

08.



*Лекции по информатике для студентов  
первого курса Высшей школы ИТИС  
2019 год*

**МИХАИЛ АБРАМСКИЙ**  
старший преподаватель  
Высшая школа ИТИС КФУ

# На самом деле

- Player – тоже наследник.
- И любой класс, объявляемый в java – наследник класса Object.
- Object – корень иерархии классов. В нем даже есть свои собственные методы
  - Попробуйте вывести Player с помощью `System.out.println`

# Выведется

Player@15db9742

Название класса + значение метода hashCode, который объявлен и реализован в Object и унаследовался в Player (возвращает число, уникальное для каждого объекта).

Но как получилась эта строка?

- В **Object** есть метод **toString**, который возвращает строковое представление объекта, который тоже есть у любого класса, а значит и у Player.
- Соответственно, println неявно вызывает метод **toString**, но реализация этого метода абсолютно нас не устраивает:
  - Player@15db9742, как информативно!

# Переопределение

- Изменение потомком реализации родительского метода.
- *Если я вывожу на экран игрока, что я хочу увидеть? Да пусть хотя бы имя!*
- В Player добавим метод:

```
public String toString() {  
    return name;  
}
```

и все будет хорошо

# Разница между перегрузкой и переопределением

- **Перегрузка** – название метода общее, наборы параметров разные
  - Существует и без наследования – мы перегружали конструкторы в Player
- **Переопределение** – полная идентичность заголовка методов
  - Существует только в контексте наследования – в одном классе не может быть двух методов с одинаковыми заголовками

# Поиск реализации метода при вызове у объекта

1. Сначала ищется реализация метода в классе
2. Если метод с нужной сигнатурой не найден, проводится поиск у родителя.
  - *Старательно умалчиваю тот факт, что в Java у класса может быть только один родитель.*
3. Если и там не найден, то у родителя родителя, и т.д. до Object
4. Если метод не найден у Object, то выведется ошибка «Symbol not Found»
  - Где Symbol – это не символ, а class-member.

# Переопределим что-нибудь еще

- Пусть HealerPlayer при выкрике боевого клича еще и хвастается, что он умеет хилить!
- Значит метод shoutBattleCry должен отличаться от Player
  - Но при этом возможно вызывает его (клич-то тоже надо выкликивать!)

# HealerPlayer

```
public void shoutBattleCry() {  
    super.shoutBattleCry();  
    System.out.println("And I can heal!");  
}
```

- Обращаемся к родителю (super), к его реализации shoutBattleCry.
- Затем добавляем то, что характерно только для HealerPlayer.



# Модификатор `final`

- У атрибута – как у переменное – константа.
- У класса:  

```
public final class MyClass
```

– теперь `MyClass` запрещено наследовать.  
– Очень многие классы в Java `final` – напр., `String`, `Scanner`
- У метода:  

```
public final void f () { ... }
```

– теперь `f()` нельзя переопределять в потомках

# Посмотрим на код еще раз

```
public class Player {  
    protected int hp;  
  
    protected String name;  
    protected String battleCry;  
  
    ...  
    public void shoutBattleCry() { ... }  
    public void kick(Player p) { ... }  
  
}  
  
public class HealerPlayer extends Player {  
    private int healPoints;  
    ...  
    public void heal(int p) {  
        ...  
    }  
}
```

# Временные ~~обзывательства~~ соглашения о названиях

Есть объект (экземпляр класса)

- Все его public-методы (их заголовки) назовем **интерфейсом** объекта (класса)
- То, как именно эти метод работают, назовем конкретной **реализацией/поведением** объекта (класса)

# Отцы и дети

- Интерфейс любого потомка включает в себя интерфейс родителя:
  - Т.к. любой объект подкласса может быть объектом суперкласса.
- Но при одинаковых методах интерфейса в них может быть разная реализация.

# И такое возможно

```
Player p = new HealerPlayer();  
p.shoutBattleCry();
```

- Интерфейс `p` определяется `Player` (левая часть, ссылка, интерфейс)
  - `p` не может вызвать `heal` (т.к. у `Player` его нет)
- Но `p.shoutBattleCry()` выведется в реализации `HealerPlayer` (правая часть, реализация)

# Смысл

- В предыдущем примере нет никакой необходимости создавать дочерние классы с интерфейсом родителя
  - ТАМ ВООБЩЕ БРЕД - поэтому происходящее там кажется бредом
    - » Не умеет лечить, но кричит об этом
- Но иногда в этом есть необходимость!

# Телефон и Смартфон, восходящее преобразование

- Класс **Phone**, метод **call**
- Класс **SmartPhone** наследует **Phone**, переопределенный метод **call**, метод **takePhoto()**

```
class Human {  
    public void callWith(Phone p) {  
        p.call();  
    }  
}
```

```
SmartPhone p = new SmartPhone();  
Human h = new Human();  
h.callWith(p);
```

- p.call() – работает, т.к. есть у Phone, но с реализацией SmartPhone
- p.takePhoto() – не работает

# Восходящее преобразование

- Сужение интерфейса потомка до интерфейса родителя.
- `Phone p = new SmartPhone ( ) ;`



# Много детей

У одного родителя может быть **несколько наследников**.

- У нас есть договоры с физ.лицами и есть с юр.лицами.
- Очевидно, что они оба могут быть наследниками общего класса Contract.

```
public class CompanyContract {  
    private String subject;  
    private Date dueTo;  
    private double cost;  
    private Company company;  
}  
  
public class IndividualContract {  
    private String subject;  
    private Date dueTo;  
    private double cost;  
    private Individual individual;  
    private Employee responsible;  
}
```

# Наводим иерархию

```
public class Contract {  
    protected String number;  
    protected String subject;  
    protected Date dueTo;  
    protected double cost;  
}
```

```
public class IndividualContract extends Contract {  
    private Individual individual;  
    private Employee responsible;  
}
```

```
public class CompanyContract extends Contract {  
    private Company company;  
}
```

# new Требование()

## ***Заказчик:***

«Слушай, Storage договоров нужно и для физ, и для юр.лица. Один, общий. Сделай плиз, генеральный бесится. Я вам срок сдачи оттяну.

И еще нужно, чтобы каждый договор умел печатать инфу о себе в System.out – предмет, сумма, срок + нужная инфа физ. или юр.лица.»

# Storage

*жестко привязать нельзя*

```
public class Storage {  
    private static final int CAPACITY = 1000;  
  
    private IndividualContract [] journal =  
new IndividualContract[CAPACITY];  
  
    private CompanyContract [] journal =  
new CompanyContract[CAPACITY];  
  
    private int storageSize = 0;  
    /// ...  
}
```

# Вспоминаем про объекты в памяти

- Объект подкласса хранит экземпляр родительского класса (super). И может подыграть в его качестве.
  - T.e. – IndividualContract – это Contract
  - И CompanyContract – это Contract

# Используем восходящее преобразование

```
public class Storage {  
    private static final int CAPACITY = 1000;  
    private Contract [] journal =  
        new Contract[CAPACITY];  
    private int storageSize = 0;  
    // ...  
}
```

# Кстати!

```
private Contract [] contracts  
    = new Contract [CAPACITY] ;
```

Ни одного объекта Contract кстати не создано!  
Создан массив ссылок на экземпляры Contract, но ни  
одного объекта.

А что тогда с объектами? А вот что!

# Добавляем новый договор в хранилище

```
public class Storage {  
    //...  
    public void add(Contract contract) {  
        contracts[storageSize] = contract;  
        storageSize++;  
    }  
}
```



# И в том месте, где реализовано использование Storage


```
Storage storage = new Storage();  
//...  
IndividualContract ic1 = new IndividualContract(  
    "Development", new Date(2016, 3, 15), 100000);  
  
CompanyContract cc1 = new CompanyContract(  
    "Awesome Development", new Date(2016, 3, 15), 500000);  
  
//В ic1 добавляют физ.лицо, ответственного, в cc1 юр.лицо  
//а потом ...  
storage.add(ic1);  
storage.add(cc1);
```

# Что произошло

- Контракты **IndividualContract** и **CompanyContract** были переданы в метод **add** как **Contract**.
- Что это? Чем является?

# Восходящее преобразование

```
public class Storage {  
    //...  
    public void add(Contract contract) {  
        contracts[storageSize] = contract;  
        storageSize++;  
    }  
}
```



IndividualContract и CompanyContract  
ограничены по родительскому интерфейсу в  
Contract.

*Ну и что? Нам и нужна их специфика.  
Мы используем их как контракты*

# Что там с информацией о договоре

```
public class IndividualContract extends Contract {
    //...
    public void printInfo() {
        System.out.println(number);
        System.out.println(subject + ". Due to " +
                               dueTo + ". Money: " + cost);
        System.out.println(individual);
        System.out.println(responsible);
    }
}


public class CompanyContract extends Contract {
    //...
    public void printInfo() {
        System.out.println(number);
        System.out.println(subject + ". Due to " +
                               dueTo + ". Money: " + cost);
        System.out.println(company);
    }
}
```

# Приводим в порядок

```
public class Contract {  
    public void printInfo() {  
        System.out.println(number);  
        System.out.println(subject + ". Due to " +  
                               dueTo + ". Money: " + cost);  
    }  
}  
  
public class CompanyContract extends Contract {  
    public void printInfo() {  
        super.printInfo();  
        System.out.println(company);  
    }  
}  
  
public class IndividualContract extends Contract {  
    public void printInfo() {  
        super.printInfo();  
        System.out.println(individual);  
        System.out.println(responsible);  
    }  
}
```

# Вывести всю инфу о всех договорах из storage

```
public class Storage {  
    //...  
  
    public void printAllInfo () {  
        for (Contract contract : contracts) {  
            contract.printInfo();  
        }  
    }  
}
```



У нас три printInfo, что вызовется??

# Связывание (binding)

- Присоединение вызова метода к телу метода.
- `contract.printInfo()` – ВЫЗОВ
- методы `printInfo()` есть у нескольких классах.

# в Java - позднее связывание

- Реализация вызываемого метода определяется в момент выполнения!
  - Компилятор не знает заранее, какая реализация `printInfo` будет использована для `contract.printInfo()`!
  - Но JVM потом разрулит все, опираясь на созданные объекты.  
» !
- У договора тип – `Contract`, но в момент вызова:
  - у тех, кто по факту `IndividualContract`, вызовется их реализация;
  - у тех, кто по факту `CompanyContract`, вызовется их реализация;



# Третий принцип (кит) ООП

## ПОЛИМОРФИЗМ

### *Разное понимание*

- ad hoc полиморфизм – один интерфейс, множество реализаций
  - сюда иногда добавляют перегрузку
- Полиморфизм наследования
  - то, что было у нас
- Параметрический полиморфизм
  - обобщение, когда тип параметра – тоже параметр

# Полиморфизм в ООП - это

- Возможность реализовывать уникальное поведение у нескольких подклассов при едином интерфейсе суперкласса.

- Device – электронное устройство.
  - Может включаться и выключаться (но КАК?)
  - Очевидно, работает (но КАК?)
- Phone и Camera – устройства.
  - Понятно, как включается
  - Понятно, как работает

# Device

```
public abstract class Device {  
    private String name;  
    public String getName() {  
        return name;  
    }  
    abstract public void on();  
    abstract public void off();  
    abstract public String getInfo();  
}
```

Абстрактный класс с тремя абстрактными методами.

Абстрактный метод не имеет реализации – значит, чего не может быть у абстрактного класса?

# Phone

```
public class Phone extends Device {  
  
    public void on() {...}  
  
    public void off() {...}  
  
    public String getInfo() {  
        return "Можно звонить, хранить номера,  
                принимать звонки";  
    }  
    public void makeCall() {...}  
}
```

Должны реализовать все абстрактные методы  
или объявить потомка тоже абстрактным

# Camera

```
public class Camera extends Device {  
    public void on() {}  
    public void off() {}  
    public String getInfo() {  
        return "Можно снимать фотографии  
                просматривать их";  
    }  
  
    public void makePhoto() {}  
}
```

# Полиморфизм

```
Device[] devices = new Device[10];  
for (int i = 0; i < devices.length; i += 2) {  
    devices[i] = new Phone();  
    devices[i+1] = new Camera();  
  
}  
  
for (Device device : devices) {  
    System.out.println(device.getInfo());  
}
```

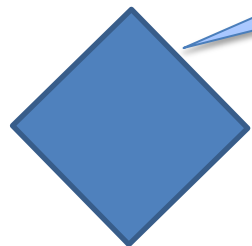
# Множественное наследование

- У класса – два родителя
- Берем все, что есть во всех родителях, в себя.



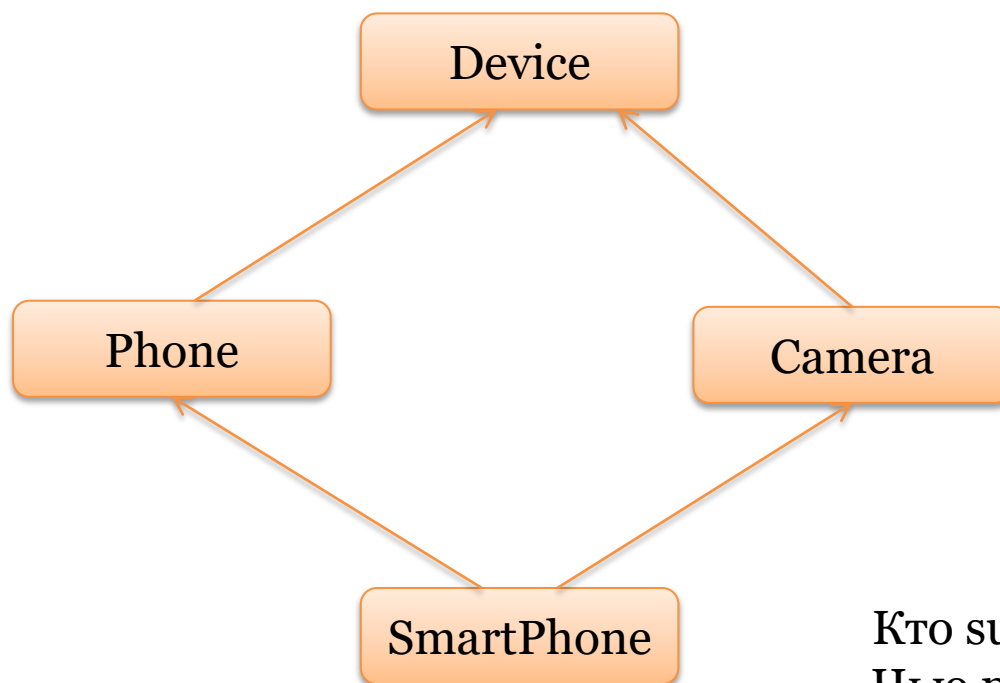
# Почему нельзя множественное наследование?

- *Вот есть же Смартфон – он и телефон, и камера. В чем проблема-то?*
- *В ромбе!*



- Ребят, я вообще не в курсе, зачем меня сюда приплели...

# Ромб наследования



Кто super для SmartPhone?  
Чью реализацию брать  
getInfo? а on? а off?

# Мешает реализация

- Заголовки-то одинаковые...
- ... а это идея!

# DeviceInterface

```
public interface DeviceInterface {  
    void on();  
    void off();  
    String getInfo();  
}
```

Все методы – public abstract (до Java 8).

Разрешены final static константы.

Модификатор – всегда public, но не пишется – не путать с default.

# CallInterface и MakePhotoInterface

```
public interface CallInterface {  
    void makeCall();  
}
```

```
public interface MakePhotoInterface {  
    void makePhoto();  
}
```

# Наследование интерфейсов - множественное

- Нет тела – нет проблем

```
public interface PhoneInterface  
    extends DeviceInterface, CallInterface {  
  
}
```

# Реализация интерфейса в классе

Класс обязан переопределить все методы интерфейса или остаться абстрактным.

```
public class Phone implements PhoneInterface {  
    public void on() {}  
    public void off() {}  
    public String getInfo() {  
        return "Можно звонить, хранить номера, принимать звонки";  
    }  
    public void makeCall() {  
        System.out.println("Calling");  
    }  
}
```

# Решение проблемы множественного наследования для SmartPhone

```
public class SmartPhone extends Phone
    implements MakePhotoInterface{
    public void makePhoto() {...}

}
```

*или*

```
public class SmartPhone
    implements PhoneInterface, MakePhotoInterface{
    public void makePhoto() {...}
    public void makeCall() {...}
    public void on() {...}
    public void off() {...}
    public String getInfo() {...}
}
```



# Полиморфизм работает на уровне интерфейсов

Не забываем про восходящее преобразование:

```
DeviceInterface phone1 = new SmartPhone();  
PhoneInterface phone2 = new SmartPhone();  
MakePhotoInterface phone3 = new SmartPhone();
```

# Разница между абстрактным классом и интерфейсом

- Абстрактный класс может иметь реализованные методы
- Наследование от класса – 1, реализованы могут быть несколько интерфейсов
- Есть атрибуты у абстрактного класса
- Могут быть `private`, `protected`, `default` (пакет) поля и методы

# Анонимный класс

## Мгновенная реализация интерфейса или абстрактного класса в коде

```
public static void main(String[] args) {  
  
    MakePhotoInterface camera = new MakePhotoInterface() {  
  
        public void makePhoto() {  
            System.out.println("Make photo as Nikon Camera");  
        }  
  
    };  
  
    camera.makePhoto();  
}
```

! lambda in Java 8

# @Override

Аннотация, проверяет, действительно ли это переопределение (подробнее в следующем семестре)

```
public static void main(String[] args) {  
  
    MakePhotoInterface camera = new MakePhotoInterface() {  
  
        @Override  
        public void makePhoto() {  
            System.out.println("Make photo as Nikon Camera");  
        }  
  
    };  
  
    camera.makePhoto();  
}
```

# java 8 - default

- Методы по умолчанию в интерфейсе

```
interface Formula {  
    double calculate(int a);  
    default double sqrt(int a) {  
        return Math.sqrt(a);  
    }  
}
```

При реализации можно не определять!  
Не о множественном наследовании!

# ПЕРЕЧИСЛЕНИЯ

# Хардкод (Hardcode).

## Случай с числами

```
public static void main(String[] args) {  
    Scanner scanner = new Scanner(System.in);  
    int[] a = new int[42];  
    for (int i = 0; i < 42; i++) {  
        a[i] = scanner.nextInt();  
    }  
    for (int i = 0; i < 42 / 2; i++) {  
        a[i] = a[42 - i - 1];  
    }  
}
```

# Хардкод. Еще хуже

```
public static void main(String[] args) {  
    Scanner scanner = new Scanner(System.in);  
    int[] a = new int[42];  
    for (int i = 0; i <= 41; i++) {  
        a[i] = scanner.nextInt();  
    }  
    for (int i = 0; i < 21; i++) {  
        a[i] = a[41 - i];  
    }  
}
```



# Константы как частный способ решения проблемы

```
public static void main(String[] args) {  
    Scanner scanner = new Scanner(System.in);  
    final int SIZE = 42;  
    int[] a = new int[SIZE];  
    for (int i = 0; i < SIZE; i++) {  
        a[i] = scanner.nextInt();  
    }  
    for (int i = 0; i < SIZE / 2; i++) {  
        a[i] = a[SIZE - i - 1];  
    }  
}
```

# Более частый случай – строковый хардкод

```
if (person.getGender().equals("Мужской")) {  
    ...  
}  
  
...  
if (direction.getName().equals("Вверх")) {  
    ...  
}  
  
...  
if (season.getName().equals("Лето")) {  
    ...  
}
```

# Одно из решений – строковые константы

```
final String SUMMER = "Summer";  
final String MALE_GENDER = "Male";
```

## Проблемы:

- название и значение дублируют друг друга,
- где хранить, чтобы обращаться?
- как быть с Summer и SUMMER?
- ...

# Другое решение

```
class Season {  
    final static int WINTER = 0;  
    final static int SPRING = 1;  
    final static int SUMMER = 2;  
    final static int FALL = 3;  
}
```

## Проблемы:

- Откуда знать весь диапазон значений и как его перебрать?
- Если `x == 0`, то `Season.WINTER == x`, но действительно ли корректно считать левую переменную `x` хранящей значение «Зима»

# Итак

Нужен тип данных:

- Чтобы у переменных этого типа явно было видно значение.
- Чтобы можно было легко перебрать все его значения,
- *Чтобы не хардкодить,*

Он есть! И это...

# Перечисления (Enumerations)

Объявление:

```
enum Season {  
    WINTER, SPRING, SUMMER, FALL  
}
```

Использование:

```
Season s = Season.SPRING;
```

# Решаем проблемы.

## Перебираем с помощью `values()`

- `values()` возвращает массив из всех значений перечисления

```
for (Season season: Season.values()) {  
    System.out.println(season);  
}
```

# Решаем проблемы. Сравнить можно только с другими значениями перечисления

```
Season season = Season.SUMMER;
```

```
...
```

```
if (season == Season.WINTER)  
    System.out.println("NEW YEAR!");
```

```
// у каждого есть свой порядковый номер  
// выведет 3
```

```
System.out.print(Season.FALL.ordinal());
```

```
// но вот такое сделать не получится
```

```
if (season == 3) {
```

```
    ...
```

```
}
```



# Решаем проблемы. Ввод

- Значение можно восстановить по строке
  - Надо вводить строку с точностью до регистра!

*// Прокатит*

```
Season season = Season.valueOf("WINTER") ;
```

*...*

*// Не прокатит*

```
Season season = Season.valueOf("Winter") ;
```

# Все гораздо интереснее

- Вы думаете, эти WINTER, SUMMER – просто константы?
- А вот и нет! Это объекты!

# Другой enum. Цвет

```
enum Color {  
    RED, GREEN, BLUE, WHITE, BLACK  
}
```

У каждого цвета есть значения RGB.

Наша потребность:

- Чтобы каждый цвет знал свои значения,
- Чтобы каждый цвет мог возвращать строку-представление RGB

Для этого изменим enum.

# «В НОВОМ ЦВЕТЕ»

```
enum Color {  
    RED(255, 0, 0), GREEN(0, 255, 0),  
    BLUE(0, 0, 255), WHITE(255, 255, 255),  
    BLACK(0, 0, 0);  
  
    private int r, g, b;  
    Color(int r, int g, int b) {  
        this.r = r;  
        this.g = g;  
        this.b = b;  
    }  
    public String getRGBValues() {  
        return "(" + r + "," + g + "," + b + ")";  
    }  
}
```

# «В новом цвете». Использование

```
Color color = Color.BLACK;  
System.out.println(color.getRGBValues());
```