# 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 extends Shape {
    private double sideLength;

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

    public double getWidth() {
        return sideLength ;
    }
    public double getHeight() {
        return sideLength ;
    }
    public 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();
  System.out.println(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() {
          System.out.println(1);
      }
  }
  ```
  ```

# Абстрактный final класс

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

# Абстрактный final 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 static final double ZERO = 0.0;

    public abstract double getWidth();
    public abstract double getHeight();
    public abstract double getArea();
}
```

Это пример, так делать не нужно

# Интерфейсы в Java

- В Java понятие интерфейса несколько иное, их нужно объявлять при помощи ключевого слова interface, а не class

- ```
  public interface Shape {
      public static final double ZERO = 0.0;

      public double getWidth();
      public double getHeight();
      public double getArea();
  }
  ```
  ```

# Интерфейсы в Java

- ```
  public interface Shape {
      public static final double ZERO = 0.0;

      public double getWidth();
      public double getHeight();
      public double getArea();
  }
  ```
- Интерфейс может иметь только public методы и public static final поля
  ```

# Сокращенная запись

- public interface Shape {
  public static final double ZERO = 0.0;

  public double getWidth();
  public double getHeight();
  public double getArea();
  }
- Сокращенная запись:
- public interface Shape {
  double ZERO = 0.0;

  double getWidth();
  double getHeight();
  double getArea();
  }

Для полей можно не указывать public static final

Для полей можно не указывать public

# Реализация интерфейса

- Так как интерфейсы в терминах ООП – абстрактные классы, то нельзя создавать их экземпляры
- От интерфейса можно наследоваться, как от обычного класса, но при этом используется другой синтаксис – слово implements
- 
```
public class Square implements Shape {
    public void getWidth() {
        // ..
    }
    // реализация методов getHeight, getArea
}
```

# Реализация интерфейса

- Вместо слова «наследоваться», про интерфейсы говорят что их «реализовывают»
- То есть **класс Square реализует интерфейс Shape**
- Implement с англ. – реализовывать
- Для реализации интерфейса нельзя использовать ключевое слово extends

# 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 implements 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();
}
```

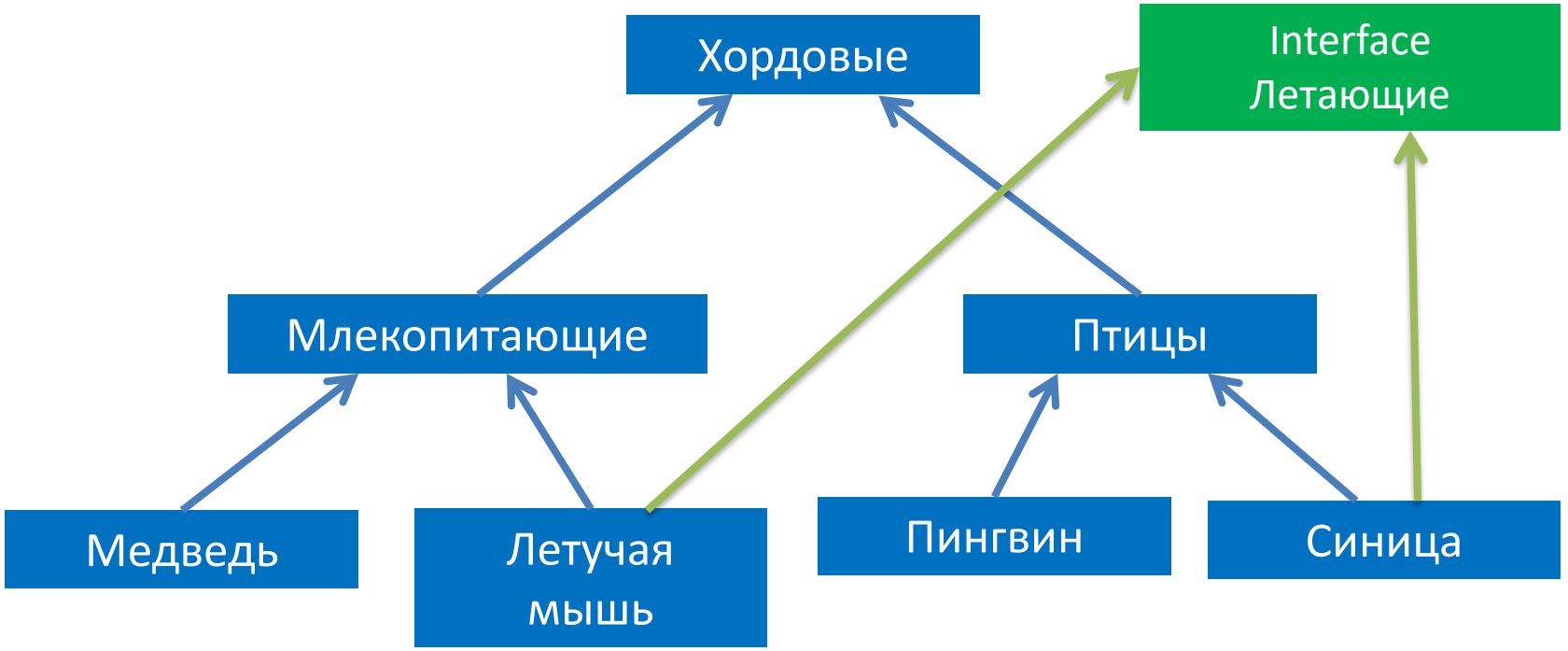
- ```
  public class A extends B implements I1, I2 {
      public void f() {
      }

      public void g() {
      }
  }
  ```

Часть extends должна идти перед implements

# 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 Logger {
      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 (это по-хорошему, но с Java 1.8 есть **default методы** и можно иметь **static методы**) | Если для абстракции совсем нет общей логики и реализации, то стоит использовать интерфейс. Абстрактный класс полезен если общая логика имеется |

# Default методы

- Начиная с Java 1.8 можно добавлять в интерфейсы реализованные методы, если пометить их словом default
- https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html
- Пример:
- ```
  interface Vehicle {
      double getSpeed();   // скорость техники в милях в час
      default double getSpeedInKmh() {
          return getSpeed() * 1.60934;
      }  // скорость в километрах в час
  }
  ```
- Класс, реализующие Vehicle, автоматически получат реализацию **getSpeedInKmh**. При желании ее можно переопределить
  ```

# Default методы

- Эта фича нужна для разработчиков библиотек
- Самим ее лучше не использовать, т.к. по сути это множественное наследование

# Наследование интерфейсов

- Интерфейсы могут наследоваться друг от друга при помощи слова extends. При этом можно наследоваться от любого количества интерфейсов

```
public interface I1 {
    void f();
}
```

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

```
public interface I3 extends I1, I2 {
    // этот интерфейс получает все члены родителей
}
```

```
public class A implements 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 Door {
      void open();
  }
  class WoodDoor implements Door {
      public void open() {
          System.out.println("Дверь открылась со скрипом");
      }
  }
  ```
- ```
  Door d = new WoodDoor();
  // неявное приведение к типу интерфейса
  ```

# Использование интерфейсов в коде

- ```
  interface Door {
      void open();
  }
  ```

  ```
  class WoodDoor implements Door {
      // …
  }
  ```

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

  ```
  public static void openDoor(Door d) {
      d.open();
      System.out.println("Дверь была открыта");
  }
  ```

- Тогда можно вызвать:

- ```
  Door 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(Door d) {
  }
- Сюда можно передать любой объект, который реализует интерфейс Door

# Использование интерфейсов в коде

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

Сюда нельзя передать объект BadDoor – он не реализует интерфейс Door

- ```
  public static void openDoor(Door d) {
  }
  ```

- Даже если в некотором классе есть все методы некоторого интерфейса, но не использовано слово implements, то нельзя использовать этот объект там, где требуется объект, реализующий этот интерфейс

# Coding time

- Интерфейс Logger и несколько разных реализаций:
  - Консоль
  - Файл
  - Показ всплывающего окна

# 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