

Kydai inc.

**ООП в Java для мультиварок, масляных
обогревателей и прочих обогревающих
элементов**

**Учебное пособие для тех,
кто собирается сдать 2 лабу по
проге хотя-бы в первом сейме**



Санкт-Петербург

2021

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЗ111

НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
“ИНСТИТУТ ТЕПЛЫХ МУЖСКИХ ОТНОШЕНИЙ”

Kydlai inc.

**ООП в Java для мультиварок, масляных
обогревателей и прочих обогревающих
элементов**

**Учебное пособие для тех,
кто собирается сдать 2 лабу по
проге хотя-бы в первом сейме**



УНИВЕРСИТЕТ ИТМО

Санкт-Петербург

2021



Эта методичка даст вам познания, необходимые для 2 и 3 лаб по проге, если вы не изучали Java. Данный способ представления материала довольно краток, поэтому отчасти будет слишком много непонятностей сразу (“В этом предложении 11 слов, 9 из которых я полчаса назад не знал, в 7 из которых я путаюсь”), поэтому в случае проблем с пониманием существует гугл, ютуб и люди, близкие к гикам чуть более чем на 0%.

Теперь поговорим об ООП. Что это такое? ООП — объектно-ориентированное программирование. Принцип заключается в разделении программы на блоки (объекты) и в их взаимодействии. Каждый объект является экземпляром класса, а сами классы образуют определенную иерархию. Формально ООП в Java можно разделить на 3 ветки — классы, интерфейсы и перечисления.

Принципы ООП

Перед началом объяснения нужно обозначить принципы (или парадигмы) ООП. На самом деле их достаточно много, но выделяют в основном 3 (в последнее время 4):

Инкапсуляция — размещение в одном компоненте данных и методов, которые с ними работают. В Java принято рассматривать инкапсуляцию вместе с сокрытием данных. Сокрытие — разграничение доступа различных частей программы ко внутренним компонентам друг друга.

Полиморфизм — возможность применения одноименных методов с объектами разных типов данных в пределах класса или группы классов

Абстракция — выделение главных (важных для текущей задачи) характеристик предмета и игнорирование всех прочих

Наследование — механизм, позволяющий описать новый класс, используя уже существующий, расширив при этом функционал

Классы

Классы — основа всех проектов на Java. Как правило отдельному классу соответствует файл с таким же названием. Именно на их взаимодействии строится всё. Определение звучит так: «Класс — некоторое общее описание множества объектов и их поведения, взаимодействия, присущего каждому объекту из множества». Например. Допустим, у нас есть множество стульев. Как мы можем охарактеризовать любой объект этого множества? Что есть у них всех? Очевидно, что у всех стульев есть ножки, и каких-то 3, а у каких-то 4, но у любого объекта типа «стул» есть какое-то количество ножек, которое может различаться, но сам параметр количества ног должен присутствовать.

Классы объявляются с помощью ключевого слова `class`. Пример общего объявления такой:

```
<модификатор доступа> class <имя класса>{  
    // тело
```

}

В Java принято называть все классы с большой буквы, а имя публичного класса должно совпадать с названием .java файла.

В теле класса описывается поведение самого класса. Разделить это можно на 2 вещи — методы и поля. **Поля** — **переменные**, находящиеся внутри класса. **Методы** — **функции**. Если с созданием полей все очевидно — создавать их вы научились в прошлом пособии, то с методами мы знакомимся только сейчас. Но перед этим хочется сказать об области видимости. Если в прошлый раз мы создавали все переменные внутри *main* и никаких проблем не возникало, то теперь вы можете столкнуться с тем, что объявленные вами переменные почему-то вам не доступны. Нужно запомнить только одну вещь — переменные существуют только в тот момент времени, пока выполняется блок, в котором их создали. Следовательно, если вы просто в теле класса создадите, допустим, `int a = 5`, то эта переменная будет существовать до тех пор, пока существует объект вашего класса. Такие переменные называются **локальными**, то есть они доступны откуда угодно, но только внутри класса, в котором они были определены. Переменные, объявленные в телах функций, циклов и т. п., существуют только во время их выполнения и по окончании пропадают, такие переменные тоже называются **локальными**. **Глобальные** переменные — **публичные статические переменные**, т.е. они доступны в любом месте проекта.

Методы

И так, вернемся к функциям, которые в Java обычно называют методами. Для их объявления существует следующий шаблон:

```
<модификатор доступа> <тип возвращаемого значения> <имя функции> (<тип аргумента 1> <имя аргумента 1>, ... <тип аргумента n> <имя аргумента n>) {  
    // тело функции  
}
```

Тип возвращаемого значения — объект класса, который вернет данная функция (`int`, `ArrayList<Integer>`, `String` и т.д.). Если функция не возвращает значение, то пишется `void`, иначе возвращение результата осуществляется через ***return <переменная>***, причем после этого выполнение функции прерывается.

Сигнатурой метода называется имя и передаваемые параметры, причем порядок самих параметров важен. Методы (т.е. функции) используются для взаимодействия с полями (т.е. переменными) классов и между классами. Одни методы могут вызвать другие, а чтобы вызвать метод у объекта класса используется эта конструкция: `<имя переменной-объекта>.<имя метода>()`


```
    }  
}
```

Мы не можем написать `legs = legs`, т.к. они обе - параметры, но если мы напишем `this.legs = legs`, то переменной `legs` класса `Chair` присвоится значение `legs`, переданное в качестве параметра. Также через `this`. Можно вызывать методы класса, а `this(<параметры>)` вызовет конструктор с соответствующим набором параметров. `Super` – это тот же `this`, но он указывает не на объект текущего класса, а родительского (отношения между классами мы разберем немного позже). Во всем остальном их использование одинаково.

Теперь немного примеров, чтобы стало понятнее.

Вернемся к примеру со стульями:

```
public class Chair{  
    private int legs = -1;  
  
    public Chair(int legs){  
        System.out.println(legs);  
        System.out.println(this.legs);  
    }  
}
```

Теперь при написании строчки `Chair chair = new Chair(2);` вызовется конструктор класса `Chair`, который напечатает следующие:

2 // Т.к. `legs` – параметр, переданный в конструктор и в данном случае в области видимости находится именно он, а не переменная класса
-1 // Т.к. `this` – указатель на объект текущего класса и `this.legs` – непосредственное обращение к переменной класса

Вошебные слова, которые помогут вам в разработке

Итак, теперь, когда вы уже знаете кусочек из ООП и можете спокойно воспринимать примеры, пришло время прояснить те оставшиеся моменты, которые вам пока не известны. Первое — ***модификаторы доступа***. С их помощью реализуется доступ к полям и методам классов и объектов. Всего их 4:

- `public`
- `default` (aka `package-private`)
- `protected`
- `private`

Сразу скажу, что сравнивать `default` и `protected` немного некорректно, т.к. защищают они от разного. Теперь по порядку.

Public – доступен всем и отовсюду. Иначе говоря, для классов — объект данного класса можно создать в любом классе, получать доступ к полям данного класса и использовать методы данного класса, для методов — данный метод можно вызвать откуда угодно, для переменных — доступ к данной переменной можно получить через любой объект класса или через сам класс, если поле — статическое (про это чуть позже).

Default – Как `public`, но работает только внутри определенного пакета. Если при объявлении поля/класса/метода вы не укажете модификатор доступа, то он будет поставлен по умолчанию (из названия это же так неочевидно). Ключевого слова под него в Жабу не завезли, поэтому писать его не надо. Теперь о механике работы. Помните, в первой методичке была строчка `package main;` (Если нет, то добро пожаловать! <https://t.me/joinchat/V7md1-rTKzgwNDEy>)? Именно эта строчка определяет, внутри какого пакета находится данный класс. Что это значит? Теперь для любого, кто находится внутри данного пакета будет возможность взаимодействовать с методом/классом/полем так, как будто он для него `public`, а для всех, кто не находится в этом пакете его не будет существовать.

Protected – использование только самим классом и его наследниками, а также всем, кто находится с ним в одном пакете. Наследование мы разберем чуть позже.

Private – использование только внутри класса и никак иначе. Для переменных/функций — обращение к ним возможно только внутри самого класса.

Еще одно магическое слово, с которым вы познакомились ещё в первой методичке — **static**. Оно просто перепривязывает сущность, к которой относится, от объекта к самому классу (если сам класс помечен `static`, то там другая история, которую мы разберем позже). Проще говоря, теперь вместо создания объекта класса для обращения к функции мы сможем обращаться к ней через имя самого класса. Разумеется, если мы создадим объект класса, то все равно сможем к ней обратиться. Очень знакомый вам пример `static` метода — `Math.abs()`. В классе `Math` метод `abs` обозначен статическим, поэтому мы можем его вызвать так. Из минусов можно отметить только то, что теперь на весь класс будет только одна переменная => при изменении её значения, меняется оно будет у всех объектов класса (это далеко не всегда минус).

Также, из-за очевидной привязки к самому классу, статические функции могут вызывать только другие статические функции и использовать только локальные и статические глобальные. В то время как нестатические функции такого ограничения не имеют => нестатическая функция может вызвать статическую.

final – еще одно полезное ключевое слово. Его реализация зависит от контекста

Для переменных — зависит от типа:

примитивный — запрещено изменять значение после первого присвоения

- остальные — запрещено изменение ссылки на объект, но изменение данных внутри самого объекта разрешено

Для функций — запрет на переопределение в классах-наследниках, для классов — запрет на наследование от данного класса.

Наследование и abstract

Ранее я уже упоминал классы могут наследоваться/расширяться от других классов. Сразу стоит обозначить, что Java не поддерживает множественное наследование => каждый класс может иметь только одного родителя

Происходит это с помощью ключевого слова ***extends***. При этом класс наследник получает все методы и параметры, а также вложенные классы (о них поговорим чуть позже). В переменную типа родительского класса можно поместить объект дочернего класса, но интерпретировать его java будет именно как родителя, поэтому без явного приведения нельзя будет вызвать дочерний метод, но это позволяет использовать ***полиморфизм***. Также можно проверить, является ли объект экземпляром класса с помощью ***instanceof***, который вернет true при условии, что объект — наследник класса или является объектом самого класса.

По умолчанию все Java-классы являются наследниками класса Object, т.е. если вы не указали класс-родитель, то Java автоматически добавит extends Object

Пример:

```
class A {
    private void printA() {
        System.out.println("A");
    }

    protected void printB() {
        System.out.println("B");
    }
}

class B extends A {

    public void print_a() {
        printA(); // ошибка, т.к. A - приватная функция, которая недоступна
                  наследникам
    }

    public void print_b() {
        printB(); // а вот protected доступен наследникам
    }
}

A a = new A();
B b = new B();
a.printA(); // Ошибка, т.к. printA() - private
a.printB(); // Выведет B, если класс Main находится в одном пакете с A
b.printB(); // Выведет B, если класс Main находится в одном пакете с B
b.print_b(); // Выведет B
```

Теперь про ***abstract***. Если мы хотим в каком-то классе объявить метод, но мы не знаем, что он будет делать, то можно пометить его как ***abstract***, тогда перед именем самого класса нужно будет написать этот модификатор. От абстрактного класса нельзя создать объект (можно, но об этом потом), т.к. у как минимум одного из его методов нет реализации и при его вызове java просто не поймет, что ей нужно сделать. Тогда при создании наследника от этого класса вам будет необходимо реализовать этот метод или оставить новый класс абстрактным.

Большие классов богу классов

На самом деле, в одном файле может находиться не только один класс — их может быть много, но главный класс всегда помечен как `public`, а его имя совпадает с именем файла. А сейчас поговорим о еще 4 видах классов:

- Вложенные внутренние классы
- Вложенные статические классы
- Локальные классы
- Анонимные классы

Теперь подробнее.

Вложенные внутренние классы — классы, объявленные внутри других классов. Например

```
public class A {  
    private class B{  
        //текст  
    }  
}
```

Такое используется, когда мы хотим создать еще один класс для упрощения работы с основным классом, но не хотим выносить его в отдельный файл.

Допустим, у нас есть класс автомобиль, и мы хотим добавить класс двигатель, который будет реализовывать логику непосредственно нашего двигателя автомобиля, причем всем остальным классам в проекте совершенно не обязательно знать, что это за двигатель, ведь они будут работать непосредственно с одним объектом — автомобилем и им не нужен какой-то абстрактный двигатель.

Особенности:

- Объекты таких классов существуют только у объектов внешних классов (очевидно, ведь для их создания нужно создать объект внешнего класса)
- Второе вытекает из первого. Внутри таких классов не может быть статических переменных, ведь они не привязаны к классу, а сам вложенный внутренний класс привязан к объекту внешнего класса.

У таких классов полный доступ к методам и полям родительского класса, даже если они помечены как `private`. Аналогично для отношений внешний-внутренний.

`this` — ссылка на объект внутреннего класса, а `A.this` — ссылка на объект внешнего класса, если `this` вызвать в `B`

Вложенные статические классы ничем не отличаются от внешнего класса, за исключением прописания пути. Например, пусть у нас есть

```
public class A{  
    public static class B{  
  
    }  
}
```

Тогда, через `A.B b = new A.B();` мы создадим объект класса `B`. Такое удобно если нужно связать несколько схожих классов, при этом не вынося их в отдельные файлы. Так, например, можно сделать с покемонами и их эволюциями во 2 лабе. Разумеется, это будет удобно только если вы помните эволюцией какого покемона является данный покемон, ну или пользуетесь автодополнением IntelliJ IDEA.

P.S. статический вложенный класс может обращаться только к статическим полям/методам родительского класса, т.к. объекта класса-родителя может не существовать.

Локальные классы обычно создаются внутри методов. По сути, это вложенные внутренние классы, но не для класса, а для метода. Не знаю, зачем это может пригодиться, но так можно.

Самые простые из этой четверки — **анонимные классы**. По сути, при использовании анонимного класса мы пишем свой класс прямо во время создания объекта. Помните, я говорил, что от абстрактного класса нельзя создать объект? Так вот, это можно сделать с помощью анонимного класса. Допустим, у нас есть

```
public abstract class A{
    public abstract void f();
}
```

При попытке создать объект напрямую через `A a = new A();` нам выведет ошибку, но уже такая запись будет корректной:

```
A a = new A() {
    @Override
    public void f() {

    }
};
```

Ведь, по сути, теперь метод `f` реализован и вызовется через полиморфизм при написании `a.f()`; Применяются такие классы когда:

- Тело нужного класса короткое
- нужен только один экземпляр класса
- класс используется в месте его создания или где-то неподалеку
- Имя класса не важно

Интерфейсы

Интерфейс похож на абстрактный класс, только если класс описывает и поведение, и состояние, то интерфейс описывает только поведение. В нем содержатся только методы без их реализации, соответственно мы не можем использовать `private` в интерфейсах. С точки зрения логики, абстрактный класс связывает классы, которые имеют близкую логическую связь, в то время как интерфейс, который просто описывает поведение, которое может использоваться в совершенно разных классах. Например, у нас есть утка и самолет. Что у них общего? Умение летать, поэтому они оба могут реализовать

интерфейс `flyable`(умеющий летать), а теперь попробуйте придумать какой-нибудь подходящий класс-родитель с точки зрения логики для этих двух классов.

Implements – ключевое слово, которое используется для обозначения реализации интерфейса, т.е. синтаксически это выглядит так: `<имя класса> implements <имена интерфейсов через запятую>`. Каждый класс в Java может реализовывать сколько угодно интерфейсов, но иметь только одного родителя. Сами интерфейсы могут наследоваться через `extends`, тогда при реализации интерфейса, классу нужно будет реализовать как его методы, так и методы родителей.

Хоть изначально интерфейсы не имеют реализации методов, но её можно поставить по умолчанию через ***default***. Выглядит это так:

```
public interface Flyable{
    public default void fly(){
        System.out.println("Я летаю")
    }
}
```

Когда вы наследуете свой класс от абстрактного, вы реализуете его абстрактные методы и над ними появляется аннотация `@Override`. Она заставляет компилятор проверить существование метода с данной сигнатурой в родительском классе => это проверка на то, что вы именно переопределяете метод, а не создаете новый. Точно такая-же появляется и при реализации методов интерфейса.

P.S. От интерфейса нельзя напрямую создать объект, но анонимные классы позволяют это сделать.

Функциональные интерфейсы и лямбда-функции

Стоит сначала сказать, что всё, что вы дальше прочтёте необходимо только для 3 и последующих лаб по проге, так что если вы готовитесь ко 2 лабе, то вам осталось лишь прочитать заключение на последней странице.

Функциональными интерфейсами называются интерфейсы, объявляющие только один абстрактный метод. Для таких случаев была придумана аннотация `@FunctionalInterface`, которая проверяет интерфейс на наличие только одной функции. Специально для них были созданы такие классные штуки как лямбда функции. По сути, это просто сокращенный анонимный класс, который под капотом превращается в обычный. Синтаксически это выглядит так:

```
(<параметры>) -> {<тело функции>} Например,
public interface Printable {
    public void print();
}

public void print(Printable p){
    p.print()
}
```

Тогда, при вызове `print()` вместо

```
print(new Printable() {  
    System.out.println("Hello world!");  
});
```

Возможно написать просто

```
print(() -> {  
    System.out.println("Hello world!");  
});
```

А если тело функции состоит только из одной строчки, то

```
print(() -> System.out.println("Hello world!"))
```

В () передаются параметры через запятую, но если передается только 1 параметр, то скобки можно опустить. Также лямбда выражения могут возвращать значения через *return*. Сами лямбда выражения могут получать доступ к внешним параметрам, только если они объявлены как *final*, причем изменять их значения они не могут.

Ссылки на методы

Теперь поговорим о ссылках на методы. Допустим, что у нас есть какая-то лямбда, которая просто вызывает какой-то метод, допустим

$x \rightarrow \text{System.out.println}(x)$. Теперь представим, что нам нужно только указать какой метод вызвать, а сами элементы указывать не нужно. Тогда нам нужно указать конструкцию

<имя класса>::*<имя метода>* для статической функции

<переменная с объектом>::*<имя метода>* для нестатической функции

<имя класса>::*<имя метода>* для класса, в котором этот метод реализован

Для использования этого нужно, чтобы у передаваемого метода и метода функционального интерфейса были одинаковые принимаемые параметры. По сути, аргументы будут переданы функции, которая указана в ссылке.

Вот простой, но достаточно корявый пример:

```
interface Inter {  
    void f(int i)  
}  
  
static void j(int i, Inter inter){  
    inter.f(i);  
}  
  
static void main(String[] args){  
    int[] a = { 1, 2, 3 };  
    for (int t : a)  
        j(t, System.out::println)  
}
```

После выполнения этого кода выведет 1, 2, 3.

Вот более сложный, но понятный для чего это существует пример

```
ArrayList<Integer> a = new ArrayList<>();  
a.add(1);  
a.add(2);  
a.add(3);  
  
a.forEach(System.out::println);
```

Тоже выведет 1 2 3, но теперь более понятно, как это работает. Функция `forEach` вызывает для каждого элемента коллекции функцию интерфейса `Consumer`, которая принимает любой параметр типа `Object` => любой созданный в Java класс, а потом выполняет с ним действия, указанные в лямбда функции. В данном случае в функцию `System.out.println()` передается каждый параметр коллекции `a`. Причем мы не вызываем сам метод, а просто передаем ссылку на него.

Enum

Когда мы пишем проги, то иногда может возникать необходимость, чтобы какая-то переменная имела строгий диапазон значений. Самый банальный пример — календарь. Пусть, нам нужно написать программу-календарь. Пусть, в ней будет переменная типа `String`, которая отвечает за название текущего дня. Очевидно, что всего дней 7, а значений — бесконечно много, в результате этого в переменной может оказаться значение «https://www.youtube.com/watch?v=X9r-4W_tPOU», что нам совершенно не нужно. Конечно, можно объявить 7 статических констант и использовать их, но это не совсем то, что мы хотим. В таких ситуациях нас спасает **Enum** или по-славянски — перечисления. На самом деле, это просто класс, наследующийся от `java.lang.Enum`. Объявляется это так:

```
<модификатор доступа> enum <имя>{  
  
}
```

в нашем случае можно сделать так:

```
public enum Days{  
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday;  
}
```

Им не нужно присваивать значения. Теперь пусть текущий день хранится в переменной `curDay = Days.Monday`. Напишем следующее

```
System.out.println(curDay == Days.Monday); // выведет true  
System.out.println(curDay == Days.Tuesday); // выведет false
```

На самом деле, когда мы просто перечисляли дни, мы создавали объекты класса Days, поэтому, если вдруг необходимо получить строковое представление текущего дня, то можно сделать так:

```
public enum Days{
    Monday («Понедельник»),
    Tuesday («Вторник»),
    Wednesday («Среда»),
    Thursday («Четверг»),
    Friday («Пятница»),
    Saturday («Суббота»),
    Sunday («Воскресенье»);

    private String title;

    private Days(String title){
        this.title = title;
    }
    public String getName(){
        return title;
    }
}
```

Теперь строчка `System.out.println(Days.Friday.getName());` выведет Пятница.

Заключение

На сегодня всё. Надеюсь, я ничего не забыл и не попутал. Если вы что-то не поняли или остались вопросы, то всегда есть гугл/одногруппники. Если вы нашли ошибку или есть какие-либо предложения по улучшению методичек, то писать сюда: <https://t.me/joinchat/uT7U0l0egD8zMDNi>

Все пособия под редакцией Никиты Кудлая распространяются на бесплатной основе, так как они помогают усваивать материал и созданы не для капитализации, но первая данная методичка имеет порядковый номер 50, а себестоимость составляет 5р. Если есть желание поддержать авторов (Kydlay inc.), то вот счет Сбера: 40817810552097130180

Главным автором данного пособия не является Н.Р. Кудлай, так что прошу указать пособие, за которое жертвуете свои кровные