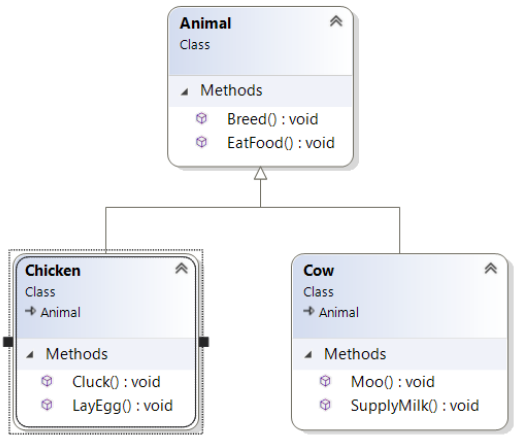
Лекция №5 Наследяване. Разширяване на типове.

Наследяването е една от най-важните концепции на ООП. То позволява да разширяваме или да правим по-специфични класове на базата на други, по-общи класове. Ако един клас `D` наследява друг клас `B` това означава, че `D` има всички членове на `B`. Класът от който се наследява се нарича **базов**, а наследникът – **производен**. В C# класовете могат да наследяват директно само от един базов клас (единично наследяване, *single inheritance*), но няма ограничение да се наследява от производен клас и така да се получи дървовидна наследствена йерархия.

Наследяването може да се използва като инструмент за избягване на дублирането на код, когато дефинираме различни класове с ясна връзка помежду си, притежаващи общ набор от възможности. Лакмусът за определяне дали да използваме наследяване е наличието на *is-a* релация между класовете.

Примери: Да разгледаме клас `Animal`, представящ домашно животно във ферма (фиг. 1). Този клас предоставя методи като `EatFood()` и `Breed()`. Кравата е животно (имаме *is-a* отношение), следователно можем да дефинираме производен клас `Cow`, който поддържа всички методи на `Animal`, но може да предоставя и свои собствени като `Moo()` и `SupplyMilk()`.



Фиг. 1. Пример за наследяване

Можем да дефинираме и втори производен клас `Chicken` с методи `Cluck()` и `LayEgg()`.

Друга типична ситуация, когато е оправдано използването на наследяване е организирането на обработката на информация за служителите в една фирма. Най-общите неща като име, ЕГН, адрес и т.н. могат да се включат в базов клас `Employee`, от който да наследяват например класовете `Manager`, `ManualWorker` и др. И ръководителите, и обикновените работници са служители (отново *is-a* релация).

Да разгледаме още един пример: Конете и китовете са бозайници и притежават общите за бозайници характеристики – дишат въздух, кърмят малките си топлокръвни са и т.н. Всеки от видовете бозайници обаче притежава и свои собствени характеристики, например конят има копита, а китът – плавници. Можем да решим, че двата вида са твърде различни и да ги имплементираме като напълно отделни класове `Horse` и `Whale`, всеки от тях представящ уникалното си поведение с методи като `Trot()` за коня и `Swim()` за кита. Какво обаче да направим с моделирането на общото за всички бозайници поведение като `Breathe()` или `SuckleYoung()`? Можем да добавим методи с тези имена към всеки от класовете, но тогава програмата става трудна за поддръжка, особено ако се наложи да моделираме и други типове бозайници като `Human` и `Aardvark` (мравояд). По-доброто решение е да се моделира общото поведение в базов клас `Mammal`,

а останалите да наследяват от него. Така Horse, Whale, Human и Aardvark автоматично ще получат функционалността на Mammal. След това можем да разширим производните класове, като добавим към тях функционалността, уникална за всеки от тях - Trot() метод за Horse и Swim() за Whale. Сега, ако се наложи да се модифицира начинът по който работи основен метод като Breathe(), ще се пипа само на едно място – в класа Mammal.

- **Деклариране на производен клас**

Декларираме, че класът DericedClass наследява класа BaseClass по следния начин:

```
class DerivedClass : BaseClass
```

Производният клас (DericedClass) наследява компонентите на базовия клас (BaseClass) и те стават част от производния. Производните класове също могат да се наследяват. Синтаксисът е същия:

```
class DerivedSubClass : DerivedClass
```

Не може да се наследява от клас, деклариран като sealed.

Да се върнем на примера с бозайниците. Можем да декларираме класовете така:

```
public class Mammal
{
    public void Breathe()
    {
        //.................
    }
    public void SuckleYoung()
    {
        //.................
    }
}
public class Horse : Mammal
{
    public void Trot()  // Тръс, бърз ход
    {
        //.................
    }
}
public sealed class Whale : Mammal
{
    public void Swim()
    {
        //.................
    }
}
```

Ако в програмата създадем обект на Horse, за него можем да извикваме методите Trot(), Breathe() и SuckleYoung().

```
Horse horse = new Horse();
horse.Trot();
horse.Breathe();
horse.SuckleYoung()
```

Аналогично, чрез обект на Whale можем да извикваме методите Swim(), Breathe() и SuckleYoung(), но не и Trot(), защото той е дефиниран само за класа Horse.

```
Whale whale = new Whale();
whale.Swim();
```

```
whale.Breathe();
whale.SuckleYoung();
```

> *Забележка: Наследяването може да се прилага само към класове, но не и към структури. Не може да се наследява структура, както и структура не може да наследява клас или друга структура… В действителност всички структури наследяват от абстрактния клас* `System.ValueType`*. Това е само детайл по имплементацията, свързан с начина по който Microsoft .NET Framework дефинира общото поведение на стек базираните value типове. Не е характерно директното използване на* `System.ValueType` *в потребителски код.*

Още за класа `System.Object`

Класът `System.Object` е базов за всички класове и стои на най-високото ниво на наследствената йерархия. Всички класове косвено наследяват от `System.Object`. Ако се върнем на примера, компилаторът мълчаливо пренаписва класа `Mammal` така:

```
public class Mammal : System.Object
{
        //.................
}
```

Всеки метод на класа `System.Object` автоматично се предава надолу по веригата към класовете, производни на `Mammal`. Това означава, че всички класове, които дефинираме, автоматично наследяват функционалност от `System.Object`. Това включва методи като `ToString()`, който се използва за преобразуване на обект до низ, обикновено за да може да се изведе на екрана.

Ето например как можем с помощта ма рефлексия да покажем всички `public` методи на класа `Horse` – както собствените, ката и наследените:

```
var horseMethods = typeof(Horse).GetMethods(
      System.Reflection.BindingFlags.Public |
      System.Reflection.BindingFlags.Instance);
foreach(var method in horseMethods)
{
    var parameters = method.GetParameters();
    Console.Write($"{method.ReturnType.Name} {method.Name}(");
    foreach (var parameter in parameters)
    {
       Console.Write($"{parameter.ParameterType.Name} {parameter.Name} ");
    }
    Console.WriteLine(")");
 }
```

- **Извикване на конструктори на базов клас**

Производният клас наследява не само методите, а всички компоненти на базовия клас, включително полетата. Тези полета обикновено изискват инициализация при създаването на обект. В повечето случаи това се прави от конструктор. Напомняме, че всеки клас има поне един конструктор. Ако програмистът не е написал такъв, компилаторът автоматично го създава. Добра практика е конструкторът на производния клас да извиква конструктор на базовия клас като част от инициализацията, засягаща наследените компоненти. Обръщението се прави с ключовата дума `base` в дефиницията на конструктора на производния клас.

Модифицираме класа `Mammal` с добавяне на поле с името на бозайника и конструктор:

```
public class Mammal
{
    private string name;
    public Mammal(string name)
    {
        this.name = name;
    }
    //..................
}
```

Яко не извикаме явно конструктора на базовия клас в конструктора на производния, компилаторът прави опит да вмъкне обръщение към подразбиращия се (*default*) конструктор на базовия клас преди да изпълни кода на конструктора на производния. В нашия пример това ще доведе до грешка при компилацията, защото компилаторът не създава *default* конструктор, ако програмистът е дефинирал конструктор с параметри. Изискват се промени и в производните класове:

```
public class Horse : Mammal
{
    public Horse(string name) : base(name)
    {}
    public void Trot()  // Тръс, бърз ход
    {
        //..................
    }
}
public class Whale : Mammal
{
    public Whale(string name) : base(name)
    {}
    public void Swim()
    {
        //..................
    }
}
```

- **Присвоявания между обекти на класове от наследствена йерархия**

Компилаторът не допуска на променлива от един тип да се присвои стойност обект от друг тип. Следният код предизвиква грешка:

```
Horse myHorse = new Horse("Bramble");
Whale myWhale = myHorse;  // Cannot implicitly convert type Horse to Whale
```

Допустимо е обаче на променлива от тип базов клас да се присвоява обект на производен клас:

```
Mammal myMammal = myHorse;
```

Това важи за променлива от всеки тип, стоящ на по-високо ниво в наследствената йерархия. Позволяването на такова присвояване е логично, доколкото всеки кон е бозайник. Просто разглеждаме коня като специален тип бозайник, който притежава всичко, което имат бозайниците плюс някои допълнителни неща, дефинирани в класа `Horse`. Променлива от тип `Mammal` може да сочи и обект на `Whale`. Производният клас разширява базовия. Има обаче едно съществено ограничение – когато обект на производен клас чрез променлива от тип базов клас имаме достъп само до компонентите, дефинирани в базовия клас. В нашия пример методите, дефинирани в `Horse` или `Whale` не са видими чрез променливата `myMammal`:

```
myMammal.Breathe(); // OK
myMammal.Trot();    // Грешка! Trot не е част от класа Mammal
```

Тъй като всички класове директно или индиректно наследяват от System.Object, на променлива от този тип може да се присвои почти всичко.

```
object obj = myHorse;
```

Но, както вече казахме, достъпни ще са само компонентите на object.

Да разгледаме обратната ситуация – на променлива от тип производен клас се присвоява обект на базов клас.

```
Mammal myMammal = new Mammal("Napoleon");
Horse myHorse = myMammal;   // Грешка!
```

Такова присвояване, без явно преобразуване на типа, не се позволява и ограничението изглежда логично – все пак не всички бозайници са коне, има и китове 😊. Присвояването е допустимо, ако се направи проверка, че обектът отдясно наистина е от типа на променливата отляво чрез операторите as и is или чрез явно преобразуване на типа.

```
Mammal myMammal = new Mammal("Napoleon");
Horse myHorse = myMammal as Horse;
```

Този код не предизвиква грешка, но тъй като обектът myMammal не е Horse, myHorseAgain се инициализира с null.

```
Horse myHorse = new Horse("Napoleon");
Mammal myMammal = myHorse;  // myMammal refers to a Horse
Horse myHorseAgain = myMammal as Horse;     // OK - myMammal was a Horse
Whale myWhale = new Whale("Abalone");
myMammal = myWhale;         // myMammal refers to a Whale
myHorseAgain = myMammal as Horse;   // returns null - myMammal was a Whale
```

- **Едноименни методи в производни и базови класове**

Не е лесно да се измислят уникални и смислени имена за идентификаторите. Ако дефинираме метод в клас, който е част от наследствена йерархия, рано или късно ще поискаме да използваме повторно това име в клас по-долу в йерархията. Това е допустимо, но ако в производен и базов клас има два метода с една и съща сигнатура, компилаторът издава предупреждение. Методът в производния клас „скрива“ метода на базовия клас, който има същата сигнатура (име и параметри).

Добавяме към класовете Mammal и Horse методи Move():

```
public class Mammal
{
    //.................
    public void Move()
    {
        Console.WriteLine("Mammal moves");
    }
}

public class Horse : Mammal
{
    //.................
    public void Move()
    {
        Console.WriteLine("Horse moves");
    }
```

```
}
```

Компилаторът предупреждава

```
Warning CS0108 'Horse.Move()' hides inherited member 'Mammal.Move()'. Use the
new keyword if hiding was intended.
```

Макар че кодът ще се компилира и работи успешно, тези предупреждения трябва да се имат предвид, защото ако класът `Horse` също има производен клас и чрез обект на този клас се извика методът `Move()`, това ще е методът на `Horse` (директният базов клас), а не на `Mammal`.

Ако искаме да потиснем предупреждението, използваме ключовата дума `new` като спецификатор в декларацията на метода в производния клас (`Horse`):

```
new public void Move()
{
    Console.WriteLine("Horse moves");
}
```

Използването на `new` в такива ситуации не променя факта, че двата едноименни метода са напълно независими и че методът на производния скрива този на базовия клас. Това само изключва предупреждението.

Възниква въпросът как производният клас може да извиква едноименните методи на своя базов клас. Достъпът до тях е възможен с ключовата дума `base`. Например в класа `Horse`:

```
new public void Move()
{
    base.Move();
    Console.WriteLine("Horse moves");
}
```

- **Виртуални методи**

Деклариране на виртуални методи

Понякога искаме да скрием имплементацията на метод в базовия клас. Типичен пример е методът `ToString()` на `System.Object`. Неговото предназначение е да преобразува обекта до представянето му като символен низ. Тъй като подобен метод е много полезен, той е включен като член на класа `System.Object` и така осигурява метод `ToString()`на всички класове. Как обаче версията на `ToString()`, имплементирана в `System.Object` знае как да преобразува инстанция на производен клас в стринг? Производният клас може да съдържа различен брой полета, които трябва да са част от низа… Затова имплементацията в `System.Object` е доста опростена – тя само преобразува обекта до низ, съдържащ името на типа (в нашия пример `Mammal` или `Horse`). Това не е много полезно и възниква въпросът защо тогава се предоставя безполезен метод? Отговорът на този въпрос изисква малко по-детайлно обяснение.

На практика не се очаква директното извикване на метода `ToString()`, дефиниран в `System.Object`. Той служи само като резервирано име. Очаква се всеки клас да предостави собствена имплементация на този метод, препокривайки (*override*) подразбиращата се имплементация в `System.Object`. Тази подразбираща се версия е само за случаи, в които класът не се нуждае от собствена версия или още за него не е имплементирана собствена версия на метода `ToString()`.

Метод, който е предназначен да бъде препокриван се нарича **виртуален** (*virtual method*). Трябва да се изясни разликата между препокриване (*overriding*) и скриване (*hiding*) на метод. Препокриването е механизъм за предоставяне на различни имплементации на един и същ метод.

Методите имат връзка помежду си, защото са предназначени за една и съща задача, но по специфичен за всеки клас начин. Скриването на метод означава замяната на един метод с друг. Методите обикновено нямат семантична връзка помежду си и може да изпълняват напълно различни задачи. Препокриването е полезна концепция, докато скриването най-често е грешка.

Метод се обявява за виртуален с ключовата дума `virtual`. Например методът `ToString()` е деклариран така:

```
public class Object
 {
     //.........................
     public virtual string ToString();
     //.........................
 }
```

За разлика от Java в C#, както и в C++, методите не са виртуални по подразбиране.

Освен методи за виртуални могат да се обявяват свойства, индексатори и събития.

Деклариране на override методи

Ако в базовия клас даден метод е деклариран като виртуален, в производния клас може да се декларира нова имплементация на този метод с ключовата дума `override`. Ще направим това с класовете `Mammal` и `Horse`.

```
public class Mammal
{
    //.................
    public override string ToString()
    {
        return name;
    }
}

public class Horse : Mammal
{
    //.................
    public override string ToString()
    {
        return $"Horse {base.ToString()}";
    }
}
```

Имплементацията в производния клас може да извика наследената имплементация от базовия клас с ключовата дума `base` (виж класа `Horse`).

Методите могат да се дефинират по-кратко като *expression-bodied* така:

```
public override string ToString() => name;

public override string ToString() => $"Horse {base.ToString()}";
```

Сега можем да използваме тези методи за извеждане на екрана с `Console.WriteLine()`.

```
Horse horse = new Horse("Napoleon");
Console.WriteLine(horse);   // Horse Napoleon
```

Има няколко важни правила, които трябва да се спазват при работа с виртуални методи:

- виртуални методи не могат да бъдат `private`. Също така `override` методи не могат да бъдат `private`, защото класът не може да променя нивото на защита на метод, който наследява;
- сигнатурите на `virtual` и `override` методите трябва да съвпада – трябва да имат еднакво име, брой и тип на параметрите си. В допълнение и двата метода трябва да имат един и същ тип на резултата;
- могат да се препокриват само виртуални методи. Ако методът в базовия клас не е виртуален и направим опит да го препокрием (`override`), ще възникне грешка по време на компилацията;
- ако производният клас не декларира метода като `override` и въпреки това го предефинира, новият метод скрива наследения, а не го препокрива. С други думи той става нов, съвсем различен метод, който просто носи същото име. Отново ще получим само предупреждение, което можем да изключим с `new`;
- `override` методът по подразбиране също е виртуален и на свой ред може да бъде препокрит в други производни класове надолу по йерархията. Не може обаче да се използва ключовата дума `virtual`, за да се обяви явно методът за виртуален.

Виртуални методи и полиморфизъм

С помощта на виртуалните методи е възможно да се извикват различни версии на един и същ метод в зависимост от типа на обекта, определен динамично по време на работа на програмата (динамично свързване).

Ще демонстрираме това като добавим нов виртуален метод `GetTypeName()` към наследствената йерархия на бозайниците.

```
public class Mammal
{
    //.................
    public virtual string GetTypeName() => "This is a mammal";
}
public class Horse : Mammal
{
    //.................
    public override string GetTypeName() => "This is a horse";
}
public class Whale : Mammal
{
    //.................
    public override string GetTypeName() => "This is a whale";
}
public class Aardvark : Mammal      // Мравояд
{
    public Aardvark(string name) : base(name)
    {}
}
```

Забележете, че методът `GetTypeName()` е препокрит в производните класове `Horse` и `Whale`, а класът `Aardvark` не притежава такъв метод.

В метода `Main()` добавяме следния код:

```
Mammal myMammal;
Horse myHorse = new Horse("Napoleon");
Whale myWhale = new Whale("Abalone");
```

```
Aardvark myAardvark = new Aardvark("Ivan");
myMammal = myHorse;
Console.WriteLine(myMammal.GetTypeName());  // This is a horse
myMammal = myWhale;
Console.WriteLine(myMammal.GetTypeName());  // This is a whale
myMammal = myAardvark;
Console.WriteLine(myMammal.GetTypeName());  // This is a mammal
```

Променливата `myMammal` е от тип `Mammal`. На нея могат да се присвояват референции на обекти на всички производни класове на `Mammal`. Тъй като виртуалният метод `GetTypeName()` е виртуален и препокрит в `Horse` и `Whale`, извикването на `myMammal.GetTypeName()`, когато `myMammal` сочи обекти на `Horse` и `Whale`, води до изпълнение на метода, дефиниран в съответния клас на обекта, за който е извикан методът. Класът `Aardvark` няма собствен метод `GetTypeName()`, затова когато `myMammal` сочи обект на `Aardvark`, се изпълнява наследения от `Mammal` метод.

Виждаме, че с една и съща синтактична конструкция (`myMammal.GetTypeName()`) се извикват различни методи в зависимост от обекта, който сочи референцията в момента на извикването. Този феномен се нарича **полиморфизъм** (*polymorphism*). Той се реализира с помощта на виртуални методи и динамично свързване.

Нека сега заменим модификаторите `override` с `new` в декларациите на класовете `Horse` и `Whale`. Сега и в трите случая на екрана ще се изведе текста „`This is a mammal`“, защото методите `GetTypeName()` в производните класове не са виртуални и за тях не се прилага динамично свързване. В този случай кой метод ще се изпълни зависи от типа на референцията, а не от типа на обекта, който тя сочи. А типа на `myMammal` е `Mammal`. Независимо от това, че `Horse` и `Whale` имат свои версии на метода `GetTypeName()` и в трите случая методът, който ще се извика е `Mammal.GetTypeName()`.

- **protected достъп до компонентите**

Знаем че модификаторът `public` прави компонентите достъпни за всички, а `private` – достъпни само за класа. Тези нива на достъп са достатъчни, ако разработваме изолирани класове. Опитните програмисти знаят, че с изолирани класове не могат да се решават сложни проблеми. Наследяването е мощен механизъм за свързване на класовете и в този случай има ясно изразена връзка между производния клас и неговия базов клас. Често се оказва полезно базовият клас да позволи на производния клас достъп до някои от неговите членове, като същевременно ги запази скрити за класовете извън наследствената йерархия. За такива ситуации е предвиден модификаторът `protected`. Производните класове имат достъп до `protected` компонентите на своя базов клас. Това важи за всички производни класове надолу по йерархията.

Препоръчва се полетата да се запазят `private`, когато е възможно и да се нарушава тази рестрикция само когато е абсолютно необходимо. `public` полетата нарушават капсулирането на класа. `protected` полетата поддържат капсулирането по отношение на потребителите на класа, но все пак позволяват капсулирането да бъде нарушено от други класове, наследяващи (директно или индиректно) от същия базов клас.

Пример: Базов клас `Vehicle` и два производни – `Airplane` и `Car`. В базовия клас е деклариран виртуален метод `Drive()`, който производните класове препокриват, съгласно особеностите си.

Файл Vehicle.cs

```
namespace Vehicles
{
    public class Vehicle
```

```
    {
        public void StartEngine(string noiseToMakeWhenStarting)
        {
            Console.WriteLine("Starting engine: {0}", noiseToMakeWhenStarting);
        }
        public void StopEngine(string noiseToMakeWhenStopping)
        {
            Console.WriteLine("Stopping engine: {0}", noiseToMakeWhenStopping);
        }
        public virtual void Drive()
        {
            Console.WriteLine("Default implementation of the Drive method");
        }
    }
}
```

Файл Airplane.cs

```
namespace Vehicles
{
    public class Airplane : Vehicle
    {
        public void TakeOff()
        {
            Console.WriteLine("Taking off");
        }
        public void Land()
        {
            Console.WriteLine("Landing");
        }
        public override void Drive()
        {
            Console.WriteLine("Flying");
        }
    }
}
```

Файл Car.cs

```
namespace Vehicles
{
    public class Car : Vehicle
    {
        public void Accelerate()
        {
            Console.WriteLine("Accelerating");
        }
        public void Brake()
        {
            Console.WriteLine("Braking");
        }
        public override void Drive()
        {
            Console.WriteLine("Motoring");
        }
    }
}
```

Демонстрация на използването на класовете (във файла Program.cs):

```
            Console.WriteLine("Journey by airplane:");
            Airplane myPlane = new Airplane();
```

```
myPlane.StartEngine("Contact");
myPlane.TakeOff();
myPlane.Drive();
myPlane.Land();
myPlane.StopEngine("Whirr");

Console.WriteLine("\nJourney by car:");
Car myCar = new Car();
myCar.StartEngine("Brm brm");
myCar.Accelerate();
myCar.Drive();
myCar.Brake();
myCar.StopEngine("Phut phut");

Console.WriteLine("\nTesting polymorphism");
Vehicle v = myCar;
v.Drive();
v = myPlane;
v.Drive();
```

Ще обърнем внимание на тестването на полиморфизма, реализиран чрез виртуалния метод `Drive()`. Създаваме референция на обект на `Car` чрез променлива от тип `Vehicle`. Това е безопасно, защото `Vehicle` е базов на `Car`. Извикването на `v.Drive()` в този случай ще доведе до изпълнението на `Car.Drive()`, тъй като за виртуалните методи се използва динамично свързване и от значение е типа на обекта, който сочи референцията в момента на извикването на метода. След това на същата променлива `v` присвояваме обект на `Airplane`. Със същото обръщение `v.Drive()` вече ще се изпълни методът `Airplane.Drive()`.

➢ Разширяващи методи (*extension methods*)

Наследяването е мощен механизъм, позволяващ разширяването на функционалността на клас чрез създаването на нов негов производен клас. Има ситуации обаче, когато наследяването не е най-подходящият механизъм за добавяне на ново поведение, особено ако се нуждаем от бързо разширяване на типа, без това да засяга съществуващия код.

Нека например искаме да добавим нова възможност към типа `int` под формата на метод `Negate()`, който връща противоположното число на текущата стойност (ясно е, че по-лесно можем да направим това с операцията унарен минус, но това е само пример). Можем да помислим за дефиниране на нов тип `NegInt32`, който наследява `System.Int32` и добавя желания метод:

```
public class NegInt32 : System.Int32
 {
     public int Negate()
     {
         //.................
     }
 }
```

Идеята е, че новият тип наследява цялата функционалност от стария и добавя новия метод. Проблемите тук са два:

- трябва навсякъде да заменим името на типа `int` с новия тип `NegInt32`;
- `System.Int32` всъщност е структура и не може да се наследява.

В такива случаи по-подходящи са разширяващите методи. Разширяването става с допълнителни статични методи, които стават налични за кода навсякъде, където се референсират данни от разширения тип.

Разширяващ метод се дефинира в статичен клас, като типът, който методът разширява, се посочва като първи параметър заедно с ключовата дума this.

```
public static class Util
{
    public static int Negate(this int x)
    {
        return -x;
    }
}

static void Main(string[] args)
{
     int i = 10;
     Console.WriteLine(i.Negate());
}
```

За да използваме разширяващия метод трябва класът Util да е видим (може да се наложи да се използва using директива, задаваща пространството на имената, на което принадлежи класът Util). Самият клас не трябва да се референсира в оператора, извикващ разширяващия метод. Компилаторът автоматично открива всички разширяващи методи за даден тип от всички статични класове, които са видими. Можем да извикваме метода Negate() и така:

```
Console.WriteLine(Util.Negate(i));
```

Пример: Ще разгледаме малко по-сложен разширяващ метод на int, който преобразува числото от десетична в друга бройна система. Числото се представя като низ.

Файл Util.cs

```
namespace ExtensionMethod
{
    public static class Util
    {
        public static string ConvertToBase(this int i, int baseToConvertTo)
        {
            string result = "";
            do
            {
                int nextDigit = i % baseToConvertTo;
                i /= baseToConvertTo;
                if(nextDigit<10)
                    result = nextDigit.ToString() + result;
                else
                    result = (char)('A'+ (nextDigit-10)) + result;
            } while (i != 0);
            return result;
        }
    }
}
```

Файл Program.cs

```
using System;

namespace ExtensionMethod
{
```

```
    class Program
    {
        static void doWork(int x)
        {
            for (int i = 2; i <= 16; i++)
            {
                Console.WriteLine($"{x} in base {i} is {x.ConvertToBase(i)}");
            }
        }
        static void Main(string[] args)
        {
            Console.Write("Enter the number to convert:");
            int x = Convert.ToInt32(Console.ReadLine());
            try
            {
                doWork(x);
            }
            catch(Exception e)
            {
                Console.WriteLine(e.Message);
            }
            Console.ReadKey();
        }
    }
}
```