Лекция 4.
Интерфейсы.
Абстрактные классы.
Отношения между классами.

Абстрактные классы

- **Абстрактный класс** – это класс, экземпляры которого нельзя создать
- Базовый класс объявляют абстрактным, если не имеет смысл создавать его экземпляры, а имеет смысл создавать только экземпляры его наследников

- Пусть класс Shape – абстрактный:
- Shape a = new Shape(); // ошибка компиляции
  // нельзя создавать экземпляр

Абстрактные классы

```
public abstract class Shape
{
    private Color color;
    public abstract double GetWidth();
    public abstract double GetHeight();
    public abstract double GetArea();

    protected Shape(Color color)
    {
        this.color = color;
    }
    protected Color GetColor()
    {
        return color;
    }
}
```

Абстрактный класс надо пометить словом abstract

Абстр. класс может иметь **абстрактные методы** – методы без реализации. Их надо пометить словом abstract

Конструктор часто делают protected - все равно создать экземпляры нельзя

Абстрактный класс может иметь обычные поля и методы

Абстрактные классы

- Экземпляры абстрактного класса нельзя создать
- `Shape a = new Shape();  // ошибка компиляции`
- Если в классе есть хотя бы один абстрактный метод или от родителей достался нереализованный абстрактный метод, то класс обязан быть абстрактным
- ```
  public class A
  {
      public abstract void F();
  }
  // ошибка компиляции – класс должен быть
  // помечен как abstract
  ```
  ```

Когда следует делать класс абстрактным?

- Когда нет смысл создавать экземпляры этого класса
- Например, на текущем уровне абстракции не понятно как реализовать какие-то методы и нет какой-то разумной реализации по умолчанию
- ```
  public abstract class Shape
  {
      public abstract double GetWidth();
      public abstract double GetHeight();
      public abstract double GetArea();
  }
  ```
- Тут не понятно каковы размеры и площадь фигуры, потому что мы не знаем её тип, положение
  ```

Когда следует делать класс абстрактным?

```
public class Square : Shape
{
    private double sideLength;

    public Square(Color color, double sideLength) : base(color) {
        this.sideLength = sideLength;
    }

    public override double GetWidth() {
        return sideLength;
    }
    public override double GetHeight() {
        return sideLength;
    }
    public override double GetArea() {
        return sideLength * sideLength;
    }
}
```

Для квадрата мы уже понимаем как посчитать размеры и площадь

Мы реализовали все абстрактные методы всех родителей, поэтому класс можно делать не абстрактным

Когда следует делать класс абстрактным?

- Объекты квадратов уже можно создавать и использовать:
- ```
  Shape s = new Square();
  Console.WriteLine(s.GetArea());
  ```
  ```

Абстрактный класс без абстр. методов

- Класс можно делать абстрактным, даже если в нем нет собственных или унаследованных от родителей абстрактных методов
-
```
public abstract class A
{
    public void F()
    {
        Console.WriteLine(1);
    }
}
```

Абстрактный sealed класс

- Абстрактный класс не может быть sealed
- Потому что экземпляры абстрактного класса создавать нельзя, а если от класса и не наследоваться, то этот класс не имеет смысла
- Аналогично для методов
-
```
public sealed abstract class A
{
    // ошибка компиляции – класс не может
    // быть sealed и abstract одновременно
    public abstract sealed void f()
    {
        // ошибка компиляции
    }
}
```

Абстрактный sealed класс

- Невиртуальные методы не могут быть abstract, будет ошибка компиляции
- То есть все static и private методы не могут быть abstract, потому что их нельзя будет переопределить в потомках

Зачем нужны абстрактные классы?

- Чтобы создавать базовые классы, которые реализуют некоторую общую логику, но при этом некоторые аспекты на данном уровне абстракции еще не известны
- Пример – абстрактный класс Shape, который мы рассматривали

Интерфейсы в терминах ООП

- **Интерфейс** – это абстрактный класс, который содержит только абстрактные методы и статические константы

- То есть в терминах ООП это был бы интерфейс:
-
```
public abstract class Shape
{
    public const double ZERO = 0.0;

    public abstract double GetWidth();
    public abstract double GetHeight();
    public abstract double GetArea();
}
```

Это пример, так делать не нужно

Интерфейсы в C#

- В C# понятие интерфейса несколько иное, их нужно объявлять при помощи ключевого слова interface, а не class
- В C# интерфейс не может содержать любые поля, даже статические константы
- Для интерфейсов принято соглашение именования – начинать их с буквы **I** (от слова Interface)

- ```
  public interface IShape
  {
      double GetWidth();
      double GetHeight();
      double GetArea();
  }
  ```
  ```

Интерфейсы в C#

- ```
  public interface IShape
  {
      double GetWidth();
      double GetHeight();
      double GetArea();
  }
  ```
- Интерфейс может иметь только public члены – методы, свойства, события
- Модификаторы видимости указывать нельзя, они всегда подразумеваются public
  ```

Реализация интерфейса

- Так как интерфейсы в терминах ООП – абстрактные классы, то нельзя создавать их экземпляры
- От интерфейса можно наследоваться, как от обычного класса, указав двоеточие
-
```
public class Square : IShape
{
    public void GetWidth()
    {
        // ..
    }
    // реализация методов GetHeight, GetArea
}
```

Реализация интерфейса

- Вместо слова «наследоваться», про интерфейсы говорят что их «реализовывают»
- То есть **класс Square реализует интерфейс IShape**
- Implement с англ. – реализовывать

Реализация интерфейса

- Если класс реализует интерфейс, то он должен реализовывать все его методы, либо быть абстрактным

Реализация нескольких интерфейсов

- Класс может реализовывать несколько интерфейсов, в этом отличие от классов
-
```
public interface I1
{
    void F();
}
```
-
```
public class A : I1, I2 {
    public void F()
    {
    }
    public void G()
    {
    }
}
```

```
public interface I2 {
    void G();
}
```

Интерфейсы указываются через запятую, их порядок не важен

Реализация интерфейсов и наследование

- Можно одновременно наследоваться от некоторого класса и реализовывать сколько угодно интерфейсов

- ```
  public interface I1
  {
      void F();
  }
  ```

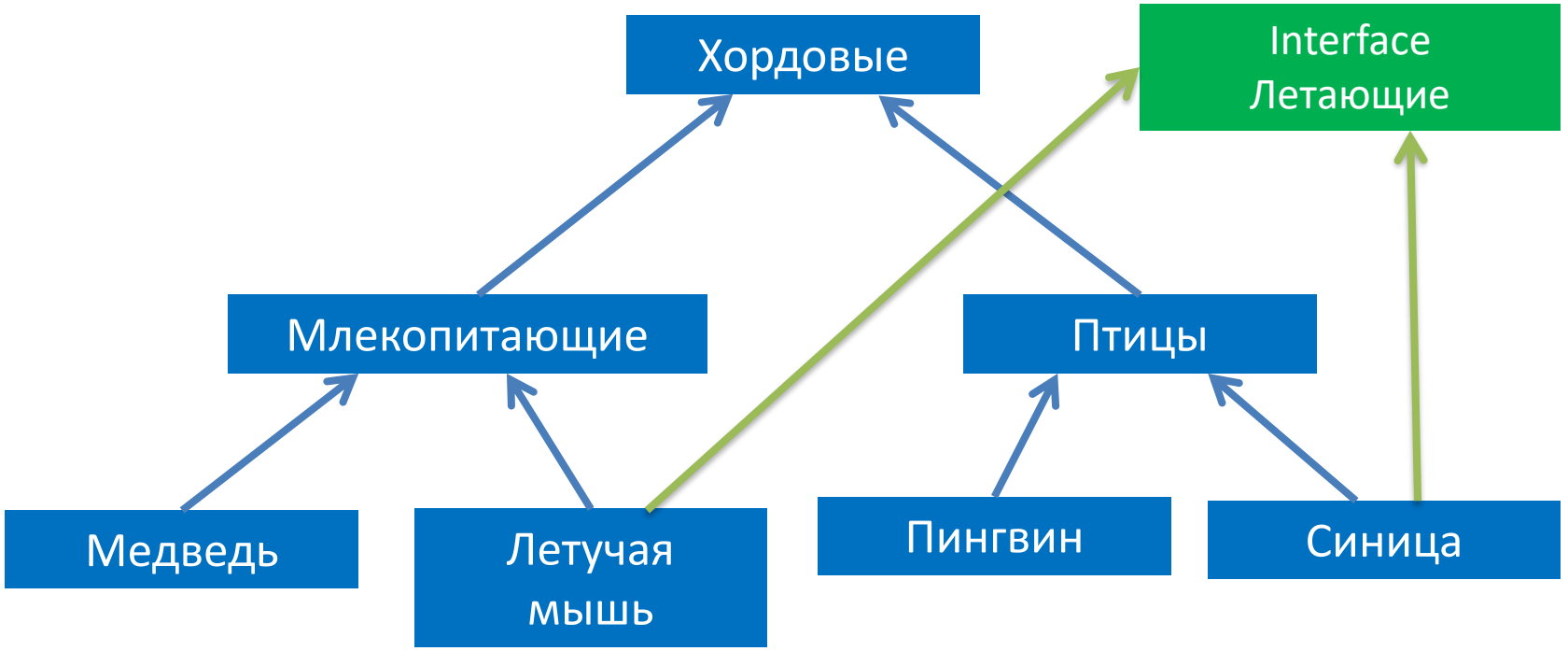
```
public interface I2
{
    void G();
}
```

- ```
  public class A : B, I1, I2
  {
      public void F() {
      }
      public void G() {
      }
  }
  ```

В этом случае класс должен идти первым после :

Когда полезны интерфейсы?

- Чтобы указать для класса признак, который не вписывается в иерархию классов



Когда полезны интерфейсы?

- Когда хочется выделить некоторую абстракцию, но не понятно как она будет реализована. Или реализации могут быть абсолютно несхожими между собой
- Пример:
- ```
  interface ILogger
  {
      void Warning(string text);  // сообщает о предупреждении
      void Error(string text);    // сообщает об ошибке
      void Info(string text);     // информационное сообщение
  }
  ```
- Конкретные логгеры могут писать сообщения в консоль, либо в файлы, либо пересылать их на почту, либо мигать сенсорами и т.д.
  ```

Что выбрать?

Абстрактные классы	Интерфейсы	Итог
Можно наследоваться от одного класса	Можно реализовывать сколько угодно интерфейсов	Часто бывает лучше обойтись интерфейсом, чтобы не потратить свой единственный шанс отнаследоваться
Могут иметь некоторую реализацию по умолчанию	Не могут иметь никакой реализации	Если для абстракции совсем нет общей логики и реализации, то стоит использовать интерфейс. Абстрактный класс полезен если общая логика имеется

Наследование интерфейсов

- Интерфейсы могут наследоваться друг от друга. При этом можно наследоваться от любого количества интерфейсов
- ```
  public interface I1 {
      void F();
  }
  ```

  ```
  public interface I2 {
      void G();
  }
  ```
- ```
  public interface I3 : I1, I2
  {
      // этот интерфейс получает все члены родителей
  }
  ```
- ```
  public class A : I3
  {
      // этот класс должен реализовать F() и G()
  }
  ```
  ```

Использование интерфейсов в коде

- Интерфейсы можно использовать в коде примерно так же, как обычные названия классов и имеет место полиморфизм
- ```
  interface IDoor {
      void Open();
  }
  class WoodDoor : IDoor {
      public void Open() {
          Console.WriteLine("Дверь открылась со скрипом");
      }
  }
  ```
- ```
  IDoor d = new WoodDoor();
  // неявное приведение к типу интерфейса
  ```

Использование интерфейсов в коде

- ```
  interface IDoor {
      void Open();
  }
  ```

  ```
  class WoodDoor : IDoor {
      // …
  }
  ```

  Пусть в классе X есть некоторая статическая функция, принимающая IDoor:

  ```
  public static void OpenDoor(IDoor d) {
      d.Open();
      Console.WriteLine("Дверь была открыта");
  }
  ```

- Тогда можно вызвать:

- ```
  IDoor d = new WoodDoor();
  X.OpenDoor(d);  // Дверь открылась со скрипом \n Дверь была открыта
  ```

Использование интерфейсов в коде

- То есть, ссылки интерфейсного типа дают все преимущества полиморфизма, как и для обычных классов
- Если класс (или один из его родителей) реализует интерфейс, то его можно использовать везде, где требуется ссылка на интерфейс
- ```
  public static void OpenDoor(IDoor d)
  {
  }
  ```
- Сюда можно передать любой объект, который реализует интерфейс IDoor
  ```

Использование интерфейсов в коде

- ```
  public class BadDoor {
      public void Open() {
          // код
      }
  }
  ```

Сюда нельзя передать объект BadDoor – он не реализует интерфейс IDoor

- ```
  public static void OpenDoor(IDoor d) {
  }
  ```

- Даже если в некотором классе есть все методы некоторого интерфейса, но не указано, что класс реализует интерфейс, то нельзя использовать этот объект там, где требуется объект, реализующий этот интерфейс

Явная реализация интерфейса

Явная реализация интерфейса

- Пусть некоторый класс реализует 2 интерфейса, в которых есть одинаковый по сигнатуре метод
- ```
  public interface IControl {
      void Paint();
  }
  ```
  ```
  public interface IShape {
      void Paint();
  }
  ```
- ```
  public class MyClass : IControl, IShape {
      public void Paint() {
          Console.WriteLine("Paint!");
      }
  }
  ```
- В таком случае реализация метода **Paint** будет использоваться для обоих интерфейсов
- А нам хочется разную реализацию

Явная реализация интерфейса

- В этом случае можно использовать фичу языка – **явная реализация интерфейса**

```
public interface IControl {
    void Paint();
}
```

```
public interface IShape {
    void Paint();
}
```

```
public class MyClass : IControl, IShape {
    void IControl.Paint() {
        Console.WriteLine("Paint control!");
    }
    void IShape.Paint() {
        Console.WriteLine("Paint shape!");
    }
}
```

Обратите внимание на синтаксис – модификатор видимости не пишется, а перед именем метода пишется название интерфейса

Явная реализация интерфейса

- Методы, реализованные через явную реализацию интерфейса, доступны только через этот интерфейс
- Поэтому в данном примере у самого класса MyClass вообще нельзя вызвать метод **Paint**
- ```
  MyClass obj = new MyClass();
  obj.Paint();  // ошибка компиляции – нет такого метода

  IControl c = obj;
  c.Paint();    // Paint control!

  IShape s = obj;
  s.Paint();    // Paint shape!
  ```
  ```

Явная реализация интерфейса

- Часто используется и такой вариант – один метод реализуют по-обычному (тогда он доступен через класс), а второй метод – через явную реализацию
-
```
public class MyClass : IControl, IShape {
    public void Paint() {
        Console.WriteLine("Paint control!");
    }
    void IShape.Paint() {
        Console.WriteLine("Paint shape!");
    }
}
```

Явная реализация интерфейса

- Рассмотрим, что будет для примера из предыдущего слайда:
- ```
  MyClass obj = new MyClass();
  obj.Paint();  // Paint control!

  IControl c = obj;
  c.Paint();    // Paint control!

  IShape s = obj;
  s.Paint();    // Paint shape!
  ```
  ```

Отношения между классами

Отношения между классами

- Кроме наследования, между классами могут быть и другие отношения:
- **Ассоциация** – один класс некоторым образом может обратиться к другому
- **Агрегация** – отношение часть-целое, когда один из классов содержит в себе один или несколько экземпляров другого класса
- **Композиция** – вид агрегации, при котором объект-целое управляет жизненным циклом объекта-части

Ассоциация

- **Ассоциация** – один класс некоторым образом может обратиться к другому
- Пусть есть классы A и B. Ассоциацией будет:
  - Внутри класса A есть поле типа B или, например, B[]
  - В некоторых методах класса A создается объекты класса B
  - Некоторые методы класса A принимают объекты класса B в качестве параметра

Агрегация

- **Агрегация** – отношение часть-целое, когда один из классов содержит в себе один или несколько экземпляров другого класса
- **Агрегация** является частным случаем **ассоциации**
- Агрегация подразумевает что в объекте-целом есть поле на объект-часть. Но не всегда, если есть поле, то это агрегация
- Примеры:
  1. Акционер и акции, которыми он владеет. Это **ассоциация**, но не **агрегация**
  2. Автомобиль и двигатель, колеса, корпус. Это **агрегация** и **ассоциация**

Композиция

- **Композиция** – вид агрегации, при котором объект-целое управляет жизненным циклом объекта-части
- При композиции если уничтожается целое, то уничтожаются и его части
- Пример композиции – организм, его части неотъемлимы от самого живого существа, и не живут без него
- Автомобиль может быть примером как композиции, так и просто агрегации
- Если предположить, что в автомобиле все детали сменные, то можно считать это **агрегацией**, но не **композицией** – перед уничтожением автомобиля мы бы забрали многие его детали

Домашняя работа «Люди»

- Пусть мы имеем классы **Голова**, **ЧастьТела**, **Рука**, **Человек**, **Нога**, **Женщина**, **ГруппаЛюдей**, **Мужчина**.
- Опишите как можно связать эти классы в терминах ООП