Лекция 2

Наследование

- позволяет создавать производные классы (классы наследники), взяв **за основу** все методы и элементы **базового класса** (класса родителя).
- Объекты производного класса свободно могут **использовать всё, что создано** и отлажено в **базовом классе**.

Наследование

- При этом, мы можем в производный класс, **дописать необходимый код** для усовершенствования программы: добавить новые элементы, методы и т.д..
- Базовый класс **останется нетронутым**.

Терминология

- Класс, **от которого** произошло наследование, называется **базовым** или **родительским** (англ. base class).
- Классы, **которые** произошли от базового, называются **потомками**, **наследниками** или производными классами (англ. derived class).

Виды наследования

От простого класса

От абстрактного класса

От интерфейса

Наследование от класса

- Производный класс наследует от базового класса ВСЕ, что он имеет. Другое дело, что воспользоваться в производном классе можно не всем наследством. Например, объект класса наследника в принципе НЕ может получить доступ к private данным — членам и функциям — членам класса-родителя.

Как организовать наследование?

- Описать базовый класс

```
class ClassParent
   {
      private int x;

      protected double y;

      public int getX { get { return x; } }
   }
```

Как организовать наследование?

- Создать класс и унаследоваться от базового

```
class ClassChild : ClassParent
    {

    }
```

protected

- Доступ к члену с модификатором protected возможен **внутри класса** и из **производных экземпляров класса**.

protected

- У родительского класса есть поле **y** типа **double** с модификатором доступа **protected**

```
class ClassParent
{

    private int x;

    protected double y;

    2 references
    public int getX { get { return x; } }
```

protected

- Дочерний класс имеет доступ к этому полю

```
class ClassChild : ClassParent
{
    1 reference
    public new void nothingMethod()
    {
        base.y = 145;
    }
```

```
class ClassChild : ClassParent
{
    0 references
    public void method()
    {
        int c = getX;

        int d = base.getX;

        //int f = base.x;

        double df = y;
    }
```

protected

- Но извне это поле недоступно

```
class ClassMain
{
    0 references
    void method()
    {
        ClassChild child = new ClassChild(45);
        child.nothingMethod();

        ClassParent parent = new ClassParent();

        parent.y = 34.6;
    }
}
```

Доступ к базовомоу классу

- Доступ к членам базового класса во вложенном классе можно получить при помощи ключевого слова **base**.

```
class ClassChild : ClassParent
{
    0 references
    public void method()
    {
        int c = getX;

        int d = base.getX;

        int f = base.x;

        double df = y;
    }
```

Переопределение функций

- Изменим базовый класс, добавим метод

```
class ClassParent
{
    private int x;

    protected double y;

    2 references
    public int getX { get { return x; } }

    0 references
    public void nothingMethod()
    {
        x = 10;
        y = 34;
    }
}
```

Переопределение функций

- Переопределим его в наследнике, используя ключевое слово new

```
class ClassChild : ClassParent
{
    0 references
    public void method()
    {
        int c = getX;

        int d = base.getX;

        //int f = base.x;

        double df = y;
    }

    0 references
    public new void nothingMethod()
    {
        base.y = 145;
    }
```

Переопределение функций

- Теперь при вызове метода nothingMethod у объекта класса ClassChild будет вызываться метод не родительского класса, а дочернего

```
class ClassMain
{
    0 references
    void method()
    {
        ClassChild child = new ClassChild();

        child.nothingMethod();
    }
}
```

Виртуальные методы

- Ключевое слово **virtual** используется для изменения объявлений методов, свойств, индексаторов и событий и **разрешения их переопределения в производном классе**.

Виртуальные методы

- Модификатор **virtual** нельзя использовать с модификаторами **static**, **abstract, private** или **override**. В следующем примере показано виртуальное свойство.

Виртуальные методы

- В родительском классе создаем метод с модификатором **vitrual**

```
class ClassParent
{
    0 references
    public virtual void method()
    {
        y += x;
    }
```

Виртуальные методы

- А в дочернем классе переопределяем его, используя модификатор **override**

```
3 references
class ClassChild : ClassParent
{
    1 reference
    public override void method()
    {
        //base.method(); // это создалось по умолчанию
        y += 40;
    }
```

Почему обязательно override

```
class A
{
    2 references
    public virtual void PrintLetter()
    {
        Console.WriteLine("A");
    }
    1 reference
    public virtual void PrintNumber()
    {
        Console.WriteLine("1");
    }
}
```

```
class B : A
{
    2 references
    public override void PrintLetter()
    {
        Console.WriteLine("B");
    }

    0 references
    public void PrintNumber()
    {
        Console.WriteLine("2");
    }
}
```

```
static void Main(string[] args)
{
    A obj = new B();

    obj.PrintLetter();

    obj.PrintNumber();
    Console.ReadKey();
}
```

Конструктор и деструктор

- Особые методы, которые есть у каждого класса (создаются по умолчанию, даже если их не прописали).
- **Конструктор** вызывается **при создании** объекта от класса и служит инициализатором всех его полей
- **Деструктор** вызывается **при уничтожении** объекта для отчистки из памяти всех его полей.

Конструктор, особые приметы

- Если не прописать, создается по умолчанию только для присвоения полям значений по умолчанию.
- Если написали конструктор, конструктор по умолчанию уже не будет создан.
- Если есть несколько конструкторов, хотя бы один должен иметь доступ public.

Конструктор как метод

- Ничего не возвращает
- Называется также как и класс

```
class ClassParent
{
    0 references
    public ClassParent()
    {
        x = 10;
        y = 23;
    }
```

```
class ClassChild : ClassParent
{
    1 reference
    public ClassChild(double d)
    {
        base.y = d;
    }
```

```
class ClassMain
{
    0 references
    void method()
    {
        ClassChild child = new ClassChild();

        child.nothingMethod();
    }
}
```

Деструктор, особые приметы

- Если не прописать, создается по умолчанию для отчистки полей.
- Имеется только один деструктор в классе (конструкторов может быть много)

Деструктор как метод

- Перед деструктором ставится значок ‘~’
- Имеет имя, такое же как класс
- Не возвращает ничего

```
class ClassParent
{
    0 references
    public ~ClassParent()
    { }
```

Выбор конструктора

- **Построение** объектов **базового** класса всегда выполняется **до** любого **производного** класса. Так, **конструктор базового** класса выполняется **перед конструктором производного** класса. Если базовый класс имеет несколько конструкторов, производный класс **может выбрать** вызываемый конструктор.

Выбор конструктора

- Дополним базовый класс двумя конструкторами, один ничего не принимает, другой принимает 2 параметра

```
class ClassParent
{
    0 references
    public ClassParent()
    {
        x = 10;
        y = 23;
    }
    0 references
    public ClassParent(int f, double g)
    {
        x = f;
        y = g;
    }
```

Выбор конструктора

- Теперь укажем, что в наследнике, перед вызов его конструктора, вызывался конструктор родителя с двумя параметрами

```
class ClassChild : ClassParent
{
    1 reference
    public ClassChild(double d) : base (34, d)
    {
    }
```

Шаблон

```
<мод. доступа> class <имя кл.> : <род. кл.>
{
        <мод. доступа> <имя кл.>(<параметр.>) :
                                        base(<параметр.>)
    {
        //код
    }

}
```

Модификатор sealed*

- При применении к классу, модификатор sealed **запрещает** другим классам **наследовать** от этого класса.

- *Если вы его применяете, то у вас большие проблемы с архитектурой проекта!

Пример

- Если к нашему родителю применить этот модификатор, то его члены перестанут быть доступны наследнику

```
sealed class ClassParent
{

    private int x;

    protected double y;

    0 references
    public int getX { get { return x; } }
```

```
class ClassChild : ClassParent
{
    0 references
    public ClassChild(double d) : base (34, d)
    {
    }

    0 references
    public void method()
    {
        int c = getX;

        int d = base.getX;

        //int f = base.x;

        double df = y;
    }
```

abstract

- Ключевое слово abstract позволяет создавать **классы** и **методы** классов, которые являются неполными и **должны быть реализованы в производном классе**.

Абстрактный класс

- **Создавать** объекты от абстрактного класса **нельзя**. Назначение абстрактного класса заключается в предоставлении **общего определения для базового класса**, которое могут совместно использовать несколько производных классов.

Пример

```
abstract class ClassAbstr
{
    0 references
    public abstract void method1();

    0 references
    public abstract int method2();
}
```

```
class ClassFromAbstr : ClassAbstr
{
    1 reference
    public override void method1()
    {
        throw new NotImplementedException();
    }

    1 reference
    public override int method2()
    {
        throw new NotImplementedException();
    }
}
```

Интерфейсы

- Интерфейс является ссылочным типом, который состоит **только из абстрактных членов**.
- Когда класс реализует интерфейс, он должен предоставить реализацию для **всех членов** интерфейса.
- В классе может быть реализовано **несколько** интерфейсов

Создание интерфейса

- Интерфейс объявляется через ключевое слова **interface**.
- Все **методы** интерфейса **публичные**!

```
interface Inter1
{
    1 reference
    void method1();

    1 reference
    void method2(int x, double y);

    1 reference
    int method3();
}
```

Создание интерфейса

- Интерфейс **не может** содержать **полей**!

```
interface Inter1
{
    int value;

    1 reference
    void method1();

    1 reference
    void method2(int x, double y);

    1 reference
    int method3();
}
```

Создание интерфейса

- Но может содержать свойства

```
interface Inter1
{
    1 reference
    int value {set; get;}

    1 reference
    void method1();

    1 reference
    void method2(int x, double y);

    1 reference
    int method3();
}
```

Наследование от интерфейса

- Если мы просто унаследуемся от интерфейса, не определив ни одного метода, то получим ошибку

```
class childInter : Inter1
{
}
```

Наследование от интерфейса

- Даже определив несколько методов интерфейса, но не все, все равно будет ошибка

```
class childInter : Inter1
{
    1 reference
    public void method1()
    {
        throw new NotImplementedException();
    }

    1 reference
    public void method2(int x, double y)
    {
        throw new NotImplementedException();
    }
}
```

Наследование от интерфейса

- Только определив все методы и свойства, ошибки не будет

```
class childInter : Inter1
{
    1 reference
    public int value
    {
        get
        {
            throw new NotImplementedException();
        }
        set
        {
            throw new NotImplementedException();
        }
    }

    1 reference
    public void method1()
    {
        throw new NotImplementedException();
    }

    1 reference
    public void method2(int x, double y)
    {
        throw new NotImplementedException();
    }

    1 reference
    public int method3()
    {
        throw new NotImplementedException();
    }
}
```

Модификатор доступа

- **Все методы**, наследуемые от интерфейса, должны иметь **публичный доступ**

```
class childInter : Inter1
{
    1 reference
    public int value...

    1 reference
    public void method1()
    {
        throw new NotImplementedException();
    }

    0 references
    private void method2(int x, double y)
    {
        throw new NotImplementedException();
    }

    1 reference
    public int method3()
    {
        throw new NotImplementedException();
    }
}
```

Множественное наследование

- Если потомок может наследоваться только от одного обычного или абстрактного класса, то в случае с интерфейсами, он может наследоваться сразу же от нескольких интерфейсов.

- Но тут может возникнуть проблемка…

Множественное наследование

```
interface Inter1
{
    1 reference
    int value {set; get;}

    1 reference
    void method1();

    1 reference
    void method2(int x, double y);

    1 reference
    int method3();
}
```

```
interface Inter2
{
    0 references
    double value2 { set; get; }

    0 references
    int method1();

    0 references
    void method2();

    0 references
    int method3(int x, double y);
}
```

```
class childInter : Inter1, Inter2
{
    1 reference
    public int value...
```

Множественное наследование

- Если в интерфейсах присутствуют методы с одинаковыми названиями, как понять метод какого из интерфейсов был реализован в классе-наследнике?

Множественное наследование

- возможную неоднозначность в именах членов можно разрешить при помощи **полного квалификатора имени свойства или метода**.

```
class childInter : Inter1, Inter2
{
    2 references
    void Inter1.method1()
    {
        throw new NotImplementedException();
    }

    1 reference
    int Inter2.method1()
    {
        throw new NotImplementedException();
    }
```

Как вызвать такой метод

- Оператор **as** используется для выполнения определенных **преобразований типов** между совместимыми ссылочными типами или тип, допускающий значение NULL.
- Оператор **is** проверяет совместимость объекта с заданным типом.

is и as. Пример

- С помощью оператора **is** убеждаемся, что наш объект относится к нужному интерфейсу, а потом приводим его к этому интерфейсу для вызова нужного метода

```
class ClassMain
{
    0 references
    void methodInter()
    {
        childInter ch = new childInter();
        if(ch is Inter1)
        {
            (ch as Inter1).method1();
        }
    }
```

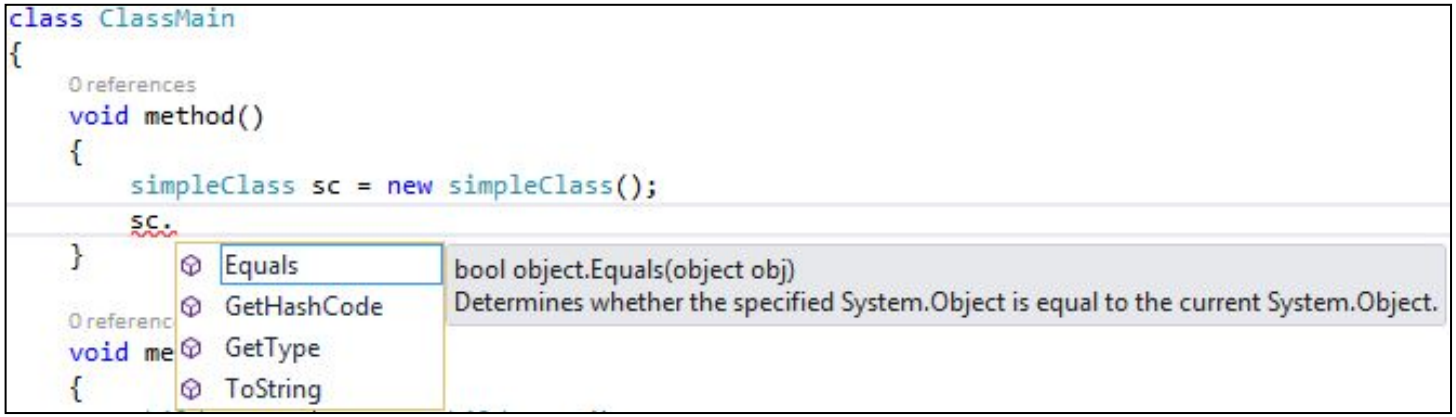
Особенности

- Оператор **as** подобен оператору приведения. Однако если преобразование невозможно, то **as** возвращает **null** вместо вызова исключения.
- Если предоставленное выражение отлично от NULL и предоставленный объект может быть приведен к предоставленному типу не вызывая исключение, выражение **is** принимает значение **true**.

Общий тип

- Создадим простой класс без содержимого
- Теперь создадим объект от этого класса и посмотрим что в нем есть

```
class simpleClass
{
}
```



Object

- В нем оказалось 4 метода, хотя мы ничего не прописывали. Откуда они?

- В унифицированной системе типов C# **все типы**, предопределенные и пользовательские, наследуют непосредственно или косвенно от **Object**.

Object

- Всеобщий базовый тип. Обязательная составляющая любого типа в .NET. Функциональные характеристики типа System.Object приводятся в таблице.

Object

Equals	Обеспечивает сравнение объектов
GetHashCode	Обеспечивает реализацию алгоритма хэширования для значений объектов
GetType	Для любого объекта создает объект типа Type, содержащий информацию о структуре типа данного объекта
ReferenceEquals	Проверка эквивалентности ссылок. Статический
ToString	Возвращает объект типа String с описанием данного объекта