# Лекция 3 ООП в Java

# План

1. Геттеры и сеттеры
2. Переопределение методов
3. Абстрактные классы Java
4. Ключевое слово Java final
5. Статические переменные Java
6. Статические методы Java
7. Статический блок Java
8. Java static import
9. Класс Object и методы класса Object
10. Интерфейс

# Геттеры и сеттеры

Если переменная имеет уровень доступа private, к ней невозможно обратиться извне класса, в котором она объявлена. Но все равно необходим способ обращения к private переменным из другого класса, иначе такие изолированные переменные не будут иметь смысла. Это достигается с помощью объявления специальных public методов. Методы, которые возвращают значение переменных, называются *геттеры*. Методы, которые изменяют значение свойств, называются *сеттеры*.

# Геттеры и сеттеры

Существуют правила объявления таких методов, рассмотрим их:

1. Если свойство НЕ типа boolean, префикс *геттера* должно быть get. Например: getName() это корректное имя *геттера* для переменной name.
2. Если свойство типа boolean, префикс имени *геттера* может быть get или is. Например, getPrinted() или isPrinted() оба являются корректными именами для переменных типа boolean.
3. Имя *сеттера* должно начинаться с префикса set. Например, setName() корректное имя для переменной name.

# Геттеры и сеттеры

4. Для создания имени *геттера* или *сеттера*, первая буква свойства должна быть изменена на большую и прибавлена к соответствующему префиксу (set, get или is).
5. Для создания имени *геттера* или *сеттера*, первая буква свойства должна быть изменена на большую и прибавлена к соответствующему префиксу (set, get или is). *Сеттер* должен быть public, возвращать void тип и иметь параметр соответствующий типу переменной. Например:

```
public void setAge(int age) {
    this.age = age;
}
```

# Геттеры и сеттеры

6. Геттер метод должен быть public, не иметь параметров метода, и возвращать значение соответствующее типу свойства. Например:

```
public int getAge() {
    return age;
}
```

```java
public class Person {
    private String fullName;
    private int age;
    private boolean retired;
    public Person() {}
    public Person(String fullName, int age, boolean retired) {
        this.fullName = fullName;
        this.age = age;
        this.retired = retired;
    }
    public String getFullName() {
        return fullName;
    }
    public void setFullName(String fullName) {
        this.fullName = fullName;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public boolean isRetired() {
        return retired;
    }
    public void setRetired(boolean retired) {
        this.retired = retired;
    }
}
```

# Геттеры и сеттеры

В языке Java при проектировании классов принято ограничивать уровень доступа к переменным с помощью модификаторов private или protected и обращаться к ним через *геттеры* и *сеттеры*.

Существует также такое понятие как *JavaBeans* классы - это классы содержащие свойства. В Java мы можем рассматривать свойства как private переменные класса. Так как они private, доступ к ним извне класса может быть осуществлен только с помощью методов класса.

Рассмотрим пример реализации концепции *JavaBeans* на классе Person, у которого есть три переменные. Они объявлены как private и доступ к ним возможен только через соответствующие *геттеры* и *сеттеры*.

# Геттеры и сеттеры

```
public class PersonDemo {
    public static void main(String[] args) {
        Person person = new Person();
        person.setFullName("Петров Иван Иванович");
        person.setAge(56);
        person.setRetired(false);

        System.out.println("Полное имя " + person.getFullName());
        System.out.println("Возраст " + person.getAge());
        System.out.println("Пенсионер? " + person.isRetired());
    }
}
```

# Геттеры и сеттеры

Геттеры и сеттеры делают код более громоздким, поэтому часто задается вопрос о том, можно ли обойтись без них. Объясню их необходимость на следующих двух примерах.

Рассмотрим класс CircleWrong, у которого две переменные - радиус и диаметр, объявленные с уровнем доступа по умолчанию.

```
public class CircleWrong {
    int radius;
    int diam;
}
```

# Геттеры и сеттеры

Любой класс в том же пакете может обратится к ним напрямую и изменить их значение. Значения этих двух переменных должно соответствовать друг другу, но пользователь этого класса может задать любые значения, например:

```
public class CircleWrongDemo {
    public static void main(String[] args) {
        CircleWrong circle = new CircleWrong();
        circle.diam = 25;
        circle.radius = 10;

        System.out.println("Диаметр: " + circle.diam);
        System.out.println("Радиус: " + circle.radius);
    }
}
```

```java
public class Circle {
    private int radius;
    private int diam;

    public int getRadius() {
        return radius;
    }

    public void setRadius(int radius) {
        this.radius = radius;
        this.diam = radius * 2;
    }

    public int getDiam() {
        return diam;
    }

    public void setDiam(int diam) {
        this.diam = diam;
        this.radius = diam / 2;
    }
}
```

# Геттеры и сеттеры

Это может привести к неправильным дальнейшим вычислениям. Перепишем этот класс с использованием концепции *JavaBeans,* но немного изменим сеттеры. Метод setRadius() вместе с радиусом задает правильное значение для диаметра. Метод setDiam() написан соответствующим образом.

# Геттеры и сеттеры

Пользователь данного класса не может напрямую добраться к переменным, доступ осуществляется только через сеттеры, где мы контролируем правильную установку значений нашим переменным:

```
public class CircleDemo {
    public static void main(String[] args) {
        Circle circle = new Circle();
        circle.setDiam(25);
        circle.setRadius(10);

        System.out.println(circle.getDiam());
        System.out.println(circle.getRadius());
    }
}
```

```java
public class User {
    private String login;
    private String password;

    public User(String login, String password) {
        this.login = login;
        this.password = password;
    }

    public String getLogin() {
        return login;
    }

    public void setLogin(String login) {
        this.login = login;
    }

    public String getPassword() {
        return password.charAt(0) + "*****";
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

# Геттеры и сеттеры

В следующем примере рассмотрим возможности геттера. Класс User содержит две переменные - логин и пароль. Пароль содержит чувствительную информацию, которую не рекомендуется показывать. Достигается это с помощью объявления переменной с модификатором private и доступом к ней только через геттер метод, в котором вместо полного пароля показываем только первую букву, а остальные заменяются звездочками.

# Геттеры и сеттеры

```
public class UserDemo {
    public static void main(String[] args) {
        User user = new User("mylogin", "mypassword");
        System.out.println("Логин: " + user.getLogin());
        System.out.println("Пароль: " + user.getPassword());
    }
}
```

Результат выполнения:

```
Логин: mylogin
Пароль: m*****
```

# Переопределение методов

Если в иерархии классов совпадают имена и сигнатуры типов методов из подкласса и суперкласса, то говорят, что метод из подкласса *переопределяет* метод из суперкласса.

*Переопределение* методов выполняется только в том случае, если имена и сигнатуры типов обоих методов одинаковы. В противном случае оба метода считаются перегружаемыми.

# Переопределение методов

В следующем примере в классе M определен метод print(). В его наследнике классе N тоже определен метод print() с такой же сигнатурой, но другим поведением. Это и называется переопределением методов:

```
public class M {
    public int i;
    public int j;

    public M(int i, int j) {
        this.i = i;
        this.j = j;
    }

    public void print() {
        System.out.println("Метод M i = " + i + " j = " + j);
    }
}
```

# Переопределение методов

```
public class N extends M {
    public int k;

    public N(int i, int j, int k) {
        super(i, j);
        this.k = k;
    }

    public void print() {
        System.out.println("Метод N k = " + k);
    }

    public void someMethod() {
        print();
    }
}
```

Когда переопределенный метод вызывается из своего подкласса, он всегда ссылается на свой вариант, определенный в подклассе. А вариант метода, определенный в суперклассе, будет скрыт. Из метода someMethod() будет вызван метод того же класса N.

# Переопределение методов

```
public class OverrideDemo {
    public static void main(String[] args) {
        M obj1 = new M(7, 8);
        obj1.print();

        N obj2 = new N(4, 5, 6);
        obj2.print();

        M obj3 = new N(1, 2, 3);
        obj3.print();
    }
}
```

Создадим три объекта и для каждого вызовем метод print().

Первая переменная obj1 типа M указывает на объект того же типа M. При вызове метода print() ожидаемо вызовется метод класса M. Вторая переменная obj2 типа N указывает на объект N. При вызове метода print() вызовется метод класса N. Третий вариант самый интересный - переменная obj3 типа M, но указывает на объект N. Какой же метод print() будет использоваться здесь? Выбор необходимого переопределенного метода выбирается JVM на основе ТИПА ОБЪЕКТА, а не типа переменной!!! Поэтому для переменной obj3 вызовется метод класса N.

# Переопределение методов

Результат выполнения:

```
Метод M i = 7 j = 8
Метод N k = 6
Метод N k = 3
```

# Переопределение методов

Существует такое понятие в Java как динамическая диспетчеризация методов - это механизм, с помощью которого вызов переопределенного метода разрешается во время выполнения, а не компиляции.

Переопределение методов это одна из форм реализации полиморфизма, который позволяет определить в общем классе методы, которые станут общими для всех производных от него классов, а в подклассах - конкретные реализации некоторых или всех этих методов.

# Переопределение методов

Рассмотрим более конкретный пример, который показывает зачем переопределяются методы.

Создадим класс Figure, описывающий какую-то абстрактную фигуру и классы наследники Triangle и Rectangle. Класс Figure содержит метод area(), подсчитывающий площадь фигуры. У каждой фигуры своя формула для подсчета площади, поэтому в классах Triangle и Rectangle метод area() переопределяется соответствующим образом.

```
public class Figure {
    double dim1;
    double dim2;

    public Figure(double dim1, double dim2) {
        this.dim1 = dim1;
        this.dim2 = dim2;
    }

    public double area() {
        System.out.println("Площадь фигуры не определена.");
        return 0;
    }
}
```

# Переопределение методов

```
public class Rectangle extends Figure {
    public Rectangle(double dim1, double dim2) {
        super(dim1, dim2);
    }

    public double area() {
        System.out.println("В области четырехугольника.");
        return dim1 * dim2;
    }
}
```

```
public class Triangle extends Figure {
    public Triangle(double dim1, double dim2) {
        super(dim1, dim2);
    }

    public double area() {
        System.out.println("В области треугольника.");
        return dim1 * dim2 / 2;
    }
}
```

# Переопределение методов

Создадим массив типа Figure, который будет содержать объекты типа Figure, Triangle и Rectangle. Подсчитаем площадь для каждого элемента перебирая элементы массива и вызывая метод area() для каждого элемента. Нам все равно какого типа объект - у каждого есть вызываемый метод area(). JVM с помощью динамической диспетчеризации выбирает нужный вариант метода, основываясь на реальном типе объекта:

```
public class FindAreas {
    public static void main(String[] args) {
        Figure[] figures = new Figure[3];
        figures[0] = new Figure(10, 10);
        figures[1] = new Rectangle(10, 10);
        figures[2] = new Triangle(10, 10);
        for (Figure figure : figures) {
            figure.area();
        }
    }
}
```

# Переопределение методов

Результат выполнения кода:

```
Площадь фигуры не определена.
В области четырехугольника.
В области треугольника.
```

# Переопределение методов

После выхода Java 5 появилась возможность при переопределении методов указывать другой тип возвращаемого значения, в качестве которого можно использовать только типы, находящиеся ниже в иерархии наследования, чем исходный тип. Такие типы еще называются ковариантными.

Например, класс S наследует класс R и переопределяет метод getInstance(). При переопределении возвращаемый тип метода может или остаться таким же - Box6, или быть изменен на любого наследника класса Box6 - HeavyBox, ColorBox или Shipment:

```
public class R {
    Box6 getInstance() {
        return new Box6();
    }
}
```

```
public class S extends R {
    HeavyBox getInstance() {
        return new HeavyBox();
    }
}
```

# Переопределение методов

Статические методы не могут быть переопределены. Класс наследник может объявлять метод с такой же сигнатурой, что и суперкласс, но это не будет переопределением. При вызове переопределенного метода JVM выбирает нужный вариант основываясь на типе объекта. Вызов же статического метода происходит без объекта. Версия вызываемого статического метода всегда определяется на этапе компиляции.

При использовании ссылки для доступа к статическому члену компилятор при выборе метода учитывает тип ссылки, а не тип объекта, ей присвоенного.

# Переопределение методов

Создадим в суперклассе и наследнике статические методы с одинаковой сигнатурой:

```
public class Base {
    public static void go() {
        System.out.println("метод из Base");
    }
}
```

```
public class Sub extends Base {
    public static void go() {
        System.out.println("метод из Sub");
    }
}
```

# Переопределение методов

Попробуем вызвать статический метод используя переменную типа Base, которая указывает на объект типа Sub. При вызове статического метода JVM найдет тип переменной и вызовет метод того же класса:

```
public class Runner {
    public static void main(String[] args) {
        Base ob = new Sub();
        ob.go();
        Sub.go();
    }
}
```

Результат выполнения:

```
метод из Base
метод из Sub
```

# Переопределение методов

Методы объявленные как private никто, кроме самого класса не видит. Поэтому их наличие/отсутствие никак не отражается на классах-наследниках. Они с легкостью могут объявлять методы с такой же сигнатурой и любыми модификаторами. Но это плохой тон! Также класс наследник может расширить видимость protected метода до public. Сузить видимость класс-наследник не может.

# Переопределение методов

Необязательная аннотация @Override используется с методом для указания того, что он переопределен. Если метод переопределен неверно, код не скомпилируется:

```
public class S extends R {
    @Override
    HeavyBox getInstance() {
        return new HeavyBox();
    }
}
```

# Абстрактные классы Java

Абстрактные методы – это методы у которых отсутствует реализация.

Общая форма:

```
abstract тип имяМетода(списокПараметоров);
```

Пример:

```
public abstract double area();
```

# Абстрактные классы Java

Абстрактные методы должны быть обязательно переопределены в подклассе.

Любой класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный. Для этого достаточно указать ключевое слово abstract перед ключевым словом class в начале объявления класса:

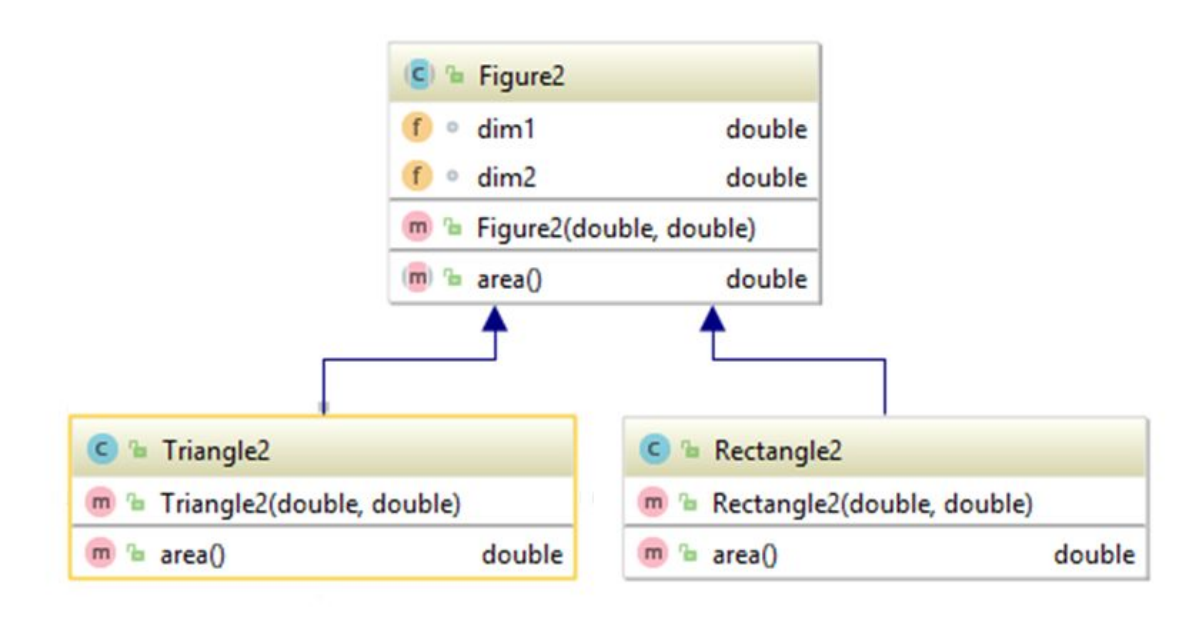
```
public abstract class Figure2 {
    …
    public abstract double area();
}
```

# Абстрактные классы Java

У абстрактного класса в Java не может быть никаких объектов. Но абстрактные классы можно применять для создания ссылок на объекты.

```
public abstract class Figure2 {


    ...
    public abstract double area();
    public static void main(String[] args) {
        // У абстрактного класса не может быть никаких объектов.
        // Figure2 figure1 = new Figure2();
        // Но абстрактные классы можно применять для создания ссылок на объекты.
     Figure2 figure2 = new Rectangle();
    }
}
```

# Абстрактные классы Java




Также нельзя объявлять абстрактные конструкторы или абстрактные статические методы.

Рассмотрим пример использования абстрактных классов и методов на примере класса Figure и его наследников Triangle и Rectangle, которые мы рассматривали в Переопределение методов. Класс Figure описывает абстрактную фигуру, для которой вычисление площади не имеет смысла, поэтому метод area() более правильно сделать абстрактным.

# Абстрактные классы Java

```
public abstract class Figure2 {
    double dim1;
    double dim2;

    public Figure2(double dim1, double dim2) {
        this.dim1 = dim1;
        this.dim2 = dim2;
    }

    public abstract double area();
}
```

```java
public class Rectangle2 extends Figure2 {
    public Rectangle2(double dim1, double dim2) {
        super(dim1, dim2);
    }

    public double area() {
        System.out.println("В области четырехугольника.");
        return dim1 * dim2;
    }
}
```

```java
public class Triangle2 extends Figure2 {
    public Triangle2(double dim1, double dim2) {
        super(dim1, dim2);
    }

    public double area() {
        System.out.println("В области треугольника.");
        return dim1 * dim2 / 2;
    }
}
```

# Абстрактные классы Java

Любой подкласс, производный от абстрактного класса, должен реализовать все абстрактные методы из своего суперкласса или же сам быть объявлен абстрактным.

# Абстрактные классы Java

У абстрактного класса не может быть объектов, но можно создать массив абстрактного класса, который будет содержать ссылки на объекты классов наследников:

```
public class FindAreas2 {
    public static void main(String[] args) {
        Figure2[] figures = new Figure2[3];
        figures[0] = new Rectangle2(20, 10);
        //  figures[0] = new Figure2(10, 10);
        figures[1] = new Rectangle2(10, 10);
        figures[2] = new Triangle2(10, 10);
        for (Figure2 figure : figures) {
            figure.area();
        }
    }
}
```

# Абстрактные классы Java

Результат выполнения программы:

```
В области четырехугольника.
В области четырехугольника.
В области треугольника.
```

# Ключевое слово Java final

Ключевое слово final означает завершенный и может быть использовано для объявления переменных, методов и классов.

Переменная может быть объявлено как final, что позволяет предотвратить изменение содержимого переменной, сделав ее, по существу, константой.

final переменная класса, объявленная как не static, должна инициализироваться при объявлении или в теле конструктора или блоке инициализации, иначе произойдет ошибка компиляции.

Кроме переменных, объявленными как final могут быть параметры метода и локальные переменные. final переменные, объявленные как static, должны быть проинициализированы при объявлении или в блоке инициализации, также объявленном как static. В противном случае, опять получится ошибка компиляции.

# Ключевое слово Java final

```
public class FinalVariables {
    public static final int FILE_NEW = 1;
    private final String someString = "something";

    public static void print(final double d) {
        //  FILE_NEW = 2;
        final String str;
        str = "someString";
        //  str = "";
        //  d = 4;
        System.out.println("FILE_NEW = " + FILE_NEW);
        System.out.println("str = " + str);
        System.out.println("d = " + d);
    }

    public static void main(String[] args) {
        print(3);
    }
}
```

Следующий пример показывает различные варианты объявления завершенных переменных.

# Ключевое слово Java final

Константы – это переменные, значение которых не меняется. Константами в Java принято называть public static final переменные класса. Имена констант следует задавать только заглавными буквами, а слова в имени разделять знаком подчеркивания: MAX_WEIGHT.

Константы часто используются для борьбы с магическими (или волшебными) числами, то есть непонятно что означающими числами или строками. Например, следующий код содержит несколько раз повторяющееся магическое число 9.81:

```
public class PhysicsMagicNumber {
    public static double potentialEnergy(double mass, double height) {
        return mass * height * 9.81;
    }

    public static double getVelocity(double time) {
        return time * 9.81;
    }

    public static double getDistance(double time) {
        return 9.81 * time * time / 2;
    }
}
```

# Ключевое слово Java final

Давайте перепишем код введя константу с именем ACCELERATION. Какие преимущества дает нам введение константы? Во-первых имя константы уже объясняет значение этого числа, и во-вторых при желании изменить значение ACCELERATION, это можно сделать в одном месте. После рефакторинга:

```
public class Physics {
    public static final double ACCELERATION = 9.81;

    public static double potentialEnergy(double mass, double height) {
        return mass * height * ACCELERATION;
    }

    public static double getVelocity(double time) {
        return time * ACCELERATION;
    }

    public static double getDistance(double time) {
        return ACCELERATION * time * time / 2;
    }
}
```

# Ключевое слово Java final

Чтобы запретить переопределение метода в классах наследниках, в начале его объявления следует указать ключевое слово final. Такие методы еще называют завершенными.

Не имеет смысла объявлять метод private final так как private метод не виден в наследниках, соответственно не может быть переопределен.

Также конструктор не может быть объявлен как final.

Класс O содержит завершенный метод, который не может быть переопределен в классе наследнике P. При попытке возникнет ошибка компиляции:

```java
public class O {
    final void method() {
        System.out.println("Это завершенный метод");
    }
}
```

```java
public class P extends O {
    //Этот метод не может быть переопределен
    /* void method() {
        System.out.println("Недопустимо");
    }*/
}
```

# Ключевое слово Java final

Для предотвращения наследование класса в начале объявления класса следует указать ключевое слово final. Объявление класса завершенным неявно делает завершенными и все его методы. Одновременное объявление класса как abstract и final недопустимо.

```
final class A{
    //…
}
class B extends A {// ОШИБКА! Класс A не может иметь подклассы
    //…
}
```

# Статические переменные Java

Ключевое слово языка Java static используется для объявления статических членов класса - методов и переменных. Также бывают статические блоки.

Разница между статическими данными и обычными членами класса состоит в том, что обращение к обычному члену класса должно осуществляться только в сочетании с объектом его класса. Когда же член класса объявлен с ключевым словом static, он доступен и без ссылки на какой-нибудь объект.

# Статические переменные Java

Статические переменные Java, объявляются внутри класса с ключевым словом static. Такие переменные, по существу, являются глобальными переменными. При объявлении объектов, копии статических переменных не создаются - создается одна статическая переменная на весь класс. Статическая переменная создается при загрузке класса.

# Статические переменные Java

Рассмотрим различие между обычными и статическими переменными на следующем примере. Для обращения к обычной переменной a, необходим объект класса StaticVars. К переменной b можно обращаться без упоминания объекта и даже без упоминания класса (если обращение происходит из того же класса):

```
public class StaticVars {
    int a;
    static int b;

    public static void main(String[] args) {
        StaticVars staticVars = new StaticVars();
        System.out.println(staticVars.a);
        System.out.println(b);
    }
}
```

# Статические переменные Java

При обращении к статической переменной из другого класса, необходимо указать имя ее класса - StaticVars.b. Можно обратиться к статической переменной используя любой объект того же класса, например - staticVars1.b или staticVars2.b. Но такой вариант не рекомендуется, так как пользователь вашего кода может решить, что это обычная переменная.

```
public class StaticVarsDemo {
    public static void main(String[] args) {
        StaticVars staticVars1 = new StaticVars();
        StaticVars staticVars2 = new StaticVars();

        System.out.println(StaticVars.b);
        System.out.println(staticVars1.b);

        staticVars1.b = 3;
        staticVars2.b = 4;

        System.out.println(staticVars1.b);
        System.out.println(staticVars2.b);
    }
}
```

# Статические переменные Java

Результат выполнения кода:

```
0
0
4
4
```

# Статические переменные Java

Рассмотрим классический пример, демонстрирующий использование статических переменных для подсчета количества созданных объектов класса Ball. Для этого определим в классе статическую переменную count, которая и будет содержать количество созданных объектов. При создании объекта всегда вызывается конструктор, поэтому именно там будем увеличивать переменную count. Для доступа к private переменной count определен метод getCount():

```
public class Ball {
    static int count = 0;
    String color = "none";

    public Ball(String color) {
        this.color = color;
        count++;
    }
}
```

```
public class BallDemo {
    public static void main(String[] args) {
        Ball ball1 = new Ball("красный");
        Ball ball2 = new Ball("голубой");
        System.out.println("Количество созданных объектов: "
                + Ball.count);
    }
}
```

# Статические переменные Java

Результат выполнения кода:

```
Количество созданных объектов: 2
```

# Статические методы Java

Статические методы можно вызывать не используя ссылку на объект. В этом их ключевое отличие от обычных методов класса. Для объявления таких методов используется ключевое слово static. На методы, объявленные как static, накладывается следующие ограничения:

- Они могут непосредственно вызывать только другие статические методы.
- Им непосредственно доступны только статические переменные.
- Они не могут делать ссылки типа this или super.

```java
public class StaticMethodClass {
    static int staticVar = 3;
    int nonStaticVar;

    public void nonStaticMethod() {
        System.out.println("Нестатический метод");
    }

    static void staticMethod(int localVar) {
        System.out.println("localVar = " + localVar);
        System.out.println("staticVar = " + staticVar);
        //Нельзя обратиться к нестатической переменной из статического метода
        //System.out.println("nonStaticVar = " + nonStaticVar);
    }

    public static void main(String[] args) {
        staticMethod(42);
        //Нельзя обратиться к нестатическому методу без указания объекта
        //nonStaticMethod();
        StaticMethodClass useStatic = new StaticMethodClass();
        useStatic.nonStaticMethod();
        useStatic.staticMethod(67);
    }
}
```

# Статические методы Java

Пример использования статических методов.

# Статические методы Java

```
public class StaticMethodDemo {
    public static void main(String[] args) {
        StaticMethodClass.staticMethod(42);
    }
}
```

# Статический блок Java

```java
import java.util.Scanner;

public class StaticBlock {
    static String a;

    static {
        System.out.println("Статический блок инициализирован.");
        Scanner scanner = new Scanner(System.in);
        a = scanner.nextLine();
    }

    public static void main(String[] args) {
        System.out.println("a = " + a);
    }
}
```

Если для инициализации статических переменных требуется произвести вычисления, то для этой цели достаточно объявить статический блок Java, который будет выполняться только один раз при первой загрузке класса. Объявляется статический блок с помощью ключевого слова static.

# Java static import

```
public class WithoutStaticImport {
    public static void main(String[] args) {
        double value = Math.cos(Math.PI * 4);
        System.out.println(value);
    }
}
```

Для того чтобы получить доступ к статическим членам классов, требуются указывать ссылку на класс.

К примеру, для вызова статического метода cos() класса Math и обращения к ее статической переменной PI, необходимо указать имя класса Math.

# Java static import

```
package oop;

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

public class StaticImport {
    public static void main(String[] args) {
        double value = cos(PI * 4);
        System.out.println(value);
    }
}
```

Чтобы улучшить читабельность кода можно импортировать статические члены класса почти так же, как и обычные классы, и получить прямой доступ к статическим членам без указания имени класса. Для импорта используется оператор Java import static, после которого указывается полное имя класса и метод или переменная.

Статический импорт Java языка располагается после указания пакета перед объявлением класса.

# Класс Object и методы класса Object

В Java определен один специальный класс, называемый Object. Все остальные классы являются подклассами, производными от этого класса, даже если в объявлении это явно не указано. В классе Object определен ряд методов, которые доступны всем классам языка Java.

# Класс Object и методы класса Object

Методы класса Object в Java:

- protected Object clone() - создает новый объект, не отличающийся от клонируемого.
- public boolean equals(Object obj) - определяет, равен ли один объект другому.
- protected void finalize() - вызывается перед удалением неиспользуемого объекта.
- public final Class<?> getClass() - получает класс объекта во время выполнения.
- public int hashCode() - возвращает хеш-код, связанный с вызывающим объектом.
- public final void notify() - возобновляет исполнение потока, ожидающего вызывающего объекта.
- public final void notifyAll() - возобновляет исполнение всех потоков, ожидающих вызывающего объекта.
- public String toString() - возвращает символьную строку, описывающую объект.
- public final void wait() - ожидает другого потока исполнения.
- public final void wait(long timeout) - ожидает другого потока исполнения.
- public final void wait(long timeout, int nanos) - ожидает другого потока исполнения.

# Класс Object и методы класса Object

В Java сравнение объектов производится с помощью метода equals() класса Object. Этот метод сравнивает содержимое объектов и выводит значение типа boolean. Значение true - если содержимое эквивалентно, и false — если нет.

Операция == не рекомендуется для сравнения объектов в Java. Дело в том, что при сравнение объектов, операция == вернет true лишь в одном случае — когда ссылки указывают на один и тот же объект. В данном случае не учитывается содержимое переменных класса.

При создании пользовательского класса, принято переопределять метод equals() таким образом, что бы учитывались переменные объекта.

```java
public class Person {
    private String fullName;
    private int age;
    private boolean retired;
    ...

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }

        Person person = (Person) o;

        if (getAge() != person.getAge()) {
            return false;
        }
        if (isRetired() != person.isRetired()) {
            return false;
        }
        return getFullName() != null
                ? getFullName().equals(person.getFullName())
                : person.getFullName() == null;

    }
}
```

# Класс Object и методы класса Object

Рассмотрим пример использования метода equals() - в классе Person определены три переменные: fullName, age и retired. В переопределенном методе equals() все они участвуют в проверке. Если вы не хотите учитывать какую-то переменную при проверке объектов на равенство, вы имеете право не проверять ее в методе equals().

# Класс Object и методы класса Object

Определим два объекта person1 и person2 типа Person с одинаковыми значениями. При их сравнении с помощью операции "==" вернется значение false, так как это разные объекты. Если же сравнивать их методом equals(), то результат будет равен true. Также в этом примере объявлена переменная person3, которой присвоена ссылка из переменной person1. Вот сравнение person1 == person3 вернет значение true, так как переменные указывают на один объект. При сравнении person1 и person3 с помощью метода equals(), тоже вернется значение true. В методе equals() первой строкой проверяются ссылки сравниваемых объектов - this == o, и если они равны, сразу же возвращается значение true.

# Класс Object и методы класса Object

```
public class PersonDemo2 {
    public static void main(String[] args) {
        Person person1 = new Person("Петров Иван Иванович", 56, false);
        Person person2 = new Person("Петров Иван Иванович", 56, false);
        Person person3 = person1;

        System.out.println("person1 == person2? " + (person1 == person2));
        System.out.println("person1 == person3? " + (person1 == person3));

        System.out.println("person1.equals(person2)? " + person1.equals(person2));
        System.out.println("person1.equals(person3)? " + person1.equals(person3));
    }
}
```

# Класс Object и методы класса Object

Результат выполнения:

```
person1 == person2? false
person1 == person3? true
person1.equals(person2)? true
person1.equals(person3)? true
```

# Класс Object и методы класса Object

Часто необходимо узнать содержимое того или иного объекта. Для этого в классе Object языка Java определен специальный метод toString(), возвращающий символьную строку описывающую объект. При создании нового класса принято переопределение  toString() таким образом, чтобы возвращающая строка содержала в себе имя класса, имена и значения всех переменных.

# Класс Object и методы класса Object

Следующий пример демонстрирует это:

```
public class Person {
    private String fullName;
    private int age;
    private boolean retired;


    ...
    @Override
    public String toString() {
        return "Person{"
                + "fullName='" + fullName + '\''
                + ", age=" + age
                + ", retired=" + retired
                + '}';
    }
}
```

# Класс Object и методы класса Object

Для вызова метода toString() необходимо просто передать нужный объект в System.out.println:

```
public class PersonDemo4 {
    public static void main(String[] args) {
        Person person = new Person("Петров Иван Иванович", 56, false);
        System.out.println(person);
    }
}
```

# Класс Object и методы класса Object

Результат выполнения программы будет такой:

```
Person{fullName='Петров Иван Иванович', age=56, retired=false}
```

Чисто теоретически можно явно вызывать метод toString() - System.out.println(person.toString()), но так не принято.

Если у класса Person не переопределен метод toString(), то при запуске класса PersonDemo4 вызовется метод toString(), определенный в классе Object. И на консоль выведется нечто такое:

```
oop.Person@5e2a3193
```

# Класс Object и методы класса Object

Приложение на языке Java может вызывать методы, написанные на языке C++.

Такие методы объявляются в языке Java с ключевым словом native, которое сообщает компилятору, что метод реализован в другом месте.

Например:

```
public native int hashCode();
```

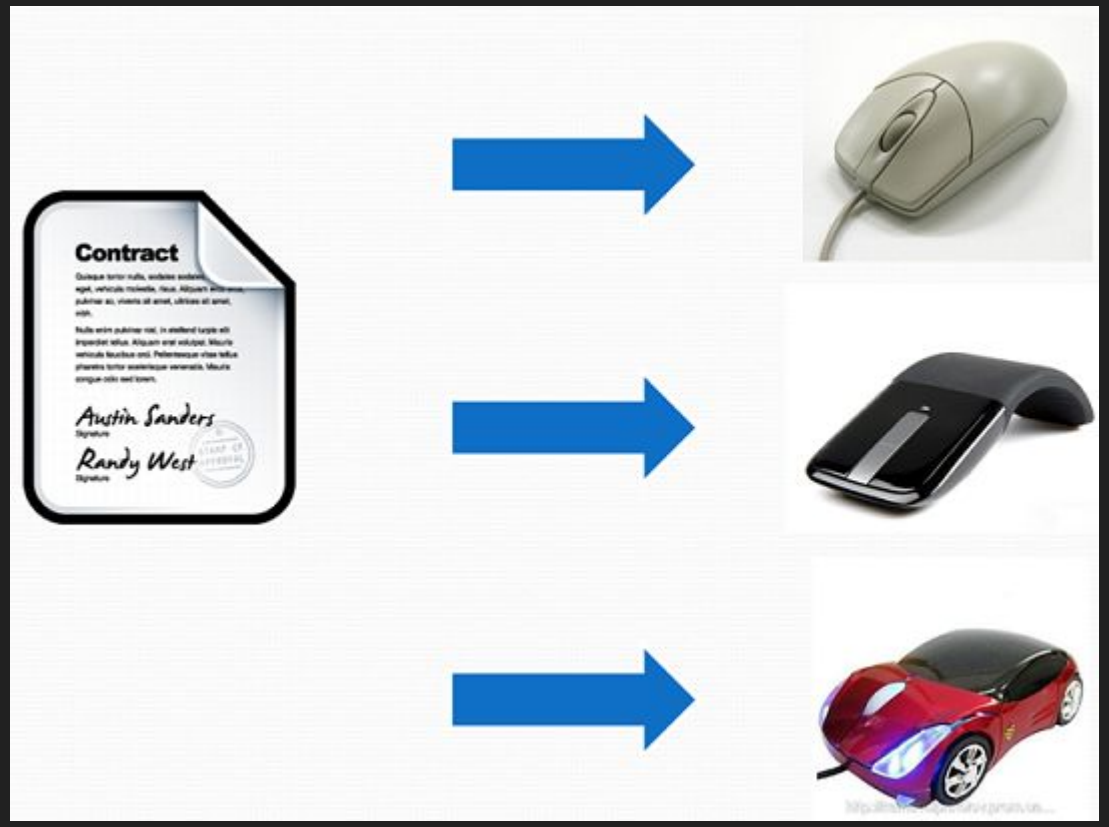
Методы, помеченные native, можно переопределять обычными методами в подклассах.

# Интерфейс

Интерфейс это конструкция языка Java, в рамках которой принято описывать абстрактные публичные (abstract public) методы и статические константы (final static).

С помощью интерфейса можно указать, что именно должен выполнять класс его реализующий, но не как это делать. Способ реализации выбирает сам класс. Интерфейсы не способны сохранять данные состояния. Интерфейсы - это один из механизмов реализации принципа полиморфизма "один интерфейс, несколько методов".

# Интерфейс




Рассмотрим следующую картинку. У нас есть контракт (интерфейс), в котором описано какие действия должна выполнять мышка. Это например, клик по правой клавише и клик по левой. Разные производители мышки (классы), реализующие данный контракт (интерфейс), обязаны спроектировать мышки, у которых будут эти действия. Но как выглядят мышки, какие дополнительные опции будут иметь - все это решает сам производитель.

# Интерфейс

Интерфейсы как и классы могут быть объявлены с уровнем доступа public или default.

Переменные интерфейса являются public static final по умолчанию и эти модификаторы необязательны при их объявлении. Например, в следующем примере объявлены переменные RIGHT, LEFT, UP, DOWN без каких-либо модификаторов. Но они будут public static final.

# Интерфейс

```
public interface Moveable {
    int RIGHT = 1;
    int LEFT = 2;
    int UP = 3;
    int DOWN = 4;

    void moveRight();

    void moveLeft();
}
```

Все методы интерфейса являются public abstract и эти модификаторы тоже необязательны. Объявляемые методы не содержат тел, их объявления завершаются точкой с запятой.

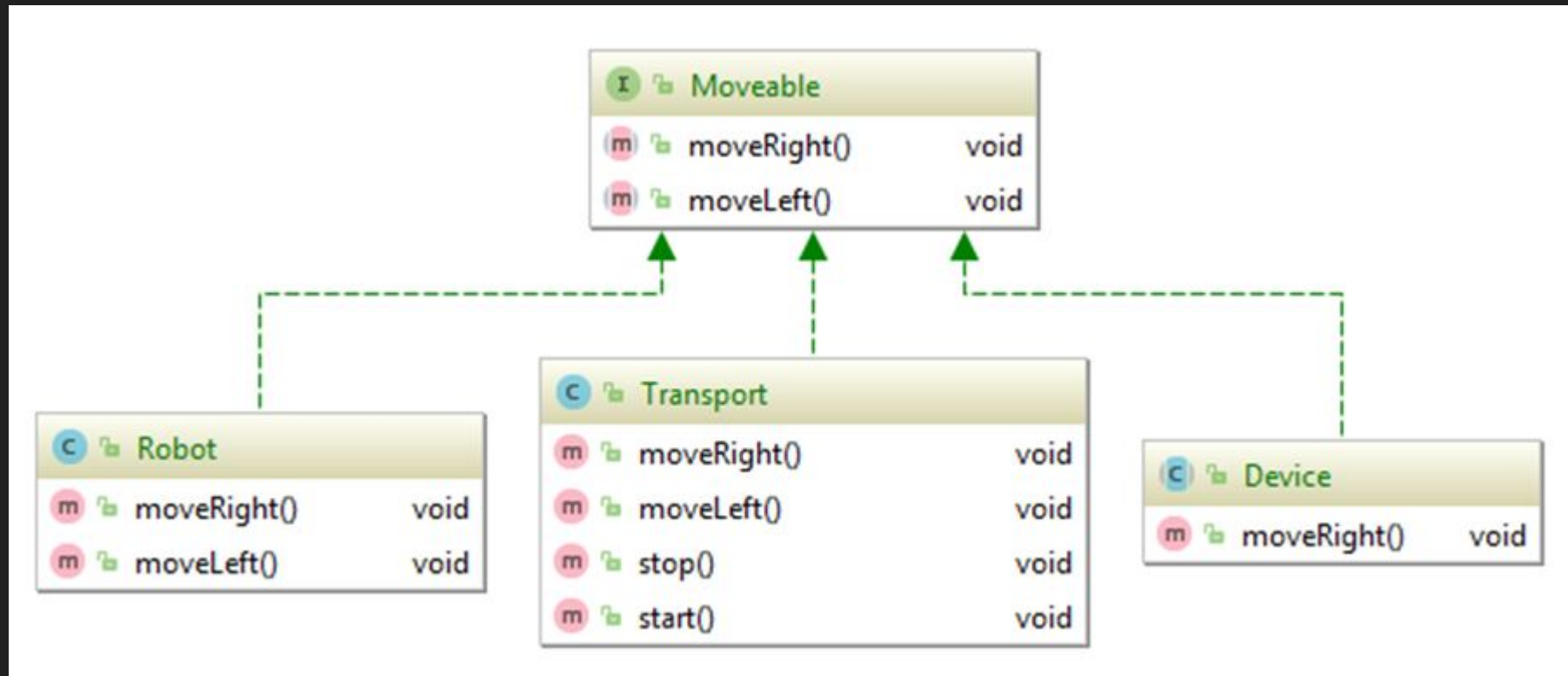
# Интерфейс

```
public class Transport implements Moveable {
    public void moveRight() {
        System.out.println("Транспорт поворачивает вправо.");
    }

    public void moveLeft() {
        System.out.println("Транспорт поворачивает влево.");
    }

    public void stop() {
        System.out.println("Транспорт останавливается.");
    }

    public void start() {
        System.out.println("Транспорт стартует.");
    }
}
```

Чтобы указать, что данный класс реализует интерфейс, в строке объявления класса указываем ключевое слово implements и имя интерфейса. Класс реализующий интерфейс должен содержать полный набор методов, определенных в этом интерфейсе. Но в каждом классе могут быть определены и свои методы. Например, следующий класс Transport реализует интерфейс Moveable. В нем реализованы методы moveRight() и moveLeft() интерфейса Moveable, и добавлены свои методы stop(), start().

# Интерфейс

Один интерфейс может быть реализован любым количеством классов. Например, в следующей схеме добавлены еще два класса Robot и Device, которые тоже реализуют интерфейс Moveable.



# Интерфейс

Класс Robot из вышеуказанной схемы:

```
public class Robot implements Moveable {
    public void moveRight() {
        System.out.println("Робот поворачивает вправо.");
    }

    public void moveLeft() {
        System.out.println("Робот поворачивает влево.");
    }
}
```

# Интерфейс

Если класс реализует интерфейс, но не полностью реализует определенные в нем методы, он должен быть объявлен как abstract.

Например, класс Device реализует только один метод интерфейса Moveable, поэтому он абстрактный:

```
public abstract class Device implements Moveable {
    public void moveRight() {
        System.out.println("Девайс поворачивает вправо.");
    }
}
```

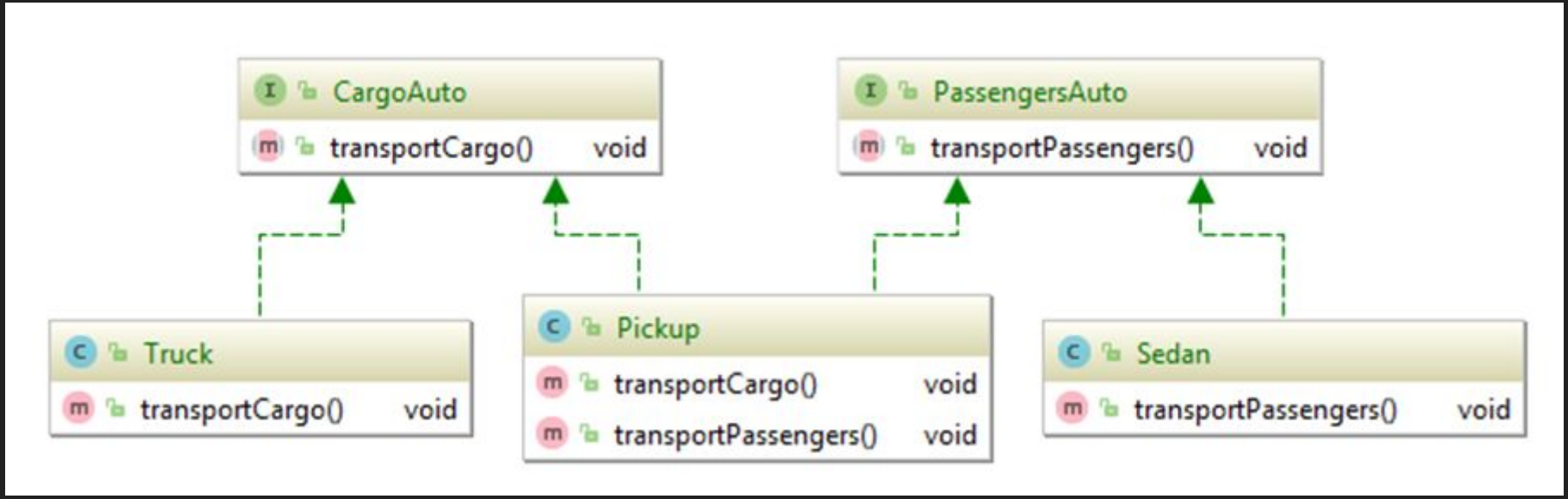
```java
public class TransportDemo {
    public static void main(String[] args) {
        Moveable moveable = new Transport();
        Transport transport = new Transport();
        Moveable robot = new Robot();
        //Moveable moveable1 = new Moveable();

        //  moveable.start();
        moveable.moveRight();
        moveable.moveLeft();
        System.out.println();

        transport.start();
        transport.moveRight();
        transport.moveLeft();
        transport.stop();
        System.out.println();

        robot.moveLeft();
        robot.moveRight();
    }
}
```

# Интерфейс

Тип интерфейса можно указывать при объявлении переменных, которые будут содержать ссылки на объекты, классы которых реализуют этот интерфейс. Например, в следующем примере переменная moveable имеет тип Moveable, и указывает она на объект Transport. Но на основе интерфейсов нельзя порождать объекты. Например, в строке Moveable moveable1 = new Moveable() будет ошибка компиляции. При использовании переменной типа интерфейс, доступны только те члены класса, которые определены в этом интерфейсе. Например, нельзя вызвать метод start(), используя переменную moveable. А для переменной transport можно.

# Интерфейс

Один класс может реализовать любое количество интерфейсов. На следующей схеме показан класс Pickup, который реализует два интерфейса CargoAuto и PassengersAuto:



# Интерфейс

```java
public interface CargoAuto {
    void transportCargo();
}
```

```java
public interface PassengersAuto {
    void transportPassengers();
}
```

# Интерфейс

Для указания того, что класс реализует несколько интерфейсов, после ключевого слова implements через запятую перечисляются нужные интерфейсы. Класс Pickup должен определить все методы реализуемых интерфейсов:

```
public class Pickup implements CargoAuto, PassengersAuto {
    public void transportCargo() {
        System.out.println("Везу груз");
    }

    public void transportPassengers() {
        System.out.println("Везу пассажиров");
    }
}
```

# Интерфейс

Интерфейсы объявленные в классах или в другие интерфейсах называются внутренние или вложенные. Например интерфейс NestedIf определен внутри класса A:

```
public class A {
    public interface NestedIf {
        boolean isNotNegative(int x);
    }
}
```

# Интерфейс

При обращении к интерфейсу NestedIf требуется указывать имя его внешнего класса - A.NestedIf:

```
public class B implements A.NestedIf {
    public boolean isNotNegative(int x) {
        return x >= 0;
    }
}
```

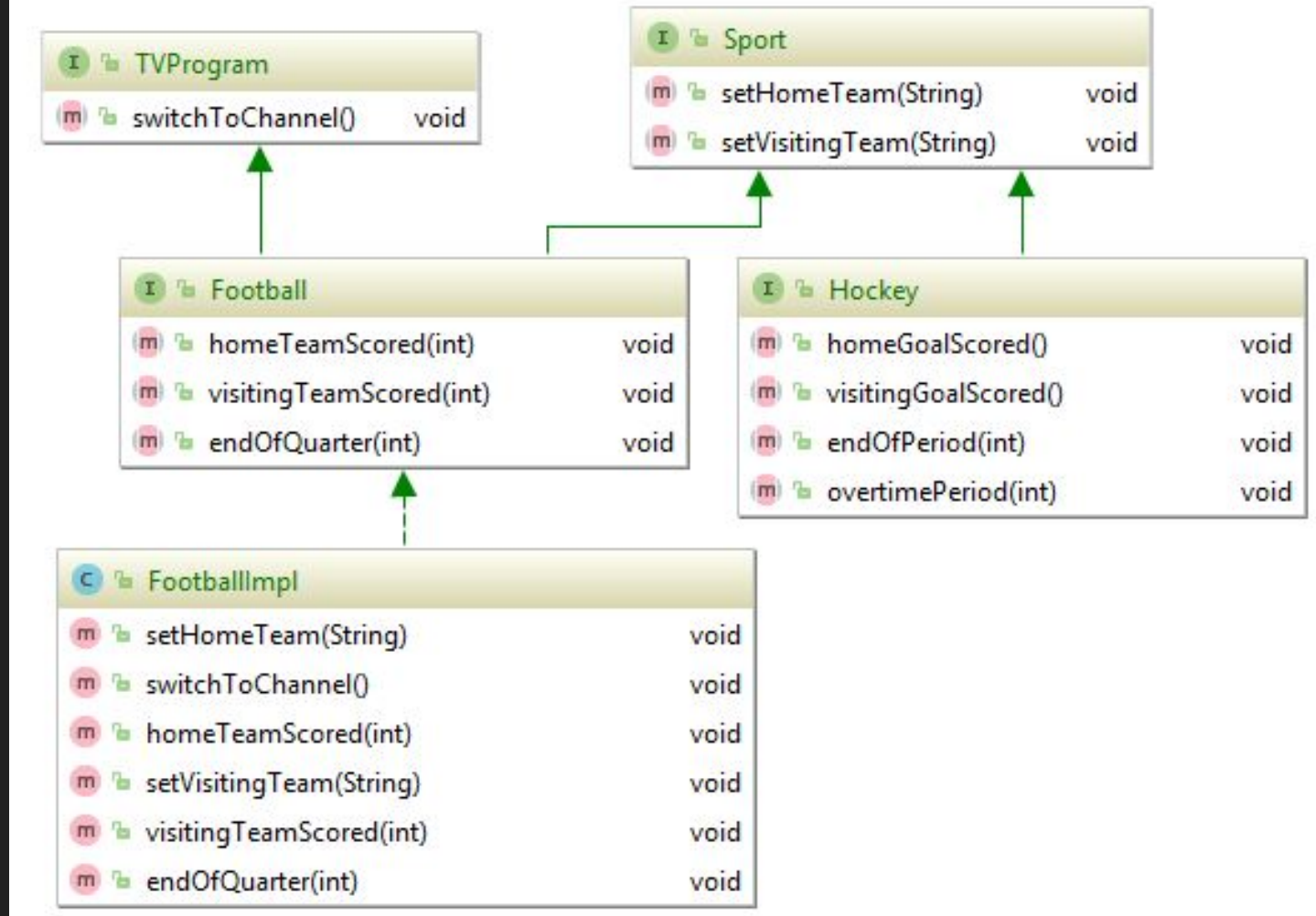
# Интерфейс

```
public class NestedIfDemo {
    public static void main(String[] args) {
        A.NestedIf nif = new B();
        if (nif.isNotNegative(10)) {
            System.out.println("Число 10 не отрицательное.");
        }
        if (nif.isNotNegative(-12)) {
            System.out.println("Это не будет выведено.");
        }
    }
}
```

# Интерфейс

Интерфейс может наследоваться от другого интерфейса через ключевое слово extends. Один интерфейс, в отличие от классов, может расширять несколько интерфейсов.

Например, интерфейс Football расширяет интерфейсы TVProgram и Sport. Класс FootballImpl, реализующий интерфейс Football, должен переопределить методы всех трех интерфейсов Football, TVProgram и Sport:



```java
public interface Sport {
    void setHomeTeam(String name);

    void setVisitingTeam(String name);
}
```

```java
public interface Hockey extends Sport {
    void homeGoalScored();

    void visitingGoalScored();

    void endOfPeriod(int period);

    void overtimePeriod(int ot);
}
```

```java
public interface TVProgram {
    void switchToChannel();
}
```

```java
public interface Football extends Sport, TVProgram {
    void homeTeamScored(int points);

    void visitingTeamScored(int points);

    void endOfQuarter(int quarter);
}
```

```java
public class FootballImpl implements Football {
    @Override
    public void setHomeTeam(String name) {
        System.out.println("Setting Home Team");
    }

    @Override
    public void switchToChannel() {
        System.out.println("Switching to channel");
    }

    @Override
    public void homeTeamScored(int points) {
        System.out.println("Scored");
    }

    @Override
    public void setVisitingTeam(String name) {
        System.out.println("Setting visiting team");
    }

    @Override
    public void visitingTeamScored(int points) {
        System.out.println("Visiting Team Scored");
    }

    @Override
    public void endOfQuarter(int quarter) {
        System.out.println("End of quarter");
    }
}
```

# Интерфейс

Интерфейсы маркеры - это интерфейсы, у которых не определены ни методы, ни переменные. Реализация этих интерфейсов придает классу определенные свойства. Например, интерфейсы Cloneable и Serializable, отвечающие за клонирование и сохранение объекта в информационном потоке, являются интерфейсами маркерами. Если класс реализует интерфейс Cloneable, это говорит о том, что объекты этого класса могут быть клонированы.

# Интерфейс

В JDK 8 в интерфейсы ввели методы по умолчанию - это методы, у которых есть реализация. Другое их название - методы расширения. Классы, реализующие интерфейсы, не обязаны переопределять такие методы, но могут если это необходимо. Методы по умолчанию определяются с ключевым словом default.

# Интерфейс

Интерфейс SomeInterface объявляет метод по умолчанию defaultMethod() с базовой реализацией:

```
public interface SomeInterface {
    default String defaultMethod() {
        return "Объект типа String по умолчанию";
    }
}
```

# Интерфейс

Класс SomeInterfaceImpl1, реализующий этот интерфейс, не переопределяет метод defaultMethod() - так можно.

```
public class SomeInterfaceImpl1 implements SomeInterface {
}
```

# Интерфейс

А если класс SomeInterfaceImpl2 не устраивает реализация по умолчанию, он переопределяет этот метод:

```
public class SomeInterfaceImpl2 implements SomeInterface {
    @Override
    public String defaultMethod() {
        return "Другая символьная строка";
    }
}
```

# Интерфейс

Создаем два объекта классов SomeInterfaceImpl1 и SomeInterfaceImpl2, и вызываем для каждого метод defaultMethod(). Для объекта класса SomeInterfaceImpl1 вызовется метод, реализованный в интерфейсе, а для объекта класса SomeInterfaceImpl2 - его собственная реализация:

```
public class DefaultMethodDemo {
    public static void main(String[] args) {
        SomeInterface obj1 = new SomeInterfaceImpl1();
        SomeInterface obj2 = new SomeInterfaceImpl2();

        System.out.println(obj1.defaultMethod());
        System.out.println(obj2.defaultMethod());
    }
}
```

# Интерфейс

Результат выполнения:

```
Объект типа String по умолчанию
Другая символьная строка
```

# Интерфейс

```
public interface MyIf {
    int getNumber();

    static int staticMethod() {
        return 0;
    }
}
```

```
public class StaticMethodDemo {
    public static void main(String[] args) {
        MyIf obj1 = new MyIfImp();

        System.out.println(obj1.getNumber());
        System.out.println(MyIf.staticMethod());
    }
}
```

В версии JDK 8, в интерфейсы добавлена еще одна возможность - определять в нем статические методы. Статические методы интерфейса, как и класса, можно вызывать независимо от любого объекта. Для вызова статического метода достаточно указать имя интерфейса и через точку имя самого метода.