Здесь мы собрали всю теорию из урока. Чтобы конспект всегда был у вас под рукой, скачайте его в PDF-формате [по ссылке](https://drive.google.com/file/d/1sH6Zoa7jrg9_usaHEwVFGnMI6Gf57Mb0/view).

**Наследование**

Под наследованием в Java подразумевают как определенный **механизм** создания нового класса, так и **принцип ООП**, который предписывает пользоваться этим механизмом.

**Наследование в Java** — механизм (способ) создания нового класса, который может использовать код и функционал уже существующего класса без повторения этого кода.

Для примера рассмотрим приложение для учета печатной продукции издательства. Сначала нам нужно реализовать классы для двух типов продукции — книг и журналов.

// Cоздаем класс Книга

class Book {

// Cтрока Название

private String name;

// Cтрока Имя автора

private String authorName;

// Cтрока Количество страниц

private int pageQuantity;

// Cтрока Содержание

private String content;

public void printContent() {

System.out.println(content);

}

// Для краткости опускаем геттеры и сеттеры

}

// Cоздаем класс Журнал

class Magazine {

// Cтрока Название

private String name;

// Cтрока Количество страниц

private int pageQuantity;

// Cтрока Содержание

private String content;

public void printContent() {

System.out.println(content);

}

// Для краткости опускаем геттеры и сеттеры

}

Классы

Book

 и

Magazine

 практически дублируют друг друга. У обоих классов есть поля с названием, количеством страниц, содержанием. Они различаются только полем

authorName

, так как в книгах принято указывать автора, а в журналах — нет.

В примере выше мы дублировали код, что снижает удобочитаемость и отнимает дополнительное время у разработчика. Избежать этого поможет наследование.

Механизм наследования позволяет вынести общий для двух объектов код в отдельный класс.

Объявим класс

PrintedProduct

, который будет хранить общий код:

class PrintedProduct {

private String name;

private int pageQuantity;

private String content;

public void printContent() {

System.out.println(content);

}

// Геттеры и сеттеры

}

С помощью механизма наследования классы (

Book

 и

Magazine

) могут использовать (наследовать) поля и методы класса

PrintedProduct

.

Ключевое слово **extends** *(от англ. extend — «расширять»)* указывает, что один класс является подклассом (потомком) другого класса. Это позволяет новому классу использовать все поля и методы класса-родителя, словно получая их по наследству.

Иными словами, подклассы (в нашем случае

Book

 и

Magazine

) умеют всё то же самое, что и класс

PrintedProduct

. При этом в подклассы можно добавить что-то еще. Для примера добавим в класс

Book

 строку с именем автора:

// Класс Book является подклассом (наследником) класса PrintedProduct,

// то есть он наследует все его поля и методы,

// а также может добавлять свои.

class Book extends PrintedProduct {

// Это поле для хранения имя автора книги.

// Оно доступно только в классе Book, так как объявлено как private.

private String authorName;

}

// Класс Magazine также подкласс класса PrintedProduct,

// У него пока нет собственных полей и методов, но их можно добавить.

class Magazine extends PrintedProduct {

}

Теперь мы можем создать объект любого из двух этих классов и вызвать метод

printContent

, который они унаследовали от родителя.

Следует знать, что **private-члены класса** (поля и методы) **не наследуются**. Это значит, что классы-наследники не будут иметь доступа к полям и методам, помеченным как

private

.

Когда мы инициализируем поле

content

 через конструктор, единственный способ обратиться к нему из кода класса

Book

 или

Magazine

 будет через геттеры и сеттеры.

Без сеттера класс «Книга» не сможет изменять поле

content

 , так как оно находится в классе-родителе и управляется исключительно им.

**Инициализация переменных, к которым нет доступа**

Java неявно вкладывает объект класса-родителя в ваш объект и присваивает его переменной

super

 (по аналогии с

this

, которая ссылается на ваш текущий объект).

По переменной

super

, соответственно, можно вызвать конструктор родителя.

Помните, что строка с вызовом конструктора родителя **всегда есть** в конструкторе вашего класса-наследника. Если вы ее не пишете, то пишет Java.

Изменим наш код, добавив в него конструкторы:

public class PrintedProduct {

private String name;

private int pageQuantity;

private String content;

public PrintedProduct(String name, int pageQuantity, String content) {

this.name = name;

this.pageQuantitiy = pageQuantity;

this.content = content;

}

public void printContent() {

System.out.println(content);

}

// Геттеры и сеттеры

}

public class Book extends PrintedProduct {

private String authorName;

public Book(String name, int pageQuantity, String authorName, String content) {

super(name, pageQuantity, content);

this.authorName = authorName;

}

// Геттер и сеттер для authorName

}

public class Magazine extends PrintedProduct {

public Magazine(String name, int pageQuantity, String content) {

super(name, pageQuantity, content);

}

}

Первый вариант кода тоже можно назвать рабочим. Так как в нем нет конструктора, Java сама создала бы пустой конструктор, а в нем вызвала пустой конструктор класса-родителя. Все поля пришлось бы устанавливать через сеттеры.

Сгенерированный конструктор выглядел бы так:

public Book() {

super();

}

Теперь создадим книгу и журнал:

Book b = new Book("War and Peace", 1000, "Lev Tolstoy", "War and Peace content");

Magazine m = new Magazine("Java Magazine", 100, "Lots of information about Java");

b.printContent(); // Будет напечатано в консоль: War and Peace content

m.printContent(); // Будет напечатано в консоль: Lots of information about Java

Если наследники должны иметь внутри себя логику по работе с содержимым полей родителя, то эти поля (или методы) должны быть помечены модификаторами

default

 (модификатор может отсутствовать, если родитель и наследник лежат в одном пакете) или

protected

.

**Переопределение и перегрузка**

**@Override и переопределение методов**

В Java необязательно пользоваться именно той версией метода, которая описана в родителе. Если у вас есть доступ к этому методу (он объявлен в родителе **не** как

private

), его можно **переписать**.

Для этого нужно объявить точно такой же (по сигнатуре) метод в наследнике и пометить его флагом

@Override

. Это позволит IDEA и читателям кода (в том числе и вам) сразу понять, что этот метод есть где-то в родителе, а вы его просто переписываете, или переопределяете.

**Переопределение** — процесс, в котором класс-потомок меняет поведение метода, унаследованного от класса-родителя.

Рассмотрим пример:

public class A {

public void doSmth() {

System.out.println("A");

}

}

public class B extends A {

@Override

public void doSmth() {

System.out.println("B");

}

// Так как метода doSmth2 нет в A, мы не должны помечать его @Override

public void doSmth2() {

System.out.println("B2");

}

}

A a = new A();

A b1 = new B();

B b2 = new B();

a.doSmth(); // Выведет A

b1.doSmth(); // Выведет B

b1.doSmth2(); // Будет ошибка, так как метода doSmth2() нет в классе A

b2.doSmth(); // Выведет B

b2.doSmth2(); // Выведет B2

Переопределять методы обязательно в следующих случаях:

* когда вам нужно изменить их логику (родительская логика не подходит);
* когда переписываемый метод в объекте является абстрактным (то есть не имеет тела).

**Перегрузка методов**

В Java есть понятие, похожее на переопределение, — **перегрузка метода**.

**Перегрузка метода** — это прием, который позволяет вызывать один и тот же метод с разными параметрами и получать совершенно разную обработку этих параметров.

Этот прием используется редко, но его надо четко отличать от **переопределения метода**.

Представим, что нам нужно написать метод, который применяет разную логику в зависимости от того, какой поступил тип данных. Например, нужно получить число и напечатать его значение в квадрате (во второй степени). Есть два варианта того, как можно это реализовать:

1. Объявить один метод с двумя параметрами и всегда передавать в него один из них.
2. Перегрузить метод, написав две его вариации под разные типы параметров.

**Первый вариант**

В случае когда нам нужно передать строку, нам придется также передать какое-то дефолтное значение в метод для числа — чтобы показать методу, что он **не должен работать с этим числом**. Таким значением может быть

−1

. В случае с отсутствием строки дефолтным значением будет

null

.

public void printSquare(int num, String numStr) {

if (num != -1) {

System.out.println(num \* num);

} else {

int parsedNum = Integer.parseInt(numStr);

System.out.println(parsedNum \* parsedNum);

}

}

int n1 = 5;

String n2 = "6";

printSquare(n1, null); // Напечатает 25

printSquare(-1, n2); // Напечатает 36

Если пользователь этого приложения забудет, что нужно передавать

−1

, — всё сломается.

**Второй вариант**

Мы можем объявить два одинаковых метода (одно имя, один возвращаемый тип), но с разным набором параметров.

Это называется **перегрузка** или **overload**.

public void printSquare(int num) {

System.out.println(num \* num);

}

public void printSquare(String str) {

int parsedNum = Integer.parseInt(str);

System.out.println(parsedNum \* parsedNum);

}

int n1 = 5;

String n2 = "6";

printSquare(n1); // Напечатает 25

printSquare(n2); // Напечатает 36

**Запрет наследования**

Мы можем как реализовать функционал родительского класса в классах наследниках, так и запретить наследование. Тогда часть функционала родителя **не будет переходить** потомкам.

Чтобы наследование от определенного класса было невозможным, нужно при его объявлении указать ключевое слово

final

:

public final class Book {

private String name;

private String authorName;

private int pageQuantity;

private String content;

// Геттеры и сеттеры

}

// Мы не можем унаследоваться от данного класса

**Также ключевое слово**

final

**может использоваться**:

1. **С методами** — такие методы невозможно переопределить.
2. **С переменными** — значение таких переменных нельзя изменить после их первоначальной инициализации.

**Абстрактные классы**

Вернемся к нашему приложению по учету печатной продукции. Что будет, если кто-то захочет создать объект

PrintedProduct

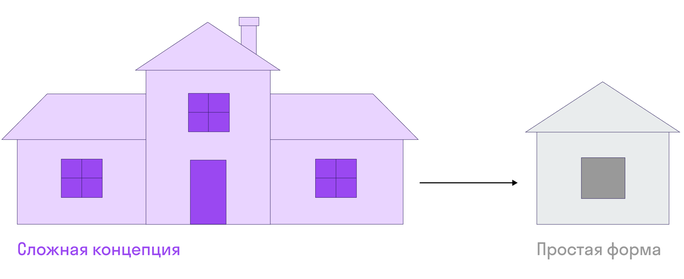
? Ведь у нас не должно быть абстрактных печатных объектов без указания их типа («газета», «журнал» или «книга»).

В этой ситуации нужно запретить создавать объект класса

PrintedProduct

 и указать, что его можно использовать только в качестве абстрактного носителя общего кода для реализаций.

Абстракция представляет сложную концепцию в более простой форме.



В этом нам поможет **ключевое слово**

abstract

. Добавим его к объявлению класса

PrintedProduct

. Теперь при попытке создать

new PrintedProduct

 Java скажет, что класс является абстрактным и его нельзя создать.

Но на этом особенности абстрактных классов не заканчиваются.

Абстрактный класс позволяет объявить внутри него **абстрактные методы** — методы, которые мы объявляем, но не пишем для них реализацию.

Это нужно для ситуаций, когда есть общий код с вызовом метода, реализация которого может различаться в зависимости от наследников.

Рассмотрим абстрактный класс, который управляет телевизором с абстрактным методом включения телевизора, и две реализации этого метода в классах-наследниках.

Так выглядит абстрактный класс:

// Логика по включению первого канала

public abstract class TVController {

public void turnOnFirstChannel() {

turnOnTV();

}

public abstract void turnOnTV();

}

В примерах ниже вы встретите **псевдокод** — текстовое описание фрагмента кода или алгоритма. Он часто используется в учебниках, так как с его помощью удобно показать суть работы алгоритма, опустив несущественные для понимания детали.

И две реализации абстрактного класса:

public class SmartSpeaker extends TVController {

@Override

public void turnOnTV() {

while (пользователь не сказал фразу "включить ТВ")) {

ждать;

}

}

}

public class RemoteController extends TVController {

@Override

public void turnOnTV() {

while (пользователь не нажал кнопку включения)) {

ждать;

}

}

}

Теперь мы можем создать функционал *«работать с обеими реализациями»* по ссылке типа

TVController

:

TVController[] controllers = new TVController[] {

new SmartSpeaker(),

new RemoveController();

};

for (TVController controller : controllers) {

controller.turnOnFirstChannel();

}

Независимо от способа управления первый канал включится на всех телевизорах, чьи пульты лежат в массиве. Реализация

turnOnTV

 будет браться из конкретного объекта с реализацией метода.

Важно запомнить, что обычный класс **не может** иметь абстрактных методов. Это значит, что любой класс, который наследует абстрактный класс с абстрактными методами, обязан их реализовать внутри себя. Или тоже стать абстрактным.

**Ключевое слово Static**

Нередко возникает потребность создать такой член класса, который бы не зависел от конкретных объектов. Такими членами могут быть переменные, методы или вложенные классы.

Если при их создании указать ключевое слово

static

, то они не будут зависеть от определенного объекта и станут статическими.

**Статические переменные**

Статическая переменная в Java используются, если нужна общая переменная для всех объектов класса. Она хранит информацию, которая должна быть доступна и единообразна для всех экземпляров класса.

**Статические переменные** — переменные, которые объявлены с помощью ключевого слова

static

 и принадлежат не отдельному объекту класса, а всему классу в целом.

Давайте на примере разберем, чем отличаются статические переменные от обычных.

public class SomeStaticClass {

int number1;

static int number2;

public static void main(String[] args) {

SomeStaticClass someStaticClass = new SomeStaticClass();

System.out.println(someStaticClass.number1);

System.out.println(number2);

}

}

Чтобы обратиться к нестатической переменной

number1

, нам нужен объект класса

SomeStaticClass

. А статическая переменная

number2

 этого не требует, так как принадлежит классу, в котором мы работаем.

Однако если мы хотим обратиться к статической переменной из другого класса, нам придется это сделать через имя класса, которому она принадлежит, —

SomeStaticClass.number2

 — или через любой объект этого же класса.

Не рекомендуется обращаться к члену другого класса через объект. Так можно запутаться, какие поля статические, а какие — нет.

Обратимся к переменной через имя ее класса:

public class SomeStaticClassDemo {

public static void main(String[] args) {

SomeStaticClass someStaticClass1 = new SomeStaticClass();

SomeStaticClass someStaticClass2 = new SomeStaticClass();

// Обращение к статической переменной через имя класса

System.out.println(SomeStaticClass.number2);

// Обращение к статической переменной через конкретный объект

System.out.println(someStaticClass1.number2);

someStaticClass1.number2 = 3;

someStaticClass2.number2 = 4;

System.out.println(someStaticClass1.number2);

System.out.println(someStaticClass2.number2);

}

}

Получим следующий результат:

0

0

4

4

Приведем стандартный пример использования статической переменной, которая подсчитывает, сколько объектов класса было создано.

Определим и инициализируем в классе статическую переменную

counter

, которая будет увеличиваться на

1

 при создании каждого нового объекта.

Для этого используем конструктор:

public class SomeClass {

// Инициализируем статическую переменную counter начальным значением 0

static int counter = 0;

// Создаем конструктор класса SomeClass

public SomeClass() {

// Увеличиваем значение статической переменной counter на 1

// при каждом вызове конструктора

counter++;

}

}

public class SomeClassTest {

public static void main(String[] args) {

SomeClass object1 = new SomeClass();

SomeClass object2 = new SomeClass();

System.out.println("Количество созданных объектов: "

+ SomeClass.counter);

}

}

Получаем такой результат:

Количество созданных объектов: 2

**Статические методы**

Статические методы, как и статические переменные, создаются с использованием ключевого слова

static

. Для вызова таких методов не нужен конкретный объект класса, в котором создан сам метод.

У статических методов есть ограничения:

* Они имеют доступ **только** к статическим переменным.
* Из такого метода можно вызывать **только** статические методы.
* Эти методы **не могут** использовать ссылки на

this

 или

super

.

Приведем пример использования статических методов:

public class SomeStaticClass2 {

static int number1 = 3;

int number2;

public void someMethod() {

System.out.println("Нестатический метод");

}

static void someStaticMethod(int localVar) {

System.out.println("localVar = " + localVar);

System.out.println("staticVar = " + number1);

// Нельзя обратиться к нестатической переменной из статического метода

// System.out.println("nonStaticVar = " + number2);

}

public static void main(String[] args) {

someStaticMethod(42);

// Нельзя обратиться к нестатическому методу без указания объекта

// someMethod();

SomeStaticClass2 someStaticObj = new SomeStaticClass2();

someStaticObj.someMethod();

someStaticObj.someStaticMethod(67);

}

}

public class SomeStaticClass2Test {

public static void main(String[] args) {

SomeStaticClass2.someStaticMethod(42);

}

}

**Статический блок**

Также в Java есть такая сущность, как статический блок. Внутри этого блока производятся те операции, которые нужно произвести единожды при загрузке программы.

**Статический блок** — часть кода, которая выполняется только один раз в момент загрузки класса (до создания любых объектов этого класса).

Для инициализации такого блока нужно написать ключевое слово

static

, а дальше в фигурных скобках обозначить все необходимые вычисления:

import java.util.Scanner;

public class SomeStaticClass3 {

static String string;

static {

System.out.println("Инициализация статического блока");

Scanner scanner = new Scanner(System.in);

string = scanner.nextLine();

}

public static void main(String[] args) {

System.out.println("String = " + string);

}

}

**Java static import**

Чтобы воспользоваться статическими членами классов, нужно указать ссылку на класс, которому они принадлежат.

Например, для использования статических методов класса

Math

 или его констант мы должны прописать класс

Math

:

public class SomeStaticClass4 {

public static void main(String[] args) {

double value = Math.cos(Math.PI \* 4);

System.out.println(value);

}

}

Также можно импортировать статические члены класса и использовать их, не прописывая имя класса, которому они принадлежат. Для импорта таких членов используется конструкция

import static

, а далее указывается полное имя класса и название константы или метода:

package oop;

import static java.lang.Math.PI;

import static java.lang.Math.cos;

public class SomeStaticClass5 {

public static void main(String[] args) {

double value = cos(PI \* 4);

System.out.println(value);

}

}

**Класс Object**

Всякий раз, когда вы создаете класс, Java автоматически делает его наследником класса Object. Простыми словами, все классы в Java — это «дети» класса Object.

**Class Object** в Java — базовый (родительский) класс для всех других классов. В нём определены некоторые методы, которые наследуются всеми другими классами.

Это означает, что **любой объект в Java имеет методы класса**

Object

. Хоть в объявлениях классов это и не указано, все они неявно наследуют класс

Object

 и реализуют его методы. Некоторые из методов можно переопределить.

**Методы класса Object в Java**

* protected Object clone()

 — создает новый объект (клон) на основе существующего.

* public boolean equals(Object obj)

 — определяет, равен ли один объект другому.

* public final Class<?> getClass()

 — получает класс объекта во время выполнения.

* public int hashCode()

 — возвращает хеш-код объекта.

* public String toString()

 — возвращает строковое представление объекта.

Подробнее о сборщике мусора и механизмах многопоточного программирования, которые перечислены ниже, вы узнаете в других уроках и модулях курса.

* protected void finalize()

 — вызывается сборщиком мусора перед удалением неиспользуемого объекта.

* public final void notify()

 — возобновляет работу одного из потоков, который был поставлен на ожидание для конкретного объекта.

* public final void notifyAll()

 — возобновляет работу всех потоков, которые были поставлены в состояние ожидания данного объекта.

* public final void wait()

 ,

public final void wait(long timeout)

 ,

public final void wait(long timeout, int nanos)

 — ожидают другого потока исполнения.

Рассмотрим подробнее некоторые из методов.

**Методы equals() и hashCode()**

Метод

equals()

 сравнивает объекты между собой и возвращает результат типа

boolean

. Данный метод необходимо переопределять в своих классах. По умолчанию этот метод сравнивает объекты через оператор

==

, и мы можем получить

true

, только если обе ссылки указывают на один и тот же объект.

Реализовывать данный метод в своих классах нужно вместе с другим методом класса

Object

 —

hashCode()

, который возвращает хеш-код (уникальный числовой код любого объекта).

Методы

equals()

 и

hashCode()

 работают вместе по определенным правилам:

1. Если два объекта равны (т. е. метод

equals()

 возвращает

true

), у них должен быть один и тот же хеш-код.

1. Если метод

hashCode()

 вызывается несколько раз на одном и том же объекте, каждый раз он должен возвращать одно и то же число.

1. Хеш-код может быть одним и тем же у двух разных объектов.

Последний пункт связан с тем, что хеш-код ограничен 32 битами. При этом количество создаваемых объектов не ограничено ничем, кроме объема памяти, доступного приложению.

Переопределить оба метода можно автоматически в среде разработки, а можно и вручную, как в этом примере:

import java.util.Objects;

class Book {

private String name;

private String authorName;

// Геттеры и сеттеры

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Book book = (Book) o;

return Objects.equals(name, book.name) && Objects.equals(authorName, book.authorName);

}

@Override

public int hashCode() {

return Objects.hash(name, authorName);

}

}

**Метод toString()**

Для корректной работы программы зачастую нужно использовать строковое представление того или иного объекта.

Для этой цели в Java в классе

Object

 есть метод

toString()

. Переопределять данный метод в Java принято в собственных классах, чтобы строковое представление содержало в себе все необходимые данные.

public class Human {

private String name;

private int age;

...

@Override

public String toString() {

return "Human{"

+ "name='" + name + '\''

+ ", age=" + age

+ '}';

}

}

Если мы передаем в параметр метода

System.out.println()

 экземпляр класса, к нему по умолчанию неявно вызывается метод

toString()

:

public class HumanTest {

public static void main(String[] args) {

Human human = new Human("Смирнов Роман Сергеевич", 23);

System.out.println(human);

}

}

Вот что получим в результате:

Human{name='Смирнов Роман Сергеевич', age=23}

Ваш вопрос будет первым