



 **enhorse** / **java-interview** Public[Code](#) [Issues](#) 9 [Pull requests](#) [Actions](#) [Projects](#) [Security](#) [Insights](#) master ▾

...

**java-interview** / **oop.md****Cupcake-master** Correcting punctuation errors in the file oop.md History 5 contributors 387 lines (287 sloc) | 42 KB

...

[Вопросы для собеседования](#)

# ООП

- [Что такое ООП?](#)
- [Назовите основные принципы ООП.](#)
- [Что такое «инкапсуляция»?](#)
- [Что такое «наследование»?](#)
- [Что такое «полиморфизм»?](#)
- [Что такое «абстракция»?](#)
- [Что представляет собой «обмен сообщениями»?](#)
- [Расскажите про основные понятия ООП: «класс», «объект», «интерфейс».](#)
- [В чем заключаются преимущества и недостатки объектно-ориентированного подхода в программировании?](#)
- [Что подразумевают в плане принципов ООП выражения «является» и «имеет»?](#)
- [В чем разница между композицией и агрегацией?](#)
- [Что такое статическое и динамическое связывание?](#)

## Что такое ООП?

**Объектно-ориентированное программирование (ООП)** — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

- объектно-ориентированное программирование использует в качестве основных логических конструктивных элементов объекты, а не алгоритмы;
- каждый объект является экземпляром определенного класса
- классы образуют иерархии.

Программа считается объектно-ориентированной, только если выполнены все три указанных требования. В частности, программирование, не использующее наследование, называется не объектно-ориентированным, а программированием с помощью абстрактных типов данных.

Согласно парадигме ООП программа состоит из объектов, обменивающихся сообщениями. Объекты могут обладать состоянием, единственный способ изменить состояние объекта — послать ему сообщение, в ответ на которое, объект может изменить собственное состояние.

[к оглавлению](#)

## Назовите основные принципы ООП.

- **Инкапсуляция** - сокрытие реализации.
- **Наследование** - создание новой сущности на базе уже существующей.
- **Полиморфизм** - возможность иметь разные формы для одной и той же сущности.
- **Абстракция** - набор общих характеристик.
- **Посылка сообщений** - форма связи, взаимодействия между сущностями.
- **Переиспользование** - все что перечислено выше работает на повторное использование кода.

Это единственно верный порядок парадигм ООП, так как каждая последующая использует предыдущие.

[к оглавлению](#)

## Что такое «инкапсуляция»?

**Инкапсуляция** — это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя, открыв только то, что необходимо при последующем использовании.

**Цель инкапсуляции** — уйти от зависимости внешнего интерфейса класса (то, что могут использовать другие классы) от реализации. Чтобы малейшее изменение в классе не влекло за собой изменение внешнего поведения класса.

Представим на минутку, что мы оказались в конце позапрошлого века, когда Генри Форд ещё не придумал конвейер, а первые попытки создать автомобиль сталкивались с критикой властей по поводу того, что эти копящие монстры загрязняют воздух и пугают лошадей. Представим, что для управления первым паровым автомобилем необходимо было знать, как устроен паровой котёл, постоянно подбрасывать уголь, следить за температурой, уровнем воды. При этом для поворота колёс использовать два рычага, каждый из которых поворачивает одно колесо в отдельности. Думаю, можно согласиться с тем, что вождение автомобиля того времени было весьма неудобным и трудным занятием.

Теперь вернёмся в сегодняшний день к современным чудесам автопрома с коробкой-автоматом. На самом деле, по сути, ничего не изменилось. Бензонасос всё так же поставляет бензин в двигатель, дифференциалы обеспечивают поворот колёс на различающиеся углы, коленвал превращает поступательное движение поршня во вращательное движение колёс. Прогресс в другом. Сейчас все эти действия скрыты от пользователя и позволяют ему крутить руль и нажимать на педаль газа, не задумываясь, что в это время происходит с инжектором, дроссельной заслонкой и распределением. Именно сокрытие внутренних процессов, происходящих в автомобиле, позволяет эффективно его использовать даже тем, кто не является профессионалом-автомехаником с двадцатилетним стажем. Это сокрытие в ООП носит название инкапсуляции.

**Пример:**

```
public class SomePhone {  
  
    private int year;  
    private String company;
```

```
public SomePhone(int year, String company) {
    this.year = year;
    this.company = company;
}
private void openConnection(){
    //findComutator
    //openNewConnection...
}
public void call() {
    openConnection();
    System.out.println("Вызываю номер");
}

public void ring() {
    System.out.println("Дзынь-дзынь");
}
}
```

Модификатор `private` делает доступными поля и методы класса только внутри данного класса. Это означает, что получить доступ к `private` полям из вне невозможно, как и нет возможности вызвать `private` методы.

Соккрытие доступа к методу `openConnection`, оставляет нам также возможность к свободному изменению внутренней реализации этого метода, так как этот метод гарантированно не используется другими объектами и не нарушит их работу.

Для работы с нашим объектом мы оставляем открытыми методы `call` и `ring` с помощью модификатора `public`. Предоставление открытых методов для работы с объектом также является частью механизма **инкапсуляции**, так как если полностью закрыть доступ к объекту – он станет бесполезным.

[к оглавлению](#)

## Что такое «наследование»?

**Наследование** – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Класс, от которого производится наследование, называется **предком**, **базовым** или **родительским**. Новый класс – **потомком**, **наследником** или **производным** классом.

Представим себя, на минуту, инженерами автомобильного завода. Нашей задачей является разработка современного автомобиля. У нас уже есть предыдущая модель, которая отлично зарекомендовала себя в течение многолетнего использования. Всё бы хорошо, но времена и технологии меняются, а наш современный завод должен стремиться повышать удобство и комфорт выпускаемой продукции и соответствовать современным стандартам.

Нам необходимо выпустить целый модельный ряд автомобилей: седан, универсал и малолитражный хэтч-бэк. Очевидно, что мы не собираемся проектировать новый автомобиль с нуля, а, взяв за основу предыдущее поколение, внесём ряд конструктивных изменений. Например, добавим гидроусилитель руля и уменьшим зазоры между крыльями и крышкой капота, поставим противотуманные фонари. Кроме того, в каждой модели будет изменена форма кузова.

Очевидно, что все три модификации будут иметь большинство свойств прежней модели (старый добрый двигатель 1970 года, непробиваемая ходовая часть, зарекомендовавшая себя отличным образом на отечественных дорогах, коробку передач и т.д.). При этом каждая из моделей будет реализовать некоторую новую функциональность или конструктивную особенность. В данном случае, мы имеем дело с наследованием.

**Пример:** Рассмотрим пример создания класса смартфон с помощью наследования. Все беспроводные телефоны работают от аккумуляторных батарей, которые имеют определенный ресурс работы в часах. Поэтому добавим это свойство в класс беспроводных телефонов:

```
public abstract class WirelessPhone extends AbstractPhone {  
  
    private int hour;  
  
    public WirelessPhone(int year, int hour) {  
        super(year);  
        this.hour = hour;  
    }  
}
```

Сотовые телефоны наследуют свойства беспроводного телефона, мы также добавили в этот класс реализацию методов call и ring:

```
public class CellPhone extends WirelessPhone {
    public CellPhone(int year, int hour) {
        super(year, hour);
    }

    @Override
    public void call(int outputNumber) {
        System.out.println("Вызываю номер " + outputNumber);
    }

    @Override
    public void ring(int inputNumber) {
        System.out.println("Вам звонит абонент " + inputNumber);
    }
}
```

И, наконец, класс смартфон, который в отличие от классических сотовых телефонов имеет полноценную операционную систему. В смартфон можно добавлять новые программы, поддерживаемые данной операционной системой, расширяя, таким образом, его функциональность. С помощью кода класс можно описать так:

```
public class Smartphone extends CellPhone {

    private String operationSystem;

    public Smartphone(int year, int hour, String operationSystem) {
        super(year, hour);
        this.operationSystem = operationSystem;
    }

    public void install(String program){
        System.out.println("Устанавливаю " + program + " для " + operationSystem);
    }

}
```

Как видите, для описания класса Smartphone мы создали совсем немного нового кода, но получили новый класс с новой функциональностью. **Использование этого принципа ООП java позволяет значительно уменьшить объем кода**, а значит, и облегчить работу программисту.

[к оглавлению](#)

## Что такое «полиморфизм»?

**Полиморфизм** — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного и того же интерфейса для задания единого набора действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор языка программирования. Отсюда следует ключевая особенность полиморфизма — использование объекта производного класса, вместо объекта базового (потомки могут изменять родительское поведение, даже если обращение к ним будет производиться по ссылке родительского типа).

Любое обучение вождению не имело бы смысла, если бы человек, научившийся водить, скажем, ВАЗ 2106 не мог потом водить ВАЗ 2110 или BMW X3. С другой стороны, трудно представить человека, который смог бы нормально управлять автомобилем, в котором педаль газа находится левее педали тормоза, а вместо руля — джойстик.

Всё дело в том, что основные элементы управления автомобиля имеют одну и ту же конструкцию, и принцип действия. Водитель точно знает, что для того, чтобы повернуть налево, он должен повернуть руль, независимо от того, есть там гидроусилитель или нет. Если человеку надо доехать с работы до дома, то он сядет за руль автомобиля и будет выполнять одни и те же действия, независимо от того, какой именно тип автомобиля он использует. По сути, можно сказать, что все автомобили имеют один и тот же интерфейс, а водитель, абстрагируясь от сущности автомобиля, работает именно с этим интерфейсом. Если водителю предстоит ехать по немецкому автобану, он, вероятно выберет быстрый автомобиль с низкой посадкой, а если предстоит возвращаться из отдалённого маральника в Горном Алтае после дождя, скорее всего, будет выбран УАЗ с армейскими мостами. Но, независимо от того, каким образом будет реализовываться движение и внутреннее функционирование машины, интерфейс останется прежним.

**Полиморфная переменная**, это переменная, которая может принимать значения разных типов, а **полиморфная функция**, это функция, у которой хотя бы один аргумент является полиморфной переменной. Выделяют два вида полиморфных функций:

- **ad hoc**, функция ведет себя по разному для разных типов аргументов

(например, функция `draw()` — рисует по разному фигуры разных типов);

- **параметрический** функция ведет себя одинаково для аргументов разных типов (например, функция `add()` — одинаково кладет в контейнер элементы разных типов).

**Принцип в ООП**, когда программа может использовать объекты с одинаковым интерфейсом без информации о внутреннем устройстве объекта, **называется полиморфизмом**.

**Пример:**

Давайте представим, что нам в программе нужно описать пользователя, который может пользоваться любыми моделями телефона, чтобы позвонить другому пользователю. Вот как можно это сделать:

```
public class User {
    private String name;

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

    public void callAnotherUser(int number, AbstractPhone phone) {
        // вот он полиморфизм - использование в коде абстрактного типа AbstractPhone phone
        phone.call(number);
    }
}
```

Теперь опишем различные модели телефонов. Одна из первых моделей телефонов:

```
public class ThomasEdisonPhone extends AbstractPhone {

    public ThomasEdisonPhone(int year) {
        super(year);
    }

    @Override
    public void call(int outputNumber) {
        System.out.println("Вращайте ручку");
        System.out.println("Сообщите номер абонента, сэр");
    }

    @Override
```



```
    public void ring(int inputNumber) {  
        System.out.println("Телефон звонит");  
    }  
}
```

Обычный стационарный телефон:

```
public class Phone extends AbstractPhone {  
  
    public Phone(int year) {  
        super(year);  
    }  
  
    @Override  
    public void call(int outputNumber) {  
        System.out.println("Вызываю номер" + outputNumber);  
    }  
  
    @Override  
    public void ring(int inputNumber) {  
        System.out.println("Телефон звонит");  
    }  
}
```

И, наконец, крутой видеотелефон:

```
public class VideoPhone extends AbstractPhone {  
  
    public VideoPhone(int year) {  
        super(year);  
    }  
  
    @Override  
    public void call(int outputNumber) {  
        System.out.println("Подключаю видеоканал для абонента " + outputNumber);  
    }  
  
    @Override  
    public void ring(int inputNumber) {  
        System.out.println("У вас входящий видеовызов..." + inputNumber);  
    }  
}
```

Создадим объекты в методе main() и протестируем метод callAnotherUser:

```
AbstractPhone firstPhone = new ThomasEdisonPhone(1879);
AbstractPhone phone = new Phone(1984);
AbstractPhone videoPhone=new VideoPhone(2018);
User user = new User("Андрей");
user.callAnotherUser(224466,firstPhone);
// Вращайте ручку
//Сообщите номер абонента, сэп
user.callAnotherUser(224466,phone);
//Вызываю номер 224466
user.callAnotherUser(224466,videoPhone);
//Подключаю видеоканал для абонента 224466
```

Используя вызов одного и того же метода объекта user, мы получили различные результаты. Выбор конкретной реализации метода call внутри метода callAnotherUser производился динамически на основании конкретного типа вызывающего его объекта в процессе выполнения программы. В этом и заключается **основное преимущество полиморфизма** – выбор реализации в процессе выполнения программы.

В примерах классов телефонов, приведенных выше, мы использовали **переопределение методов** – прием, при котором изменяется реализация метода, определенная в базовом классе, без изменения сигнатуры метода. По сути, это является заменой метода, и именно новый метод, определенный в подклассе, вызывается при выполнении программы.

Обычно, при переопределении метода, используется аннотация @Override, которая подсказывает компилятору о необходимости проверить сигнатуры переопределяемого и переопределяющего методов.

[к оглавлению](#)

## Что такое «абстракция»?

**Абстрагирование** – это способ выделить набор общих характеристик объекта, исключая из рассмотрения частные и незначимые. Соответственно, **абстракция** – это набор всех таких характеристик.

Представьте, что водитель едет в автомобиле по оживлённому участку движения. Понятно, что в этот момент он не будет задумываться о химическом составе краски автомобиля, особенностях взаимодействия шестерёнок в коробке передач или влияния формы кузова на скорость (разве что, автомобиль стоит в глухой пробке и водителю абсолютно нечем заняться). Однако, руль, педали, указатель поворота он будет использовать регулярно.

#### Пример:

```
// Abstract class
abstract class Animal {
    // Abstract method (does not have a body)
    public abstract void animalSound();

    // Regular method
    public void sleep() {
        System.out.println("Zzz");
    }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
}

class MyMainClass {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```

[к оглавлению](#)

**Что представляет собой «обмен сообщениями»?**

Объекты взаимодействуют, посылая и получая сообщения. **Сообщение** — это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия. В ООП **посылка сообщения** (**вызов метода**) — это единственный путь передать управление объекту. Если объект должен «отвечать» на это сообщение, то у него должна иметься соответствующий данному сообщению метод. Так же объекты, используя свои методы, могут и сами посылать сообщения другим объектам. **Обмен сообщениями** реализуется с помощью динамических вызовов, что приводит к **чрезвычайно позднему связыванию** (extreme late binding).

Пусть требуется создать физическую модель, описывающую сталкивающиеся шары разных размеров. Традиционный подход к решению этой задачи примерно таков: определяется набор данных, описывающих каждый шар (например, его координаты, массу и ускорение); каждому шару присваивается уникальный идентификатор (например, организуется массив, значение индекса которого соответствует номеру шара), который позволит отличать каждый из шаров от всех других. Наконец, пишется подпрограмма с названием, скажем, `bounce`; эта процедура должна на основе номера шара и его начальных параметров соответствующим образом изменять данные, описывающие шар. В отличие от традиционного подхода объектно-ориентированная версия программы моделирует каждый из шаров посредством объекта. При этом объект, соответствующий конкретному шару, содержит не только его параметры, но и весь код, описывающий поведение шара при различных взаимодействиях. Так, каждый шар будет иметь собственный метод `bounce()`. Вместо того, чтобы вызывать подпрограмму `bounce` с аргументом, определяющим, скажем, шар №3, необходимо будет передать объекту «шар №3» сообщение, предписывающее ему выполнить столкновение.

[к оглавлению](#)

## Расскажите про основные понятия ООП: «класс», «объект», «интерфейс».

**Класс** — это способ описания сущности, определяющий состояние и поведение, зависящее от этого состояния, а также правила для взаимодействия с данной сущностью (контракт).

С точки зрения программирования класс можно рассматривать как набор данных (полей, атрибутов, членов класса) и функций для работы с ними (методов).

С точки зрения структуры программы, **класс** является сложным типом данных.

**Объект (экземпляр)** — это отдельный представитель **класса**, имеющий конкретное состояние и поведение, полностью определяемое классом. Каждый **объект** имеет конкретные значения атрибутов и методы, работающие с этими значениями на основе правил, заданных в классе.

**Интерфейс** — это набор методов класса, доступных для использования.

**Интерфейсом класса** будет являться набор всех его публичных методов в совокупности с набором публичных атрибутов. По сути, интерфейс специфицирует класс, чётко определяя все возможные действия над ним.

[к оглавлению](#)

## В чем заключаются преимущества и недостатки объектно-ориентированного подхода в программировании?

### Преимущества:

- Объектная **модель** вполне естественна, поскольку в первую очередь **ориентирована на человеческое восприятие мира**, а не на компьютерную реализацию.
- Классы позволяют проводить конструирование из полезных компонентов, обладающих простыми инструментами, что позволяет **абстрагироваться от деталей реализации**.
- **Данные и операции над ними образуют определенную сущность, и они не разносятся по всей программе**, как нередко бывает в случае процедурного программирования, а описываются вместе. **Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения**.
- **Инкапсуляция** **позволяет привнести свойство модульности**, что облегчает распараллеливание выполнения задачи между несколькими исполнителями и обновление версий отдельных компонентов.
- Возможность создавать расширяемые системы.
- **Использование полиморфизма** **оказывается полезным при:**
  - **Обработке разнородных структур данных**. Программы могут работать, не различая вида объектов, что существенно упрощает код. Новые виды могут быть добавлены в любой момент.
  - **Изменении поведения во время исполнения**. На этапе исполнения один

объект может быть заменен другим, что позволяет легко, без изменения кода, адаптировать алгоритм в зависимости от того, какой используется объект.

- Реализации работы с наследниками. Алгоритмы можно обобщить настолько, что они уже смогут работать более чем с одним видом объектов.
- Возможности описать независимые от приложения части предметной области в виде набора универсальных классов, или фреймворка, который в дальнейшем будет расширен за счет добавления частей, специфичных для конкретного приложения.
- **Повторное использование кода:**
  - Сокращается время на разработку, которое может быть отдано другим задачам.
  - Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
  - Когда некий компонент используется сразу несколькими клиентами, улучшения, вносимые в его код, одновременно оказывают положительное влияние и на множество работающих с ним программ.
  - Если программа опирается на стандартные компоненты, ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает использование.

#### Недостатки:

- В сложных иерархиях классов поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному классу.
- Код для обработки сообщения иногда «размазан» по многим методам (иначе говоря, обработка сообщения требует не одного, а многих методов, которые могут быть описаны в разных классах).
- Документирование классов - задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но и о том, в каком контексте он вызывается.
- Неэффективность и неэкономное распределения памяти на этапе выполнения (по причине издержек на динамическое связывание и проверки типов на этапе выполнения).

- Излишняя универсальность. Часто содержится больше методов, чем это реально необходимо текущей программе. А поскольку лишние методы не могут быть удалены, они становятся мертвым грузом.

[к оглавлению](#)

## Что подразумевают в плане принципов ООП выражения «является» и «имеет»?

«является» подразумевает наследование. «имеет» подразумевает ассоциацию (агрегацию или композицию).

[к оглавлению](#)

## В чем разница между композицией и агрегацией?

Ассоциация обозначает связь между объектами. Композиция и агрегация — частные случаи ассоциации «часть-целое».

Агрегация предполагает, что объекты связаны взаимоотношением «part-of» (часть). Композиция более строгий вариант агрегации. Дополнительно к требованию «part-of» накладывается условие, что экземпляр «части» может входить только в одно целое (или никуда не входить), в то время как в случае агрегации экземпляр «части» может входить в несколько целых.

Например, книга состоит из страниц, и мы не можем вырвать страницу из книги и вложить в другую книгу. Страницы четко привязаны к конкретной книге, поэтому это композиция. В тоже время мы можем взять и перенести книгу из одной библиотеки в другую - это уже агрегация.

[к оглавлению](#)

## Что такое статическое и динамическое связывание?

Присоединение вызова метода к телу метода называется связыванием. Если связывание проводится компилятором (компоновщиком) перед запуском программы, то оно называется статическим или ранним связыванием (early binding).

В свою очередь, **позднее связывание** (*late binding*) это связывание, **проводимое непосредственно во время выполнения программы**, в зависимости от типа объекта. **Позднее связывание** также называют **динамическим** (*dynamic*) или **связыванием на стадии выполнения** (*runtime binding*). В языках, реализующих позднее связывание, **должен существовать механизм определения фактического типа объекта во время работы программы**, для вызова подходящего метода. Иначе говоря, компилятор не знает тип объекта, но механизм вызова методов определяет его и вызывает соответствующее тело метода. **Механизм позднего связывания** зависит от конкретного языка, но нетрудно предположить, что для его реализации в объекты должна включаться какая-то дополнительная информация.

Для всех методов Java используется механизм **позднего (динамического) связывания**, если только метод не был объявлен как **final** (приватные методы являются **final** по умолчанию).

[к оглавлению](#)

## Источники

- [DevColibri](#)
- [Хабрахабр](#)
- [Википедия](#)

[Вопросы для собеседования](#)