Наследование (inheritance)

Наследование (**inheritance**) является одним из ключевых моментов **ООП**. Благодаря наследованию один класс может унаследовать функциональность другого класса.

Пусть у нас есть следующий класс **Person**, который описывает отдельного человека:

```
class Person
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
    public void Display()
    {
        Console.WriteLine(Name);
    }
}
```

Но вдруг нам потребовался класс, описывающий сотрудника предприятия - класс **Employee**. Поскольку этот класс будет реализовывать тот же функционал, что и класс **Person**, так как сотрудник - это также и человек, то было бы рационально сделать класс **Employee** производным (или наследником, или подклассом) от класса **Person**, который, в свою очередь, называется базовым классом или родителем (или суперклассом):

```
class Employee : Person
{

}
```

После двоеточия мы указываем базовый класс для данного класса. Для класса **Employee** базовым является **Person**, и поэтому класс **Employee** наследует все те же **свойства**, **методы**, **поля**, которые есть в классе **Person**.

>>> Единственное, что не передается при наследовании, это конструкторы базового класса.

Наследование (inheritance)

Таким образом, наследование реализует отношение **is-a** (**является**), объект класса **Employee** также является объектом класса Person:

- Отношение "**is-a**" — это отношение "обобщение-детализация", отношение большей или меньшей абстракции, и ему соответствует наследование классов.
- Отношение "**has-a**" — это отношение "целое-часть", ему соответствует вложение.

```
static void Main(string[] args)
{
    Person p = new Person { Name = "Tom" };
    p.Display();
    p = new Employee { Name = "Sam" };
    p.Display();
    Console.Read();
}
```

И поскольку объект Employee является также и объектом Person, то мы можем так определить переменную: `Person p = new Employee()`.

По умолчанию все классы наследуются от базового класса **Object**, даже если мы явным образом не устанавливаем наследование. Поэтому выше определенные классы **Person** и **Employee** кроме своих собственных методов, также будут иметь и методы класса **Object**: **ToString**(), **Equals**(), **GetHashCode**() и **GetType**().

Наследование (inheritance)

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

- Не поддерживается множественное наследование, класс может наследоваться только от одного класса.
- При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**.

В C# применяются следующие модификаторы доступа:

- **public**: публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.
- **private**: закрытый класс или член класса. Представляет полную противоположность модификатору public. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.
- **protected**: такой член класса доступен из любого места в текущем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
- **internal**: класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором public).
- **protected internal**: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.
- **private protected**: такой член класса доступен из любого места в текущем классе или в производных классах, которые определены в той же сборке.

    - Если для **полей** и **методов** не определен модификатор доступа, то по умолчанию для них применяется модификатор **private**.
    - **Классы** и **структуры**, объявленные без модификатора, по умолчанию имеют доступ **internal**.
    - Все **классы** и **структуры**, определенные напрямую в пространствах имен и не являющиеся вложенными в другие классы, могут иметь только модификаторы public или internal.

Однако следует также учитывать, что если базовый и производный класс находятся в разных сборках (проектах), то в этом случае производый класс может наследовать только от класса, который имеет модификатор public.

Наследование (inheritance)

- Если класс объявлен с модификатором **sealed**, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

```
sealed class Admin
{
}
```

- Нельзя унаследовать класс от статического класса.

Доступ к членам базового класса из класса-наследника

Вернемся к нашим классам Person и Employee. Хотя Employee наследует весь функционал от класса Person, посмотрим, что будет в следующем случае:

```
class Employee : Person
{
    public void Display()
    {
        Console.WriteLine(_name);
    }
}
```

Этот код не сработает и выдаст **ошибку**, так как переменная `_name` объявлена с модификатором `private` и поэтому к ней доступ имеет только класс `Person`. Но зато в классе **Person** определено общедоступное свойство **Name**, которое мы можем использовать, поэтому следующий код у нас будет работать нормально:

```
class Employee : Person
{
    public void Display()
    {
        Console.WriteLine(Name);
    }
}
```

Наследование (inheritance)

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами **private protected** (если базовый и производный класс находятся в одной сборке), **public**, **internal** (если базовый и производный класс находятся в одной сборке), **protected** и **protected internal**.

Ключевое слово base

Теперь добавим в наши классы конструкторы:

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

    public Person(string name)
    {
        Name = name;
    }

    public void Display()
    {
        Console.WriteLine(Name);
    }
}

class Employee : Person
{
    public string Company { get; set; }

    public Employee(string name, string company)
        : base(name)
    {
        Company = company;
    }
}
```

Наследование (inheritance)

Класс **Person** имеет конструктор, который устанавливает свойство **Name**. Поскольку класс **Employee** наследует и устанавливает то же свойство **Name**, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса **Person**. К тому же свойств, которые надо установить в конструкторе базового класса, и параметров может быть гораздо больше.

С помощью ключевого слова **base** мы можем обратиться к базовому классу. В нашем случае в конструкторе класса **Employee** нам надо установить имя и компанию. Но имя мы передаем на установку в конструктор базового класса, то есть в конструктор класса **Person**, с помощью выражения **base(name)**.

```
static void Main(string[] args)
{
    Person p = new Person("Bill");
    p.Display();
    Employee emp = new Employee("Tom", "Microsoft");
    emp.Display();
    Console.Read();
}
```

Наследование (inheritance)

Конструкторы в производных классах

Конструкторы не передаются производному классу при наследовании. И если в базовом классе **не определен** конструктор по умолчанию без параметров, а только конструкторы с параметрами (как в случае с базовым классом **Person**), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово **base**. Например, из класса **Employee** уберем определение конструктора:

```
class Employee : Person
{
    public string Company { get; set; }
}
```

В данном случае мы получим ошибку, так как класс **Employee** не соответствует классу **Person**, а именно не вызывает конструктор базового класса. Даже если бы мы добавили какой-нибудь конструктор, который бы устанавливал все те же свойства, то мы все равно бы получили ошибку:

```
public Employee(string name, string company)
{
    Name = name;
    Company = company;
}
```

То есть в классе **Employee** через ключевое слово **base** надо явным образом вызвать конструктор класса **Person**:

```
public Employee(string name, string company)
        : base(name)
{
    Company = company;
}
```

Либо в качестве альтернативы мы могли бы определить в базовом классе конструктор без параметров:

```
class Person
{
    // остальной код класса
    // конструктор по умолчанию
    public Person()
    {
        FirstName = "Tom";
        Console.WriteLine("Вызов конструктора без параметров");
    }
}
```

Наследование (inheritance)

Тогда в любом конструкторе производного класса, где нет обращения конструктору базового класса, все равно неявно вызывался бы этот конструктор по умолчанию. Например, следующий конструктор

```
public Employee(string company)
{
    Company = company;
}
```

Фактически был бы эквивалентен следующему конструктору:

```
public Employee(string company)
    : base()
{
    Company = company;
}
```

Наследование (inheritance)

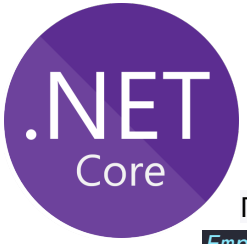
Порядок вызова конструкторов

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных. Например, возьмем следующие классы:

```
class Person
{
    string name;
    int age;

    public Person(string name)
    {
        this.name = name;
        Console.WriteLine("Person(string name)");
    }
    public Person(string name, int age) : this(name)
    {
        this.age = age;
        Console.WriteLine("Person(string name, int age)");
    }
}
class Employee : Person
{
    string company;

    public Employee(string name, int age, string company) : base(name, age)
    {
        this.company = company;
        Console.WriteLine("Employee(string name, int age, string company)");
    }
}
```



Наследование (inheritance)

При создании объекта **Employee**:

```
Employee tom = new Employee("Tom", 22, "Microsoft");
```

Мы получим следующий консольный вывод:

```
Person(string name)
Person(string name, int age)
Employee(string name, int age, string company)
```

В итоге мы получаем следующую цепь выполнений.

1. Вначале вызывается конструктор **`Employee(string name, int age, string company)`**. Он делегирует выполнение конструктору **`Person(string name, int age)`**
2. Вызывается конструктор **`Person(string name, int age)`**, который сам пока не выполняется и передает выполнение конструктору **`Person(string name)`**
3. Вызывается конструктор **`Person(string name)`**, который передает выполнение конструктору класса **System.Object**, так как это базовый по умолчанию класс для **Person**.
4. Выполняется конструктор **`System.Object.Object()`**, затем выполнение возвращается конструктору **`Person(string name)`**
5. Выполняется тело конструктора **`Person(string name)`**, затем выполнение возвращается конструктору `Person(string name, int age)`
6. Выполняется тело конструктора **`Person(string name, int age)`**, затем выполнение возвращается конструктору **`Employee(string name, int age, string company)`**
7. Выполняется тело конструктора **`Employee(string name, int age, string company)`**. В итоге создается объект **Employee**