

# Статика в Java

© Составление, Колесов А.А., 2012

# План лекции

---

- Статические методы и поля
- Статические блоки
- Статический импорт
- Класс Class
- Порядок загрузки
- Параметризованные классы и методы (generic)

# Структура класса

```
модификатор class имя_класса {  
    поля  
    методы  
    блоки инициализации  
    конструктор  
    финализатор  
}
```

В Java есть статические поля и статические методы. Для указания того, что поле или метод являются статическими, используется описатель **static** перед именем типа поля или метода.

```
class SomeClass {  
    static int t = 0; // статическое поле  
    . . .  
    public static void f() { // статический метод  
        . . . }  
}
```

# Статические поля

- Поле создается в единственном экземпляре вне зависимости от количества объектов данного класса
- Существуют без создания экземпляра класса
- Статические поля класса создаются в момент первого обращения к данному классу.

Использование:

- С модификатором `final` объявляют константы (например число `PI` в классе `Math`)
- Использовать одну переменную для всех экземпляров класса
- `Singleton`

# Пример со статическим полем

```
public class Proba {  
    int a = 10; // обычное поле  
    static int cnt = 0; // статическое поле  
    public void print() {  
        System.out.println("cnt = " + cnt);  
        System.out.println("a = " + a); }  
    public static void main(String args[]) {  
        Proba obj1 = new Proba();  
        cnt++; // увеличим cnt на 1  
        obj1.print();  
        Proba obj2 = new Proba();  
        cnt++; // увеличим cnt на 1  
        obj2.a = 0;  
        obj1.print();  
        obj2.print(); }  
}
```

# Статические методы

По аналогии со статическими полями, статические методы не привязаны к конкретному объекту класса. При вызове статического метода перед ним можно указать не ссылку, а имя класса:

```
class SomeClass {  
    static int t = 0; // статическое поле  
    . . .  
    public static void f() { // статический метод  
        . . .  
    }  
}  
  
. . .  
SomeClass.f();  
  
. . .
```

# Ограничения на `static` методы:

- Они могут вызывать только другие статические методы.
- Они должны осуществлять доступ только к статическим переменным.
- Они ни коим образом не могут ссылаться на члены типа `this` или `super`.
- Если статический метод определен как `final` -метод, то он не может быть переопределен.
- статические методы не могут быть абстрактными;
- статические методы переопределяются в подклассах только как статические.



# Пример статических методов

- `System.out.println(...);`
- `public static void main(String[] args) {...}`
- Методы класса `Math`

# Задание:

---

Создать класс Automobile, в котором определить поле, которое будет содержать порядковый заводской номер автомобиля. Так же определить статический метод, который будет возвращать это значение.

# Статический блок

За словом `static` следует блок кода, окруженного фигурными скобками. Вы можете использовать статический блок для инициализации статических данных.

```
static List<char> alphabet;  
static {  
    alphabet = new ArrayList<char>();  
    for (char c='a'; c<='z'; c++)  
        alphabet.add(c);  
}
```

Из-за статичности блок запрашивается единожды, когда создается класс. Теперь вам не нужен конструктор, и вы можете пользоваться данными без предшествующего создания класса.

# Какой результат?

```
public class Proba {  
    public static int value;  
    static{  
        System.out.println("static block");  
        value = 50;  
    }  
  
    public Proba(){  
        System.out.println("constructor");  
        value = 100;  
    }  
}  
public class General {  
    public static void main(String[] args) {  
        System.out.println(" General ");  
    }  
}
```

# Задание:

---

В классе определить статическое поле-массив, которое инициализируется в статическом блоке.

# Статический импорт

Для того чтобы получить доступ к статическим членам классов, требуются указать ссылку на класс. К примеру, необходимо указать имя класса Math:

```
double r = Math.cos(Math.PI * theta);
```

Конструкция статического импорта позволяет получить прямой доступ к статическим членам класса, который содержит эти статические члены:

```
import static java.lang.Math.PI;
```

или все целиком:

```
import static java.lang.Math.*;
```

Однажды импортированный статический член может быть использован без указания имени класса:

```
double r =cos(PI * theta);
```

# Когда использовать статический импорт?

- Если иначе вы вынуждены объявлять локальные копии констант
- Постоянное использование статических членов одного класса из одного или двух других классов.
- Чрезмерное использование статического импорта может сделать вашу программу нечитаемой из-за увеличения пространства имен
- Импортирование всех статических методов из класса может быть частично вредно для читабельности. Импортируйте их по-отдельности
- Использованный умело, статический импорт может сделать вашу программу более наглядной

# Пример без статического импорта

```
class Hypot {  
    public static void main(String args[]) {  
        double side1, side2;  
        double hypot;  
        side1 = 3.0;  
        side2 = 4.0;  
        hypot = Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));  
        System.out.println("Given sides of lengths " +  
            side1 + " and " + side2 +  
            " the hypotenuse is " + hypot);  
    }  
}
```



# Задание:

Переписать программу вычисления гипотенузы прямоугольного треугольника, с использованием статического импорта. Использовать два статических метода из встроенного в язык Java класса `Math`, являющегося частью пакета `Java.lang`. Первый метод, `Math.pow()`, возвращает значение, возведенное в определенную степень. Второй — `Math.sqrt()` — возвращает квадратный корень своего аргумента. Имена `sqrt` и `pow` импортировать в область видимости операторами статического импорта.

# Класс Class

Класс с именем `class` представляет характеристики класса, экземпляром которого является объект:

- хранит информацию о том, не является ли объект на самом деле интерфейсом, массивом или примитивным типом.
- каков суперкласс объекта
- имя класса
- какие в нем конструкторы, поля, методы и вложенные классы

# Класс Class

В классе `Class` нет конструкторов, экземпляр этого класса создается исполняющей системой Java во время загрузки класса и предоставляется методом `getClass()` класса:

```
String s = "Это строка";
```

```
Class c = s.getClass();
```

Статический метод `forName(string class)` возвращает объект класса `Class` для класса, указанного в аргументе:

```
Class cl = Class.forName("Java.lang.String"); (устаревший)
```

```
или Class c2 = Java.lang.String.class;
```

# Пример:

```
import java.lang.reflect.*;
class ClassTest{
    public static void main(String[] args){
        Class c = null, c1 = null, c2 = null;
        Field[] fld = null;
        String s = "Some string";
        c = s.getClass();
        cl = Class.forName("Java.lang.String"); // Старый стиль
        c2 = Java.lang.String.class;           // Новый стиль
        if (!c1.isPrimitive())
            fid = cl.getDeclaredFields();       // Все поля класса String
        System.out.println("Class    c: " + c);
        System.out.println("Class    cl: " + cl);
        System.out.println("Class    c2: " + c2);
        System.out.println("Superclass c: " + c.getSuperclass()); //получить суперкласс
        System.out.println("Package   c: " + c.getPackage()); //получить пакет
        for(int i = 0; i < fid.length; i++)
            System.out.println(fld[i]);
    }
}
```

# Задание:

- Получить Class объекта и с помощью логических методов `isArray()`, `isInterface()`, `isPrimitive()` уточнить, не является ли объект массивом, интерфейсом или примитивным типом.
- Если объект ссылочного типа, извлечь сведения о вложенных классах, конструкторах, методах и полях методами `getDeclaredClasses()`, `getDeclaredConstructors()`, `getDeclaredMethods()`, `getDeclaredFields()`, в виде массива классов, соответственно, `Class`, `Constructor`, `Method`, `Field`. Последние три класса расположены в пакете `java.lang.reflect` и содержат сведения о конструкторах, полях и методах аналогично тому, как класс `Class` хранит сведения о классах.
- Получить данные, с помощью методов `getClasses()`, `getConstructors()`, `getInterfaces()`, `getMethods()`, `getFields()` которые возвращают такие же массивы, но не всех, а только открытых членов класса.

# Порядок загрузки

- Статические поля инициализируются во время загрузки класса.
- Инициализация статического блока происходит во время загрузки класса.
- Инициализация статических полей в месте объявления и статические блоки выполняются в порядке их объявления в классе.
- В отличие от статических полей класса, поля объекта инициализируются во время конструирования экземпляра класса

# Порядок инициализации объекта

- инициализация полей в месте объявления и в инициализационном блоке происходит до инициализации в конструкторе
- инициализации полей в месте объявления и в инициализационных блоках выполняются в порядке их объявления в классе
- инициализация полей базового класса происходит полностью до инициализации производного класса, т.е. сначала выполняются все инициализаторы базового класса, а потом все инициализаторы производного класса.

# Параметризованные классы и методы (Generic)

- Параметризованные (generic) классы и методы, позволяют использовать более гибкую и в то же время достаточно строгую типизацию, что особенно важно при работе с коллекциями.
- Параметризация позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр.



# Пример generic-класса с двумя параметрами:

```
public class Subject <T1, T2> {  
    private T1 name;  
    private T2 id;  
    public Subject(T2 ids, T1 names) {  
        id = ids;  
        name = names;  
    }  
}
```

Объект класса **Subject** можно создать, например, следующим образом:

```
Subject<String, Integer> sub = new Subject<String,Integer>();  
char ch[] = {'J','a','v','a'};  
Subject<char[], Double> sub2 = new Subject<char[],Double>(ch, 71.5 );
```

**T1, T2** – фиктивные объектные типы, которые используются при объявлении членов класса и обрабатываемых данных. При создании объекта компилятор заменит все фиктивные типы на реальные и создаст соответствующий им объект.

# Пример параметризованного класса с конструктором и методами

```
public class Optional <T> {  
    private T value;  
    public Optional(T value) {  
        this.value = value;  
    }  
    public T getValue() {  
        return value;  
    }  
    public void setValue(T val) {  
        value = val;  
    }  
    public String toString() {  
        if (value == null) return null;  
        return value.getClass().getName() + " " + value;  
    }  
}
```

# Использование класса Option

```
public class Runner {  
    public static void main(String[] args) {  
        Optional<Integer> ob1 = new Optional<Integer>(); //параметризация типом Integer  
        ob1.setValue(1); //попробуйте задать значение "2" ob1.setValue("2");  
        int v1 = ob1.getValue();  
        System.out.println(v1);  
        Optional<String> ob2 = new Optional<String>("Java"); //параметризация типом String  
        String v2 = ob2.getValue();  
        System.out.println(v2);  
        //попробуйте присвоить ob1 = ob2;  
        Optional ob3 = new Optional(); //параметризация по умолчанию – Object  
        System.out.println(ob3.getValue());  
        ob3.setValue("Java SE 6");  
        System.out.println(ob3.toString()); /* выводится тип объекта, а не тип параметризации */  
        ob3.setValue(71);  
        System.out.println(ob3.toString());  
        ob3.setValue(null);  
    }  
}
```

# Расширение возможностей

- Объявление generic-типа в виде **<T>**, несмотря на возможность использовать любой тип в качестве параметра, ограничивает область применения разрабатываемого класса. Переменные такого типа могут вызывать только методы класса **Object**. Доступ к другим методам ограничивает компилятор.
- Чтобы расширить возможности параметризованных членов класса, можно ввести ограничения на используемые типы при помощи следующего объявления класса:

```
public class OptionalExt <T extends Тип> {  
    private T value;  
}
```

В качестве типа **T** разрешено применять только классы, являющиеся наследниками (суперклассами) класса **Тип**, и соответственно появляется возможность вызова методов ограничивающих типов.

Бывает необходимость в метод параметризованного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом. В этом случае при определении метода следует применить метасимвол ?. Метасимвол также может использоваться с ограничением extends для передаваемого типа.

```
public class Mark<T extends Number> {  
    public T mark;  
    public Mark(T value) {  
        mark = value;  
    }  
    public T getMark() {  
        return mark;  
    }  
    public int roundMark() {  
        return Math.round(mark.floatValue());  
    }  
    /* вместо */ // public boolean sameAny(Mark<T> ob) {  
    public boolean sameAny(Mark<?> ob) {  
        return roundMark() == ob.roundMark();  
    }  
    public boolean same(Mark<T> ob) {  
        return getMark() == ob.getMark();  
    }  
}
```

```
public class Runner {  
    public static void main(String[] args) {  
        // попробуйme Mark<String> ms = new Mark<String>("7");  
        Mark<Double> md = new Mark<Double>(71.4D); // 71.5d  
        System.out.println(md.sameAny(md));  
        Mark<Integer> mi = new Mark<Integer>(71);  
        System.out.println(md.sameAny(mi));  
        // попробуйme md.same(mi);  
        System.out.println(md.roundMark());  
    }  
}
```

# Ограничения generic типов

- Невозможно выполнить явный вызов конструктора generic-типа

```
class Optional <T> {  
    T value = new T();  
}
```

- generic-поля не могут быть статическими
- Статические методы не могут иметь generic-параметры или обращаться к generic-полям

```
class Failed <T1, T2> {  
    static T1 value;  
    T2 id;  
    static T1 getValue() {  
        return value;  
    }  
    static void use() {  
        System.out.print(id);  
    }  
}
```

Спасибо за внимание!