



# Классы и интерфейсы

## Лекция 3 (продолжение)

# Повторение

- Что такое объекты и классы?
- Что такое конструктор?
- Объяснить различие между перегрузкой и замещением методов
- Объясните принципы инкапсуляции, полиморфизма и наследования.
- Какие модификаторы доступа методов вы знаете?
- Что означает ключевое слово **super**? **this**?
- Что такое пакеты?

# Объекты и классы

- **Объект** – образец класса, который создается при помощи ключевого слова **new**.
- Обращаться к переменным и методам объекта следует с помощью операции "точка":  
имяОбъекта.имяПеременной;  
имяОбъекта.имяМетода(параметры);

# Статические члены класса

- Класс содержит члены двух видов: *поля* и *методы*.
- Для каждого из них задается атрибут, определяющий возможности наследования и доступа.
- Модификатор **static** может использоваться переменной, методом или блоком кода.

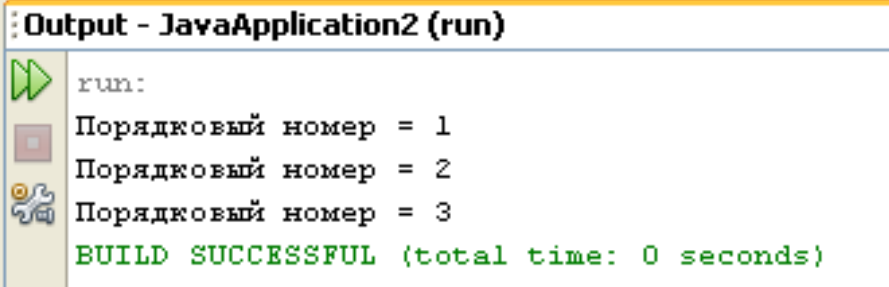
# Статические переменные

- Иногда надо определить поле, общее для всего класса, изменение которого в одном экземпляре повлечет изменение того же поля во всех экземплярах. Такие поля называются **переменными класса**.
- Для переменных класса выделяется только одна ячейка памяти, общая для всех экземпляров.
- Переменные типа **static** подобны глобальным переменным, то есть доступны из любого места кода.

# Статические переменные

```
class Students {  
    private static int number;  
    Students() {  
        number++;  
        System.out.println("Порядковый номер = "+number);  
    }  
}
```

```
public class StudentsTest{  
    public static void main(String[] args){  
        Students ivanov = new Students(),  
        petrov = new Students(),  
        sidorov = new Students()  
    }  
}
```



```
Output - JavaApplication2 (run)  
run:  
Порядковый номер = 1  
Порядковый номер = 2  
Порядковый номер = 3  
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Статические методы

- Иногда требуется создать метод, который можно было бы использовать вне контекста какого-либо объекта его класса.
- **Статические методы** вызываются для целого класса, а не для каждого конкретного объекта, созданного на его основе.
- Статический метод может выполнять задачи, общие для всех объектов класса.
- Более того, они могут выполняться, даже если не создан ни один экземпляр класса.
- Статические методы могут непосредственно обращаться только к другим статическим методам.
- В статических методах ни в каком виде не допускается использование ссылок **this** и **super**.

# Статические методы

- Статический метод работает лишь со статическими переменными и статическими методами класса.
- Статический метод не может быть отменен, то есть стать не статическим.
- Достаточно уточнить имя метода **именем класса** (а не именем объекта), чтобы метод мог работать.

`Math.abs(x)`

`Math.sqrt(x)`

`System.out.println()`

`main()`



# Статические блоки

- Класс также может содержать **блоки статической инициализации**, которые присваивают значения статическим полям или выполняют иную необходимую работу.
- **Блок инициализации** – блок кода между фигурными скобками, который выполняется прежде, чем будет создан объект класса.
- **Статический блок инициализации** – определенный блок, использующий ключевое слово *static*.

# Статические блоки

- Статические переменные инициализируются еще до начала работы конструктора, но при инициализации можно использовать только константные выражения.
- Если же инициализация требует сложных вычислений, например, циклов для задания значений элементам статических массивов или обращений к методам, то эти вычисления заключают в блок, помеченный словом **static**, который тоже будет выполнен до запуска конструктора:

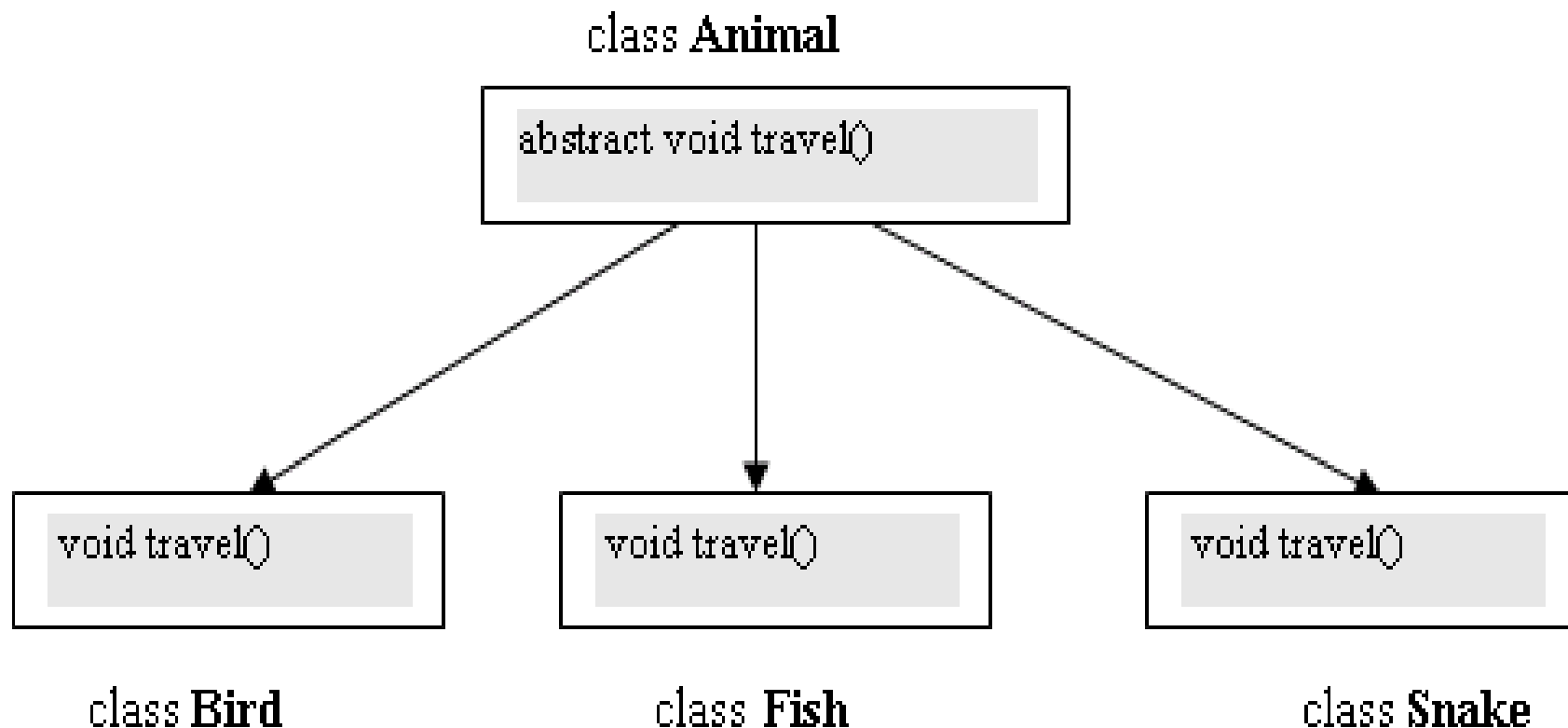
```
static int[] a = new int[10];  
static {  
    for(int k = 0; k < a.length; k++)  
        a[k] = k * k;  
}
```

# Абстрактные классы

- **Абстрактным** называется класс, который содержит хотя бы один абстрактный метод.
- **Абстрактный метод** содержит только описание (заголовок с параметрами), но не содержит тела метода.
- И в определении класса, и в описании метода нужно указывать ключевое слово **abstract**.
- Объекты абстрактного класса создавать нельзя.
- Абстрактный метод указывает, что выполнение метода должно быть обеспечено в подклассе данного абстрактного класса.

# Абстрактные классы

abstract void **travel** ();



# Абстрактные классы

```
abstract class Figure
{
    public int x, y, width, height;
    public Figure(int x, int y, int width, int height) {
        this.x=x;
        this.y=y;
        this.width=width;
        this.height=height;
    }
    abstract double getArea();
    abstract double getPerimeter();
}
```

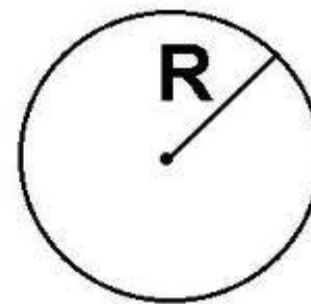
# Абстрактные классы

```
class Circle extends Figure
```

```
{  
    public double r;  
    public Circle(int x, int y, int width) {  
        super(x, y, width, width);  
        r=(double)width / 2.0;  
    }  
}
```

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

```
    public double getPerimeter() {  
        return (2 * Math.Pi * r);  
    }  
}
```



$$S = \pi R^2$$

# Интерфейсы

- **Интерфейс** в языке **Java** представляет собой "чисто абстрактный класс", т.е. класс, все методы которого являются абстрактными.
  - Производный от интерфейса класс "**раскрывает**" (**implements**) интерфейс, предоставляя коды для **всех** его методов.
- Класс может **расширять** только **один** базовый класс, но **раскрывать** он может **несколько** интерфейсов.

# Интерфейсы

- Интерфейсы похожи на классы, но в отличие от последних у интерфейсов нет переменных представителей, а в объявлениях методов отсутствует реализация.
- **Интерфейс** – это шаблон поведения (в форме методов), который должны реализовать другие классы.
- Интерфейсы являются аналогом механизма множественного наследования.
- Методы интерфейса всегда являются открытыми и имеют тип ***public***.



# interface

- Определение интерфейса сходно с определением класса, отличие состоит в том, что в интерфейсе отсутствуют объявления данных и конструкторов.

```
interface имя {  
    тип имя_метода (список параметров);  
    тип имя_final-переменной = значение;  
}
```

- У объявляемых в интерфейсе методов операторы тела отсутствуют.
- Объявление методов завершается символом ; (точка с запятой).
- В интерфейсе можно объявлять и переменные, при этом они неявно объявляются **final** - переменными.

# Интерфейсы

```
public interface myinterface
```

```
{
```

```
    public void add (int x, int y);
```

```
    public void volume (int x,int y, int z);
```

```
}
```

или

```
public interface myconstants
```

```
{
```

```
    public static final double price = 1450.00;
```

```
    public static final int counter = 5;
```

```
}
```

# Оператор **implements**

- Оператор **implements** - это дополнение к определению класса, реализующего некоторый интерфейс(ы).

```
class имя_класса [extends суперкласс]  
                [implements интерфейс0, интерфейс1..]  
    { тело класса }
```

- Если в классе реализуется несколько интерфейсов, то их имена разделяются запятыми.

```
class MyClass implements MyInterface, HisInterface {  
    void myMethod (int n) {  
        System.out.println("Number: " + n);  
    }  
}
```

# Иерархия наследования интерфейсов

- На множестве интерфейсов также определена иерархия наследования, которая не пересекается с иерархией классов.

```
public interface University {  
    // тело интерфейса  
}
```

```
public interface Faculty extends University {  
    // тело интерфейса  
}
```

- Класс, который будет реализовывать интерфейс **Faculty**, должен будет раскрыть все методы из цепочки наследования интерфейсов.

# Пример

```
interface Figure {  
    double getArea();  
    double getPerimeter();  
}  
  
class Rectangle implements Figure {  
    private double width, height;  
    public Rectangle (double width, double height) {  
        this.width=width;  
        this.height=height;  
    }  
    public double getArea() {  
        return (width * height);  
    }  
    public double getPerimeter() {  
        return (2 * (width + height));  
    }  
}}
```

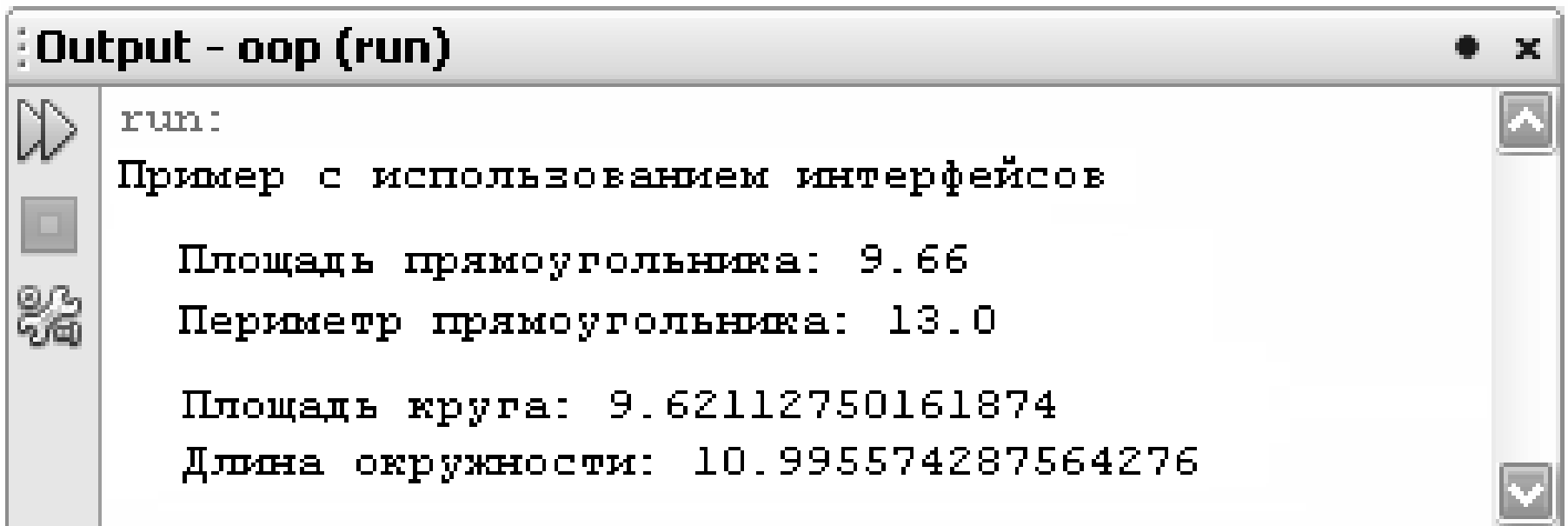
# Пример

```
class Circle implements Figure {  
    public double r;  
    public Circle (double d)    {  
        this.r=d/2;  
    }  
    public double getArea() {  
        return (r * r * Math.PI);  
    }  
    public double getPerimeter() {  
        return (2 * Math.PI * r);  
    }  
}
```

# Пример

```
public class FiguresRunner {  
    public static void main(String args[]) {  
        Rectangle r = new Rectangle(4.2, 2.3);  
        Circle c = new Circle(3.5);  
        System.out.print("Пример с  
                           использованием интерфейсов");  
        System.out.println("Площадь прямоугольника: "  
                             + r.getArea() + "\n"  
                             + "Периметр прямоугольника: "  
                             + r.getPerimeter());  
        System.out.println("Площадь круга: " + c.getArea()  
                             + "\n" + "Длина окружности: "  
                             + c.getPerimeter());  
    }  
}
```

# Результат



The screenshot shows a standard IDE output window. The title bar reads "Output - oop (run)". On the left side of the window, there is a vertical toolbar with four icons: a double right-pointing arrow (run), a square (stop), a pair of scissors (copy), and a document with a plus sign (paste). The main area of the window contains the following text, which is rendered in a monospaced font: "run:" followed by a blank line, then "Пример с использованием интерфейсов" followed by a blank line. Below that, there are four lines of calculated values, each indented: "Площадь прямоугольника: 9.66", "Периметр прямоугольника: 13.0", "Площадь круга: 9.62112750161874", and "Длина окружности: 10.995574287564276". On the right side of the text area, there is a vertical scrollbar with up and down arrow buttons at the top and bottom.

```
run:  
Пример с использованием интерфейсов  
  
Площадь прямоугольника: 9.66  
Периметр прямоугольника: 13.0  
  
Площадь круга: 9.62112750161874  
Длина окружности: 10.995574287564276
```



# Интерфейсы и абстрактные классы

Между интерфейсами и абстрактными классами существует **два важных отличия**:

Интерфейсы	Абстрактные классы
Собственный класс может реализовать несколько интерфейсов.	Собственный класс может расширить только один абстрактный класс.
Интерфейс ограничивается открытыми методами, для которых не задается реализация, и константами.	Абстрактный класс может содержать частичную реализацию, защищенные компоненты, статические методы и т. д.

# Вложенные классы

- **Вложенные классы** (nested classes) - это классы, объявленные внутри другого класса.
- Область видимости вложенного класса ограничивается включающим классом.
- Существуют два типа вложенных классов:
  - статические (static);
  - нестатические (non-static).

# Внутренние классы

- Самый важный тип вложенного класса - **внутренний класс**.
- Все **нестатические вложенные классы** называются **внутренними**.
- В них нельзя объявлять статические члены.
- Класс вне внутреннего класса называется **внешним классом**.

# Внутренние классы

- Внутренний класс имеет доступ ко всем переменным и методам своего внешнего класса и может обратиться к ним непосредственно тем же самым способом, которым это делают все другие нестатические члены внешнего класса.
- Доступ же к элементам внутреннего класса возможен из внешнего класса только через объект внутреннего класса, который должен быть создан в коде метода внешнего класса.

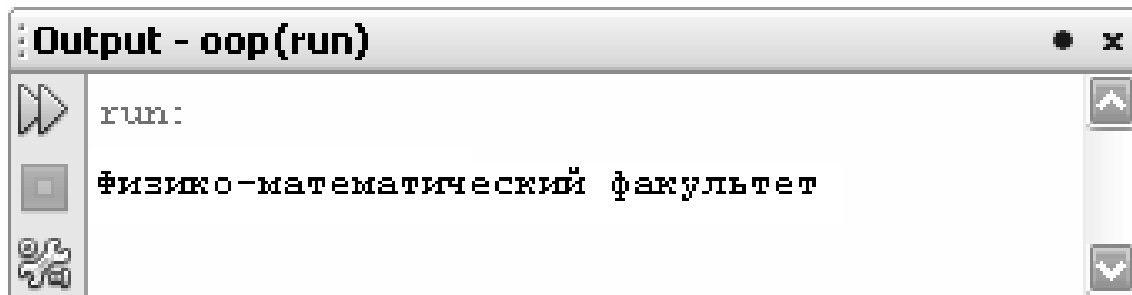
# Пример

```
package oop8;

public class University {
    public class Faculty {
        public void typeName() {
            System.out.print("Физико-математический
                             факультет");
        }
    }

    public void init() {
        Faculty fmf = new Faculty();
        fmf.typeName();
    }

    public static void main (String args[]) {
        University
        obj.init();
    }
}
```



# Классы-оболочки

- Один из основных принципов Java:  
"Всё является объектом" – "Everything is an object".
- **Примитивные типы данных** – это не объекты. Следовательно, они не могут быть созданы или получить обращение к методам.
- Чтобы создавать объекты примитивных типов данных или управлять ими, необходимо использовать классы-оболочки.
- **Классы-оболочки** предназначены не для вычислений, а для действий, типичных при работе с классами - создания, преобразования объектов, получения численных значений объектов в разных формах и передачи объектов в методы по ссылке.

# Классы-оболочки

- Классы оболочки размещены в пакете **java.lang**, который всегда подключается по умолчанию (его не нужно импортировать) и содержит наиболее часто используемые классы.

# Классы-оболочки

Data type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

- Они содержат, в основном, методы для преобразования данных из одного типа в другой.