنی نوع داینامیک نیست و در لحظه کامپایل نوع آن مشخص می‌شود.

Generic Lists

- تعریف Generic Lists از نوع ابرکلاس برای ذخیره اشیایی از نوع زیر کلاس‌ها

```
public class Persons
{
    public string Name { get; set; }
}

public class Student : Persons
{
    public string StudentNumber { get; set; }
}

public class Teacher : Persons
{
    public int TeacherID { get; set; }
}
```

Generic Lists

```
var st = new Student { Name = "Student", StudentNumber = "N4534" };
var tch = new Teacher { Name = "Teacher", TeacherID = 324324 };

var persons = new List<Persons>();
persons.Add(st);
persons.Add(tch);

foreach (var person in persons)
{
        Console.WriteLine(person.Name);
}
```

Generic Lists

```
foreach (var person in persons)
{
 if (person.GetType() == typeof(Student))
 {
  var stp = (Student)person;
  Console.WriteLine(string.Format("{0} - {1}", stp.Name, stp.StudentNumber));
 }
 else
 {
  var stp = (Teacher)person;
  Console.WriteLine(string.Format("{0} - {1}", stp.Name, stp.TeacherID));
 }
}
```

```
public abstract class Persons
{
    public string Name { get; set; }
    public abstract string GetInfo();
}

public class Student : Persons
{
    public string StudentNumber { get; set; }

    public override string GetInfo()
    {
        return string.Format("{0} - {1}", Name, StudentNumber);
    }
}

public class Teacher : Persons
{
    public int TeacherID { get; set; }

    public override string GetInfo()
    {
        return string.Format("{0} - {1}", Name, TeacherID);
    }
}
```

```
foreach (var person in persons)
{
       Console.WriteLine(person.GetInfo());
}
```

مروری سریع بر LINQ

LINQ چیست؟

- LINQ مخفف عبارت Language-Integrated Query است
  .
- یک تکنولوژی که با آن می‌توان با همه نوع منبع داده‌ای به یک روش یکسان ارتباط برقرار کرد.
- LINQ یک راه یکسان برای برقراری ارتباط و بازیابی اطلاعات از هر شی که رابط IEnumerable را پیاده سازی کرده باشد فراهم می‌کند.
  - آرایه‌ها و مجموعه‌ها (Collections)، پایگاه داده‌های رابطه‌ای و اسناد XML

LINQ چیست؟

- LINQ یک مجموعه دستورات توانمند ارائه می‌کند که به وسیله آنها می‌توان پرسجوهایی را روی منبع داده‌ای اجرا کرد.
- در این پرسجوها می‌توان از مواردی همانند Joinها، توابع Aggregation، مرتب سازی، فیلتر و... استفاده کرد.
  - این دستورات را language-level می‌نامند.

The LINQ Project

C# 3.0	Visual Basic 9.0	Others

.NET Language Integrated Query

LINQ to Objects	LINQ to DataSets	LINQ to SQL	LINQ to Entities	LINQ to XML

Objects

Relational

XML

مزایا

1. امکان پذیر ساختن اشکال زدایی از پرسجوها.
   - زیرا پرسجوها بجای اینکه به صورت رشته‌ای نوشته شوند، بخشی از کد هستند. به همین دلیل این تکنیک `Language Integrated Query` نامگذاری شده است.
2. استفاده از یک گرامر ثابت و یکسان جهت نوشتن پرسجو بدون توجه به نوع منبع داده.
3. افزایش سرعت در تولید نرم افزار با توجه به گرفتن خطاهای برنامه در زمان اجرا
4. افزایش امنیت وب سایت‌ها (در مواردی از قبیل `Sql Injection`)

اولین برنامه با LINQ

```
List<string> files = new List<string>()
{
        "report1.pdf",
        "report2.pdf",
        "order.doc",
        "data.txt",
        "letter.doc",
        "aftereffects.doc",
        "letter2.doc"
};

var docFiles =
        from f in files
        select f;

foreach (var docFile in docFiles)
{
        Console.WriteLine(docFile);
}
```

اولین برنامه با LINQ

```
List<string> files = new List<string>()
{
        "report1.pdf",
        "report2.pdf",
        "order.doc",
        "data.txt",
        "letter.doc",
        "aftereffects.doc",
        "letter2.doc"
};

var docFiles =
        from f in files
        where f.EndsWith(".doc")
        select f;

foreach (var docFile in docFiles)
{
        Console.WriteLine(docFile);
}
```

ترتیب دستورات در LINQ

- همانند ترتیب اجرای بندها در T-SQL است:

- (7) SELECT
- (8) TOP
- (1) FROM
- (3) JOIN
- (2) ON
- (4) WHERE
- (5) GROUP BY
- (6) HAVING
- (10) ORDER BY

شروع دستور LINQ با FROM این امکان را فراهم می‌کند که با مشخص شدن منبع داده، که می‌خواهیم روی آن کوئری را اجرا کنیم، قابلیت Intellisense می‌تواند Propertyهای مربوطه را تشخیص داده و فعال شود.

کلاس Developer

```
public class Developer
{
        private string _name;
        private string _language;
        private int _age;

        public string Name
        {
                get { return _name; }
        }

        public string Language
        {
                get { return _language; }
        }

        public int Age
        {
                get { return _age; }
        }

        public Developer(string n, string l, int a)
        {
                _name = n;
                _language = l;
                _age = a;
        }
}
```

یک لیست از Developer

```
List<Developer> devs = new List<Developer>();

devs.Add(new Developer("Ali", "C#", 20));
devs.Add(new Developer("Ahmad", "C#", 23));
devs.Add(new Developer("Jalal", "VB", 32));
```

چند پرسجو روی Developer

- توسعه دهندگاه C#

```
var cSharpers =
       from dev in devs
       where dev.Language.Equals("C#")
       select dev;
```

- توسعه دهندگان با سن بیشتر از ۳۰ سال

```
var oldDevs =
       from dev in devs
       where dev.Age > 30
       select dev;
```

چند پرسجو روی Developer

- انتخاب فقط برخی از فیلدها

```
var oldDevsName =
        from dev in devs
        where dev.Age > 30
        select new { dev.Name, More20 = dev.Age - 20 };

foreach (var dev in oldDevsName)
{
        Console.WriteLine(dev.Name + " " + dev.More20 + " " + dev.Age);
}
```

کسر سن از ۲۰ را نشان می دهد.

استفاده از Age مجاز نیست چون در پرسجو انتخاب نشده است.

خطای کامپایل – در پرسجو انتخاب نشده است

عملگر OfType

- فیلتر کردن بر حسب نوع در مجموعه
  - مثال زیر فقط اعضا از نوع Developer بازگردانده می‌شوند

```
ArrayList complexList = new ArrayList();

complexList.Add("Test String 1");
complexList.Add(new DateTime(2011, 1, 1));
complexList.Add(10);
complexList.Add(new Developer("Ali", "C#", 20));

var query = complexList.OfType<Developer>();

foreach (var item in query)
       Console.WriteLine(item.Name + " " + item.Language);
```

کلاس مشتری

```
class Customer
{
        public int Id { get; set; }
        public string Name { get; set; }
        public string Family { get; set; }
        public string City { get; set; }
}
```

کلاس سفارش

- یک مشتری صاحب هر سفارش است.
  - فیلد CustomerID ارتباط با اشیا از کلاس Customer را برقرار می‌کند.

```
class Order
{
        public int CustomerId { get; set; }
        public DateTime OrderDate { get; set; }
}
```

مشتریان و سفارشات!

```
List<Customer> customers = new List<Customer>()
{
   new Customer {Id = 1, Name = "Ali", Family = "Ahmadi", City = "Tehran"},
   new Customer {Id = 2, Name = "Amir", Family = "Nasiri", City = "Tehran"},
   new Customer {Id = 3, Name = "Arash", Family = "Novin", City = "Yazd"}
};

List<Order> orders = new List<Order>()
{
   new Order { CustomerId = 1, OrderDate = new DateTime(2010,1,1) },
   new Order { CustomerId = 2, OrderDate = new DateTime(2011,1,1)}
};
```

هر سفارش (Order) توسط یک مشتری (Customer) داده شده است

عملگر Join

- باز گرداندن تاریخ سفارش و نام سفارش دهنده
  - اطلاعات در دو شی متفاوت از کلاس‌های Customer و Order ذخیره شده‌اند.
  - عملگر Join ارتباط را با توجه به تساوی مقدار فیلد مشخص شده توسط کلمه کلیدی on برقرار می‌کند.

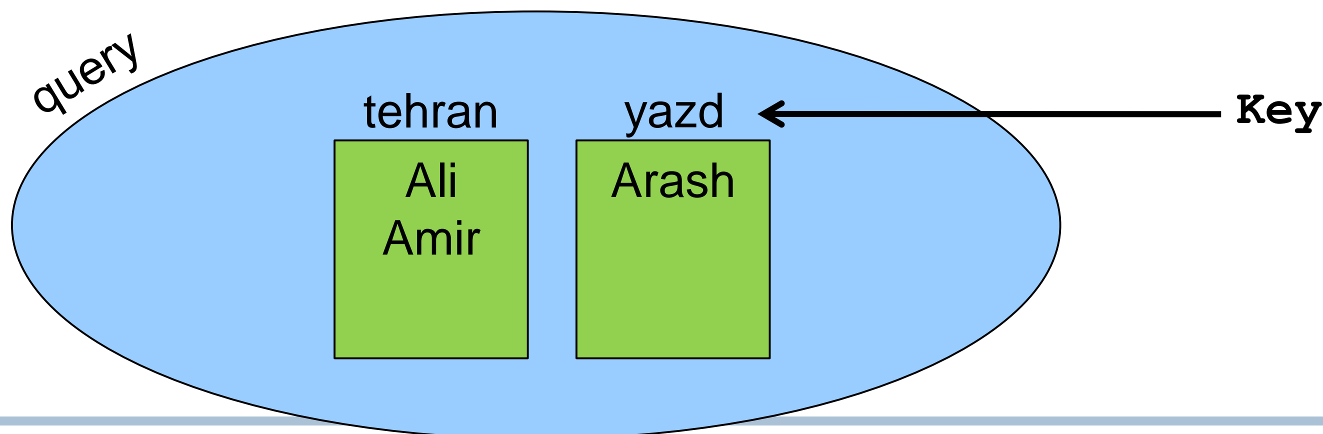
عملگر Join

```
var query =
       from c in customers
       join o in orders on c.Id equals o.CustomerId
       select new
       {
              FullName = c.Name + " " + c.Family,
              c.Id,
              o.OrderDate
       };

foreach (var item in query)
       Console.WriteLine(item.FullName +
              " ,ID= " + item.Id +
              " ,Order Date= " + item.OrderDate);
```

دسته‌بندی رکوردها با GroupBy

- گروه بندی مشتریان بر اساس شهر

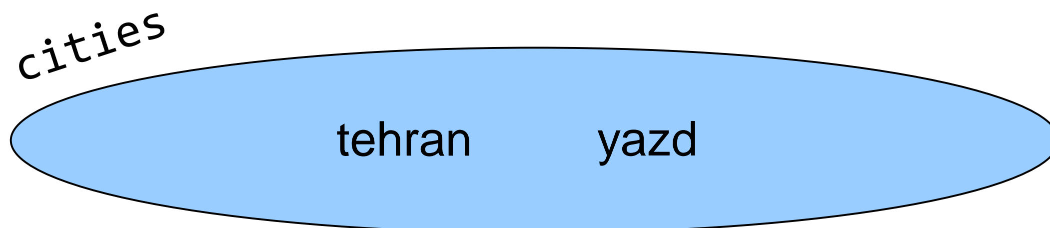
```
var query =
        from c in customers
        group c by c.City;

foreach (var CityGroup in query)
{
        Console.WriteLine(CityGroup.Key);
        foreach (var customerInGroup in CityGroup)
        {
                Console.WriteLine("\t" + customerInGroup.Name);
        }
}
```



دسته‌بندی رکوردها با GroupBy

- انتخاب شهرها

```
var query =
        from c in customers
        group c by c.City into cities
        select new { city = cities.Key };

foreach (var item in query)
{
        Console.WriteLine(item.city);
}
```



Agreagate Operators – عملگرهای تجمعی

- انجام محاسبات روی یک مجموعه از نمونه‌ها
  - شمارش نمونه‌ها

```
int query = customers.Count();
```

  - جمع

```
int[] integers = { 5, 3, 8, 9, 1, 7 };
int sum = integers.Sum();
Console.WriteLine("Sum of Numbers : {0}",
sum.ToString());
```

  - کوچکترین و بزرگترین مقدار

```
int max = integers.Max();
int min = integers.Min();
```

استفاده از عملگرهای تجمعی

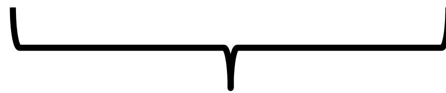
- تعداد مشتریان در هر شهر

```
var query =
        from c in customers
        group c by c.City into cities
        select new { city = cities.Key, Count = cities.Count() };

foreach (var item in query)
{
        Console.WriteLine(item.Count + " Customer from " + item.city);
}
```

استفاده از عملگرهای تجمعی

- آخرین سفارش داده شده

```
var query =
        from c in customers
        join o in orders
        on c.Id equals o.CustomerId
        select new { c.Id, c.Family, o.OrderDate };

Console.WriteLine(query.Max(m => m.OrderDate));
```



Lambda Expression

Lambda Expression

- Lambda در C# 3.0 معرفی شد.
- برای معرفی متدهای anonymous به پارامتر یک تابع با ارائه‌ی سینتکسی ساده استفاده می‌شود.
- متدهای anonymous متدهایی هستند که بدنه‌ی آنها، به عنوان پارامتر متد دیگری که یک متد را به عنوان پارامتر ورودی می‌گیرند تعریف می‌شوند.
- در این حالت به جای اینکه نام متد به عنوان پارامتر به متد دیگری ارسال شود، بدنه‌ی آن با استفاده از delegate به متد ارسال می‌شود.
- به عنوان یک شبه دستور:

```
int result;
result = Add (delegate(int height, int width) {return height +
width;});
```

Lambda Expression

- عبارت Lambda برای ساده‌تر کردن نحوه‌ی فراخوانی یک متد anonymous معرفی شده است.

```
int result;
result = Add((int height, int width) =>
{return height + width;});


int result;
result = Add((height, width) => {height
+ width;});
```

مرتب سازی رکوردها

```
var query =
        from c in customers
        orderby c.Family
        select c;
```

```
var query =
        from c in customers
        orderby c.Family descending
        select c;
```

```
var query =
        from c in customers
        orderby c.Name, c.Family
        select c;
```

Entity Framework

چرا Entity Framework؟

- در طور سالیان گذشته، روش‌های متفاوتی برای دستیابی و دستکاری داده‌های ذخیره شده در یک پایگاه داده از طریق یک زبان برنامه‌نویسی سطح بالا معرفی شده است. پس از معرفی ADO.NET (که خود بعد از معرفی OBDC و OLE توسط مایکروسافت معرفی شد) توسعه‌دهندگان نرم‌افزار احساس کردند که تکنولوژی دسترسی به داده‌ها به یک ثباتی رسیده است. اما این تمام حقیقت نبود.
- در صورتی که Queryهای نوشته شده برای اجرای ADO.NET دارای خطاهایی باشد، برنامه شما به درستی کامپایل می‌شود و خطا تا زمان اجرا خود را نشان نمی‌دهد. به عبارت بهتر، یکپارچگی‌ای بین برنامه و پایگاه داده برقرار نیست و در صورتی تغییراتی بدون اطلاع برنامه‌نویسان در پایگاه داده رخ دهد، تا زمان اجرای برنامه و برخوردن به خطاهای هنگام اجرای ناشی از این تغییرات، کسی متوجه وجود خطا نمی‌شود.

یک کد ساده ADO.NET

```
string connectionString = "data source=.;initial
catalog=NewsSystem;integrated security=True;";


using (SqlConnection conn = new SqlConnection(connectionString))
{
    conn.Open();
    using (SqlCommand cmd = new SqlCommand())
    {
        cmd.Connection = conn;
        cmd.CommandText = "SELECT * FROM News";

        SqlDataReader rdr = cmd.ExecuteReader();
        while (rdr.Read())
        {
            Console.WriteLine(rdr[1]);
        }
    }
}
```

چرا EF؟

- آنچه به واقع به آن نیاز داریم، مدلی است که با پایگاه داده و برنامه کار کند و شکاف میان این دو را از بین ببرد.
- این ارتباط به شکل مفهومی توسط روش‌های زیر قابل برقراری است
  - ERM (Entity Relationship Model): ارتباط منطقی میان موجودیت‌های پایگاه داده مدل می‌کند
  - DFD (Data Flow Diagram) و UML (Unified Modeling Language): برای مدل کردن موجودیت‌ها و نمایش نحوه جریان داده‌ها بین آنها.

ORM (Object-Relational Mapping)

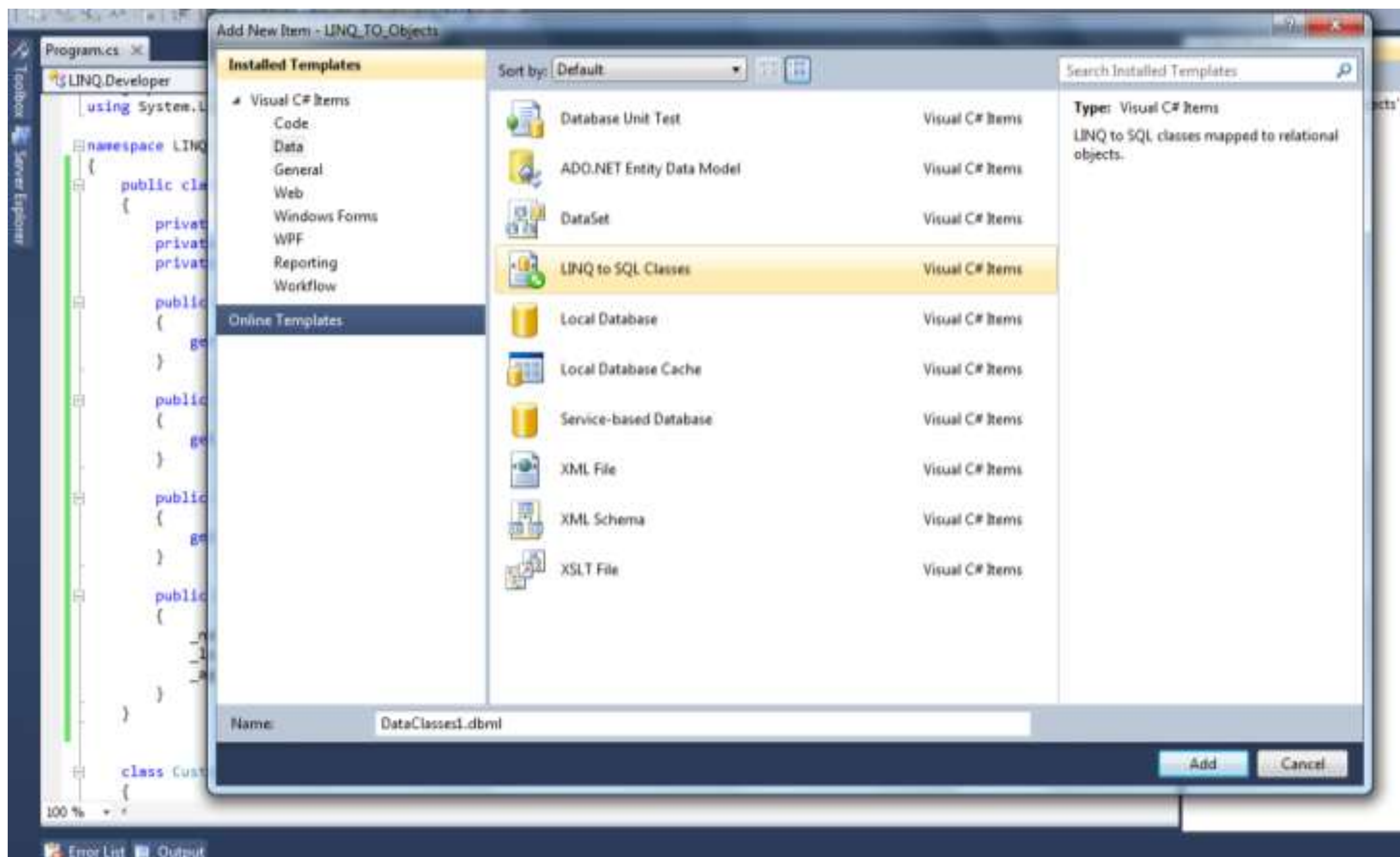
- با برقرای این ارتباط مفهومی نیز باز ممکن است ناسازگاری‌های بین کد و پایگاه داده در پیاده‌سازی بوجود بیاید.
- ORM (Object-Relational Mapping) روشی برای برقراری تناظر میان اجزای پایگاه داده‌ی رابطه‌ای با زبان‌های سطح بالای شی‌گرا است.
- ORM عملا یک لایه مترجم بین زبان برنامه‌نویسی شی‌گرا و پایگاه داده رابطه‌ای است که این دو را به هم تبدیل می‌کند و در عمل باعث می‌شود که این دو حیطه کاملا متفاوت به لحاظ مفهومی (موجودیت‌های رابطه‌ای و اشیای سلسله مراتبی) زبان یکدیگر را به خوبی بشناسند و بتوانند با هم تبادل اطلاعات داشته باشند.
- وظیفه ORM ایجاد تناظر خودکار اشیای موجود در برنامه روی جداول در پایگاه داده رابطه‌ای می‌باشد که برای این کار از متادیتاهایی برای نگاشت بین آبجکت‌ها و پایگاه داده استفاده می‌نماید.

ORM

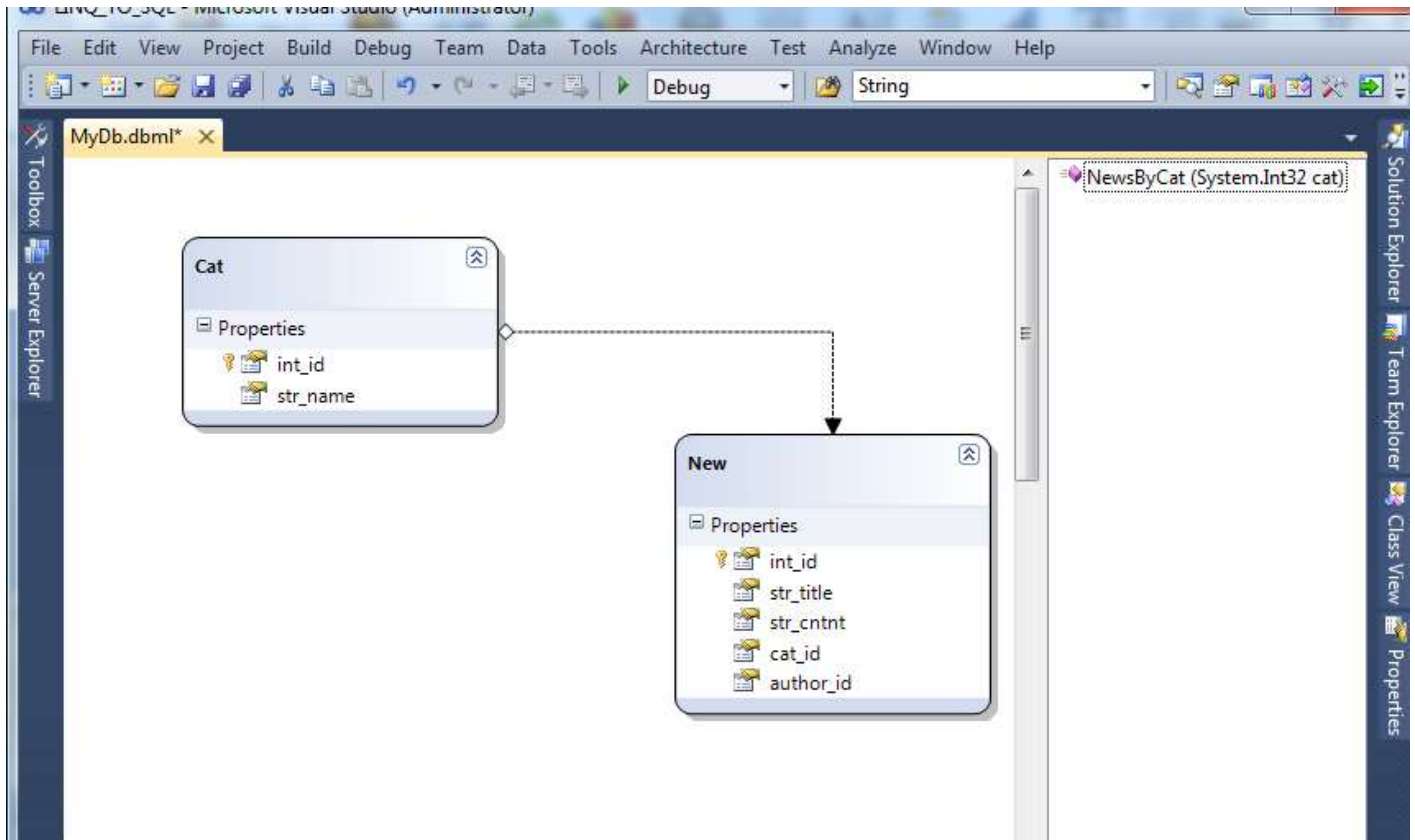
- از ORMهایی که پیش از معرفی Entitiy Framework در محیط NET. از آن بهره برده می‌شد می‌توان به موارد زیر اشاره کرد که از نظر قابلیت‌ها، امکانات و قابلیت اطمینان با Entity Framework چندان قابل مقایسه نیستند.
  - NHibernate (ایجاد شده از روی Hibernate در توسعه جاوا)
  - LINQ To SQL

LINQ To SQL

- ایجاد مدل از جدول‌های رابطه‌ای



LINQ To SQL



LINQ To SQL

- استفاده از یک شی از کلاس MyDbDataContext

```
MyDbDataContext  db = new MyDbDataContext();
```

- حال می‌توان به جدول‌ها همانند اشیا دسترسی داشت:

```
var news =
        from n in db.News
        orderby n.int_id descending
        select new { n.int_id, n.str_title, n.Cat.str_name };
```

با توجه به ارتباط دو جدول

- فراخوانی یک روال ذخیره شده

- `db.NewsByCat(1)`

Entity Framework

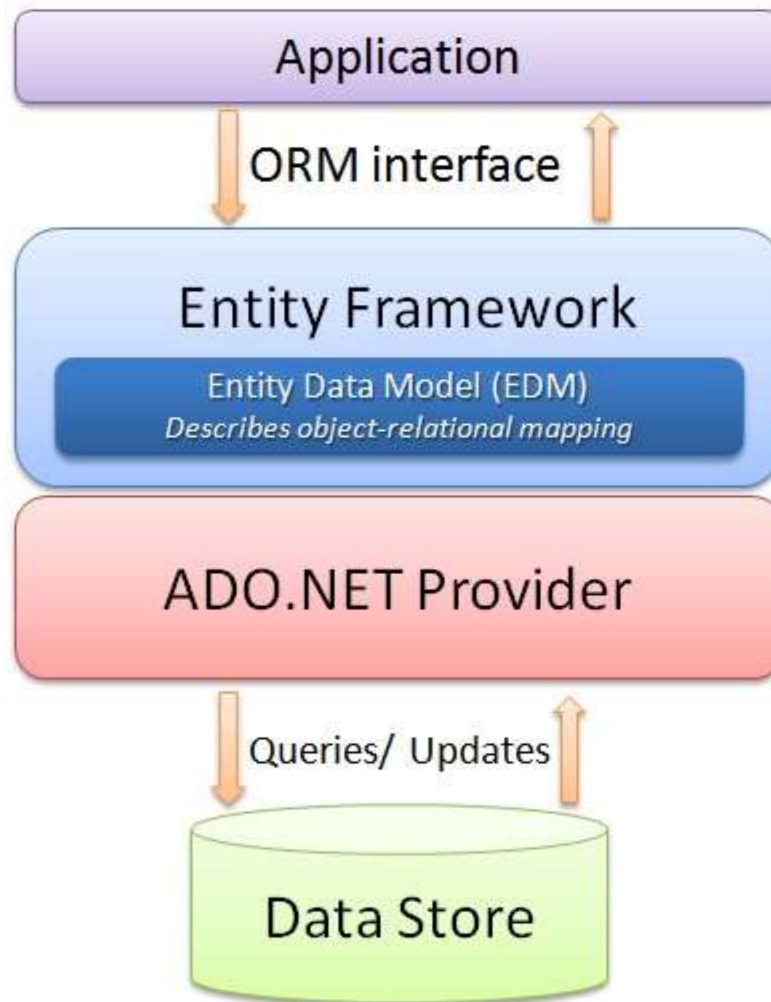
- Entity Framework برای اولین بار در سال ۲۰۰۸ و به همراه .NET Framework 3.5 منتشر شد.
- این معرفی پس از موفقیت LINQ to SQL که روشی برای مدیریت داده‌ها به صورت شی است بود.
  - در LINQ to SQL به جای برنامه‌نویسی مستقیم برای جداول پایگاه داده، برنامه‌نویسی روی مدلی که از روی ارتباط میان جداول موجود در پایگاه داده استخراج شده انجام می‌شود.
  - این مفهوم صرفا یک نگاشت یک به یک بین جداول، توابع، روال‌ها و... را فراهم می‌کند و در ابعاد یک ORM ظاهر نمی‌شود
    - مثلا در EF می‌توان یک کلاس را به چند جدول نگاشت کرد و یک ارتباط M-M را فراهم آورد).

Entity Framework

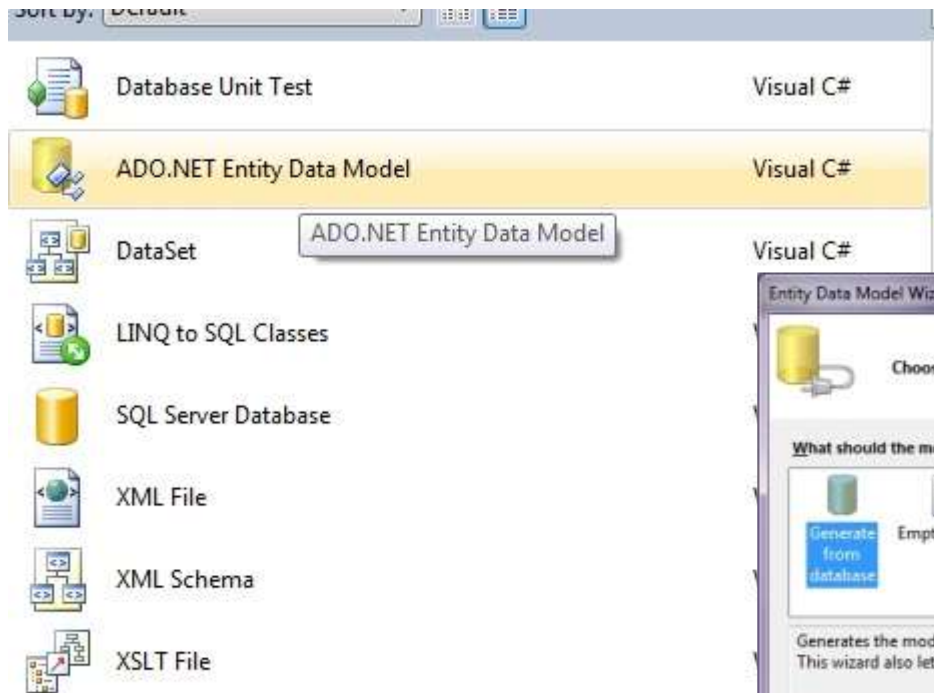
- EF برای از بین بردن شکاف میان برنامه‌نویسی شی‌گرا و پایگاه داده‌ی رابطه‌ای، از تناظر بین جدول‌ها و کلاس‌ها و فیلدها با propertyهای تعریف شده در کلاس متناظر استفاده می‌کند.
- به عبارت دیگر به ازای هر جدول یک کلاس در برنامه ایجاد می‌شود که propertyهای آن متناظر با فیلدهای پایگاه داده است.
  - در حالت کلی این تناظر لزوما یک به یک نمی‌باشد.

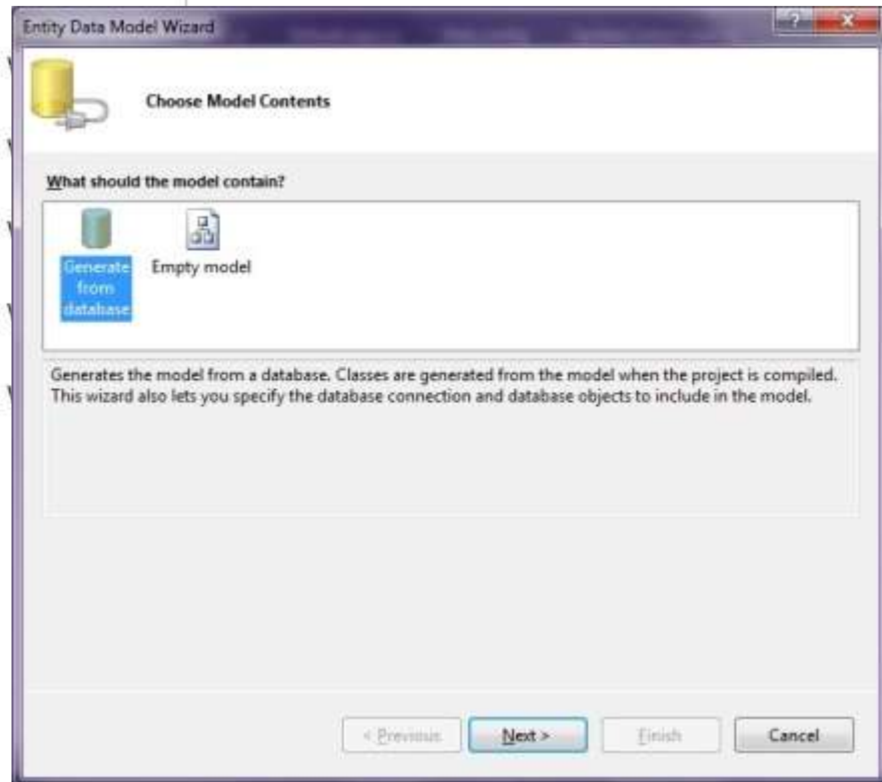
EDM (Entity Data Model)

- مدلی که EF از روی پایگاه داده ایجاد می‌کند (که لزوما نباید منطبق بر مدل پایگاه داده باشد)، ساختار پایگاه داده را در یک قالب انتزاعی ارائه می‌کند که به آن EDM (Entity Data Model) گفته می‌شود.
- در EF 4.1 به سه روش می‌توان EDM را ایجاد کرد:
  - DB First: در این روش که از نسخه اولیه EF در دسترس بود، می‌توان مدل انتزاعی را از روی پایگاه داده‌های موجود ایجاد کرد.
  - Model First: در این روش که در EF 3.5 معرفی شده است می‌توان مدل مورد نظر را درون EDM Designer ایجاد کرد، سپس پایگاه داده و کدهای کلاس مورد نیاز به صورت خودکار از روی آن تولید خواهند شد.
  - Code First: در این روش که در EF 4.1 معرفی شده، کلاس‌های معرف موجودیت‌ها توسط برنامه‌نویس نوشته می‌شود (که به آنها Plain Old CLR Objects گفته می‌شود) سپس EF از روی آنها ایجاد می‌شود. در این روش فایل‌های با پسوند EDMX که وظیفه نگهداری مدل انتزاعی را دارند، وجود ندارد.

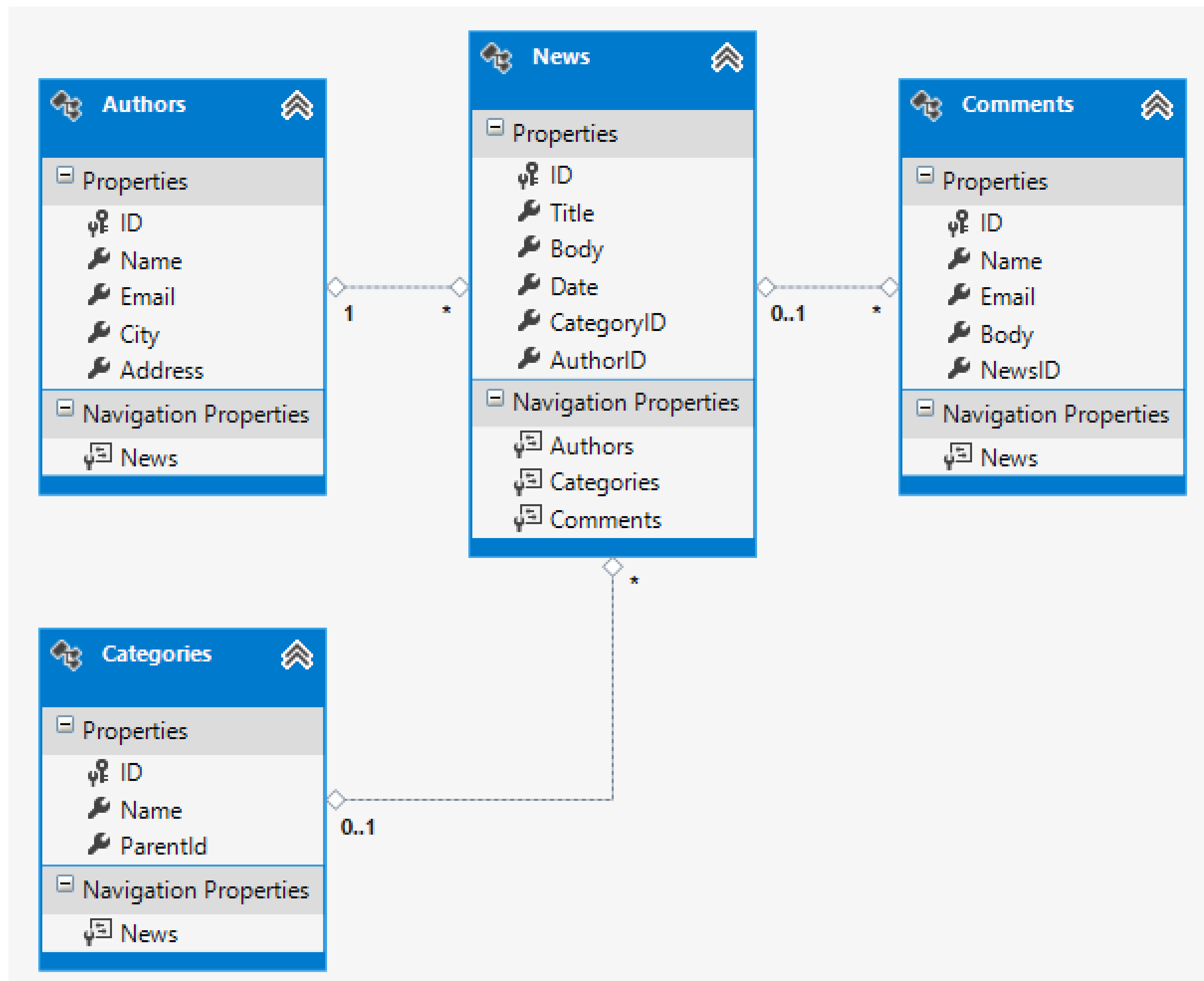

Application
ORM interface
Entity Framework
Entity Data Model (EDM)
Describes object-relational mapping
ADO.NET Provider
Queries/ Updates
Data Store

DB First – EDM ایجاد





DB First – EDM ایجاد



روش‌های انجام پرسجو روی EDM ایجاد شده

- LINQ to Entity
  - Query Syntax
  - Method Syntax
- Entity SQL

LINQ to Entities

- هدف، دسترسی به مدل ایجاد شده توسط EF و اعمال پرسجو روی آن و دریافت نتایج است:
- زمانی که یک EDM تولید می‌کنید، فایل کلاسی به برنامه اضافه می‌شود (همنام با مدل شما با پسوند Designer.cs.) که شامل Partial کلاس‌های متعدی است:
  - یک کلاس برای Context و یک کلاس برای هرکدام از موجودیت‌ها.
  - کلاس Context واسطی برای کار با موجودیت‌ها است.
    - نام آن به طور پیش‌فرض نام پایگاه داده + Entities است.
    - از کلاس ObjectContext به ارث برده شده.
    - وظیفه ارتباط با پایگاه داده و تبدیل داده‌ها به شی و مدیریت وضعیت آنها را بر عهده دارد.

LINQ to Entities

- بنابراین برای کار با موجودیت‌های EF در LINQ، ایجاد یک شی از Context مورد نظر است.

```
var context = new NewsSystemEntities();
```

- حال می‌توان به موجودیت‌های درون مدل دسترسی داشت!

```
var news = context.News;

foreach (var item in news)
      Console.WriteLine(item.Title);

foreach (var item in news)
      Console.WriteLine(item.Authors.Email);
```

LINQ to Entities

```
var context = new NewsSystemEntities();
var news =
   from n in context.News
   select n;

   foreach (var n in news)
   {
      Console.WriteLine(
         n.Title +
         ": " +
         n.Authors.Name
      );
   }
```

اعمال پرسجوهای LINQ

- Query Syntax

```
var authors =
       from a in context.Authors
       where a.Email.Contains("@")
       select a;
```

- Method Syntax

```
var authors =
       context.Authors
       .Where(a => a.Email.Contains("@"));
```

اعمال پرسجوهای LINQ

- Query Syntax

```
var autors =
       from a in context.Authors
       select new {
              AuthorName = a.Name,
              AuthorEmail = a.Email
       };
```

- Method Syntax

```
var authors =
       context.Authors
       .Select(a => new {
              AuthorName = a.Name,
              AuthorEmail = a.Email
       });
```

ایجاد Context در EF

```
public class UserContext : DbContext
{
    public DbSet<BankAccount> BankAccounts { set; get; }
    public DbSet<Transaction> Transactions { set; get; }

    public UserContext(bool mars = true) : base("MyConnectionString")
    {
        this.Database
           .Connection
           .ConnectionString += (mars ?";MultipleActiveResultSets=true;" : "");
    }
}
```

```
 <connectionStrings>
    <add name="MyConnectionString" connectionString="Data Source=.;Initial
Catalog=BK;Integrated Security = true" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

تعریف مدل

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<BankAccount>()
                .ToTable(“BankAccounts")
                .HasKey(s => s.Id);

    modelBuilder.Entity<Transaction>()
                .ToTable(“Transactions")
                .HasKey(s => s.Id);
}
```

تعریف روابط در مدل (روش Fluent API)

```
modelBuilder.Entity<Transaction>()
                .HasRequired<BankAccount>(t => t.BankAccount)
                .WithMany(b => b.Transactions)
                .HasForeignKey(t => t.BankAccountId);
```

الگوی Unit of Work

```
public interface IUnitOfWork
{
    IDbSet<TEntity> Set<TEntity>() where TEntity : class;
    int SaveAllChanges();
}
```

پیاده سازی UOW در Context

```
public new IDbSet<TEntity> Set<TEntity>() where TEntity : class
{
     return base.Set<TEntity>();
}

public int SaveAllChanges()
{
    return base.SaveChanges();
}
```

استفاده از Context با کمک UOW

```
IUnitOfWork _uow = new UserContext();
IDbSet<BankAccount> BankAccount =
_uow.Set<BankAccount>();

BankAccount.Add(......);

_uow.SaveAllChanges();
```