Обектно ориентирано програмиране

Основни концепции в JAVA

Емилия Бояджиева, Пламен Миленков

22/10/2014

Public

Софийски университет "Св. Климент Охридски"
Факултет по математика и информатика

Agenda

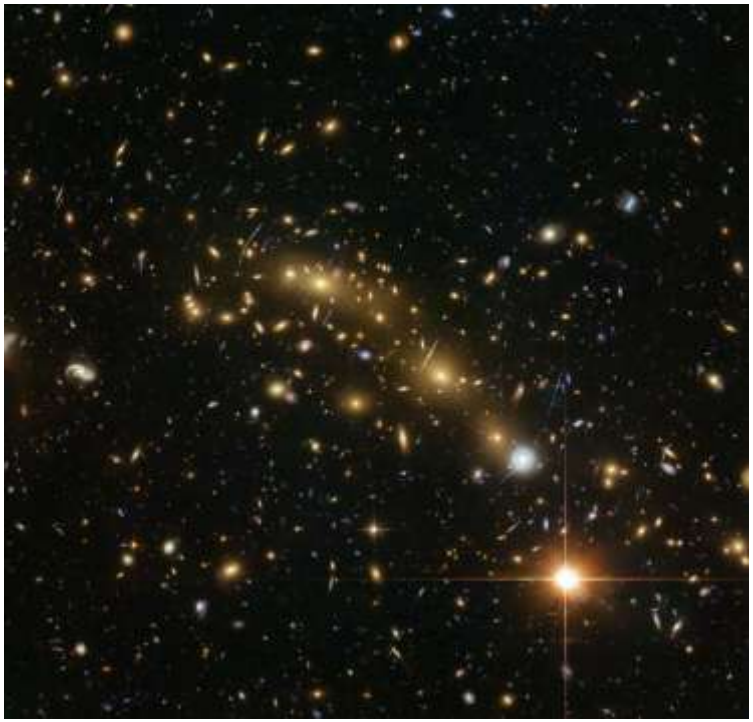
- Класове и обекти
- Енкапсулиране
- Наследяване
- Полиморфизъм
  - Overloading и Overriding
  - Интерфейси
  - Абстрактни класове
- Enum
- equals() и hashCode()
- Вложени класове
- Анонимни класове
- SOLID принципи

ООП
История и значение

За първи път ООП се появява през 1970 в езика SmallTalk

Парадигма за дизайн и имплементация на приложения като съвкупност от обекти, които

- съдържат състояние (член-променливи)
- съдържат поведение (методи)
- комуникират през съобщения



Клас

Клас - дефиниция на обект

- описва състояние – член променливи
- описва поведение – методи
- Декларация и тяло на класа
- Конструктор
- Ключова дума **this**
- Метод – функция за манипулиране на член променливите на класа
  - сигнатура на метод, - параметри
  - тяло на метод - локални променливи
  - return тип

```
public class Human {
    private String name;

    public Human(String name) {
        this.name = name;
    }

    public void whoAmI(){
        System.out.println("My name is " + name);
    }

}
```

Обекти

Обект – конкретна инстанция на класа, !обектът не е клас!

```
public class MainApp {

    public static void main(String[] args) {
        Human ivan = new Human("Ivan");
        ivan.whoAmI();
        Human petar = new Human("Petar");
        petar.whoAmI();
    }
}
```

```
My name is Ivan
My name is Petar
```

Пакети и модификатори за достъп

Пакети: йерархично организиране на код базата

винаги с малки букви

компаниите използват обърнат домейн адрес

sofia-fmi.bg -> bg.uni.sofia.fmi.corejava….

Модификатори за достъп

public - достъпен за всички пакети

private - достъпен само за методите на класа

protected - достъпен за методите на класа и неговите наследници

default **-** достъпен за методите на класове от същия пакет

Енкапсулиране (Encapsulation)

Основен принцип в ООП!

В JAVA се постига чрез модификаторите за достъп

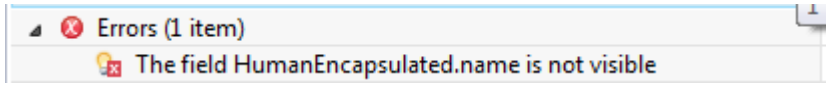
Само вътрешните методи на даден обект имат достъп до неговото състояние, като правят невъзможни неочакваните промени на състоянието от външния свят

Пример 1

```
public class Human {
    String name;

    public Human(String name) {
        this.name = name;
    }


public static void main(String[] args) {
    Human ivan = new Human("Ivan");
    ivan.name = "Faked Ivan";
}
```

Пример 2

```
public class HumanEncapsulated {
    private String name;

    public HumanEncapsulated(String name) {
        this.name = name;
    }


    public static void main(String[] args) {
        HumanEncapsulated ivan = new HumanEncapsulated("Ivan");
        ivan.name = "Faked Ivan";
    }
```



Наследяване (Inheritance)

Основен принцип в ООП!

Позволява преизползване и разширяване на поведение и състояние на вече съществуващи класове

В JAVA се реализира с ключовата дума extends

super

JAVA не поддържа множествено наследяване

```
public class Student extends Human {
    private int number;

    public Student(String name) {
        super(name);
    }
```

```
public static void main(String[] args) {
    Student ivan = new Student("Ivan");
    ivan.whoAmI();
}
```

```
My name is Ivan
```

Polymorphism

От гръцки poly (много) + morphe (форма)

Дефиниция от биологията - съществуване на морфологично различни индивиди в границите на един вид

ООП - наследниците на даден клас споделят поведение от надкласа, но могат да дефинират и собствено поведение

Всички Java обекти са полиморфични, понеже

всеки обект наследява Object класа

Декларация vs. Инициализация

Полиморфизъм
Overriding

Overriding - подкласа предефинира поведението на надкласа

Runtime полиморфизъм

```java
public class Human {
    private String name;

    public Human(String name) {
        this.name = name;
    }

    public void whoAmI(){
        System.out.println("My name is " + name);
    }

}
```

```java
public class Student extends Human {
    private int number;

    public Student(String name,int number) {
        super(name);
        this.number = number;
    }

    public void whoAmI(){
        super.whoAmI();
        System.out.println("My number is "+this.number);
    }
}
```

```
My name is Ivan
My number is 1234
```

Полиморфизъм
Overloading

Overloading - класа може да декларира методи с едно и също име и различен брой параметри

Compile-time полиморфизъм

```
public void move(){
    System.out.println("I am walking using two legs");
}

public void move(String vehicle){
    System.out.println("I move using a "+vehicle);
}
```

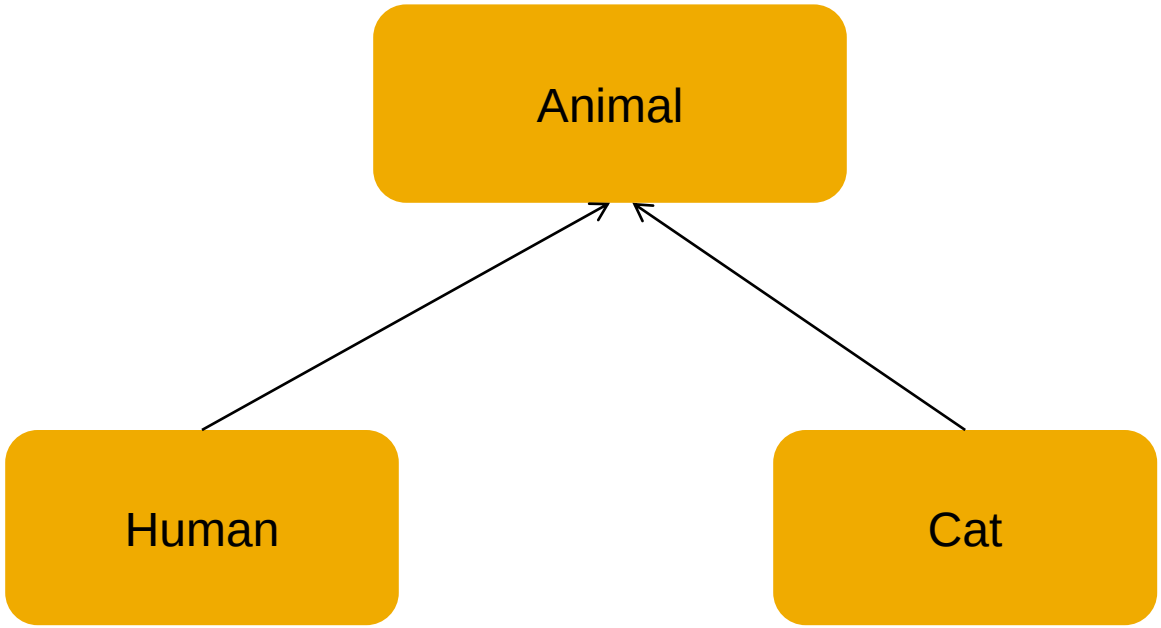
```
    public static void main(String[] args) {
        Human ivan = new Human("Ivan");
        ivan.move();
        ivan.move(4);

    }
```

```
I am walking using two legs
I move using a Car
```

Overloading vs Overriding

	Overloading	Overriding
Кога	Compile-time	Runtime
Къде	In the same class	In subclasses
Performance	Better	
Return type	Може да бъде различен	Запазва се
static, private & final methods	Да	Не
Binding	Статично	динамично
Argument list	Различен	Запазва се

Demo



Inheritance vs subtyping

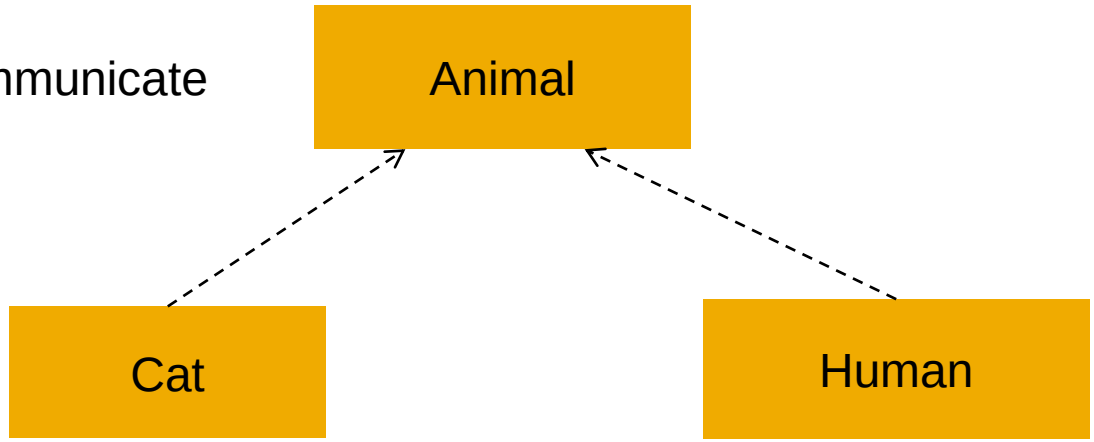
Не винаги наследяване означава подтип !

Обекта A е подтип на обекта B ако във всеки контекст B може да се замени с A

Пр. Студент - Човек

Не винаги обаче това е реализирано чрез наследяване

Пр. eat,move, communicate



Интерфейс

Съвкупност от декларации на методи без имплементация

Описва формално поведение без да го имплементира

Може да съдържа static final член-променлива = константи

```
public interface Animal {

    public void move();

    public void communicate();

}
```

```
public class Human implements Animal{
    private String name;

    public Human(String name) {
        this.name = name;
    }

    public void communicate() {
        System.out.println("I speak");

    }

    public void move(){
        System.out.println("I am walking using two legs");
    }
```

```
public class Cat implements Animal{

    public void move(){
        System.out.println("I am walking with 4 toes");
    }

    public void communicate() {
        System.out.println("I mew");

    }

}
```

Абстрактни класове

Дефинират се с ключовата дума **abstract**

Нямат инстанции

Имат методи без имплементация, които се дефинират с **abstract**

```
public abstract  class Cat implements Animal{

    public void move(){
        System.out.println("I am walking with 4 toes");
    }

    public void communicate() {
        System.out.println("I mew");

    }

    public abstract  void eat();
}
```

```
public class DomesticCat extends Cat{

    public void eat() {
        System.out.println("I eat sometimes rest from my adopters");

    }

}
```

```
public class Leopard extends Cat{

    public void eat() {
        System.out.println("I eat any prey");

    }

}
```

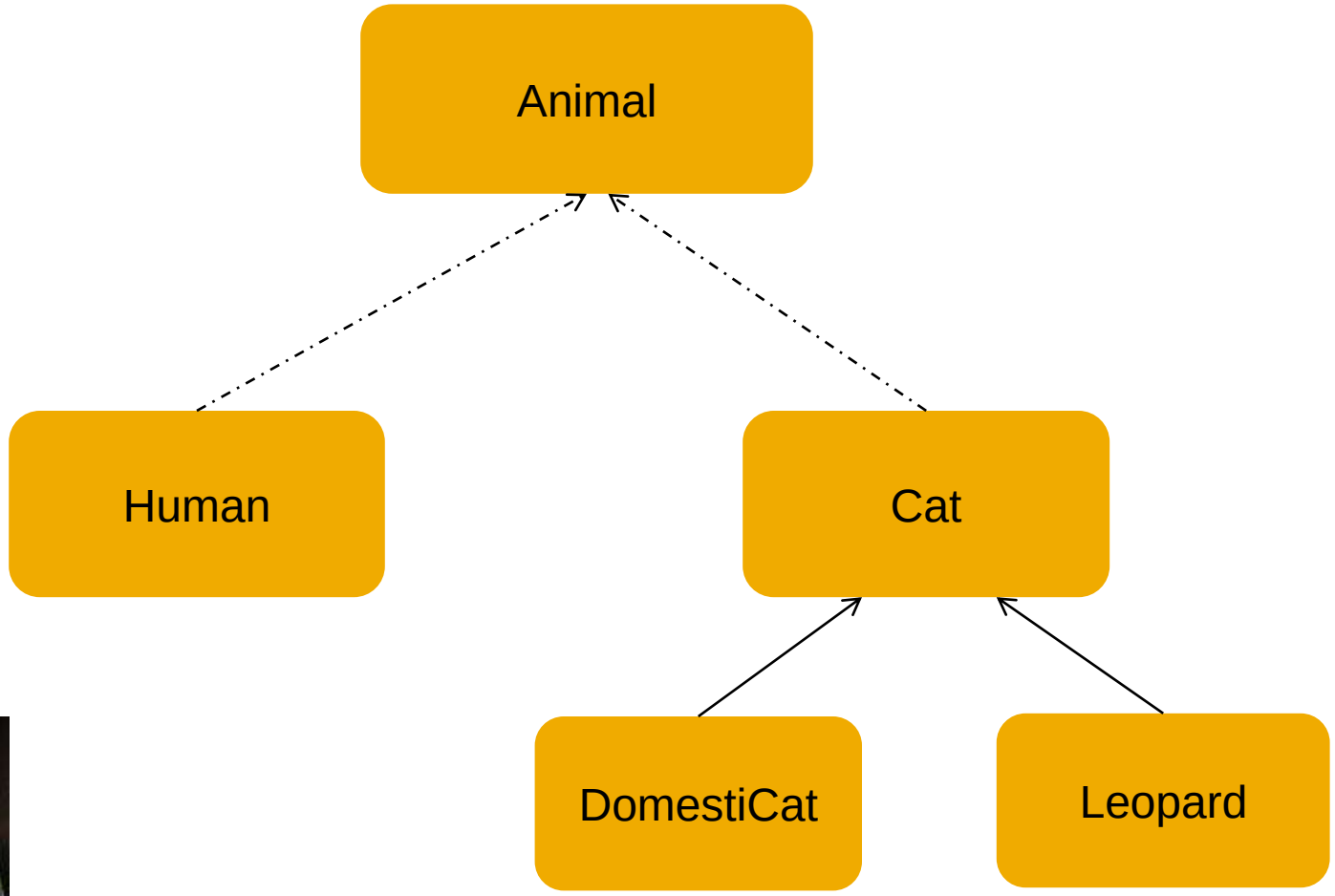
Абстракция (Abstraction)

Основен принцип в ООП!

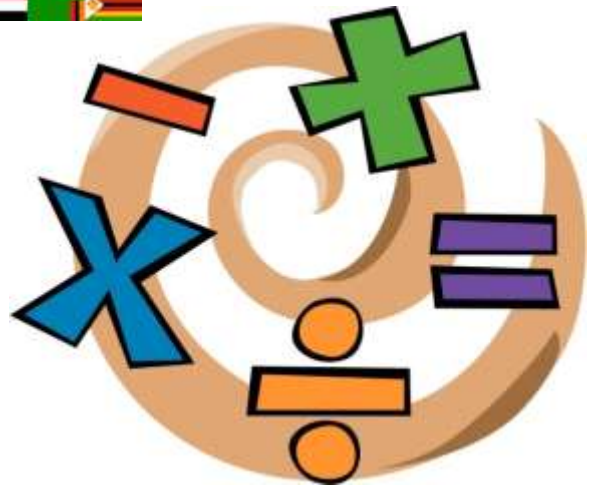
Постига се чрез интерфейси и абстрактни класове

Всяка конкретна имплементация на поведение е скрита в своя обект, за външния свят е видимо само поведението

Demo



Enum



Enum тип

Специален тип, чрез който се дефинира променлива като множество от константи

В Java enum типовете има много повече възможности в сравнение със други ОО езици:

Тялото на enum класа може да съдържа методи и други полета

Всеки **enum** има статичните методи:

- **values()** връща масив съдържаш всички стойности в enum в реда в който са декларирани
- **valueOf(String name)** връща стойността на подадения параметър, който съвпада с идентификатор на декларираните константи в enum.

Enum
Пример

```java
public enum Weekdays {
    MONDAY(1, "MON"),
    TUESDAY(2, "TUE"),
    WEDNESDAY(3, "WED"),
    THURSDAY(4, "THU"),
    FRIDAY(5, "FRI"),
    SATURDAY(6, "SAT"),
    SUNDAY(7, "SUN");

    private final int dayNumber;
    private final String shortName;

    private Weekdays(int dayNumber, String shortName) {
        this.shortName = shortName;
        this.dayNumber = dayNumber;
    }

    public int getDayNumber() {
        return dayNumber;
    }

    public String getShortName() {
        return shortName;
    }

    public String toString() {
        return String.format("%d %s", this.getDayNumber(), this.getShortName());
    }
}
```

```java
public class WeekdaysTest {
    public static void main(String[] args) {
        System.out.println(Weekdays.MONDAY);
        System.out.println(Weekdays.SUNDAY);

        for (Weekdays weekday : Weekdays.values()) {
            System.out.println(weekday);
        }
    }
}
```

```
1 MON
7 SUN
1 MON
2 TUE
3 WED
4 THU
5 FRI
6 SAT
7 SUN
```

equals()

Кога трябва да го предефинираме?
Ако сравняваме два обекта по тяхното физическо представяне, а не по адреса им в паметта

Как да го предефинираме?
С една дума „Умно“
С много думи „Effective Java” Item 8 and Item9

Къде се използва?
В повечето колекции, за да се провери дали такъв елемент вече съществува

hashCode()

Кога трябва да го предефинираме?
Когато сме предефинирали equals()

Как да го предефинираме?
С една дума „Умно“
С много думи „Effective Java” Item9

Къде се използва?
В HashTable, HashMap и HashSet

Забележка при предефинирането на hashCode()
Ако equals връща true, hashCode-а на съответните обекти трябва да е равен
Ако hashCode-а на два обекта е равен, не е задължително equal да връща true

Вложен клас (Inner class)

Позволява управление на достъпа между различните вътрешни класове

Вътрешен клас != Композиция

Независимо от външния клас наследяване

Статични и не-статични вътрешни класове

Вложен клас
Създаване

Статичен вътрешен клас:

```
OutterClass.StaticInnerClass sic =
        new OutterClass.StaticInnerClass();
```

Не-статичен вътрешен клас:

```
OutterClass oc = new OutterClass();
OutterClass.InnerClass ic = os.new InnerClass();
```

Вложен клас
Достъп

Достъп до обекти от вътрешен клас

Статичен вътрешен клас:

```
public static InnerClass getStaticInnerClass() {
        return new StaticInnerClass();
}
```

Не-статичен вътрешен клас:

```
public OutterClass getOutterClass() {
        return new InnerClass();
}
```

Singleton?

Вложен клас
Полза от вътрешен клас имплементиращ интерфейс

```
public class Students {
        ….
        private class ForwardIterator implements Iterator {
                …
        }
        private class ReverseIterator implements Iterator {
                …
        }
        public Iterator getForwardIterator() {
                return new ForwardIterator ();
        }
        public Iterator getReverseOrderIterator() {
                return new ReverseIterator();
        }
        …
}
```

Анонимен клас (Anonymous class)

Анонимният клас е израз

Едновременна декларация и инициализация

За разлика от вложените класовете, анонимните нямат декларация (учудващо :O)

Подходящи за употреба ако имате нужда само веднъж от класа

Анонимен клас
Структура на израза

- Оператор new
- class / interface name
- Списък с аргументи, подадени към конструктора
- Тяло на класа с имплементация на методи

```
public interface Greetings {
        public void wishes();
}
```

```
public class AnonymousEx {

    public static void main(String[] args) {
        Greetings hNY = new Greetings() {
            public void wishes() {
                System.out.println("Happy New Year");
                }
            };
        hNY.wishes();
        }
}
```

```
Happy New Year
```

SOLID Принципи

SOLID

- **S**ingle responsibility principle
- **O**pen closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

SOLID
Single responsibility principle

Смисъл

За промяна в даден клас трябва да има една единствена причина да бъде променен –> една отговорност = една причина за промяна

Пример

Полза

Идентифицирането на отговорности позволява да видим и създадем по-добри абстракции в кода

SOLID
Open closed principle

Смисъл

Софтуерните компоненти като класове, модули, методи трябва да са отворени за разширяване но затворени за модификации

Пример

Ползи

За да се направи дизайна на кода лесен за адаптация и поддръжка се изисква време и въвеждане на правилно ниво на абстракции, за да не се увеличава сложността на кода. Този принцип се прилага в области, които се променят често или има шанс да се променят често.

SOLID

Liskov substitution principle

Смисъл

Подтиповете могат изцяло да се заменят от техните надкласове

Пример

Ползи

Този принцип разширява Open Close Principle в смисъл, че новите класове разширяват базовите но не бива да променят поведението на базовия клас

SOLID

Interface segregation principle

Смисъл

Клиентите не трябва да се принуждават да имплементират интерфейси, които не използват.

Пример

SOLID

Dependency inversion principle

Смисъл

High-level модули не трябва да зависят от low-level модули и двете трябва да зависят от абстракции. Абстракциите не трябва да зависят от детайли, а детайлите да зависят от абстракции.

Пример

Ползи

Когато се прилага този принцип означава, че high level класове не трябва да работят директно с low level класове. В този случай създаване на low level обекти не става чрез оператора new а се използват creational design patterns като Factory Method, Abstract Factory, Prototype.

Използвана литература

Java Tutorial: Basics

Robert C. Martin Clean Code: A Handbook of Agile Software Craftsmanship

Joshua Bloch Effective Java

Thank you

Contact information:

Емилия Бояджиева и Пламен Миленков
emiliya.boyadjieva@sap.com
plamen.milenkov@sap.com