Лекция 2
Программирование на Java

ФИО преподавателя: Зорина Наталья Валентиновна

e-mail: zorina@mirea.ru,zorina_n@mail.ru

Тема лекции:

«ООП в Java»

null

```
public class Circle {
    public String color;
}
Circle c1 = new Circle();
System.out.println(c1.color); // null
System.out.println(c1.color.length());
// NullPointerException
```

null

Чтобы избежать NullPointerException, желательно в конструкторе класса инициализировать все поля.

null

Явное использование null:

String str = **null**;

Полиморфизм – зачем?

```
public class Circle {

   private double x;
   private double y;
   private double radius;


   public double getArea() {
      return Math.PI * radius * radius;
   }
   // конструкторы/get/set
}
```

```
public class Rectangle {

   private double x;
   private double y;
   private double width;
   private double height;

   public double getArea() {
      return width * height;
   }
   // конструкторы/get/set
}
```

Полиморфизм – зачем?

Как создать массив, содержащий и круги, и прямоугольники?

```
CircleOrRectangle[] shapes = …;


class CircleOrRectangle {
    …
}
```

Полиморфизм – зачем?

Минусы – сложно расширять:

```
CircleOrRectangleOrTriangleOrStar[] shapes = …;
class CircleOrRectangleOrTriangleOrStar {
    … // должен включать свойства всех
      // возможных фигур
}
```

Полиморфизм – зачем?

```
public class Shape {

    private double x;
    private double y;
}
```

Класс-предок

```
public class Circle extends Shape {

    private double radius;

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    public double getArea() {
        return width * height;
    }
}
```

Классы-потомки

Полиморфизм – зачем?

class Circle **extends** Shape

extends выражает отношение “является” или “является частным случаем”, “is a”.

В данном случае Круг является Фигурой (Circle is a Shape).

```
Circle c1 = new Circle(0, 0, 2);
Shape s1 = c1; // корректное присваивание
```

Полиморфизм – зачем?

```
Circle c1 = new Circle(0, 0, 2);
Rectangle r1 = new Rectangle(0, 0, 2, 1);
Shape[] shapes = {c1, r1};
for (Shape s : shapes) {
    System.out.println(s.getX());
}
```

Методы класса Shape наследуются в потомках

Полиморфизм – как?

```
public class Shape {

   private double x;
   private double y;

   public Shape(double x, double y) {
      this.x = x;
      this.y = y;
   }

   public double getX() {
      return x;
   }

   public double getY() {
      return y;
   }
}
```

Полиморфизм – как?

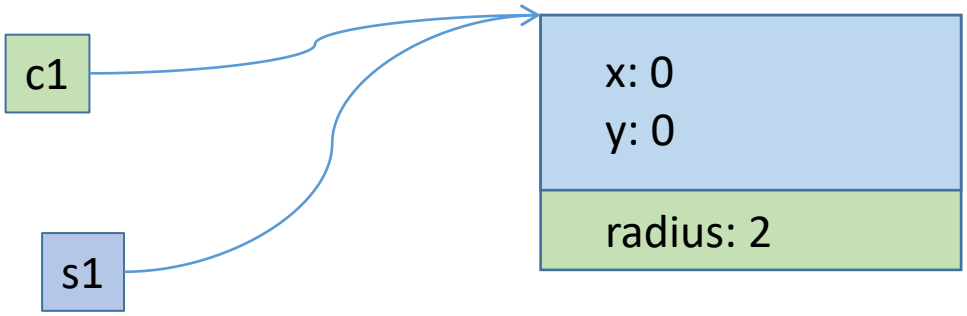
```
public class Circle extends Shape {

    private double radius;

    public Circle(double x, double y, double radius) {
        super(x, y); // Вызов конструктора базового класса
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

Полиморфизм – как?

```
public class Rectangle extends Shape {

   private double width;
   private double height;

   public Rectangle(double x, double y, double width, double height) {
      super(x, y);
      this.width = width;
      this.height = height;
   }

   public double getWidth() {
      return width;
   }

   public double getHeight() {
      return height;
   }

   public double getArea() {
      return width * height;
   }
}
```

Полиморфизм – как?

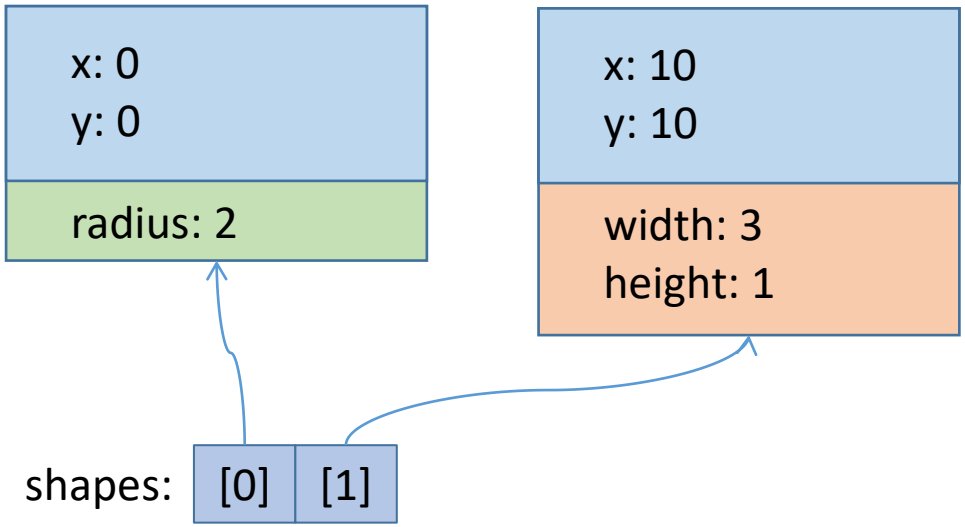
```
Circle c1 = new Circle(0, 0, 2);
Shape s1 = c1;
```



s1 и c1 указывают на один и тот же объект, но s1 “видит” только поля x и y

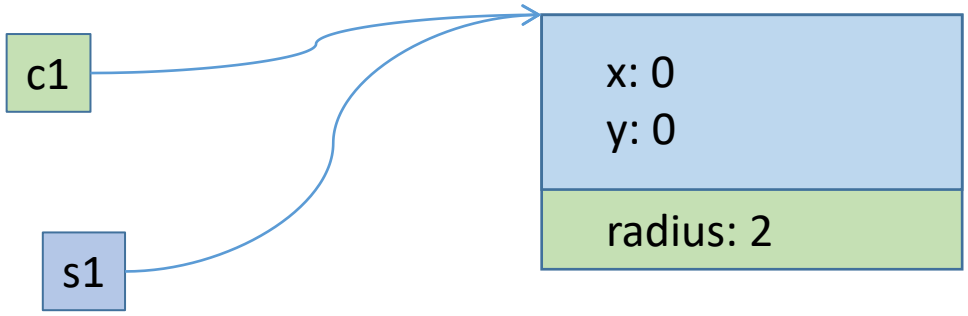
Полиморфизм – как?

```
Circle c1 = new Circle(0, 0, 2);
Rectangle r1 = new Rectangle(10, 10, 3, 1);
Shape[] shapes = {c1, r1};
```

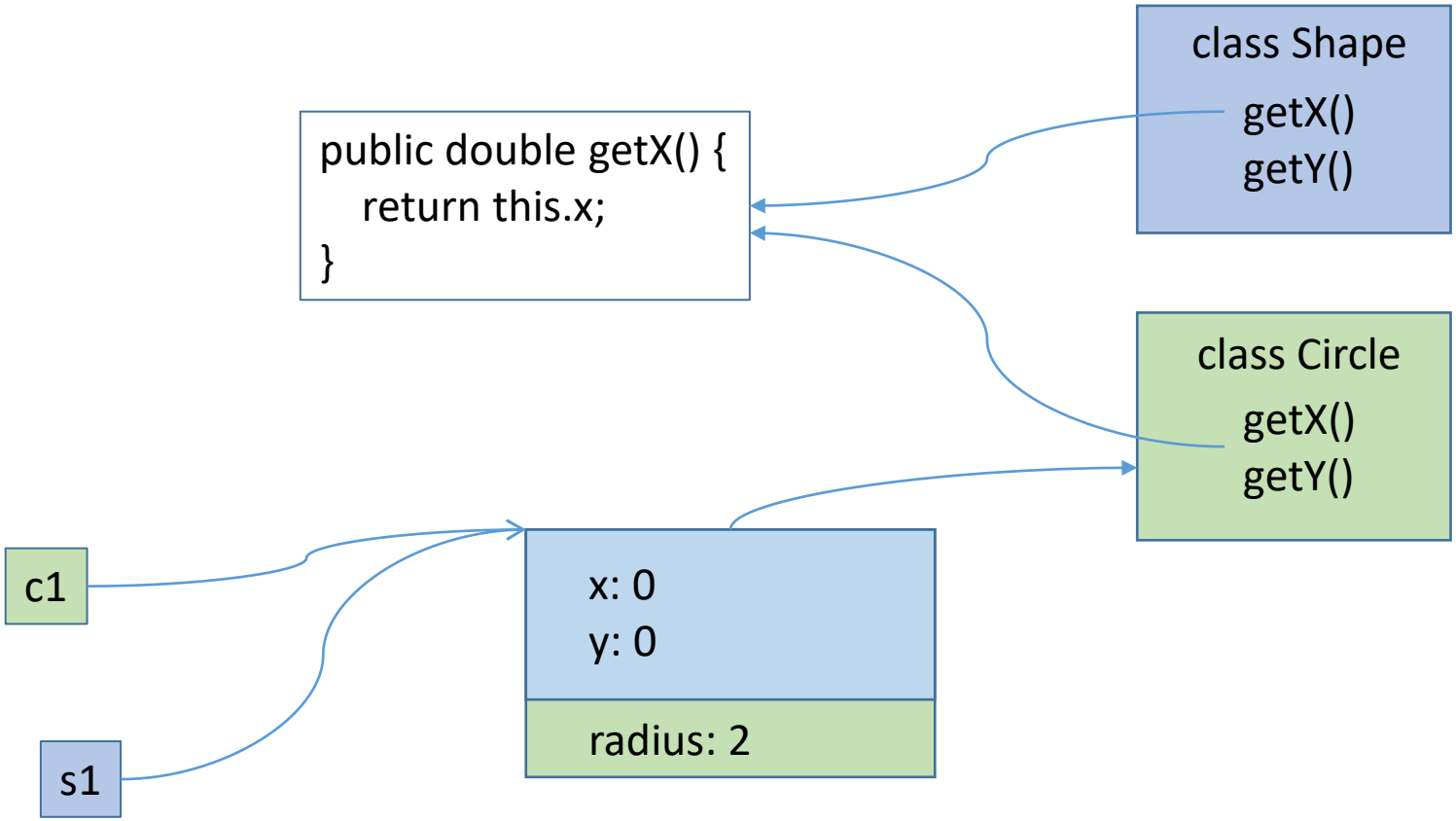


Полиморфизм – как?

```
Circle c1 = new Circle(0, 0, 2);
Shape s1 = c1;
System.out.println(s1.getX());
```



Полиморфизм – как?



Полиморфизм – как?

```
Circle c1 = new Circle(0, 0, 2);
Shape s1 = c1;
System.out.println(s1.getClass());
```

Каждый объект имеет ссылку на описание класса, доступное через метод obj.getClass()

Описание класса включает в себя список методов этого класса

Полиморфизм – как?

При наследовании “Circle **extends** Shape” методы копируются из описания класса Shape в описание класса Circle, так что вызов Circle.getX() и Shape.getX() – вызов одного и того же метода.

Полиморфизм – как?

```
Circle c1 = new Circle(0, 0, 2);
Shape s1 = c1;
System.out.println(s1.getArea()); // ошибка!
```

Статический тип s1 – Shape, в этом классе не определен метод getArea().

Полиморфизм – как?

```
Circle c1 = new Circle(0, 0, 2);
Shape s1 = c1;
if (s1 instanceof Circle) {
    Circle c2 = (Circle) s1; // приведение типа
    System.out.println(c2.getArea());
}
```

Оператор **instanceof** проверяет объект на принадлежность классу

Полиморфизм – как?

Для null instanceof всегда возвращает false, так как тип времени выполнения отсутствует

Java 15

```
Circle c1 = new Circle(0, 0, 2);
Shape s1 = c1;
if (s1 instanceof Circle c2) {
    System.out.println(c2.getArea());
}
```

Полиморфизм – как?

```
Circle c1 = new Circle(0, 0, 2);
Shape s1 = c1;
Rectangle r1 = (Rectangle) s1;
// ClassCastException – s1 является Circle
```

Переопределение методов

```
public class Shape {

    public double getArea() {
        return 0;
    }
}
```

Класс-предок

```
public class Circle extends Shape {

    private double radius;

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    public double getArea() {
        return width * height;
    }
}
```

Классы-потомки

Переопределение методов

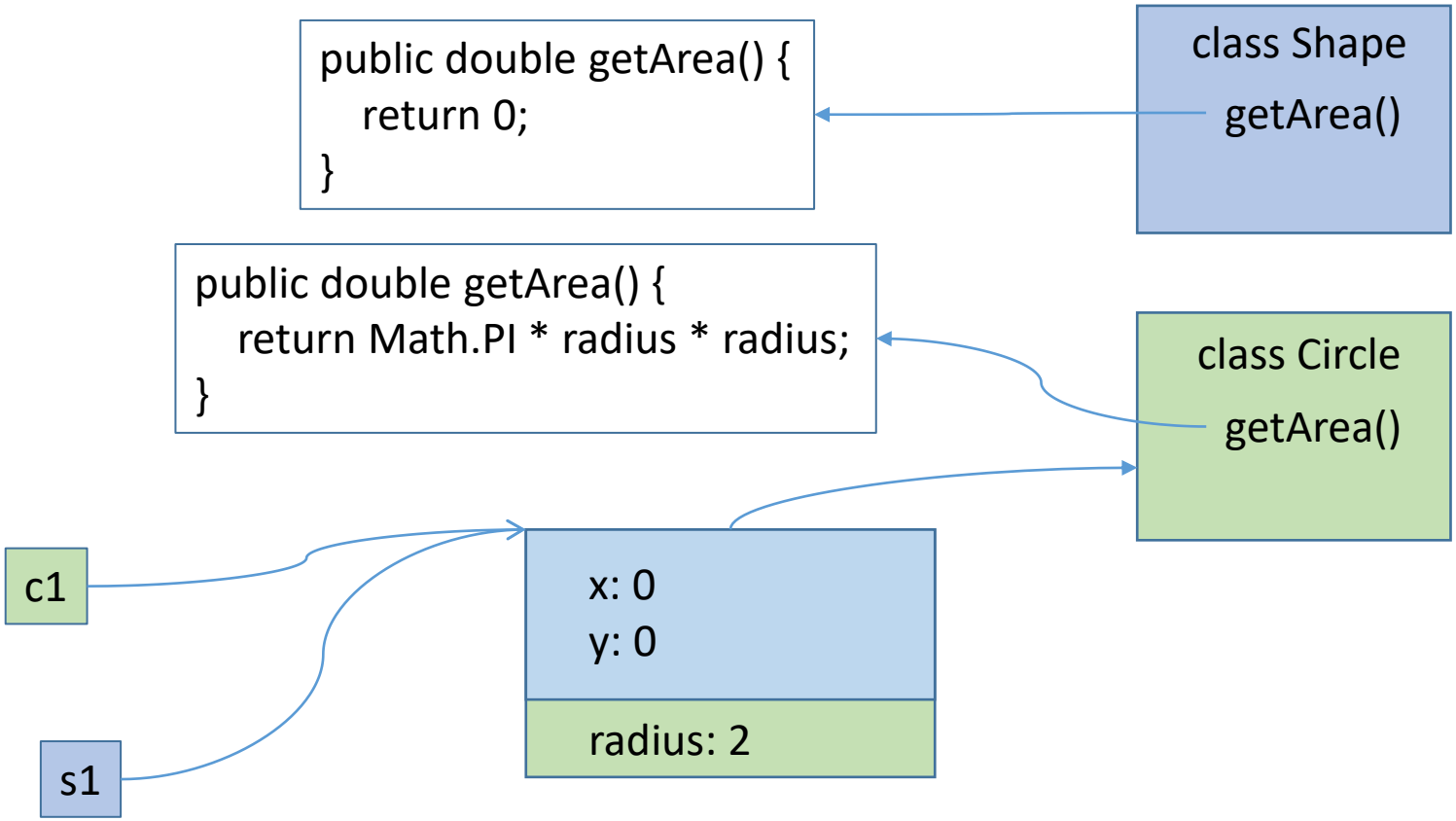
```
Circle c1 = new Circle(0, 0, 2);
Shape s1 = c1;
System.out.println(s1.getArea());
// печатает 12.566370614359172
```

Метод getArea() в классе Circle переопределяет (overrides) этот метод в классе Shape. В Java методы всегда виртуальные.

Переопределение методов

Если класс-потомок определяет метод с той же сигнатурой (именем и типами параметров), что и класс-предок, то этот метод переопределяется.

Переопределение методов



Методы getArea() в классах Shape и Circle ссылаются на разный код

Переопределение методов

Аннотация @Override:

```
public class Circle extends Shape {

   private double radius;

   @Override
   public double getArea() {
      return Math.PI * radius * radius;
   }
}
```

Если класс-предок не определяет метод getArea(), то это ошибка компиляции

Переопределение методов

Аннотация @Override используется для упрощения долгосрочной поддержки больших проектов. Если сигнатура метода в классе-предке будет изменена, то наличие аннотации @Override в классе-потомке гарантирует, что этот метод действительно переопределяет метод класса-предка (базового класса).

Класс Object

Все классы неявно наследуются от класса java.lang.Object:

```
public class Shape extends java.lang.Object {
}
```

Класс Object

Класс Object определяет методы:

- String toString()
- Class getClass()
- boolean equals(Object that)
- int hashCode()

Класс Object

Использование Object:

```
public class PrintStream {
    // Печать строки:
    public void println(String str) { ... }
    // Печать произвольного объекта:
    public void println(Object obj) {
        println(obj.toString());
    }
}
System.out.println(new Circle(0, 0, 2));
```

Переопределение методов

```
public class Thing {

    private int value;

    public Thing(int value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return "value: " + value;
    }
}
```

Переопределение методов

```
public class PrettyThing extends Thing {

    public PrettyThing(int value) {
        super(value); // Вызов конструктора базового класса
    }

    @Override
    public String toString() {
        // Вызов метода базового класса:
        return "{{{" + super.toString() + "}}}";
    }
}
```

Переопределение методов

```
public class Thing {

    private int value;

    public Thing(int value) {
        this.value = value;
    }
    // Модификатор final запрещает переопределение метода:
    public final int getValue() {
        return value;
    }
}
```

Переопределение методов

```
// Модификатор final запрещает наследование от класса:
public final class Thing {

    private int value;

    public Thing(int value) {
        this.value = value;
    }
}

public class OtherThing extends Thing { // Ошибка
```

Модификаторы доступа

Модификатор доступа protected дает доступ к члену класса для:

- всех наследников класса
- всех классов в том же пакете

Модификаторы доступа

- public: доступ открыт для всех
- protected: классы-потомки и классы в том же пакете
- без модификатора (доступ по умолчанию): классы в том же пакете
- private: доступен только внутри класса

Абстрактные классы

```
public class Shape ...
public class Circle extends Shape ...
public class Rectangle extends Shape ...

Shape s1 = new Circle(0, 0, 2);
Shape s2 = new Rectangle(10, 10, 3, 1);
Shape s3 = new Shape(1, 1); // Что это за фигура?
```

Создание объектов класса Shape не имеет смысла, он нужен только как предок других классов

Абстрактные классы

```
public abstract class Shape {

    private double x;
    private double y;

    // Для абстрактных классов конструктор обычно
    // имеет доступ protected, так как вызывается только
    // из наследников:
    protected Shape(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

Абстрактные классы

Экземпляр абстрактного класса невозможно создать:

Shape s0 = new Shape(0, 0); **// Ошибка**

Абстрактные классы используются для выражения того, что класс используется только в качестве базового класса для наследования, а не сам по себе

Абстрактные классы

Абстрактные (и только абстрактные) классы могут иметь абстрактные методы:

```
public abstract class Shape {

    private double x;
    private double y;

    protected Shape(double x, double y) {
        this.x = x;
        this.y = y;
    }
    // Площадь 0 не имеет смысла для конкретной фигуры:
    public abstract double getArea();
}
```

Абстрактные классы

Метод, являющийся абстрактным, должен быть определен в классе-потомке (так как у него нет реализации, которая могла бы быть унаследована от базового класса)

```
public class Circle extends Shape {
    // Ошибка: метод getArea() не определен
}
```

Абстрактные классы

Про метод, переопределяющий абстрактный метод, говорят, что он реализует (implements) этот метод базового класса

В среде разработки:

- Ctrl+O: override (переопределить) конкретный (т.е. не абстрактный) метод
- Ctrl+I: implement (реализовать) абстрактный метод

Абстрактные классы

В классе-предке:

- final: модификатор для конкретного метода, нельзя переопределить в классах-наследниках
- abstract: модификатор для абстрактного метода, должен быть реализован в классах-наследниках

ОО проектирование

Мы хотим работать с разными объектами одного вида (например, геометрическими фигурами). Для этого мы создаем абстрактный базовый класс, определяющий свойства и методы всех фигур и наследуем от него конкретные классы, реализующие методы для конкретной фигуры. При этом конкретные классы обычно final, т.е. иерархия наследования имеет только два уровня.

ОО проектирование

Код, использующий фигуры, может использовать базовый класс, не заботясь о деталях реализации в конкретных классах:

```
public Shape maxArea(Shape s1, Shape s2) {
    // Выбор фигуры с макс. площадью
    if (s1.getArea() > s2.getArea()) return s1;
    else return s2;
}
```

ОО проектирование

Принцип подстановки Барбары Лисков (Liskov Substitution Principle, LSP):

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Мы вызываем метод getArea(), не заботясь о том, что он реально делает.

ОО проектирование

Пример нарушения LSP:

```
public abstract class Shape {
    // Должно быть всегда > 0
    public abstract double getArea();
}
public class BadShape extends Shape {
    public double getArea() { return -1; }
}
```

Проблемы наследования

Квадрат является частным случаем прямоугольника:

```
public class Rectangle {

   private double width;
   private double height;

   public Rectangle(double width, double height) {
      this.width = width;
      this.height = height;
   }

   public void setHeight(double height) {
      this.height = height;
   }
}
```

```
public class Square extends Rectangle {

   public Square(double side) {
      super(side, side);
   }
}
```

```
Square s = new Square(3);
s.setHeight(10);
// Теперь s уже не является квадратом!
```

Класс-потомок, как правило, не может добавлять ограничения на допустимые значения

Проблемы наследования

```
public class Collection {

   public int size() { ... }

   public void add(String element) {
      ...
   }

    public void addAll(Collection other) {
      ...
   }
}
```

```
public class CountCollection extends Collection {
   // Сколько всего добавлено:
   private int count = 0;
   public void add(String element) {
      super.add(element);
      count++;
   }

   public void addAll(Collection other) {
      super.addAll(other);
      count += other.size();
   }
}
```

Если Collection.addAll вызывает add для каждого элемента, то count будет увеличиваться в два раза больше, чем надо, при вызове addAll! Это нарушение инкапсуляции – детали реализации метода addAll оказывают влияние на работу программы

Проблемы наследования

В Java нет множественного наследования. Множественное наследование создает еще одну проблему – Ромбовидное наследование (Diamond problem)

Советы по программированию

Джошуа Блох. Java. Эффективное программирование

Joshua Bloch. *Effective Java: Programming Language Guide*