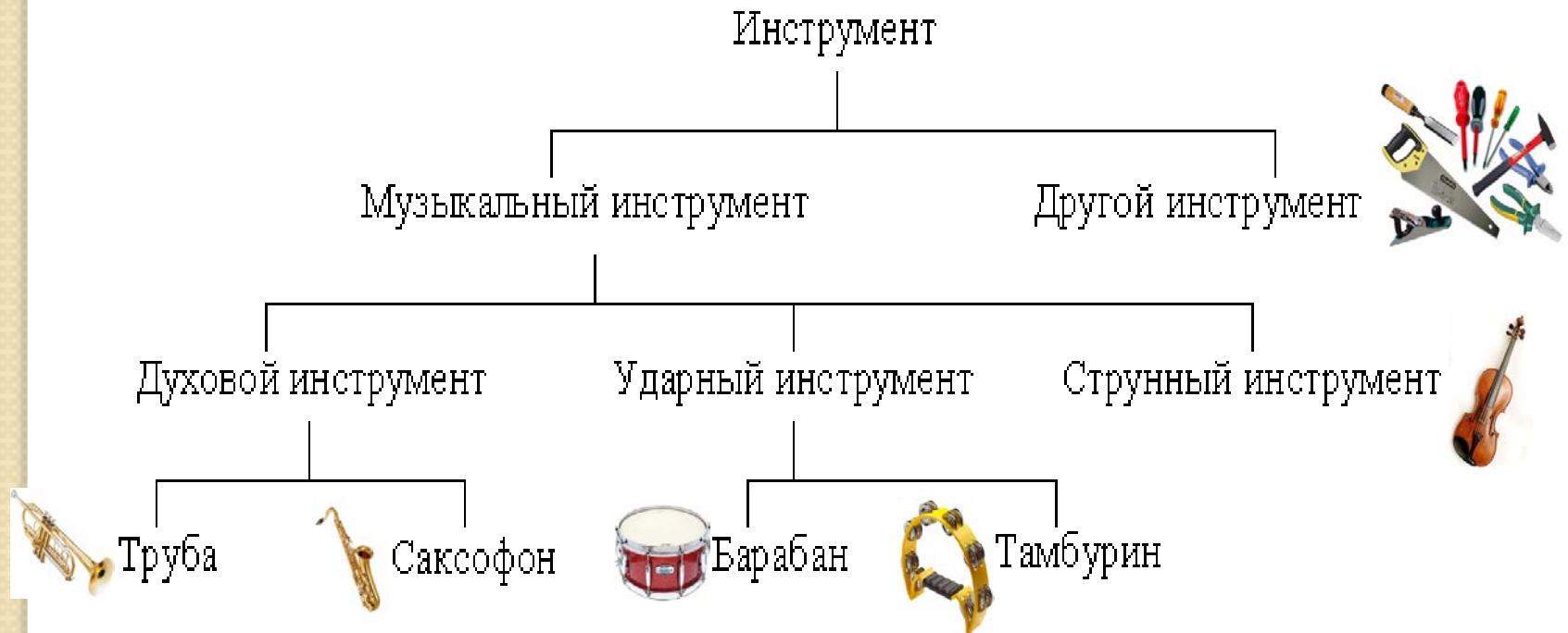


Введение в ООП

- ❑ **Объектно-ориентированное программирование (ООП)** — это подход к разработке программного обеспечения, основанный на представлении программы в виде совокупности *объектов*, каждый из которых является экземпляром определённого *класса*, которые образуют иерархические конструкции:
 - такой подход отражает реальный (повседневный) мир.



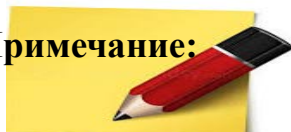
Введение в ООП



Например,

- Велосипед — это объект, который был построен согласно чертежам;
- Чертежи играют роль классов, т.е. классы — это некоторые описания, схемы, чертежи по которым создаются объекты;
- Классы имеют свои функции, которые называются методами класса (передвижение велосипеда осуществляется за счёт вращения педалей, т.е. механизм вращения педалей — это метод класса);
- Каждый велосипед имеет свой цвет, вес, различные составляющие — всё это свойства (у каждого созданного объекта свойства могут различаться);

Примечание:



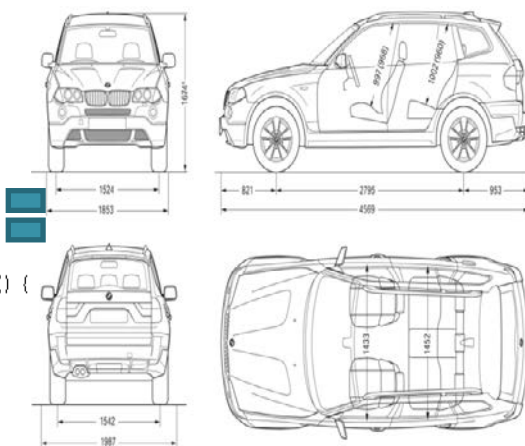
Имея один класс, можно создать неограниченно количество объектов (велосипедов), каждый из которых будет обладать одинаковым набором методов: механизм вращения педалей, колёс, срабатывания системы торможения.

С точки зрения ООП

- ❑ **Класс** – определяет структуру и поведение некоторого набора элементов предметной области, для которой разрабатывается программная модель.
 - **Класс** – это абстрактный тип данных, который определяет форму и поведение объекта.
 - ✓ *Свойства (данные, атрибуты) объекта называются полями.*
 - ✓ *Функции, определяющие поведение объекта (описывающие операции, выполняемые над данными) называются методами.*

Описание класса

```
public class Car {  
    private String model;  
    private int maxSpeed;  
    private int year;  
    //...  
    public int getMaxSpeed() {  
        return maxSpeed;  
    }  
    //...  
}
```

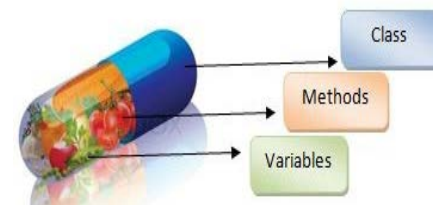


Объекты класса

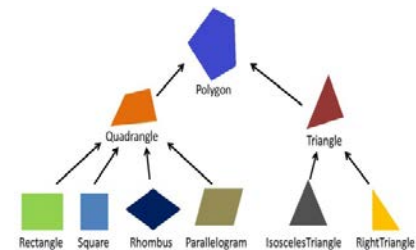


- ❑ В ООП существует три основных принципа построения классов:

- **Инкапсуляция** — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.



- **Наследование** — это свойство, позволяющее создать новый класс на основе уже существующего, при этом характеристики класса родителя присваиваются классу-потомку.



- **Полиморфизм** — свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.





Описание классов

Описание классов

- ❑ Описание класса - определяет новое имя (индикатор) ссылочного типа.

```
[Модификатор_Класса] class Идентификатор  
[Параметры_Типа] [Супер_Класс] [Супер_Интерфейс]  
{ Тело_класса }
```

- **Модификатор_Класса** – устанавливает правила использования класса (область видимости);
- **Идентификатор** – имя класса (т.е. типа данных);
- **Параметры_Типа** – определяет обобщенность, используемых данных класса;
- **Супер_Класс** – определяет родителя;
- **Супер_Интерфейс** – устанавливает некоторое поведение класса;
- **Тело_класса** – определяет поля и методы класса.

Примечание:



Компоненты, заключенные в квадратные скобки являются необязательными.

Описание классов

Пример 1,

Пакет

```
package com.knu.oop;
```

Модификатор, который определяет класс общедоступным

```
public class Car {
```

Идентификатор

```
private String model;
```

```
private int maxSpeed;
```

```
private int year;
```

```
private int speed;
```

Поля

```
public Car(String model, int year) {
```

```
    this.model = model;
```

```
    this.year = year;
```

Конструктор

```
}
```

Примечание:

 Поля – это объявление или описание переменной.

```
public int getMaxSpeed() {
```

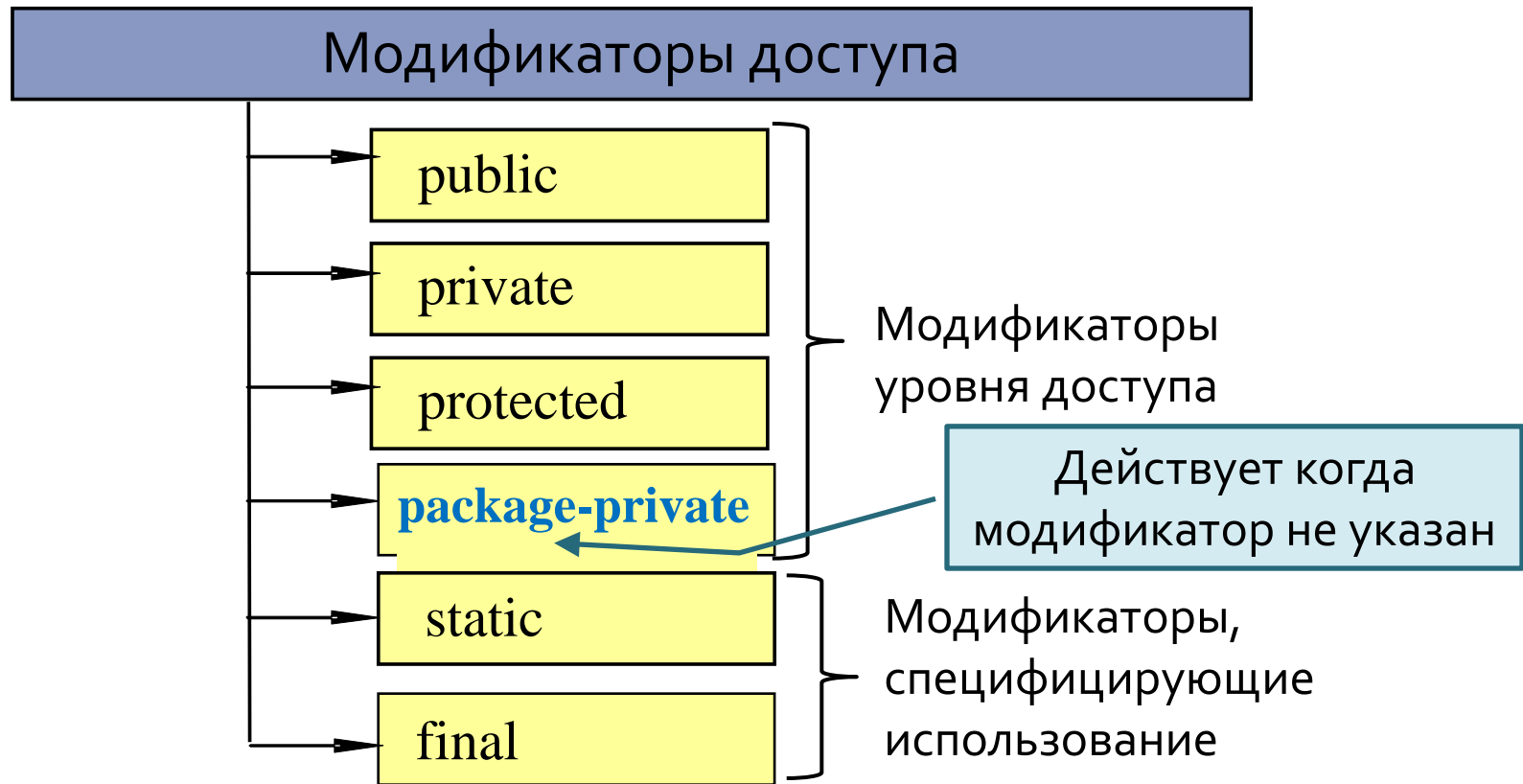
```
    return maxSpeed;
```

Метод

```
}
```

```
}
```


- ❑ **Модификаторы доступа** определяют как другие классы могут использовать этот класс, определенное поле и вызвать определенный метод.



- ❑ Есть два уровня контроля доступа:
 - верхний уровень: открытый (**public**) или пакетный (**package-private** - неявный модификатор);
 - уровень членов – открытый (**public**), закрытый (**private**), защищенный (**protected**) или пакетный (**package-private** - неявный модификатор).

- ❑ Модификатор **public** (*открытый, общедоступный*) может использоваться перед описанием класса, метода, конструктора и поля:
 - для класса определяет доступ из любой Java-программы;
 - для члена класса – доступ извне через объект, имя класса или из любого метода этого класса.

Пример 2,

```
public class Car {  
    //...  
    public int speed;  
    //...  
    public int getSpeed() {  
        return speed;  
    }  
    //...  
    public void testModifier() {  
        getSpeed();  
    }  
}
```

Доступ к методу
этого же класса

Описание классов

- ❑ Если модификатор уровня доступа не указан (**package-private**), то по умолчанию класс или член класса будет видимым и доступным только для классов и подклассов в этом же пакете.

Пример 3,

```
class Car {  
    //...  
    int speed;  
    //...  
    int getSpeed() {  
        return speed;  
    }  
    //...  
    public void testModifier() {  
        getSpeed();  
    }  
}
```

Нет явно указанного модификатора уровня доступа

Доступ к методу этого же класса, т.к. они находятся в одной области видимости (тело класса)

Описание классов

- ❑ Модификатор **private** (закрытый) может использоваться перед описанием метода, конструктора и поля, а также вложенного класса/интерфейса:
 - определяет, что доступ к члену класса возможен только из методов этого класса.

Пример 4,

```
public class Car {  
    //...  
    private int speed;  
    //...  
    private int getSpeed()  
        return speed;  
}  
    //...  
    public void testModifier() {  
        getSpeed();  
    }  
}
```

Доступ к закрытому методу этого же класса, т.к. они находятся в одной области видимости (тело класса)

Описание классов

- ❑ Модификатор **protected** (защищенный) может использоваться перед описанием метода, конструктора и поля, а также вложенного класса/интерфейса:
 - определяет, что доступ к члену класса возможен только из методов этого класса или его подклассов, даже если они объявлены в другом пакете.

Пример 5,

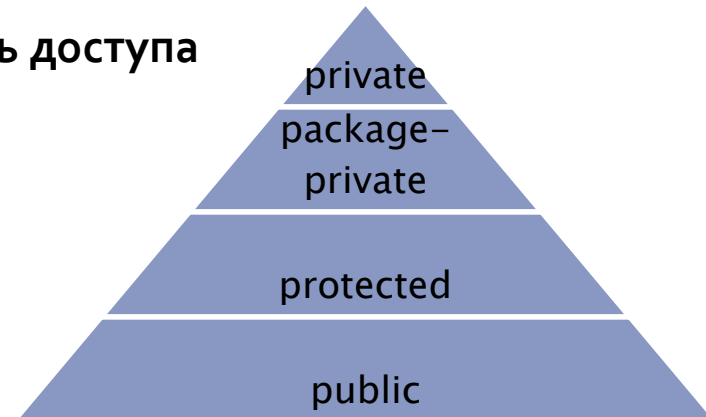
```
public class Car {  
    //...  
    protected int speed;  
    //...  
    protected int getSpeed() {  
        return speed;  
    }  
    //...  
    public void testModifier() {  
        getSpeed();  
    }  
}
```

Доступ к защищенному методу этого же класса, т.к. они находятся в одной области видимости (тело класса)

Описание классов

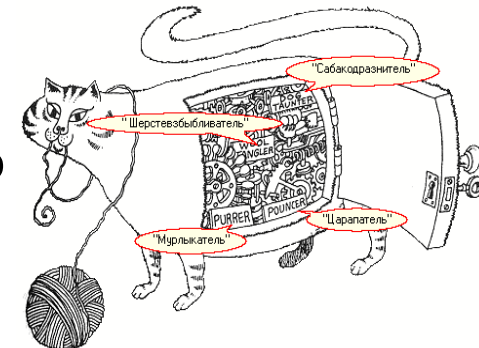
Видимость	Public	Protected	Package-private	Private
Из этого же класса	+	+	+	+
Из любого класса в этом же пакете	+	+	+	-
Из подкласса в этом же пакете	+	+	+	-
Из подкласса вне этого пакета	+	+	-	-
Из любого не подкласса вне этого пакета	+	-	-	-

Уровень доступа



ИНКАПСУЛЯЦИЯ

- ❑ Использование модификаторов уровня членов класса позволяет управлять доступом к данным класса и его реализации, тем самым обеспечивая различный уровень инкапсуляции класса:
 - защитный барьер, который предотвращает произвольный доступ к коду и данным с помощью кода, определенного вне этого класса.
- ❑ Практически это означает наличие двух частей в классе: **интерфейса** и **реализации**:
 - *в интерфейсной части* собрано все, что касается взаимодействия данного объекта с любыми другими объектами;
 - *реализация* скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов.



Рекомендации по инкапсуляции

- Используйте самый ограничительный уровень доступа, который имеет смысл для конкретного члена (т.е. используйте **private**, если нет *хорошей* причины не делать этого).
- Избегайте открытых полей, кроме для постоянных (открытые поля, как правило, отправляют к конкретной реализации и ограничивают гибкость в изменении кода).
- Для доступа к закрытым полям класса используются специальные методы доступа, называемые геттерами и сеттерами (для каждого поля).
 - ✓ Геттеры – это методы для получения значения поля, имя метода начинается с глагола **get**, а дальше следует имя поля в стиле CamelCase;
 - ✓ Сеттеры – это методы для установки значения поля, имя метода начинается с глагола **set**, а дальше следует имя поля в стиле CamelCase.



Пример 6,

```
public class Student {  
    private String firstName;  
    public String getFirstName() {  
        return firstName;  
    }  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
    // .....  
}
```

Геттер

Сеттер

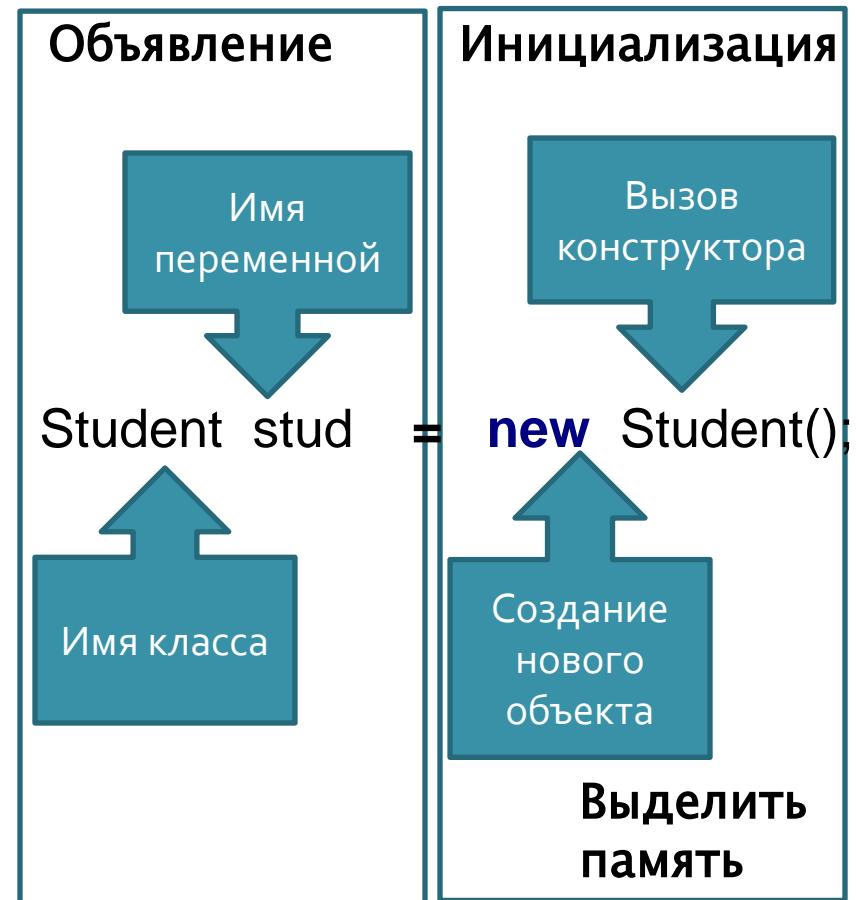


Создание объектов

Создание объектов

Создание объектов состоит из трех частей:

- ✓ Объявление: объявления переменных, которые связывают имя переменной с типом объекта (создается ссылочная переменная);
- ✓ Выделение памяти: ключевое слово **new** – это оператор Java, который создает объект;
- ✓ Инициализация: за оператором **new** следует вызов конструктора, который инициализирует новый объект.



Создание объектов

- При использовании ключевого слова **new** система выделяет место для нового объекта в «Куче», а затем вызывает конструктор для инициализации объекта.
- Переменные и методы экземпляра (объекта) доступны через созданный объект.

```
public static void main(String[] arg) {  
    Student myStud1 = new Student();  
    Student myStud2 = new Student();  
    Student myStud3 = new Student();  
    String name = myStud1.getFirstName();  
    System.out.println(name);  
}
```

Переменная ссылочного типа

Вызов метода на экземпляре

«Куча»



Создание объектов

- ❑ Ключевое слово **this** – это ссылка на текущий объект внутри класса, т.е. объект, чей метод или конструктор выполняются:
 - используется для обращения к любому члену текущего объекта изнутри метода экземпляра или конструктора.

Например,

Наиболее распространенная причина для использования этого ключевого слова - различать поля и локальные переменные/параметры, если они имеют одинаковые имена

```
public class Car {  
    private String model;  
    private int maxSpeed;  
    private int year;  
    private int speed;  
    public void setSpeed(int speed) {  
        this.speed = speed;  
    }  
}
```



Конструкторы

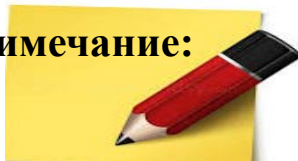
Конструкторы

- ❑ **Конструкторы** – это специальные методы, определенные в классе, которые инициализируют и возвращают объект класса, в котором они определены.

Особенности конструкторов

- Обозначение конструктора совпадает с именем класса;
- Конструктор не имеет явно указанного типа возвращаемого значения;
- Конструктор вызывается только через оператор **new**.

Примечание:



Если при описании конструктора указать тип возвращаемого значения, то Java воспримет его как метод класса, а не конструктор.

Например,

```
void Student(String name) { }
```


Конструкторы

Пример 7,

```
public class Student {  
    private String firstName;  
    public Student(String firstName) {  
        this.firstName = firstName;  
    }  
    public String getFirstName() {  
        return firstName;  
    }  
}
```

Конструктор
с параметром

Продолжение примера 7,

```
public class Main {  
    public static void main(String[] arg) {  
        Student myStud1 = new Student("Peter");  
        Student myStud2 = new Student("Alex");  
        Student myStud3 = new Student("Nina");  
        System.out.println(myStud1.getFirstName());  
        System.out.println(myStud2.getFirstName());  
        System.out.println(myStud3.getFirstName());  
    }  
}
```

Вывод в консоли:

Peter

Alex

Nina

ТИПЫ КОНСТРУКТОРОВ

- без параметров (по умолчанию);
- с параметрами.
- ❑ Если в классе нет явного описания конструктора, то Java создает конструктор по умолчанию, который при вызове инициализирует все переменные экземпляра нулевыми значениями своего типа.

```
public class Student {  
    private String firstName;  
    public String getFirstName() {  
        return firstName;  
    }  
}
```

Будет конструктор типа: **public** Student() { }

Пример 8: использование конструктора по умолчанию:

```
public class Main {  
    public static void main(String[] arg) {  
        Student myStud1 = new Student();  
        System.out.println(myStud1.getFirstName());  
    }  
}
```

Вывод в консоли:
null

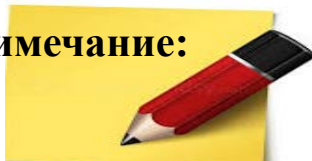
Конструкторы

- ❑ Если в классе явно описан конструктор с параметрами, то Java сама не создает конструктор по умолчанию.

```
public class Main {  
    public static void main(String[] arg) {  
        Student myStud1 = new Student();  
        System.out.println(myStud1.getFirstName());  
    }  
}
```

Ошибка
компиляции

Примечание:



Для его применения требуется явно описать конструктор без аргументов!

Конструкторы

Пример 9: Конструкторы можно перегружать.

```
public class Student {  
    private String firstName, lastName;
```

```
    public Student() { }
```

Конструктор по
умолчанию

```
    public Student(String firstName) {  
        this.firstName = firstName;  
    }
```

Конструктор с одним
параметром

```
    public Student(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }
```

Конструктор с двумя
параметрами

```
    // геттеры и сеттеры  
}  
// .....
```

Продолжение примера 9:

```
public class Main {  
    public static void main(String[] arg) {  
        Student myStud1 = new Student();  
        Student myStud2 = new Student("Alex");  
        Student myStud3 = new Student("Alex", "Petrov");  
        System.out.println(myStud1.getFirstName() + " "  
                             + myStud1.getLastName());  
        System.out.println(myStud2.getFirstName() + " "  
                             + myStud2.getLastName());  
        System.out.println(myStud3.getFirstName() + " "  
                             + myStud3.getLastName());  
    }  
}
```

Вывод в консоли:

```
null null  
Alex null  
Alex Petrov
```

Конструкторы

- ❑ Конструктор может иметь любой модификатор доступа:
 - если конструктор в классе имеет доступ **private**, то это означает, что конструктор не может быть вызван из любого места за пределами класса.

Пример 10,

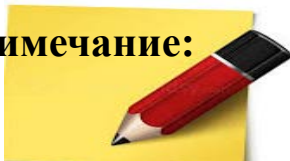
```
public class Student {  
    private String firstName, lastName;  
    public Student() { }  
    public Student(String firstName) {  
        this.firstName = firstName;  
    }  
    private Student(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}  
// .....
```


Продолжение примера 10:

```
public class Main {  
    public static void main(String[] arg) {  
        Student myStud1 = new Student();  
        Student myStud2 = new Student("Alex");  
        Student myStud3 = new Student("Alex", "Petrov");  
    }  
}
```

Ошибка компиляции:
закрытый конструктор не
может быть вызван вне тела
класса

Примечание:

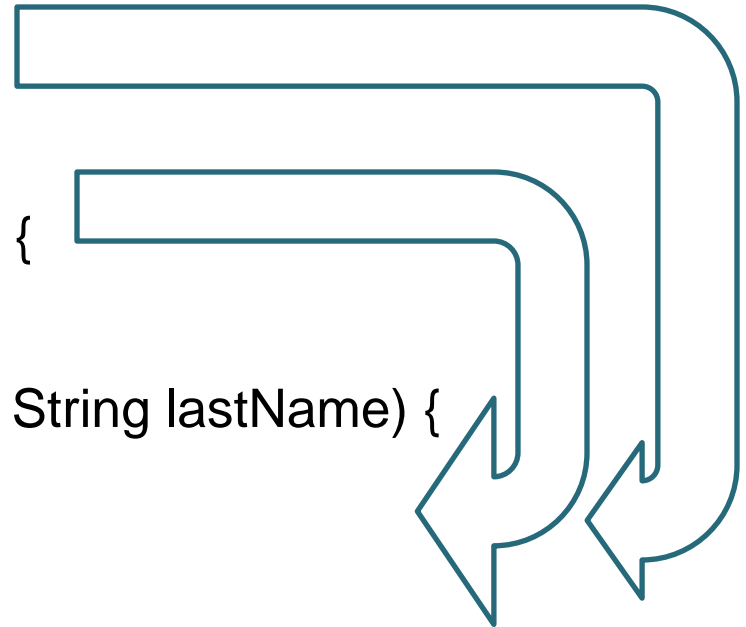


Закрытые конструкторы могут вызываться из других конструкторов или из статических методов этого же класса.

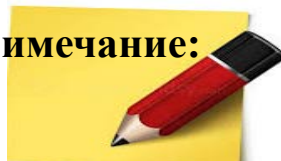
Конструкторы

- Любой конструктор класса в своем теле может вызывать другой конструктор этого класса с использованием **this**.

```
public class Student {  
    private String firstName, lastName;  
    public Student() {  
        this("Peter", "Petrov");  
        System.out.println("Init");  
    }  
    public Student(String firstName) {  
        this(firstName, "Petrov");  
    }  
    public Student(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}
```



Примечание:




Обращение к другому конструктору этого же класса должно быть первым оператором тела конструктора

Конструкторы

Пример 11, не верное обращение к конструктору из другого конструктора.

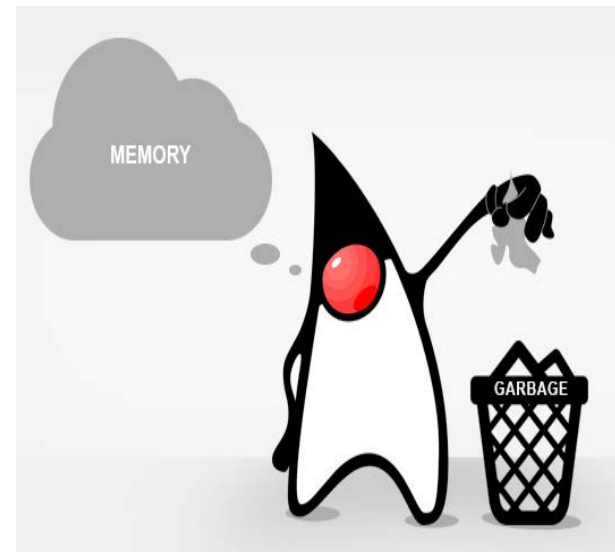
```
public class Student {  
    private String firstName, lastName;  
    public Student() {  
        System.out.println("Init");  
        this("Peter", "Petrov");  
    }  
    public Student(String firstName) {  
        this(firstName, "Petrov");  
    }  
    public Student(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}
```



Вызов конструктора должен быть первым оператором в теле другого конструктора

Уничтожение объектов

- ❑ В Java нет специальных методов и операторов (деструкторов) для уничтожения объектов класса.
- ❑ Уничтожение неиспользуемых объектов осуществляется автоматически «сборщиком мусора» (garbage collector) – специальным компонентом JVM.
- ❑ Объект удаляется, когда в программе на него нет ни одного обращения (ссылки).

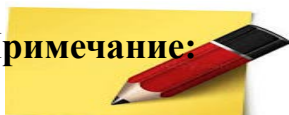


Модификатор `final`

Модификатор `final`

- ❑ Если используется перед описанием *поля*, то этот член класса не может изменить своего значения после инициализации.
- ❑ Если используется перед описанием *метода*, то такой метод нельзя переопределить (поэтому компилятор Java может встроить код *final-метода* в точку его вызова, чтобы устранить затраты времени на обращение к нему).
- ❑ Если используется перед описанием *класса*, то такой класс не может наследоваться.

Примечание:



Потокобезопасность
Производительность

finalьный


Модификатор final

Пример 12:

```
public class TestFinal {  
    private final String mName;  
    public TestFinal(String mName) {  
        this.mName = mName;  
    }  
    public String getmName() {  
        return mName;  
    }  
    public void setmName(String name) {  
        this.mName = mName;  
    }  
}
```

Ошибка
компиляции

Cannot assign a value to final variable 'mName'

Примечание:



Поле типа *final* может быть проинициализировано либо при объявлении, либо в конструкторе, либо в логическом блоке

Модификатор `static`

Модификатор *static*

- ❑ Модификатор *static* определяет общие члены класса для всех экземпляров (объектов) этого класса, т.е. описание поля или метода со **static** говорит, что они принадлежат классу, а не экземпляру класса.
- ❑ Их еще называют:
 - поле класса;
 - метод класса.
- ❑ Каждый экземпляр класса разделяет *поле класса*, которое находится в одном определенном месте в памяти.
- ❑ Любой объект может изменить значение *поля класса*, но полями класса также можно управлять без создания экземпляра класса.

Модификатор static

Пример 13:

```
class StaticDemo {  
    static int a = 42;  
    static int b = 99;  
    static void callme() {  
        System.out.println("a = " + a);  
    }  
}  
  
public class Demo {  
    public static void main(String[] s) {  
        StaticDemo.callme();  
        System.out.print("b = " + StaticDemo.b);  
    }  
}
```

Поля
класса

Метод
класса

Вызов метода
класса без
создания
экземпляра
класса

Обращение к
полю класса
напрямую

Модификатор static

Когда нужны статические поля?

Пример 14: Как вариант: нужно вести учет объектов некоторого типа.

```
public class Student {  
    private int numOfStudents;  
    //...  
    public Student() {  
        numOfStudents++;  
    }  
    //...  
    public int getNumOfStudents() {  
        return numOfStudents;  
    }  
}
```

Модификатор static

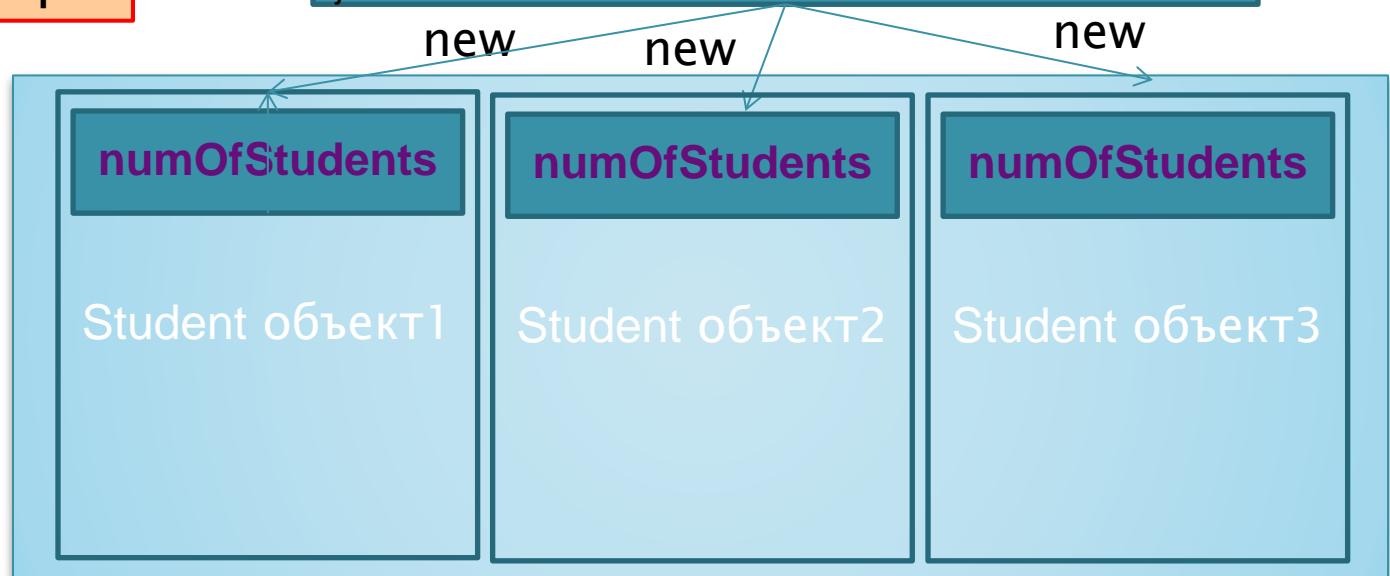
```
public static void main(String[] arg) {  
    Student myStud1 = new Student();  
    Student myStud2 = new Student();  
    Student myStud3 = new Student();  
    System.out.println(myStud1.getNumOfStudents());  
}
```

Вывод в консоли:

1

Поле
экземпляра

```
public class Student {  
    private int numOfStudents;  
    ...  
}
```



Модификатор static

Изменение примера 14: правильно описать класс следующим образом:

Вывод в консоли:

3

```
public class Student {  
    private static int numOfStudents;  
    //...  
    public Student() {  
        numOfStudents++;  
    }  
    //...  
    public static int getNumOfStudents() {  
        return numOfStudents;  
    }  
}
```

Методы, которые используют только статические данные класса лучше всего описывать тоже как статические без создания экземпляра класса

Модификатор static

Поле класса

```
public class Student {  
    private static int numOfStudents;  
    ...  
}
```

numOfStudents

new

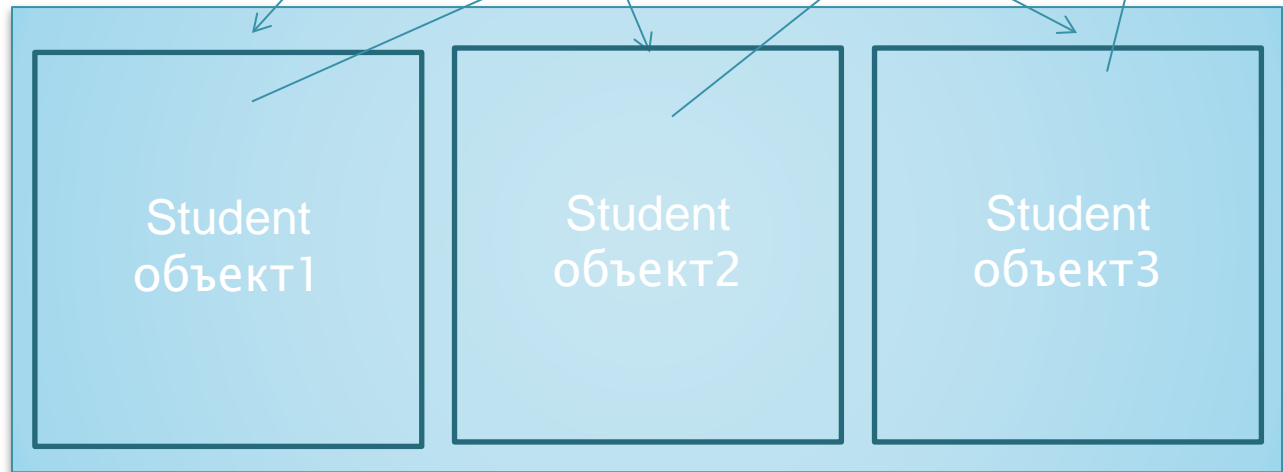
new

new

Student
объект1

Student
объект2

Student
объект3



Поля класса (статические)

- ❑ Создаются при первом обращении к классу;
- ❑ Создаются в единственном числе;
- ❑ Существуют независимо от экземпляров класса;
- ❑ Допускают обращение до создания экземпляра класса.

Пример 15,

```
public class TestStatic {  
    public static int x;  
    public static int y;  
    public static int LengthVector() {  
        return (int)Math.sqrt(x*x + y*y);  
    }  
}
```

Модификатор static

Продолжение примера 15,

```
public static void main(String[] args) {  
    TestStatic.x = 3;  
    TestStatic.y = 4;  
    System.out.println("length = " +  
        TestStatic.LengthVector());  
}
```

Вывод в консоли:

length = 5

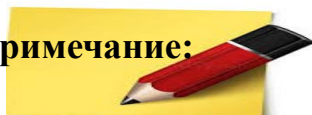
Условия использование статических методов

- ❑ Когда методу не нужен доступ к информации о состоянии объекта, поскольку все необходимые параметры задаются явно.
- ❑ Когда методу нужен доступ только к статическим полям класса.

Пример 16:

```
public class MyMath {  
    public static int add(int x, int y) {  
        return x + y;  
    }  
    public static int multiply(int x, int y) {  
        return x * y;  
    }  
}
```

Примечание:



Методы для своей работы все получают в параметрах, а значит являются вспомогательными и описываются как статические

Ограничения для статических методов

- ❑ статические методы могут вызывать напрямую только статические методы;
- ❑ статические методы могут обращаться напрямую только к статическим данным;
- ❑ на статические методы нельзя ссылаться через ссылки *this* и *super*;
- ❑ статические методы могут перегружаться нестатическими и наоборот.

Модификатор static

Пример 17:

```
public class Car {  
    public static final int NUM_OF_WHEELS = 4;  
    private int mileage;  
    //...  
    public static void testCar(){  
        int numOfBolt = NUM_OF_WHEELS * 5; // OK!  
        int numOfDiag = mileage/4000;  
    }  
}
```

Ошибка компиляции: нельзя
на прямую обращаться к полю
экземпляра

Модификатор static

Пример 18:

Обычно static и final
используются совместно

```
public class Car {  
    public static final int NUM_OF_WHEELS = 4;  
    private int mileage;  
    //...  
    public static void testCar(){  
        int numOfBolt = NUM_OF_WHEELS * 5; // OK!  
        int numOfDiag = this.mileage/4000;  
    }  
}
```

Ошибка компиляции: нельзя
использовать ссылку на
объект в статическом
контексте