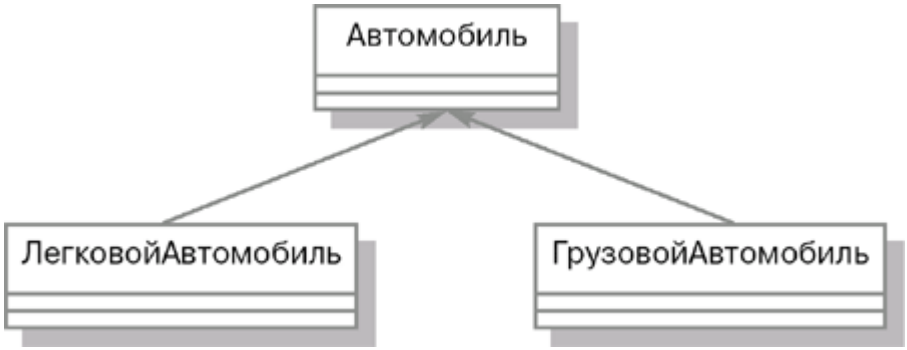
Лекция 3.
Наследование.
Виртуальные функции.
Модификаторы видимости

Наследование

- **Наследование** – процесс создания на основе старого класса нового класса, который может переопределять члены класса-родителя и добавлять новые члены
- Класс-родитель называют **базовым классом** или **суперклассом**
- Класс-наследник называют **подклассом** или **производным классом**



Пример наследования из жизни

- Пример – класс **Млекопитающие** и класс **Человек**
- Можно сказать что класс **Человек** наследуется от класса **Млекопитающие**. Человек имеет все свойства, присущие млекопитающим – например, кормление детей молоком. И при этом добавляет новые свойства – например, прямохождение, отсутствие хвоста, развитый мозг и т.д.
- Каждый человек является млекопитающим, но не каждое млекопитающее – человек. Везде, где требуется некоторое млекопитающее, можно использовать человека

Наследование

- Объекты производных классов обычно являются более узко-специализированными
- Пример из жизни: общий класс – **Число**. Конкретный класс – **Целое число**.
- Или: общий класс – **Хищник**. Производный класс – **Кошка**
- Каждый объект производного класса является и объектом базового класса и может использоваться везде, где ожидается ссылка на базовый класс

Наследование

- Объекты производных классов обычно являются более узко-специализированными

- Все объекты класса **ЛегковойАвтомобиль** являются и объектами класса **Автомобиль**
- Но не все объекта класса **Автомобиль** являются объектами класса **ЛегковойАвтомобиль** (они могут быть объектами класса **ГрузовойАвтомобиль**)

Наследование

```
public class Robot
{
    private int power;
    public void Move()
    {
        //…
    }
}
```

```
public class RobotCleaner : Robot
{
    public void Clean()
    {
        // ..
    }
}
```

Класс RobotCleaner наследуется от Robot и добавляет новый метод Clean()

```
public static void Main()
{
    RobotCleaner robot = new RobotCleaner();
    robot.Clean();
    robot.Move();
}
```

Экземпляры класса RobotCleaner наследуют (получают) члены своих родителей – метод Move и поле power

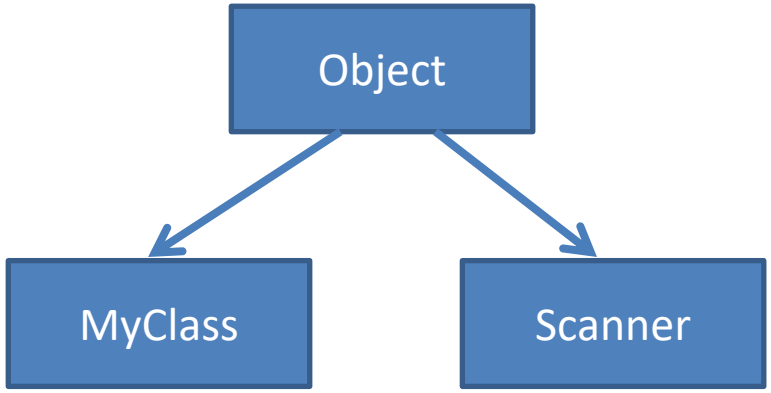
Важные моменты при наследовании

- При наследовании наследуется все, кроме конструкторов

- В классах-наследниках нет доступа к private членам родителей несмотря на то, что они наследуются
- Потому что модификатор private для членов класса означает, что «доступно внутри класса», а класс-наследник – это уже другой класс

Наследование в C#

- В C# каждый тип данных может непосредственно наследоваться только от одного класса-родителя
- Все типы данных в C# в конечном итоге наследуются от System.Object и наследуют его методы
- Если у класса при объявлении не указан родитель, то неявно родителем считается Object



Наследование от Object

```
public class A
{
    private int a;

    public void F()
    {
        //…
    }
}
```

Класс A неявно наследуется от Object
Это то же самое, что написать
A : Object

Конструкторы при наследовании

- При наследовании каждый конструктор дочернего класса **обязан** вызвать какой-либо конструктор родительского класса
- Это нужно, т.к. у класса родителя могут быть поля, которые надо заполнить
- Это делается при помощи ключевого слова base
-

```
public class A {
    private int x;

    public A(int x) {
        this.x = x;
    }
}
```

```
public class B : A {
    private int y;

    public B(int x, int y) : base(x) {
        this.y = y;
    }
}
```

Конструкторы при наследовании

- В круглых скобках передаются аргументы конструктору класса-родителя
- Если не писать base, то неявно пишется base() – вызов конструктора родителя без аргументов (а его может не быть, тогда будет ошибка компиляции)

-
```
public class A {
    private int x;

    public A(int x) /* : base() */ {
        this.x = x;
    }
}
```

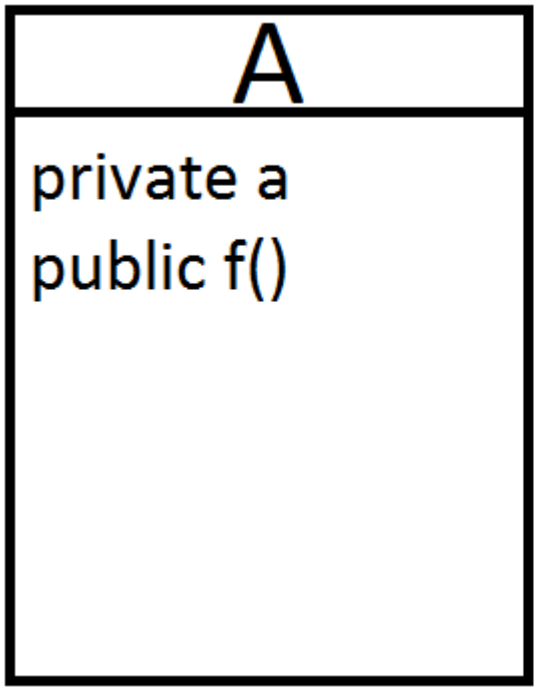
Неявно

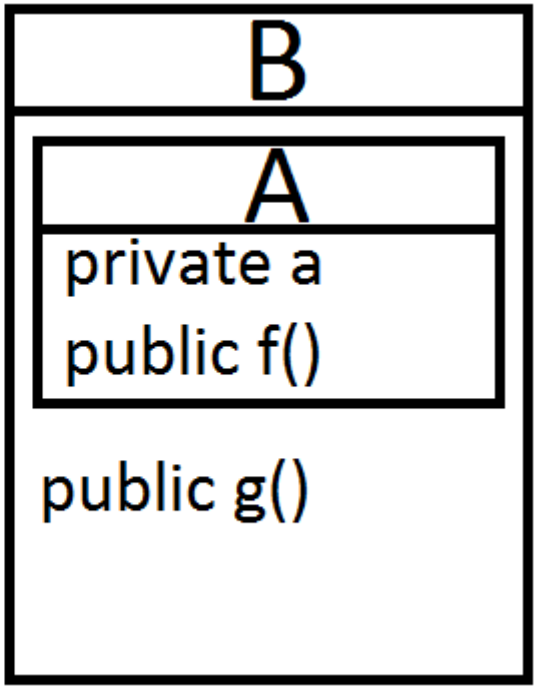
```
public class B : A {
    private int y;

    public B(int x, int y) : base(x) {
        this.y = y;
    }
}
```

Наследование

```
public class A  {
    private int a;
    public void F() {
        //…
    }
}
```

```
public class B : A {
    public void G() {
        // ..
    }
}
```





Приведение ссылочных типов

Приведение ссылочных типов

- Объекты классов-наследников можно использовать везде, где требуется класс-родитель
- Пусть класс RobotCleaner наследуется от класса Robot
- ```
  Robot a = new Robot();
  RobotCleaner b = new RobotCleaner();
  Robot c = new RobotCleaner();  // неявное приведение к
                                 // базовому типу
  ```
- Для переменной **c** – сам объект принадлежит классу RobotCleaner, но переменная – ссылка на класс Robot
  ```

Приведение ссылочных типов

- Robot c = new RobotCleaner(); // неявное приведение к
  // базовому типу
- При приведении типа от наследника к родителю сам объект никак не изменяется – он не усекается, поля не удаляются, новый объект не создается и т.д.
- Переменная **c** ссылается на тот же самый объект, но т.к. тип переменной теперь Robot, а не RobotCleaner, то компилятор разрешает обращаться только к тем членам класса, которые объявлены в Robot
- c.Clean(); // ошибка компиляции
  c.Move(); // нормально работает

Приведение ссылочных типов

- ```
  public class Utils {
      public static void UseRobot(Robot a) {
      }
  }
  ```
- В этот метод можно передавать и наследников Robot

- ```
  Robot a = new Robot();
  Utils.UseRobot(a);   // OK
  ```
- ```
  RobotCleaner b = new RobotCleaner();
  Utils.UseRobot(b);   // OK, неявное приведение к
                       // базовому типу Robot
  ```
- ```
  Robot c = new RobotCleaner();
  Utils.UseRobot(c);   // OK
  ```

Приведение ссылочных типов

- ```
  public class Utils {
      public static void UseRobot(RobotCleaner b) {
      }
  }
  ```
- ```
  Robot a = new Robot();
  Utils.UseRobot(a);  // ошибка компиляции – не каждый
                      // Robot является RobotCleaner
  ```
- ```
  RobotCleaner b = new RobotCleaner();
  Utils.UseRobot(b);  // OK
  ```
- ```
  Robot c = new RobotCleaner();
  Utils.UseRobot(c);  // ошибка компиляции – переменная
                      // типа Robot, компилятор не понимает,
                      // что это в самом деле RobotCleaner
  ```

Явное приведение типов

- Если мы точно знаем, что хоть и переменная ссылается на базовый тип, а сам объект принадлежит производному типу, то мы можем выполнить явное приведение типа

- ```
  Robot a = new Robot();
  RobotCleaner b = new RobotCleaner();
  Robot c = new RobotCleaner();
  ```

- ```
  RobotCleaner d = (RobotCleaner)c;  // OK
  RobotCleaner e = (RobotCleaner)a;  // ошибка во время
                                     // исполнения, нельзя
                                     // преобразовать объект
                                     // Robot в RobotCleaner
  ```

Проверка принадлежности классу

- Оператор проверки, что объект принадлежит классу или является потомком этого класса – is

- Пример:
- if (“abc” is string)
  {
  // выполнится – строка принадлежит типу string
  }
- if (“abc” is object)
  {
  // выполнится – все классы наслед-ся от object
  }

Проверка принадлежности классу

- Примеры:
- if (null is <ЛюбойТип>)
  {
  // не выполнится - всегда false
  }
- if (“abc” is StringBuilder)
  {
  // не выполнится – строка не наследуется от
  // StringBuilder
  }

Оператор as

- Иногда нужно выполнить приведение типа, но неизвестно, принадлежит ли объект этому типу
- Приведение типа в этом случае бросает исключение
- ```
  object o = new object();
  string s = (string)o;  // программа упадет
  ```
- Придется использовать оператор is:
- ```
  string s = null;
  if (o is string)
  {
      s = (string)o;
  }
  ```

Оператор as

- Есть более удобный вариант – оператор as
- ```
  object o = new object();
  string s = o as string;  // null
  ```
- Он не бросает исключение
- Если объект принадлежит типу (или наследнику), то выполняется приведение типа
- Иначе – результатом будет null

- ```
  object o = “123”;
  string s = o as string;  // 123
  ```

Виртуальные функции. Сокрытие методов

Сокрытие методов

- В классе потомке можно объявить в точности такую же функцию, как в родителе
- Это называется сокрытием (**hiding**), потому что если будут обращаться к этой функции, то вызовется функция потомка
- Рассмотрим, что будет происходить в этом случае

Переопределение методов

```
public class A
{
    public void F()
    {
        Console.WriteLine(1);
    }
}
```

```
public class B : A
{
    public void F()
    {
        Console.WriteLine(2);
    }
}
```

- В классах A и B есть функция с одинаковой сигнатурой
- ```
  A a = new A();
  a.F(); // 1
  ```
- ```
  B b = new B();
  b.F(); // 2
  ```
- ```
  A c = new B();
  c.F(); // 1
  ```

Для невиртуальной функции вызывается та реализация, которая соответствует типу переменной (ссылки)
  ```

new при сокрытии методов

- Кстати, в том примере в классе B будет warning при объявлении функции **F**
- Дело в том, что если мы делаем сокрытие метода, то по-хорошему, мы должны указать ключевое слово new, хоть оно и не обязательно
- ```
  public class B : A
  {
      public new void F()
      {
          Console.WriteLine(2);
      }
  }
  ```
- Тем самым мы говорим компилятору, что мы понимаем, что тут будет сокрытие метода
  ```

Переопределение методов

- Но есть и другой, более полезный вариант переопределения методов
- Он называется **переопределением метода (overriding)**
- Функции, которые можно переопределить в классах-потомках, называются **виртуальными (virtual functions)**
- **Виртуальная функция –** функция, которую можно переопределить в классах-наследниках так, что при ее вызове будет использоваться реализация, соответствующая настоящему типу объекта

Переопределение методов

- В C# чтобы функция стала виртуальной, её нужно пометить ключевым словом virtual
- Если мы переопределяем функцию в классе-наследнике, то мы должны указать ключевое слово override
- Если забыть слово override, то подразумевается new

Переопределение методов

```
public class A
{
    public virtual void F()
    {
        Console.WriteLine(1);
    }
}
```

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

- В классах A и B есть функция с одинаковой сигнатурой
- ```
  A a = new A();
  a.F(); // 1
  ```
- ```
  B b = new B();
  b.F(); // 2
  ```
- ```
  A c = new B();
  c.F(); // 2
  ```

Для виртуальной функции вызывается та реализация, которая определена для фактического типа **объекта**, а не для типа ссылки

Для полей ничего такого нет
  ```

Виртуальные функции

- **Виртуальные функции** – являются еще одним примером **полиморфизма**
- ```
  public static void WorkWithA(A a) {
      a.F();
  }
  ```
- Функция **WorkWithA** ничего не знает о том, какой именно класс у объекта **a**
  - Но она точно знает что этот класс либо A, либо наследник класса A
  - Поэтому **WorkWithA** может вызвать метод **F** класса A
- Если передать объект класса A, то вызовется реализация класса A. Если передать объект класса B, и в нем переопределен метод **F**, то вызовется метод **F** класса B
  ```

Пример – геометрические фигуры

- Пусть есть базовый класс Shape (фигура)
- public class Shape
  {
  public **virtual** double GetArea()
  {
  return 0; // нет разумной реализации, поэтому пока так
  }
  }

Пример – геометрические фигуры

- А дальше создаем классы-наследники для прямоугольника, треугольника, круга и т.д., и в них правильно реализуем этот метод
- ```
  public class Rectangle : Shape
  {
      // опущен код полей и конструктора
      public override double GetArea()
      {
          return width * height;
      }
  }
  ```
  ```

Пример – геометрические фигуры

- И тогда эти примеры будут работать правильно
- ```
  Shape s1 = new Rectangle(10, 2);
  Console.WriteLine(s1.GetArea());  // 20
  // вызывается реализация для прямоугольника
  ```
- ```
  Shape s2 = new Triangle(0, 0, 3, 0, 0, 4);
  Console.WriteLine(s2.GetArea());  // 6
  // вызывается реализация для треугольника
  ```

Пример использования полиморфизма

- ```
  public class Employee
  {
      public virtual int GetSalary() { return 20000; }
  }
  ```
- ```
  public class Director : Employee
  {
      public override int GetSalary() { return 50000; }
  }
  ```
- ```
  public class Manager : Employee
  {
      public override int GetSalary() { return 30000; }
  }
  ```
  ```

Пример использования полиморфизма

```
public static int GetTotalSalary(Employee[] employees) {
    // выдает суммарную зарплату по всем переданным
    int result = 0;
    foreach (Employee e in employees)
    {
        result += e.GetSalary();
    }
    return result;
}
```

Очень простая реализация за счет полиморфизма и виртуальной функции GetSalary

Код метода GetTotalSalary никогда не изменится и будет работать и для новых типов сотрудников

```
public static void Main()
{
    Employee[] e = { new Director(), new Employee (),
      new Manager (), new Manager (), new Employee () };
    Console.WriteLine(GetTotalSalary(e));
}
```

Пример отсутствия полиморфизма

- ```
  public enum EmployeeType
  {
      Employee = 0,
      Director = 1,
      Manager = 2
  }
  ```
  ```

Пример отсутствия полиморфизма

```
public class Employee
{
    private EmployeeType type;
    public int GetSalary()
    {
        if (type == EmployeeType.Employee)
        {
            return 20000;
        }
        if (type == EmployeeType.Director)
        {
            return 50000;
        }
        return 30000;
    }
}
```

Код не будет работать для новых типов сотрудников, нужно дописывать ветку в if

Таблица виртуальных функций

- Как реализованы виртуальные функции?
- Для каждого класса в C# есть **таблица виртуальных функций**, она одна на весь класс
- В этой таблице хранятся адреса функций (функции тоже хранятся в памяти, у них есть адрес)
- Внутри каждого объекта хранится ссылка на таблицу, которая соответствует типу объекта

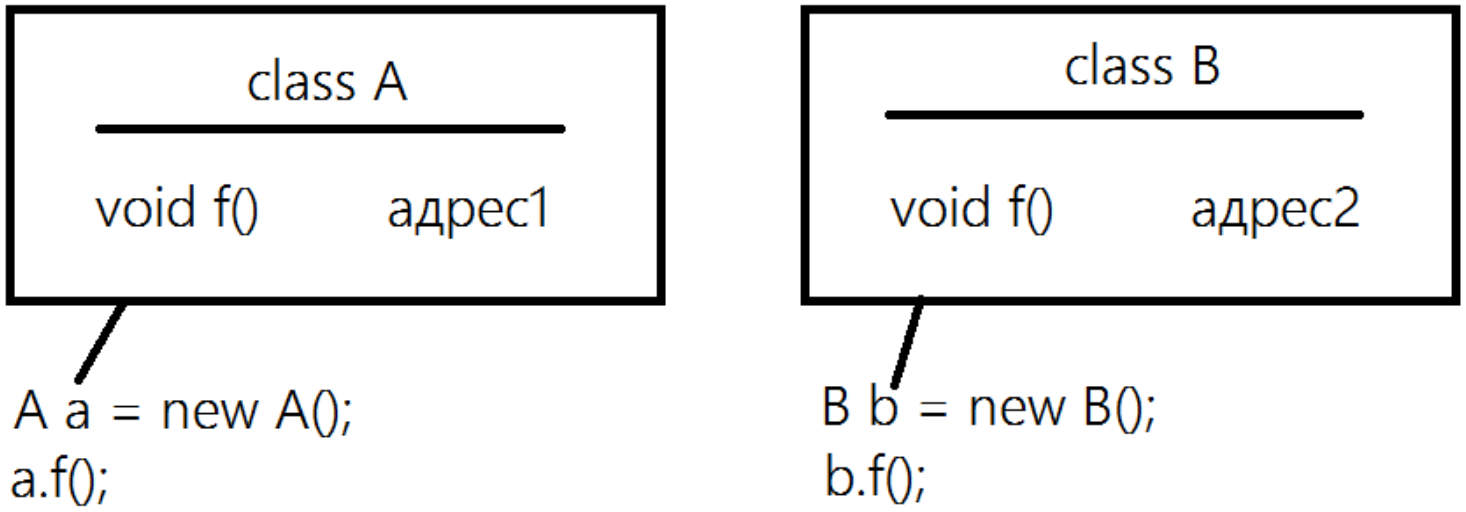
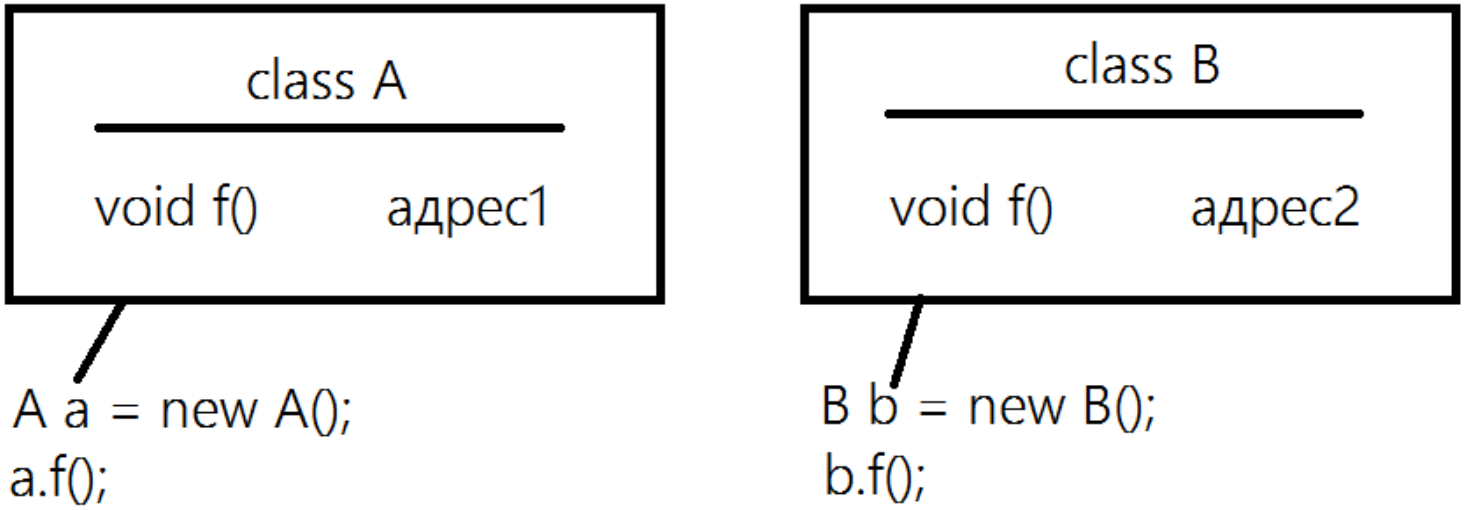


Таблица виртуальных функций

- При вызове метода **f**, CLR смотрит в таблицу виртуальных функций для текущего объекта, и вызывает функцию по адресу, который указан в таблице
- Если объект по факту принадлежит классу A, то вызовется реализация по **адресу1**, если классу B – то по **адресу2**



Модификаторы доступа

- В C# есть 6 вариантов модификаторов доступа для полей и методов классов
-
```
public class A
{
    private int x;          // доступен только внутри класса A
    public int y;           // доступен везде
    internal int z;         // доступен только в сборке
    protected int w;        // доступен классу и всем наследникам
    protected internal v;   // доступен в классе, наследниках
                            // и всем из сборки
    private protected u;    // доступен в классе и наследниках
                            // только из этой же сборки
}
```

Модификаторы доступа

Модификатор	Пояснение
private	Видимость только внутри текущего класса
private protected	Видимость внутри текущего класса и внутри наследников только из сборки, где объявлен класс
protected *	Видимость внутри текущего класса и внутри его наследников, независимо от их сборки
internal *	Видимость внутри всей сборки, где объявлен класс
protected internal	Видимость внутри всей сборки, где объявлен класс. И видимость внутри всех классов-наследниках, независимо от сборки
public	Видимость везде

* Строгость protected и internal нельзя сравнивать между собой. Но будем считать, что в сборке больше классов, чем обычно наследников у класса

Модификаторы доступа

- private члены недоступны даже классам-наследникам
- Если в классе-наследнике нужно обратиться к private членам классов-предков, то можно применить следующие варианты:
  - поменять этим членам класса модификатор на protected
  - если это поля, то поля можно оставить private, но сделать protected getter и/или setter
- protected internal и private protected на практике встречаются очень редко

Какие функции являются виртуальными?

- В C# виртуальными не могут быть:
  - Все private функции
  - Все static функции

- Остальные функции являются виртуальными, если они помечены ключевым словом virtual или override
- Эти функции можно переопределить в классах-потомках, но они не будут виртуальными

Перекрытие полей (hiding)

```
public class A
{
    public int x = 1;
}
```

```
public class B : A
{
    public int x = 2;
}
```

```
B b1 = new B();
A b2 = b1;

Console.WriteLine(b1.x);  // 2
Console.WriteLine(b2.x);  // 1
```

Создание невиртуального члена с таким же именем и сигнатурой в потомке называется перекрытием (hiding)

Поля не являются виртуальными. К какому полю пойдет обращение, зависит от типа ссылки

Слово base для методов и полей

- Из дочернего класса можно обратиться к полям и методам непосредственного родителя при помощи слова base

- ```
  public class A
  {
      public virtual void F()
      {
          Console.WriteLine(1);
      }
  }
  ```

- ```
  A b = new B();
  b.F();
  // 1
  // 2
  ```

```
public class B : A
{
    public override void F()
    {
        base.F();
        Console.WriteLine(2);
    }
}
```

Здесь слово base не обязано быть первой командой в методе

Полезно когда мы хотим «дополнить» реализацию метода

Слово base для методов и полей

- Слово base позволяет классам-наследникам обращаться только к членам непосредственного родителя
- К членам родителя родителя и т.д. обратиться таким образом нельзя

Слово base для методов и полей

- Из дочернего класса можно обратиться к полям и методам непосредственного родителя при помощи слова base

```
public class A
{
    protected int x = 1;

    public virtual void F()
    {
        Console.WriteLine(x);
    }
}
```

```
public class B : A
{
    protected int x = 2;

    public override void F()
    {
        Console.WriteLine(x + base.x);
    }
}
```

base.x указывает именно на поле в классе-родителе

```
A b = new B();
b.F(); // 3
```

Слово sealed для классов

- Можно запретить наследоваться от класса, добавив к его объявлению ключевое слово sealed

- ```
  public sealed class A
  {
      // код
  }
  ```
- ```
  public class B : A
  {
      // ошибка компиляции, нельзя наследоваться
      // от sealed класса
  }
  ```

Зачем запрещать наследование?

- Из соображений безопасности – ведь наследники могут переопределять методы как хотят и обращаться к protected членам, а хочется запретить менять реализацию класса
- Это улучшает производительность
  - Виртуальные функции замедляют работу программы (это связано с тем, как они реализованы, – требуется дополнительный переход к таблице виртуальных функций)
  - sealed методы, так как не могут быть переопределены, реализованы как не виртуальные, поэтому их вызов быстрее

Слово sealed для методов

- Можно запретить наследникам переопределять метод, указав для него модификатор sealed

```
public class A {
    public virtual string GetName() { return “1”; }
}
```

```
public class B : A {
    public sealed override string GetName() { return “2”; }
}
```

```
public class C : B {
    public override string GetName() {
        return “3”; // ошибка компиляции
                    // нельзя переопределять sealed метод
    }
}
```

Зачем sealed для методов?

- Аналогично мотивам использования sealed для классов – запретить изменение реализации или повысить производительность, только мы не хотим запрещать наследоваться от класса, а запрещаем переопределять только некоторые методы

Зачем нужно наследование?

- Помогает избавиться от дублирования кода:
  для иерархии классов можно создать базовый класс, который реализует основную логику, а от него будут наследоваться классы-наследники и переопределять лишь некоторую часть методов
- Например, если мы наследуемся от класса Form, который представляет в C# окно, то мы автоматически получаем все его методы, и чтобы создать своё окно таким как хочется, нужно лишь переопределить и добавить некоторые методы

Зачем нужно наследование?

- Но главное, ради чего стоит наследоваться – это полиморфизм
- Он позволяет создавать свои классы, которые можно использовать в уже существующем коде **библиотек** и **фреймворков**
- Например, библиотека представляет набор базовых классов, от которых можно создать наследников, чтобы решать свои задачи

Когда не нужно наследоваться

- Если мы просто хотим использовать некоторые методы класса-родителя, чтобы выполнить свою работу
- При этом наш класс логически не сильно связан с классом-родителем, либо бОльшая часть методов класса-родителя ему вообще не нужна

Когда не нужно наследоваться

- Допустим, есть класс «Окно операционной системы». Он очень сложно устроен и много чего умеет, например, отрисовываться на экране, получать события от пользователя о нажатиях мыши и клавиатуры и т.д. И еще у него есть ширина и высота и методы для работы с ними
- Допустим, мы хотим создать свой класс для геометрических фигур, и нам тоже надо уметь работать с шириной и высотой. Отнаследовавшись от окна, мы бы получили реализацию этих методов
- Но тем самым мы:
  - Получили много лишнего кода
  - Наш класс может использоваться везде, где нужны окна, а это не нужно

Когда не нужно наследоваться

- Общее правило такое – если вы при наследовании не переопределили ни один виртуальный метод, то наследование не нужно
- Тогда можно просто обойтись полем-ссылкой на нужный объект