JAVA

The best ever Metodichka for Java-backender

IT MENTOR

СОДЕРЖАНИЕ

IT MENTOR

IT MENTOR

JAVA CORE

ПРИМИТИВЫ, ЦИКЛЫ, МАССИВЫ, СТРОКИ

IT MENTOR

УСТАНОВКА ИНСТРУМЕНТОВ РАЗРАБОТЧИКА

IDE — интегрированная среда разработки, или программа, в которой можно писать, тестировать и запускать код, написанный разработчиком.

IntelliJ IDEA — интегрированная среда разработки, которую создала компания JetBrains. Эта программа подходит для написания кода на разных языках, и на ней можно писать код на Java.

Git — инструмент (или система контроля версий), который помогает отслеживать, фиксировать и отменять какие-либо изменения в коде.

Репозиторий — это хранилище кода. По умолчанию код хранится на компьютере, его копия отправляется на удаленный репозиторий, например GitHub, или другое веб-приложение (сайт).

GitHub/Bitbucket/GitLab — веб-приложения (сайты), в которых можно хранить, изменять, обновлять код. Это удаленные репозитории по хранению кода, которые помогают организовывать командную работу, например ревью (проверку) кода

Commit — запись в истории изменений, которая содержит commit message (описание внесенных изменений), а также обновленные файлы, подвергшиеся изменениям.

Push — отправка новых коммитов на удаленный репозиторий, например GitHub.

Ветка (branch) — обособленная копия проекта, в которой хранится код и история его изменений. В ветке можно изменять, удалять код, при этом эти изменения не затронут код в других ветках проекта. Веток может быть сколько угодно, они создаются отдельно друг от друга.

Master — основная ветка, которая создается тогда, когда создается сам репозиторий (хранилище). В master хранится самый проверенный и стабильный код. Обновить его можно только через слияние с другими ветками, которые были созданы от него же (копии). Напрямую в master что-то обновлять или менять можно только в крайних случаях.

Pull request — запрос на слияние (иногда называется merge request) двух веток, с помощью которого можно узнать, какие различия есть между двумя ветками (проектами).

Merge — непосредственно сам процесс слияния (объединения) двух веток. При этом ветка-получатель получает отсутствующие коммиты от ветки-отправителя.

Pull — обновление локальной ветки за счет коммитов на удаленном репозитории.

IT MENTOR

ПЕРЕМЕННЫЕ И УСЛОВНЫЕ ОПЕРАТОРЫ

ПЕРЕМЕННЫЕ

Переменная в Java — это контейнер, в котором может храниться некоторое значение данных для дальнейшего использования в программе.
Типы переменных в Java распределены на две категории: примитивные (простые) и ссылочные (объектные).
Ссылочные типы — это массивы, классы, интерфейсы и т. д.

Целые числа (целочисленные). Эта группа включает в себя типы данных **byte**, **short**, **int** и **long**, представляющие целые числа со знаком.**Числа с плавающей точкой (вещественные).** Эта группа включает в себя типы данных **float** и **double**, представляющие числа с точностью до определенного знака после десятичной точки.

Символы. Эта группа включает в себя тип данных **char**, представляющий символы, например буквы и цифры, из определенного набора.
Логические значения. Эта группа включает в себя тип данных **boolean**, специально предназначенный для представления логических значений.

Типы данных		диапазон значений	Объем памяти
Целочисленные	byte	–128 …127	1 байт
	short	–32 768 … 32 767	2 байта
	int	–2 147 483 648 … 2 147 483 647	4 байта
	long	–9 223 372 036 854 775 808 … 9 223 372 036 854 775 807	8 байт
С плавающей точкой	float	–3.4e+38 … 3.4e+38	4 байта
	double	–1.7e+308 … 1.7e+308	8 байт
Символы	char	0…65 535	2 байта
Логические	boolean	true или false	*

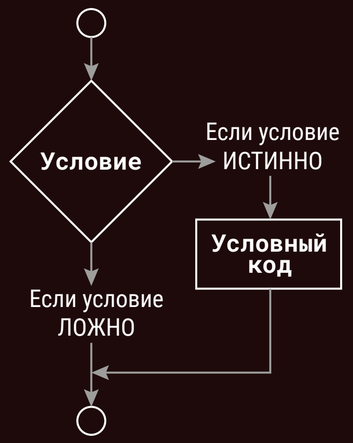
Дефолтные значения для примитивных числовых типов равно **нулю** (*`0` для целочисленных, `0.0d` и `0.0f` для double и float соответственно, для boolean — `false`, а для типа char — `\u0000`*).

IT MENTOR

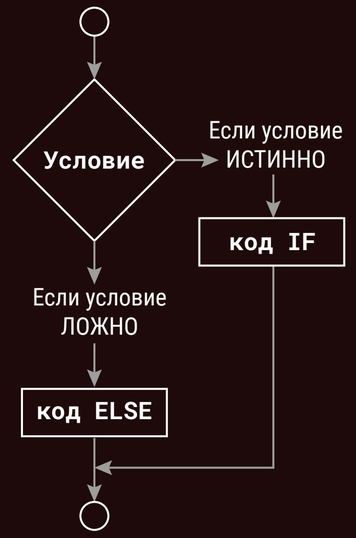
УСЛОВНЫЙ ОПЕРАТОР

Условные операторы — конструкции, которые, проверяя условия, запускают в действие необходимый код.

Оператор if



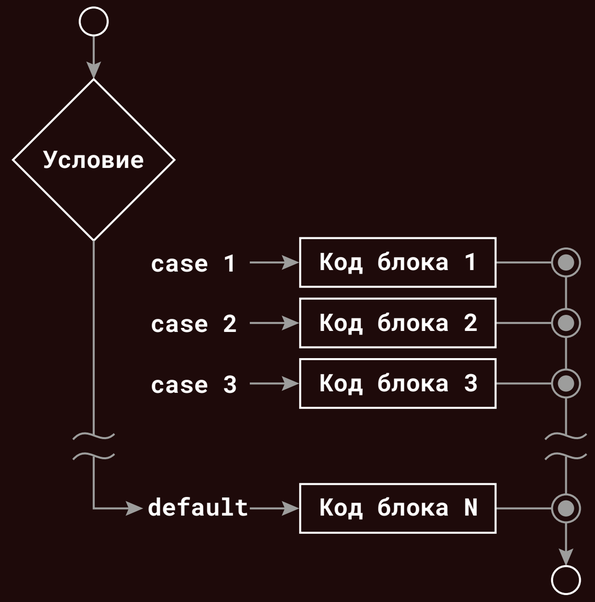
Оператор if else



При использовании операторов **if**, **else if**, **else** в Java есть несколько моментов, которые необходимо иметь в виду:

- if может не иметь ни одного или иметь один else, который должен идти после любого if.
- После того как if... else будет успешно выполнен, из оставшихся else if или else ничего не будет выполнено (проверено).

Оператор switch



Оператор **switch** проверяет переменную на равенство в отношении списка значений. Каждое значение называется case, и переменная, переключаясь, проверяется для каждого case.

Переменные, которые используются в операторе switch, могут быть только целые числа (**byte**, **short**, **integer**, **char**), **string**, **enum**.

IT MENTOR

Составные арифметические операции с присваиванием:	
`+=`	сложение с присваиванием
`-=`	вычитание с присваиванием
`*=`	умножение с присваиванием
`/=`	деление с присваиванием
`%=`	деление по модулю с присваиванием

Операторы сравнения:	
`==`	равно
`!=`	не равно
`>`	больше
`<`	меньше
`>=`	больше или равно
`<=`	меньше или равно

Логические операторы:	
`&`	Логическая операция И (AND), или конъюнкция
`|`	Логическая операция ИЛИ (OR), или дизъюнкция
`^`	Логическая операция, исключающая ИЛИ (XOR)
`!`	Логическая унарная операция НЕ (NOT)
`||`	Укороченная логическая операция ИЛИ (short-circuit)
`&&`	Укороченная логическая операция И (short-circuit)
`==`	Равенство
`!=`	Неравенство

Операнд — *значение или переменная, над которой производится операция.*

Логические операторы **`OR`**, **`AND`**, **`XOR`** являются **бинарными** — они требуют два оператора.

`NOT` — это унарный логический оператор, только один оператор участвует в операции.

A	B	A|B
false	false	false
true	false	true
false	true	true
true	true	true

A & B	A ^ B	!A
false	false	true
false	true	false
false	true	true
true	false	false

IT MENTOR

ЦИКЛЫ

Цикл — конструкция кода, которая повторяет одно и то же действие несколько (столько, сколько нам потребуется) раз.

Итерация — один повтор какого-то действия.

Инкремент — операция, которая увеличивает значение переменной.

Цикл **for** цикл помогает выполнять многократно повторяющиеся действия.

```
for ( int i=0; i<10; i= i + 1 ){
```

Блок 1. *Объявление переменной*

Блок 2. *Условие, которое должно соблюдаться*

Блок 3. *Действие, которое должно выполняться*

Цикл **while** проверяет истинность условий: пока условие истинно, код цикла выполняется.

Цикл **do-while**. Сначала совершается действие, а затем проверяется условие.

- × Если переменная со счетчиком в цикле нужна, и она меняется строго в конце итерации, удобнее **for**.
- × Если переменная со счетчиком цикла не нужна или она может измениться в любой момент, то разработчики предпочитают **while**.

Оператор **break** прерывает цикл в любой ситуации и независимо от условия. Его используют даже в тех случаях, когда условие выхода из цикла указано и цикл может завершиться самостоятельно, без ручного прерывания с помощью **break**.

Оператор **continue** принудительно пропускает выполнение шага цикла в зависимости от условий.

IT MENTOR

МАССИВЫ

Массив — это структура данных, ссылочный тип данных, которая позволяет хранить несколько значений одного типа.

Массив хранит в себе данные того типа, которым он **инициализирован**.

Размер массива изменять **нельзя**.

ОСНОВНЫЕ ОПЕРАЦИИ С МАССИВАМИ

СОЗДАЕМ МАССИВ С ПОМОЩЬЮ СЛОВА NEW:

```
int[] arr = new int[10];
```

Порядковый номер элемента массива называется **индексом**.

Первый элемент массива находится в ячейке под номером **0**.

ЗАПИСЫВАЕМ ЗНАЧЕНИЯ ЭЛЕМЕНТОВ МАССИВА:

```
int[] arr = new int[10]; // Создали массив

arr[0] = 5; // Положили в ячейку 0 значение 5
```

В массив можно положить значение из ранее созданной переменной:

```
int i = 10; // Создали переменную

arr[1] = i; // Положили значение переменной i в первую ячейку
```

Чтобы получить элемент из ячейки по известному нам номеру, напишем:

```
arr[5] = 10;
// Положить в 5-ю ячейку значение 10

int i = arr[5];
// Создать переменную
// Инициализировать ее значение из 5-й ячейки массива arr
```

Свойство length возвращает длину массива, то есть количество элементов.

```
int[] arr = new int[10]; // Создали массив

int arrSize = arr.length;
// Присвоили переменной arrSize значение длины массива (=10)
```

Важно: значение **arr.length** всегда будет иметь индекс последнего элемента **+1**.

(последний элемент лежит по индексу в ячейке 9, а length хранит в себе 10)

IT MENTOR

ОБЪЯВЛЯЕМ И СРАЗУ ИНИЦИАЛИЗИРУЕМ МАССИВ:

```
int[] arr = new int[]{1, 2, 3};
```

Размер массива вычисляется на основе количества значений из {}.

Массив при создании формирует не просто ячейки типов,
но и заполняет их стандартными значениями.

Тип переменной	Значение по умолчанию
byte	**0**
short	**0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**
char	**0**
String (или другой объект)	**null**
boolean	**false**

ПЕЧАТАЕМ МАССИВ:

```
int[] arr = new int[10];

for(int index = 0;index<arr.length;index++){
    if(index == arr.length-1){
        System.out.println(arr[index]);
        break;
    }
    System.out.print(arr[index]+" ");
}
```

Если формат вывода элементов массива не принципиален, удобнее воспользоваться методом **Arrays.toString**. С его помощью мы печатаем массив без дополнительного написания циклов.

```
int[] arr = new int[10];
System.out.println(Arrays.toString(arr));
```

При расширении массива Java не может гарантировать, что следующие ячейки в памяти не заняты. Поэтому длину массива **нельзя** изменять.

IT MENTOR

СРАВНИВАЕМ МАССИВЫ С ПОМОЩЬЮ ЦИКЛА FOR:

Массивы являются объектами, поэтому сравнивать их через знак равенства (==) нельзя.

```
int[] arr = {1, 2, 3};
int[] arr2 = {1, 2, 3};
boolean arraysAreEqual = true;
arraysAreEqual = arr.length == arr2.length;
if (arraysAreEqual) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] != arr2[i]) {
            arraysAreEqual = false;
        }
    }
}
if (arraysAreEqual) {
    System.out.println(“Массивы одинаковые”);
} else {
    System.out.println(“Массивы разные”);
}
```

ЦИКЛ FOR EACH:

For each переводится с английского — «для каждого». Само словосочетание for each в цикле не используется.

```
for (тип_переменной имя : массив) {
// действия с переменной, которая создана в первом блоке
}
```

ПЕЧАТАЕМ ЭЛЕМЕНТЫ МАССИВА:

```
for (int i : arr) {
    System.out.print(i + “ “);
}
```

Считаем сумму элементов в массиве:

```
int[] arr = {1, 2, 3};
int sum = 0;
for (int element : arr) {
    sum += element;
```

IT MENTOR

КЛАСС ARRAYS

ПЕЧАТАЕМ МАССИВ.

Напечатаем элементы массива с помощью метода **Arrays.toString** без дополнительного написания циклов:

```
int[] arr = new int[2];
arr[0] = 1;
arr[1] = 2;
System.out.println(Arrays.toString(arr));
```

ЗАПОЛНЯЕМ МАССИВ.

Метод **Arrays.fill** заполняет массив одинаковыми значениями.

```
Arrays.fill(имя массива, значение);
```

СРАВНИВАЕМ МАССИВЫ.

Для сравнения двух массивов используется метод из класса Arrays — Arrays.equals. Это позволяет нам сравнить два массива стандартным способом, который мы писали выше в виде цикла.

```
int[] arr = {1, 2};
int[] arr2 = {1, 2};
System.out.println(Arrays.equals( arr , arr2 ));
```

В консоли печатается результат сравнения в формате boolean-значения, а именно true или false.

КОПИРУЕМ МАССИВ.

Метод **Arrays.copyOf** помогает создать копию уже существующего массива.

```
тип [] имя2 = Arrays.copyOf(имя, длина);
```

Обратите внимание:

- Если длина нового массива меньше длины существующего, то лишние значения игнорируются.
- Если длина нового массива больше длины старого, ячейки заполняются значениями по умолчанию.

СОРТИРУЕМ МАССИВ.

С помощью метода **Arrays.sort** массив сортируется так же просто, как печатается или копируется:

```
Arrays.sort(имя_массива);
```

Метод Arrays.sort сортирует массив **по возрастанию**.

IT MENTOR

СТРОКИ

Чтобы хранить последовательность символов, в Java используют **строки** — тип данных **String**.

Важно: в случае с переменной **char** символ в переменной может быть только **один**, тогда как в строке **String** символов может быть **несколько**, а именно столько, сколько нужно.

В этом случае следует запомнить, что строка — **неизменяемый** объект. Если вы хотите изменить строку, то нужно создать **новую**.

Методы — это действия, которые описывают поведение объектов в определенных условиях.

КЛАСС SCANNER

Scanner — это класс в языке Java, который позволяет считывать данные из разных источников, в том числе и с консоли.

```
import java.util.Scanner;

public class Test {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter your age");
        int age = scanner.nextInt();
// вводим в консоль 19
        System.out.println("Enter your name");
        String name = scanner.next();
// вводим в консоль Java-master

        if (age < 18) {
            System.out.println(name
                    + " this page is adults only");
        } else {
            System.out.println(
                "Welcome to the application " + name);
        }
    }
}
```

IT MENTOR

СОЗДАНИЕ СТРОКИ.

Обратите внимание, что **String** пишется с большой буквы, как и все остальные объекты в Java, кроме массивов.

```
String s = “123”;
```

или

```
String s = new String(“123”);
```

СЛОЖЕНИЕ СТРОК.

Конкатенация — операция склеивания объектов линейной структуры, обычно строк.

```
String helloWorld = “Hello, “ + “world!”;
```

Мы создали новую строку, которая была сконструирована в результате слияния одной неизменяемой строки и другой неизменяемой строки. В результате этого действия в памяти оказались три строки, две из которых будут **удалены** Java как неиспользуемые, а итоговая сохранится для дальнейшего использования.

Строки, как и массивы, являются объектами. Если им не присвоить значение (не инициализировать), они будут содержать в себе null.

Строка — **неизменяемый массив символов**. Строка имеет внутренние свойства. Главное из них — массив типа **char**, где и хранятся все символы конкретной строки.

Строка — **неизменяемый объект**. Неизменяемость строк обусловлена способом их хранения в памяти, а также соображениями безопасности.

Пул строк (String pool) — один из внутренних механизмов Java, благодаря которому в памяти сохраняется только один экземпляр строки идентичного содержания.

Если бы мы могли изменять строку, в пуле строка тоже меняла бы свое содержимое, а значит, во всех местах программы, где она была, состояние бы менялось.

МЕТОДЫ ДЛЯ РАБОТЫ СО СТРОКАМИ:

СРАВНЕНИЕ СТРОК.

Для сравнения строк используют метод **equals()**. Этот метод в качестве параметра принимает другую строку и возвращает true/false.

```
String s = “Anna”;
String s2 = “ANNA”;
boolean sEqualsS2 = s.equals(s2);
```

Чтобы сравнить строки без учета регистра, заменим метод equals() на метод **equalsIgnoreCase()**:

```
String s = “Anna”;
String s2 = “ANNA”;
boolean sEqualsIgnoreCaseS2 = s.equalsIgnoreCase(s2);
```

IT MENTOR

РАБОТА С РАЗМЕРОМ СТРОКИ.

length() – длина строки

```
String s = “abc”;
int sLength = s.length();
// sLength будет присвоено значение 3
```

isEmpty(). Он проверяет, равна ли длина строки нулю:

```
String s = “abcde”;
boolean sIsEmpty = s.isEmpty();
// Длина строки s не равна 0
// Будет присвоено значение false
```

isBlank() тоже проверяет, равна ли длина строки 0.
Но в отличии от isEmpty() isBlank() проверяет еще на наличие пробелов:

```
String s = “abcde”;
boolean sIsBlank = s.isBlank();
// Длина строки s не равна 0
// Будет присвоено значение false
```

Если в строке будет только пробел (“ “), то метод isEmpty() вернет false, метод isBlank() — true.

РАЗЛИЧНЫЕ ПРОВЕРКИ СТРОКИ.

Метод **contains()** проверяет последовательность символов:

```
String s = “abcde”;
boolean sContains = s.contains(“bcd”);
// Строка s содержит внутри себя последовательность символов “bcd”
// Будет присвоено значение true
```

Метод **endsWith()** проверяет, на что заканчивается строка:

```
String s = “abcde”;
boolean sEndsWith = s.endsWith(“de”);
// Строка s заканчивается на “de”
// Будет присвоено значение true
```

Метод **startsWith()** проверяет, на что начинается строка:

```
String s = “abcde”;
boolean sStartsWith = s.startsWith(“ab”);
// Строка s начинается на “ab”
// Будет присвоено значение true
```

IT MENTOR

ИЗВЛЕЧЕНИЕ СИМВОЛОВ.

Метод **charAt()** возвращает символ строки с указанным индексом (позицией):

```
String s = “abcde”;
char c = s.charAt(2); // Вернет «c»
```

Метод **substring()** извлекает символы, начиная с первого параметра в ячейке включительно, заканчивая вторым параметром не включительно.

```
s = “abcdef”;
String s1 = s.substring(2, 4);
```

РАБОТА С РЕГИСТРАМИ БУКВ В СТРОКЕ

Метод **toUpperCase()** возвращает значение строки, преобразованное в верхний регистр:

```
s = “abcd”;
String s1 = s.toUpperCase();
```

Метод **toLowerCase()** возвращает значение строки, преобразованное в нижний регистр:

```
s = “ABCD”;
String s1 = s.toLowerCase();
```

ОБРЕЗКА СТРОКИ

Метод **trim()** удаляет пробельные символы с начала и конца строки.

```
s = “   abcd   “;
String s1 = s.trim();
```

ПРЕОБРАЗОВАНИЕ СТРОКИ В МАССИВ

Метод **split()** создает из строки массив, разбив ее на части. Разделитель указывается в скобках:

```
s = “My name is Ivan”;
String[] words = s.split(“ “);
```

Метод **toCharArray()** преобразует строку в массив символов:

```
s = “abcd”;
char[] c = s.toCharArray();
```

ЗАМЕНА В СТРОКЕ

Метод **replace()** возвращает новую строку с параметрами, замененными на указанные:

```
String s = “ab c de”;
String s2 = s.replace(‘a’, ‘b’);
```

IT MENTOR

ПОВТОР СТРОКИ

Метод **repeat()** возвращает новую строку, которая содержит указанное количество соединенных вместе копий строки, на которой был вызван метод:

```
s = "#";
String s1 = s.repeat(10);
```

ИЗМЕНЯЕМЫЕ СТРОКИ (STRINGBUILDER)

StringBuilder — это сущность, которую можно создать на основе существующей строки или абсолютно новой (пустой).

```
StringBuilder sb = new StringBuilder();
// Создается сущность StringBuilder на основе пустой строки
```

```
StringBuilder sb = new StringBuilder("123");
// Создается сущность StringBuilder на основе строки "123"
```

Добавить к текущей строке, хранящейся в StringBuilder, другие строки или символы можно через метод **append()**:

```
StringBuilder sb = new StringBuilder("123");
sb.append("456");
// Добавляем к сущности sb (с "123" внутри) строку "456"
// Содержимое изменится на "123456"
```

ОБЩИЕ МЕТОДЫ STRINGBUIDER СО СТРОКАМИ:

```
StringBuilder sb = new StringBuilder("123");
sb.append("456");

char c = sb.charAt(1);
// Вернет символ по индексу 1, т. е. c получит значение '2',
// Так как именно этот символ находится на второй позиции

int sbLength = sb.length();
// Получит значение 6
// Так как такая длина у строки "123456", что лежит внутри сущности

sb.replace('1','2');
// Заменит все единицы в содержимом сущности на двойки: "223456"
```

Обратите внимание, что писать sb1 = перед sb.replace() или sb.append() не требуется.

Это происходит потому, что StringBuilder изменяет сам себя, а не создает новый измененный объект и возвращает его. Именно поэтому он называется изменяемым в отличие от String.

IT MENTOR

МЕТОДЫ STRINGBUILDER, ИЗМЕНЯЮЩИЕ ТЕКУЩЕЕ СОСТОЯНИЕ:

```
sb.setCharAt(1, '5');
// Установит вместо символа по индексу 1 символ '5'
// Текущая строка изменится на "153456"
```

```
sb.insert(1, "abc");
// Установит в ячейки, начиная с первой, символы 'a', 'b' и 'c'
// Ранее находившиеся там ячейки сдвинутся на позиции вперед
// т. е. "1abc23456"
```

```
sb.delete(1, 3);
// Удалит из строки символы, начиная от ячейки с индексом 1
// и до ячейки с индексом 3 (не включительно)
// 2-й и 3-й символы строки будут удалены, получится: "1456"
```

```
sb.deleteCharAt(1);
// Удалит из строки символ, находящийся в ячейке по индексу 1
// т. е. строка изменит свое состояние на "13456"
```

```
sb.reverse;
// Полностью развернет текущую строку
// т. е. превратит ее в "654321"
```

Чтобы **присвоить результат StringBuilder** строке:

```
String sbResult = sb.toString();
или
System.out.println(sb);
```

IT MENTOR

ООП

МЕТОДЫ, ОБЪЕКТЫ, ПЕРЕЧИСЛЕНИЯ

IT MENTOR

МЕТОДЫ

Метод — блок кода, который выполняет определенную функцию и позволяет переиспользовать себя в нескольких местах без необходимости снова писать один и тот же код.

Один метод = одна задача. Если вы захотите написать свой метод, он тоже должен выполнять строго одну функцию.

Расположение методов. Методы располагаются внутри фигурных скобок класса, параллельно другим методам, т. е. на одном уровне.

ОБЪЯВЛЕНИЕ МЕТОДА

```
public static void main(String[] args) {
... // Код
}
```

1. Слово public — это модификатор доступа. Определяет «места», откуда можно вызвать метод.
   public — является самым открытым, т. е. метод, который имеет данный модификатор, будет доступен для вызова в любом другом классе программы.
   private — метод будет доступен только в том классе/файле, где был объявлен.
   package-private или **default** (он же отсутствующий) — метод будет доступен в том пакете, где объявлен. Т. е. вызвать его смогут все классы, которые имеют тот же package (находятся в той же папке).
   protected — метод будет доступен в пакете, а также наследникам класса, где объявлен метод, даже если они в других пакетах.
2. После модификатора доступа идет слово **static**:
   Методы бывают двух типов: **статические** и **нестатические**.
   Статические методы — методы, которые принадлежат классу. Т. е. вам не требуется иметь объект для его вызова.
   Пример: Arrays.toString() или любой другой метод сущности Arrays.
   Нестатические методы — методы, которые принадлежат объекту. Для их использования нам нужно самим создать объект, инициализировать его и вызвать у этого объекта метод.
3. Следующее слово — **void**:
   void обозначает тип возвращаемого значения методом, т. е. результат метода. **void** сообщает, что не будет возвращено ничего, т. е. метод выполняет какой-то код и не отчитывается о своем выполнении. Если бы мы написали int, мы были бы обязаны вернуть числовой результат.
4. Следующим словом стоит **main**:
   main — это имя метода.
5. После имени метода открываются скобки и пишется **String[] args**:

В скобках указывается **параметр метода**. В данном случае при вызове метода main ему передается массив строк, который затем присваивается переменной args, объявленной в скобках.

Параметры метода. Имена переменных при передаче в метод не имеют значения. Внутри метода они в любом случае будут присвоены тем переменным, которые объявлены в скобках при объявлении метода.

Идентичность методов. Даже имея одинаковые названия, методы не идентичны друг другу. Такими они являются при полном совпадении сигнатур.

Сигнатура метода — это имя и параметры метода, причем порядок параметров внутри скобок имеет значение.

```
public static void calculateSum(int[] arr)
// calculateSum – имя метода
// (int[] arr) – параметры метода

public static void calculateSum(int a, int b)
// calculateSum – имя метода
// (int a, int b) – параметры метода
// Параметры методов отличаются
```

ВОЗВРАЩЕНИЕ ЗНАЧЕНИЙ ИЗ МЕТОДА

Методы делятся на два типа:

- **Возвращающие** результат
  (имеют перед именем метода указание типа).
- **Не возвращающие** результат
  (имеют перед именем метода void).

Для возврата значения нужно написать ключевое слово return и указать после него результат. Можно возвращать любые значения любого типа, но этот любой тип должен быть указан перед именем метода в сигнатуре.

ОПЕРАТОР RETURN.

У оператора две функции:

1. **Прервать** выполнение метода. Эту функцию оператор выполняет всегда.
2. **Вернуть результат** в то место (в ту строку), где метод был вызван. Эту функцию оператор выполняет в тех случаях, когда значение возвращаемое.

Пример использования return в условных операторах

```
public static int getClientOS(String name) {
    if (name.equals("iOS")) {
        return 0;
    } else {
        return 1;
    }
}
```

Важно запомнить, что **return** должен быть во всех возможных вариантах развитиях событий, которые мы покрываем методом.

IT MENTOR

Пример использования return в методе void

```
public static void printSeason(int monthNumber) {
    if (monthNumber <= 0 || monthNumber > 12) {
        System.out.println("Некорректное значение
месяца");
        return;
    }
    switch (monthNumber) {
        case 12,1,2:
            System.out.println("Зима");
            break;
        case 3,4,5:
            System.out.println("Весна");
            break;
        case 6,7,8:
            System.out.println("Лето");
            break;
        case 9,10,11:
            System.out.println("Осень");
   }
}
```

В примере выше мы применили оператор **return**, который в случае некорректного значения прервет метод и не даст коду далее выполниться.

ОБЛАСТЬ ВИДИМОСТИ ПАРАМЕТРОВ

При создании метода с параметрами мы создаем переменные, которым присваиваем то, что было передано в скобки при вызове метода. Эти переменные можно использовать только **внутри** метода.

Область видимости переменных (scope) — это свойство переменных, которое определяет, можно ли использовать переменную в тех или иных частях кода.

ПОВЕДЕНИЕ РАЗНЫХ ТИПОВ ДАННЫХ В ПАРАМЕТРАХ МЕТОДА

Примитивным типом является простое значение. Например, число или символ. Примитивы не имеют своего поведения, возможностей и лишь используются как **тип-значение**.

Ссылочные типы не просто хранят число или символ, они имеют какие-то **свойства** *(как length у массивов)* и **методы** *(как toUpperCase() или replace() у строк).*

IT MENTOR

Параметры в Java передаются по значению:

- Если это тип-значение (примитив), то копируется само значение.
- Если же это ссылочный тип (объект), то копируется ссылка.

Если мы модифицируем примитив внутри метода, это значение вне метода **не поменяется**, но если мы модифицируем в методе объект, значение объекта вне метода **поменяется** тоже.

```
public static void changeValues ( int a, int[] arr2){
// Объявляем метод
            a = 5;
            arr2[0] = 5;
        }

public static void main (String[]args){
            int a = 1;
            // Примитивный тип
            int[] arr = {1, 2, 3};
            // Ссылочный тип
            changeValues(a, arr);
            // Вызываем метод
            System.out.println(a);
            System.out.println(Arrays.toString(arr));
        }
```

В консоли будет:

```
1           // Значение не поменялось, потому что int a – примитив
[5, 2, 3] // Значение поменялось, потому что int[] arr – объект
```

IT MENTOR

ОБЪЕКТЫ И КЛАССЫ

ОБЪЕКТЫ

Типы данных делятся на **примитивные** (int, float, boolean и др.) и **ссылочные** (строки и массивы).

В **примитивные типы** мы можем положить значение переменной, которое будет там храниться. В **ссылочных типах** мы не можем хранить сам объект — только ссылку на область памяти, где данный объект находится.

В основе языка Java используются объекты и классы, поэтому **Java – это объектно-ориентированный язык.**

Шаблоном, или описанием объекта, является **класс**, а **объект** представляет экземпляр этого класса.

Объекты имеют как **состояние** (характеристики или свойства), так и **поведение** (умения, навыки или просто методы).

В языке Java каждый новый класс принято записывать в отдельном файле проекта.

Переменные, тип которых мы создаем сами, являются объектами. Все объекты записываются через ключевое слово **new**.

```
public class Main {
    public static void main(String[] args) {
        Book warAndPeace = new Book();
    }
}
```

Каждый **объект** имеет свою зону ответственности и отвечает за **одну** обязанность.

КЛАСС В JAVA

Класс в Java — это сущность, которая описывает состояние (переменные), поведение (методы) и способы создания своих объектов, если они подразумеваются.

Класс в себе содержит все общие свойства, которые не могут отличаться у любых объектов по данному чертежу.

Классы бывают разных типов:

1. **Хранитель данных**. Обычно эти классы не имеют «умений» и созданы только для агрегации в себе некоторых данных. Например, объект «Книга». У нее могут быть свойства или состояние (название, год выпуска, автор, номер издания и т. д.), но все ее «умения» заключаются только в том, что вы можете ее открыть и взять текст, информацию об авторе, годе издания и т. д.
2. **Утилити-класс**. Обычно эти классы имеют только «умения» и созданы для работы с другими объектами, например класс Arrays и массивы.
3. **Самодостаточные классы**. Например, String. Имеют и состояние (массив символов, в виде которого хранятся данные строки), и поведение (методы, которые с этим состоянием работают, например как replace).

IT MENTOR

КОНСТРУКТОРЫ (КОНСТРУИРУЮЩИЕ МЕТОДЫ)

Конструкторы подчиняются тем же правилам, что и обычные методы. Разница в том, что конструкторы не могут быть вызваны в произвольное время, а вызываются именно при создании объекта.

Задача **конструктора** в том, чтобы в случае необходимости создать объект, мы обязаны корректно инициализировать его.

В том случае, когда вы объявили свой конструктор, стандартный конструктор Java не добавит, т. е. класс можно будет создать только по тем конструкторам, что объявили вы.

Классы могут иметь состояние.

Состояние — это переменные класса, или, как их принято называть, поля класса.

КЛЮЧЕВОЕ СЛОВО THIS

this является переменной, которая всегда ссылается на ваш конкретный объект.

Следует помнить, что если поле класса совпадает по имени с параметром метода, необходимо использовать **this.имя_поля** для обращения к полю, так как параметр или локальная переменная (объявлена в конкретном методе с тем же именем, что и поле) будут «затенять» поле.

Важно заметить, что поля **static final** являются константами (значения переменных с модификатором final нельзя изменять) и пишутся капсом с разделением в виде ‘_’ между словами.

```
public static final int KEY_SIZE = 5;
```

Чтобы обратиться к **статическому** полю, если оно public, нужно использовать вместо имя_объекта.имя_поля конструкцию имя_класса.имя_поля.

Нужно запомнить, что **нестатические** поля могут различаться для каждого из объектов и именно их значения мы инициализируем в конструкторе.

Статические же переменные мы обычно задаем вручную (инициализируем) и меняем (в случае со счетчиком) внутри класса.

ИНКАПСУЛЯЦИЯ

Инкапсуляция — концепция, согласно которой мы не даем доступ к свойствам объекта, а получаем их значения через методы.

Пользователь не должен иметь доступ к методам, которые ему не положено вызывать, или к полям, которые ему запрещено читать или даже изменять.

Именно поэтому мы должны закрыть пользователю доступ для обращения к полю напрямую, объявив поле с модификатором доступа **private**.

IT MENTOR

ГЕТТЕРЫ И СЕТТЕРЫ

Сеттер(setter) — это метод, который изменяет (устанавливает; от слова set) значение поля. **Геттер(getter)** — это метод, который возвращает (от слова get) нам значение какого-то поля.

Автор не должен изменяться после создания (т. е. после инициализации полей в конструкторе), а вот издательство поменять можно, ведь книга может начать издаваться в другом месте. Поэтому для первого мы создадим только геттер, а для второго — еще и сеттер.

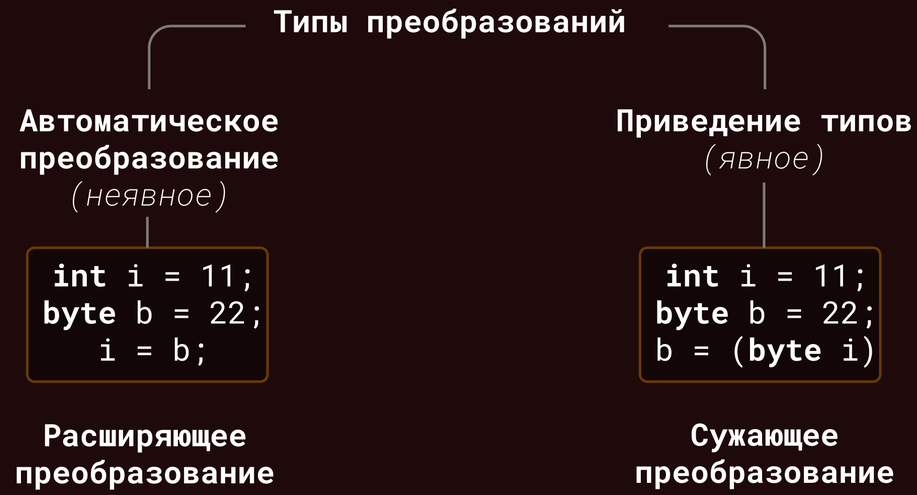
```
public String getAuthor() {
    return author;
}

public String getPublisher() {
    return publisher;
}

public void setPublisher(String publisher) {
    this.publisher = publisher;
```

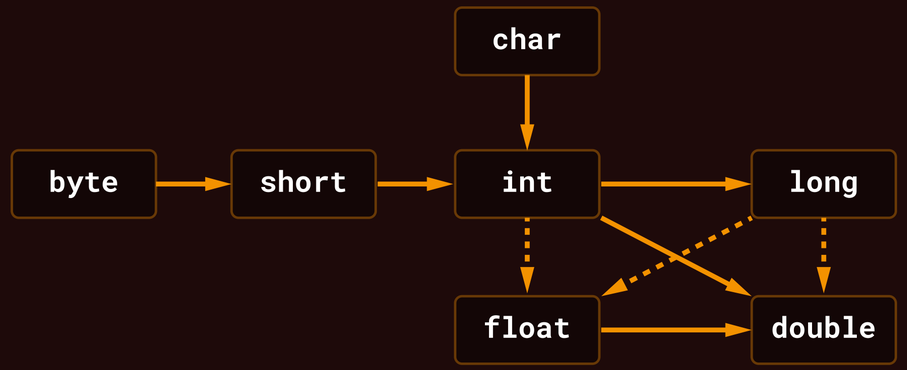
ПРИВЕДЕНИЕ ТИПОВ

В Java существуют два типа преобразований: **автоматическое преобразование** (неявное) и **приведение типов** (явное преобразование).



IT MENTOR

1. **Автоматическое преобразование типов Java.**
   Для автоматического преобразования типа должны выполняться два условия:
   - Оба типа должны быть совместимы.
   - Вместимость целевого типа должна быть больше совместимости исходного типа.



Сплошные линии обозначают преобразования, выполняемые без потери данных. Штриховые линии говорят о том, что при преобразовании может произойти потеря точности.

2. **Приведение типов Java.** Чтобы выполнить преобразование двух несовместимых типов данных, нужно воспользоваться приведением типов. Приведение — это всего лишь явное преобразование типов. Общая форма приведения типов имеет следующий вид:

(целевой_тип) значение

где параметр целевой_тип обозначает тип, в который нужно преобразовать указанное значение.

3. **Автоматическое продвижение типов в выражениях.** Помимо операций присваивания, определенное преобразование типов может выполняться и в выражениях.

В языке Java действуют следующие правила:

- Если один операнд имеет тип double, другой тоже преобразуется к типу double.
- Иначе, если один операнд имеет тип float, другой тоже преобразуется к типу float.
- Иначе, если один операнд имеет тип long, другой тоже преобразуется к типу long.
- Иначе оба операнда преобразуются к типу int.
- В выражениях совмещенного присваивания (+=,-=,*=,/=) нет необходимости делать приведение.

Приведем пример:При умножении переменной b1 (byte) на 2 (int) результат будет типа int. Поэтому при попытке присвоить результат в переменную b2 (byte) возникнет ошибка компиляции. Но при использовании совмещенной операции присваивания (*=) такой проблемы не возникнет:

```
byte b1 = 1;
byte b2 = 2 * b1; //Ошибка компиляции

int i1 = 2 * b1;
b2 *= 2;
```

IT MENTOR

АВТОУПАКОВКА И АВТОРАСПАКОВКА

К оболочкам типов относятся классы:

Double, Float, Long, **Integer**, Short, Byte, **Character**, Boolean.

Автоупаковка и распаковка — это процесс преобразования примитивных типов в объектные и наоборот. Весь процесс выполняется автоматически средой выполнения Java.

```
public class AutoBoxDemo1 {
    public static void main(String[] args) {
        Integer iOb = 100; // упаковать значение int
        int i = iOb; // распаковать
        System.out.println(i + " " + iOb);
    }
}
```

Unboxing происходит:

При присвоении экземпляра класса-обёртки переменной соответствующего примитивного типа.

В выражениях, в которых один или оба аргумента являются экземплярами классов-обёрток (кроме операции == и !=).

При передаче объекта класса-обёртки в метод, ожидающий соответствующий примитивный тип.

IMMUTABLE КЛАССЫ

Иммутабельный (неизменяемый, immutable) класс — это класс, который после инициализации не может изменить свое состояние. То есть если в коде есть ссылка на экземпляр иммутабельного класса, то любые изменения в нем приводят к созданию нового экземпляра.

Чтобы класс был иммутабельным, он должен соответствовать следующим требованиям:

- Должен быть объявлен как final, чтобы от него нельзя было наследоваться. Иначе дочерние классы могут нарушить иммутабельность.
- Все поля класса должны быть приватными в соответствии с принципами инкапсуляции (понятие инкапсуляции будет рассмотрено в одном из следующих уроков).
- Для корректного создания экземпляра в нем должны быть параметризованные конструкторы, через которые осуществляется первоначальная инициализация полей класса.
- Для исключения возможности изменения состояния после инстанцирования в классе не должно быть сеттеров.
- Для полей-коллекций необходимо делать глубокие копии, чтобы гарантировать их неизменность.

IT MENTOR

```
public final class SomeClass {
    private String someValue;
    private int someNumber;
    public SomeClass(String someValue, int someNum-
ber) {
        this.someValue = someValue;
        this.someNumber = someNumber;
    }
    public String getSomeValue() {
        return someValue;
    }
    public int getSomeNumber() {
        return someNumber;
    }
}
```

Иммутабельность строк дает следующие преимущества:

- Строки потокобезопасны. Класс является потокобезопасным, если он ведет себя корректно во время доступа из многочисленных потоков.
- Для строк можно использовать специальную область памяти, называемую пулом строк, благодаря которой две разные переменные типа String с одинаковым значением будут указывать на одну и ту же область памяти.
- Строки — отличный кандидат для ключей в коллекциях, поскольку они не могут быть изменены по ошибке.
- Класс String кэширует хеш-код, что улучшает производительность хеш-коллекций, использующих String.
- Чувствительные данные, такие как имена пользователей и пароли, нельзя изменить по ошибке во время выполнения, даже при передаче ссылок на них между разными методами.

Методы могут быть созданы самим разработчиком. Так вот в Java таких стандартных инструментов придумано уже очень много. И все они хранятся в одном месте — **в стандартной библиотеке Java.**

Так и стандартная библиотека Java хранит в себе эти инструменты, что очень экономит время разработчика — **бери стандартный код и используй.**

При этом каждый раз, когда версия языка Java обновляется, то обновляется и стандартная **библиотека языка.**

IT MENTOR

`final` — это модификатор, который позволяет объявлять константные поля в классе. Если у вас есть некоторое свойство проектируемого вами объекта, значение которого не будет меняться, то вы можете воспользоваться этим модификатором. Любая попытка переопределить значение поля с модификатором final приводит к выбросу исключения.

Методы, которые возвращают строковый вид нашего объекта, обязаны называться toString, иначе методы стандартной библиотеки или фреймворков и библиотек, которые опираются на эти правила, не будут корректно работать с вашим объектом.

МЕТОДЫ С «КОНТРАКТОМ»

Чтобы задать свое поведение, нужно эти сгенерированные методы «переписать», т. е. реализовать самим внутри нашего класса и пометить флагом `@Override.`

В Java такие «переписанные» методы называются **предопределенными**, а сам процесс называется **переопределением**.

1. `Метод toString.` Используется для приведения нашего объекта к строковому виду. В нем мы должны вернуть строку в том формате, в котором хотим видеть наш объект в печати. Массив без обертки в виде Arrays.toString «странно» печатается в консоль из-за отсутствия переопределенного toString. Это самое «странно» получается благодаря вызову стандартного сгенерированного Java toString.
2. `Метод equals.` Позволяет считать объекты равными, если совпадают те данные этих объектов, которые нужны.
3. `Метод hashCode.` Выполняет функцию превращения объекта в число - кодировки.
   - Если объекты равны, хеш-коды обязательно равные.
   - Если объекты не равны, хеш-коды не обязательно должны быть разные.
   - Если хеш-коды разные, объекты точно не будут равны.

`Метод hashCode` выполняет первую и наименее точную проверку на равенство объектов.

IT MENTOR

ООП: ВВЕДЕНИЕ, OBJECT

Объектно-ориентированное программирование (в дальнейшем ООП) — методология программирования, в которой основными концепциями являются понятия объектов и классов.

ПРИНЦИПЫ ООП

В центре ООП находится понятие объекта.

Объект — это сущность, экземпляр класса, которой можно посылать сообщения и которая может на них реагировать, используя свои данные.

1. **Инкапсуляция** — это свойство системы, позволяющее объединить данные и методы, работающие с ними в классе, и ограничить детали реализации от пользователя.
2. **Наследование** — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником или производным классом.
3. **Полиморфизм** — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Существует мнение, что в ООП стоит выделять еще одну немаловажную характеристику — **абстракцию**.

Абстрагирование — это способ выделить набор значимых характеристик объекта, исключая из рассмотрения незначимые. Соответственно, абстракция — процесс скрытия деталей реализации от пользователя, предоставляя ему только функционал.

ЧТО ТАКОЕ КОНСТРУКТОР?

Конструктор инициализирует объект непосредственно во время его создания.

1. **Конструктор по умолчанию** инициализирует все переменные экземпляра устанавливаемыми по умолчанию значениями.
2. **Конструктор с параметрами.** Используется при создании объектов зачастую необходимо сразу присваивать им конкретные значения полей.

Ключевое слово this является переменной, которая всегда ссылается на ваш конкретный объект.

Метод — блок кода, который выполняет определенную функцию и позволяет переиспользовать себя в нескольких местах без необходимости снова писать один и тот же код.

IT MENTOR

ООП: ИНКАПСУЛЯЦИЯ

ИНКАПСУЛЯЦИЯ

Инкапсуляция — концепция, согласно которой мы ограничиваем прямой доступ к свойствам объекта, а получаем их значения через методы. Способность писать более читаемый и легко поддерживаемый код, уменьшить вероятность ошибок.

Именно поэтому мы должны закрыть пользователю доступ для обращения к полю напрямую, объявив поле с модификатором доступа `private`.

Основные способы регулировки доступа к нашим данным:

- Модификаторы доступа
- Геттеры и сеттеры

1. Модификаторы доступа (или модификаторы видимости) — это специальные ключевые слова, которые показывают, кому нельзя, а кому можно пользоваться данными.

В Java определены четыре модификатора доступа:

- `public` — публичный. Доступный в любой точке программы.
- `default` — по умолчанию. Если мы не указываем модификатор доступа явно, он по умолчанию имеет значение default. Данные с этим модификатором видны в пределах папки (пакета).
- `protected` — защищенный. Имеет почти ту же видимость, что и default, только доступ распространяется на классы-наследники.
- `private` — приватный. Данные, помеченные этим модификатором доступа, видны только внутри класса.

2. **Геттер** (от англ. get — получать) — это метод, с помощью которого мы получаем значение переменной, т. е. ее читаем.

Сеттер (от англ. set — устанавливать) — это метод, с помощью которого мы меняем или задаем значение переменной.

ПАКЕТЫ

Пакеты очень похожи на папки в файловой системе. По сути, это и есть папки для хранения классов или файлов в программе.

Даже в не самых больших проектах есть очень много классов и различных файлов, хранить которые неструктурированно в одном месте неудобно. К тому же и нефункционально, так как разные программисты могут иметь необходимость создать классы с одинаковым названием в пределах одной программы.

Для этого и существуют пакеты.

IT MENTOR

Чтобы **добавить** класс в пакет, необходимо использовать оператор **package**, который всегда прописан в самой первой строке файла.

```
package lessons;

public class MyClass {
    public static void main(String[] args) {
        System.out.print(“Hello world”);
    }
}
```

В Java есть возможность создавать **многоуровневые** пакеты, для этого всю иерархию пакетов необходимо прописать через точки.

```
package com.skypro.javacourse.lessons;

 public class MyClass {
     public static void main(String[] args) {
         System.out.print(“Hello world”);
     }
 }
```

В коммерческой разработке именование пакетов начинается с com(домена), далее идет название компании и имя проекта. Далее названия пакетов распределяются по функциональному признаку.

Полное название класса состоит в том числе и из названия пакета. В первом примере полное название класса — lessons.MyClass, а во втором примере — com.skypro.javacourse.lessons.MyClass

ИСПОЛЬЗОВАНИЕ ОПЕРАТОРА IMPORT

В Java оператор import находится между оператором package и объявлением класса.

```
package one;

import two.Ex2;

public class Ex1 {
    public static void main(String[] args) {
        Ex1 ex1 = new Ex1();
        System.out.print(“Done!!!”);
    }
}
//////////////////////////////////////
package two;

public class Ex2 {
}
```

У нас есть возможность импортировать в класс сразу несколько пакетов. Однако среди них могут быть пакеты, которые содержат в себе классы с **одинаковыми** названиями. В таком случае нужно обращаться к ним по полному имени, иначе компилятор не поймет, о каком именно классе идет речь.

IT MENTOR

ВЛОЖЕННЫЕ КЛАССЫ

Вложенные, или внутренние, классы (inner class) — это классы, которые определены внутри основного класса. Область действия вложенного класса ограничена областью действия внешнего класса. Внутренний класс не может существовать самостоятельно без основного класса, внутри которого он создан. Вложенный класс имеет доступ к членам (даже приватным) класса, в котором он создан и реализован.

Типы вложенных классов:

1. **Обычный вложенный (нестатичесчкий) класс.** Внутренний класс создается внутри основного класса. Для создания объекта внутреннего класса должен быть создан объект внешнего класса. Внутренний и внешний класс взаимно доступны друг для друга, в том числе и элементы с модификатором private.

```
Car.Key keyOptions1 = lada.new Key(true, false);
```

2. **Локальный класс (local class).** Зачастую определяется в методе уже созданного класса. Он так же является полноценным членом класса, внутри которого он был объявлен, как поля и методы. Локальный класс может обращаться к локальным параметрам метода, в котором он был создан, и к его переменным, если они являются эффективно-финальными или имеют модификатор final.Эффективно-финальной является переменная, которая не имеет модификатора final, но ее значение остается неизменным.
3. **Анонимный класс (anonymous class)** — это класс, который не имеет имени. Используется в тех случаях, когда необходимо переопределить метод интерфейса или класса. Анонимный класс инициализируется в момент объявления.

```
public class Moto {
    public void drive() {
        System.out.println(“Едем на мотоцикле”);
    }
}
//////////////////////////////////////////
public class Food {
    public static void main(String[] args) {
        Moto moto = new Moto() {
            @Override
            public void drive() {
                System.out.println(“Едем на мотоцикле
в анонимном классе.”);
            }
        };
        moto.drive();
    }
}
```

Анонимный класс может также и добавлять собственные методы, но они не могут быть вызваны за его пределами.

В каких случаях следует использовать анонимный класс:

- × Тело класса является очень коротким.
- × Нужен единственный экземпляр класса.
- × Класс используется только в том месте, где он создан.
- × Нам не нужно имя класса.

IT MENTOR

Анонимные классы, помимо расширения классов, могут также реализовывать интерфейсы:

```
public interface Moto {
    void ride();
    void stop();
}

public class MotoTest {
    public static void main(String[] args) {
        Moto moto = new Moto() {
            @Override
            public void ride() {
            System.out.println("ride a motorcycle!!!");
            }

            @Override
            public void stop() {
                System.out.println("stop motorcycle");
            }
        };
        moveable.ride();
moveable.stop();
    }
}
```

4. **Статический вложенный класс (static nested class)** — это тот же самый обычный внутренний класс, с той лишь разницей, что он объявлен с модификатором static. Объект этого класса можно создать без объекта основного класса.

Статический вложенный класс имеет прямой доступ только к статическим полям и методам внешнего класса. А к нестатическим может обращаться только через ссылку на объект внешнего класса.

```
Car.Key keyOptions1 = new Car.Key(true, false);
```

IT MENTOR

ООП: ПОЛИМОРФИЗМ, GENERIC

ПОЛИМОРФИЗМ

Данный принцип описывает механизм работы с разными типами объектов как с одним.

Т. е. если наши объекты имеют какого-то общего «родственника», мы можем объединить их в массив и работать с ними так, как будто работаем с их общим родителем.

Полиморфизм
позволяет перегружать одноименные методы родительского класса в классах-потомках, то есть название метода остается прежним, а его «начинка» изменяется под нужды конкретного класса

Любой прямой или косвенный наследник класса может быть использован в качестве экземпляра своего родителя.

За набор полей и методов отвечает тип ссылки.

За код, который выполняется при вызове этих методов, отвечает уже реализация *(то, что после = new)*.

Обязательно нужно запомнить, что наследовать можно исключительно от **одного** класса.

Множественное наследование в Java **запрещено**.

ИНТЕРФЕЙСЫ

Интерфейс — это конструкция языка Java. Он схож с классом. Это совокупность абстрактных методов и констант. Класс реализует **интерфейс**, таким образом наследуя абстрактные методы **интерфейса**.

Для создания интерфейса необходимо использовать ключевое слово **interface**:

```
interface Printable {
    void print();
}
```

Надо отметить, что у нас нет возможности создавать экземпляры интерфейсов, так как интерфейс определяет **только** поведение и **не определяет** свойства!

IT MENTOR

Модификаторы доступа у членов интерфейса указывать не нужно, так как по умолчанию все методы интерфейса являются **public**, а переменные имеют модификаторы **public static final**, то есть являются константами.

```
public class Test {
    public static void main(String[] args) {
        Book book = new Book("War and Peace", "Lev Tol-
stoy");
        book.print();
    }
}
//////////////////////////////////////////////////////
interface Printable {
    void print();
}
//////////////////////////////////////////////////////
class Book implements Printable {
    String name;
    String author;
    Book(String name, String author) {
        this.name = name;
        this.author = author;

    }
    public void print() {
        System.out.println(name + " - " + author);
    }
}
```

Как вы можете заметить, наш интерфейс Printable **имплементирован** (от implements — **реализует**) классом Book, в котором реализован метод print(). Если класс, имплементирующий интерфейс, не является абстрактным, он должен реализовывать все методы интерфейса.

ИНТЕРФЕЙСЫ В ПРЕОБРАЗОВАНИЯХ ТИПОВ.

Для интерфейсов очень актуальны свойства полиморфизма. То есть для нашего примера можно сказать, что на объект класса Book указывает ссылка типа Printable.

```
Printable printable = new Book("War and Peace", "Lev
Tolstoy");
printable.print();
// Интерфейс не имеет метода getName, необходимо явное приведение
String name = (Book)printable.getName();
// Тут используется явное приведение, так как в интерфейсе нет метода get-
Name

System.out.println(name);
```

Для того чтобы через ссылку интерфейса вызывать методы класса, к которому относится наш объект, необходимо производить явное приведение типов — (Book)printable.getName();

IT MENTOR

МЕТОДЫ ПО УМОЛЧАНИЮ

В восьмой версии Java была добавлена возможность прописывать реализацию методов прямо в интерфейсе. Данная реализация распространяется на все классы, имплементирующие интерфейс, однако ничего не мешает нам переопределить эти методы в реализующих классах, если такая необходимость у нас есть. Такие методы называются **дефолтными** или **методами по умолчанию** и обозначаются ключевым словом **default**.

```
interface Printable {
    default void print() {
        System.out.println("Default method");
    }
}
```

По сути, **дефолтный метод** — это самый обычный метод, который без переопределения будет одинаково работать во всех имплементирующих классах.

СТАТИЧЕСКИЕ МЕТОДЫ

Также в Java 8 была добавлена возможность реализации статических методов в интерфейсах, статические интерфейсы используются исключительно для обслуживания самого интерфейса.

```
interface Printable {
    void print();
    static void read() {
        System.out.println("Static method");
    }
}
```

Для того чтобы вызвать этот метод, необходимо прописать сначала имя интерфейса, а потом через точку и название метода. Всё как в статических методах классов.

```
public static void main(String[] args) {
    Printable.read();
}
```

КОНСТАНТЫ В ИНТЕРФЕЙСАХ

Интерфейсы могут иметь переменные, и хотя они не имеют явных модификаторов доступа, все они являются public static final, что делает их **константами**.

```
interface SomeInterface {
    int const1 = 1;
    int const2 = 0;
    void printConst(int value);
}
```

Соответственно, константы можно использовать в **любой** точке программы.

IT MENTOR

МНОЖЕСТВЕННАЯ РЕАЛИЗАЦИЯ ИНТЕРФЕЙСОВ

Чтобы расширить свой функционал, один класс может имплементировать сразу несколько интерфейсов. Для этого после слова Implements достаточно перечислить все необходимые интерфейсы через запятую.

```
interface Printable {
    // методы интерфейса
}
//////////////////////////
interface Searchable {
    // методы интерфейса
}
//////////////////////////
class Book implements Printable, Searchable{
    // реализация класса
}
```

НАСЛЕДОВАНИЕ ИНТЕРФЕЙСОВ

Также, чтобы расширить функционал применяемых интерфейсов, у нас есть возможность наследовать интерфейсы аналогично классам.

```
interface MagazinePrintable extends Printable {
    void paint();
}
```

В таком случае классы, имплементирующие самый младший интерфейс, должны будут реализовывать методы всех интерфейсов, наследующих друг друга.

ИНТЕРФЕЙСЫ КАК ПАРАМЕТРЫ И РЕЗУЛЬТАТЫ МЕТОДОВ

Еще одним свойством интерфейсов является то, что их можно использовать как тип возвращаемого значения или передавать в качестве типа параметра метода.

```
public class Test {
    public static void main(String[] args) {
        Printable printable = createPrintable("National
Geographic",false);
        printable.print();
        read(new Book("Dead Souls", "N. Gogol'"));
        read(new Magazine("New York Times"));
    }
    static void read(Printable p) {
        p.print();
    }
    static Printable createPrintable(String name,
boolean option) {
        if(option) {
            return new Book(name, "Undefined");
        } else {
return new Magazine(name);
        }
    }
```

IT MENTOR

```
////////////////////////////////////////////////////////////////
interface Printable {
    void print();
}
////////////////////////////////////////////////////////////////
class Book implements Printable {
    String name;
    String author;
    Book(String name, String author) {
        this.name = name;
        this.author = author;
    }
    public void print() {
        System.out.println(name + " - " + author);

}
    public void print() {
        System.out.println(name + " - " + author);
    }
}
////////////////////////////////////////////////////////////////
class Magazine implements Printable {
    private String name;
    String getName() {
        return name;
    }
    Magazine(String name) {
        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
}
```

Так как метод read()принимает объект, а метод createPrintable() возвращает объект интерфейса Printable, мы можем в обоих случаях использовать типы классов, реализующих данный интерфейс.

ДЖЕНЕРИКИ, ИЛИ ПАРАМЕТРИЧЕСКИЙ ПОЛИМОРФИЗМ

Дженерики (или обобщения) — это параметризованные типы. Или конструкции и средства, в Java, которые позволяют объявлять классы, интерфейсы, и методы, где тип данных, с которыми они будут оперировать, заранее указан в виде параметра.

IT MENTOR

ПАРАМЕТРИЗОВАННЫЕ КЛАССЫ.

Класс является **параметризованным**, если при его объявлении явно указан тип данных, с которым он работает. С попощью дженериков можно создать единственный класс, которые будет работать с разными типами данных.

```
public class ClassArrays<T> {
    private T[] array;
    public ClassArrays(T[] array) {
        this.array = array;
    }
    public static void main(String[] args) {
        ClassArrays<Byte> byteArray = new ClassAr-
rays<>(new Byte[4]);
        ClassArrays<Double> doubleArray =
new ClassArrays<>(new Double[3]);
        ClassArrays<Integer> integerArray = new Clas-
sArrays<>(new Integer[9]);
    }
}
```

Благодаря тому что при объявлении класса ClassArrays мы указали параметр <T>, нам нет необходимости создавать объекты разных классов. В данном случае у нас только одна версия класса, которая корректно будет работать с каждым из типов.

Если мы указываем в качестве заполнителя какой-то универсальный символ, например <T>, это будет означать, что в дальнейшем при выполнении какой-либо работы нам необходимо будет подставить конкретный тип данных вместо этого символа и, соответственно, этот тип будет учитываться при выполнении логики, в которой учитывается параметр типа.

Имейте в виду, что в качестве параметра в дженериках **нельзя** указывать примитивы!

ОГРАНИЧЕННЫЕ ТИПЫ.

В Java есть возможность ограничить параметр типа.

`class SomeClass <T extends SomeSuperClass>`

В данном случае нашим параметром типа может быть любой класс-наследник класса SomeSuperClass, включая его самого.

Также в качестве типов ограничений мы можем использовать и интерфейсы.

```
public class SomeClass<T extends Autoclosable>
class SomeClass <T extends SomeSuperClass & Autoclosable>
```

IT MENTOR

В таком случае необходимо перечислять их через разделитель-амперсанд (&), причем на первой позиции должен быть указан именно тип класса, а уже за ним можно перечислять интерфейсы.

В данном случае мы можем использовать в качестве параметра типа класс, который является классом SomeSuperClass или его наследником, а также реализовывать интерфейс Autoclosable.

Wildcard - Обозначается знаком ? и представляет неизвестный тип, то есть на месте этого символа можно использовать любой тип данных в одной реализации метода.

class SomeClass <? super SomeYoungestClass>

В данном случае нашим параметром типа может быть любой класс-предок класса SomeYoungestClass, в том числе и класс Object, включая его самого.

Предположим, что нам требуется сравнивать средние значения массивов в методе sameAvg()при условии, что типы этих массивов отличаются.

```
Integer intArray[] = {1, 5, 2, 4, 3};
Double doubleArray[] = {2.4, 7.3, 4.4, 15.1};
Average<Integer> aveInt = new Average<>(intArray);
Average<Double> aveDouble = new Average<>(doubleArray);
if (aveInt.sameAvg(aveDouble)) {
    System.out.println("are the same.");
} else {
    System.out.println("differ.");
}
boolean sameAvg (Average < ? > object){
    return average() == object.average();
}
```

По сути, wildcard является аналогом записи <Object>.

Wildcard можно ограничивать так же, как мы изучали ранее, только вместо записи **<T extends SomeSuperClass>** можем использовать запись **<? extends SomeSuperClass>**.

```
class SomeClass <? extends SomeSuperClass & Autoclosable>
```

ПАРАМЕТРИЗОВАННЫЕ МЕТОДЫ И КОНСТРУКТОРЫ

Мы можем создать метод, который будет параметризован одним ли несколькими параметрами типа.

Параметризованными могут быть даже конструкторы.

```
public class SomeConstructorClassEx {
    private double value;
    public <T extends Number> SomeConstructorClassEx(T
arg) {
        value = arg.doubleValue();
    }
    public void printValue() {
        System.out.println("value: " + value);
    }
}
```

IT MENTOR

```
public class SomeConstructorClassExTest {
    public static void main(String[] args) {
        SomeConstructorClassEx constr1 = new SomeCon-
structorClassEx(100);
        SomeConstructorClassEx constr2 = new SomeCon-
structorClassEx(123.5F);
        constr1.printValue();
        constr2.printValue();
    }
```

ПАРАМЕТРИЗОВАННЫЕ ИНТЕРФЕЙСЫ

Наряду с классами параметризованными часто являются интерфейсы. Общая логика реализации дженериков в интерфейсах не отличается от реализации в классах.

```
public interface SomeInterface<T> {
    T someMethod(T t);
}
public class SomeClass<T> implements SomeInterface<T> {
    @Override
    public T someMethod(T t) {
        return t;
    }
    public static void main(String[] args) {
        SomeInterface<String> obj = new SomeClass<>();
        String string = obj.someMethod("some string");
    }
}
```

ИЕРАРХИИ ПАРАМЕТРИЗОВАННЫХ КЛАССОВ

Так же как и непараметризованные классы, классы, объявленные с параметром типа, могут строить иерархию с той лишь разницей, что параметр должен передаваться по всем поколениям классов.

```
public class SomeSuperClass<T> {
    private T obj;
    public SomeSuperClass(T obj) {
        this.obj = obj;
    }
    private T getObj() {
        return obj;
    }
}
public class SomeYoungerClass<T> extends SomeSuper-
Class<T> {
    public SomeYoungerClass(T obj) {
        super(obj);
    }
}
```

- Подкласс параметризованного суперкласса необязательно должен быть параметризованным, но в нем всё же должны быть указаны параметры типа, требующиеся его параметризованному суперклассу.
- Подкласс может быть дополнен и своими параметрами типа, если требуется.
- Суперклассом для параметризованного класса может быть непараметризованный класс.

IT MENTOR

ENUM

ЧТО ТАКОЕ ПЕРЕЧИСЛЕНИЯ?

Enum(перечисления) — конструкция в языке Java, с помощью которых можно создать ограниченный список значений.

Чтобы создать перечисление, необходимо воспользоваться ключевым словом **enum**, а внутри фигурных скобок через запятую перечислить те самые константы.

```
public class Program{
    public static void main(String[] args) {
        Day current = Day.MONDAY;
        System.out.println(current);     // MONDAY
    }
}
enum Day{
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

ИСПОЛЬЗОВАНИЕ ПЕРЕЧИСЛЕНИЙ В ОПЕРАТОРЕ SWITCH

Отметим, что в **теле оператора switch можно использовать вложенные операторы switch**, при этом в ключевых словах case **можно использовать** одинаковые константные выражения.

```
public class Test{
    public static void main(String[] args) {
        Book book = new Book("Dracula", "Bram Stoker",
Genre.HORROR);
        System.out.println("Book " + book.name
                           + " has a type " + book.genre);
        switch(book.genre){
            case FICTION:
                System.out.println("Fiction");
                break;
            case HORROR:
                System.out.println("Horror");
                break;
            case FAIRY_TALE:
                System.out.println("Fairy tale");
                break;
            case FANTASY:
                System.out.println("Fantasy");
                break;
```

IT MENTOR

```
        }
    }
}
class Book{
    String name;
    Genre genre;
    String author;
    Book(String name, String author, Genre genre){
        this.genre = genre;
        this.name = name;
        this.author = author;
    }
}
enum Genre
{
    HORROR,
    FICTION,
    FANTASY,
    FAIRY_TALE

}
```

КЛАСС JAVA.LANG.ENUM. МЕТОДЫ

Хоть по свей сути перечисления и являются типами классов, но они никоим образом **не могут** быть частью иерархий наследования, то есть они не могут быть наследниками других классов и от них наследоваться **нельзя**.

Метод values() возвращает массив, который содержит полный набор всех констант, определенных в текущем перечислении.

```
public class Test{
    public static void main(String[] args) {
        Genre[] genres = Genre.values();
        for (Genre genre : genres) { System.out.printl-
n(genre); }
    }
}
enum Genre
{
    HORROR,
    FICTION,
    FANTASY,
    FAIRY_TALE
}
```

Метод valueOf() мы можем получить одну из констант, которую содержит наш enum. Данный метод имеет логику, обратную методу to-String(), так как мы получаем константу, передавая в параметр этого метода ее строковое представление.

```
public class Test {
    public static void main(String[] args) {
        Genre genre = Genre.valueOf("HORROR");
        System.out.println("Выбран жанр " + genre);
    }
}
```

Метод ordinal() С помощью метода ordinal()мы можем получить порядковый номер константы, который соответствует ее расположению в нашем перечислении. В enum порядок начинается с **нуля**.

IT MENTOR

```
public class Test {
    public static void main(String[] args) {
        for (Genre genre : Genre.values()) {
            System.out.println(genre + " "
                               + genre.ordinal());
        }
    }
}
```

ВОЗМОЖНОСТИ ПЕРЕЧИСЛЕНИЯ.

Основным отличием является то, что конструктор у перечисления может быть только **приватный**. Это обеспечивает невозможность создать экземпляр перечисления вне самого перечисления.

```
enum Day{
    MONDAY("Weekday"),
    TUESDAY("Weekday"),
    WEDNESDAY("Weekday"),
    THURSDAY("Weekday"),
    FRIDAY("Weekday"),
    SATURDAY("Day off"),
    SUNDAY("Day off")
    private String dayType;
    Day(String dayType) {
        this.dayType = dayType;
    }

    public String getDayType() {
        return dayType;
    }
}
```

Для enum методы работают аналогично их работе в классах. Давайте получим тип дня для каждой константы.

```
public class Test {
    public static void main(String[] args) {
        for (Day day : Day.values()) {
            System.out.println("Type of day is " + day.
getDayType());
        }
    }
}
```

ОБЪЯВЛЕНИЕ ПЕРЕЧИСЛЕНИЙ

enum как **отдельный** класс:

```
enum Day{
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

IT MENTOR

enum как член другого класса:

```
class Book{
    String name;
    Genre genre;
    String author;
    enum Genre
    {
        HORROR,
        FICTION,
        FANTASY,
        FAIRY_TALE
    }
    Book(String name, String author, Genre genre){
        this.genre = genre;
        this.name = name;
        this.author = author;
    }
}
```

ПЕРЕЧИСЛЕНИЯ И ИНТЕРФЕЙСЫ.

Как мы с вами определили ранее, перечисления не могут являться звеном иерархии наследования, однако они имеют возможность имплементировать интерфейсы.

```
public enum Currency implements Runnable {
    PENNY(1), NICKLE(5), DIME(10), QUARTER(25);
    private int value;
    Currency(int value) {
        this.value = value;
    }
    @Override
    public void run() {
        System.out.println("Перечисления в Java могут
реализовывать    интерфейсы");
    }
}
```

JAVA CORE

ИСКЛЮЧЕНИЯ

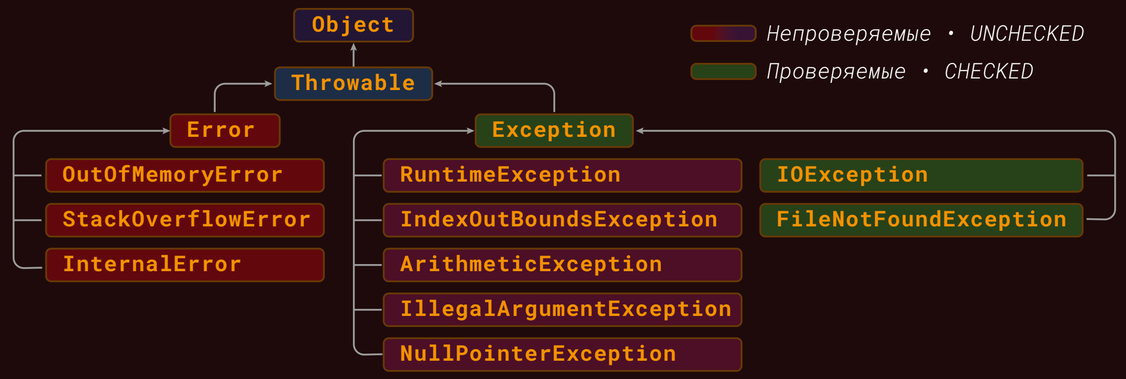
IT MENTOR

IT MENTOR

РАБОТА С ИСКЛЮЧЕНИЯМИ.

Исключения – это ситуация, при возникновении которой программа не может продолжить работу или ее работа становится бессмысленной.

ИЕРАРХИЯ ИСКЛЮЧЕНИЙ



Во главе всего стоит **Throwable**. Этот класс является родителем всего «выбрасываемого» *(от англ. «throw» — бросать)* в Java, т. е. всех ошибок.

Далее эти самые ошибки делятся на две большие группы:

1. **Error** (ошибка) представляет собой критические ошибки, возникающие, когда кончается память в случае некорректной рекурсии (бесконечный вызов методом самого себя), бесконечного цикла, и другие ошибки JVM (Java Virtual Machine — виртуальная машина Java — система, которая исполняет Java-код).
2. **Exception** (исключение) представляет собой ошибки, которые напрямую связаны с кодом. Именно их выкидывает код в случае некорректных данных, отсутствующих данных, разрыва соединения с сервером, некорректного запроса к базе данных, выхода за пределы массива, обращения к null в качестве объекта, деления на 0 и т.д.

Класс **Exception** делится на два больших типа исключений:

3. **Проверяемые** исключения являются прямыми наследниками класса **Exception**. Это те исключения, которые **могут быть предсказаны на момент написания кода,** следовательно, Java потребует от вас их обработать или «прокинуть» методу выше по цепочке вызовов. В случае отсутствия обработки проверяемых исключений Java нам укажет на это и не даст скомпилировать код. Следовательно, о запуске тоже можно забыть.
4. **Непроверяемые** исключения являются прямыми наследниками класса **RuntimeException**. Это те исключения, которые **возникают в момент выполнения приложения.**

В любом случае рекомендуется проверять данные, индексы массивов и прочие параметры. Это избавит вас от появления RuntimeException и его «детей» в вашем приложении.

IT MENTOR

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Шаг №1

ПОЙМАТЬ ОШИБКУ
с помощью try

```
try{
   выполняем
   действие,
   которое
   может вызвать
   ошибку
}
```

Шаг №2

ОБРАБОТАТЬ ОШИБКУ
с помощью catch()

```
catch(){
    что именно
    должно
    произойти,
    если ошибка
    найдена
}
```

Шаг №3

(необязательный)
Вывести блок кода, который выполняется НЕЗАВИСИМО от того, что произошло в блоке try и в блоке catch()

```
finally(){
   return "Finally block";
}
```

```
public static void checkFile(String path) {
    File file = new File(path);
    try {
        check(file);
    } catch (FileNotFoundException e) {
        System.out.println("Файл по пути " + path + "
не найден");
    }
    System.out.println("Проверка завершена");
}
```

На примере выше мы видим, как организована в Java обработка исключения.

- Для обработки исключения (независимо от того, проверяемое оно или нет) мы должны написать ключевое слово try и открыть блок кода.
- В этом блоке мы пишем код, который может выкинуть исключение и который не должен быть выполнен, если вдруг исключение выпадет. Это значит, что если в блоке try есть три строки кода, а исключение вылетит из метода на первой строке, две оставшиеся выполнены не будут.
- Далее с помощью ключевого слова catch мы производим обработку, т. е. пишем код, который должен быть выполнен в том случае, когда исключение было выброшено в блоке try. Следует заметить, что после catch кодовый блок начинается не сразу.
- Обязательно нужно открыть круглые скобки и в них указать тип исключения, которое мы будем отлавливать, и имя, которое будет присвоено исключению в этом кодовом блоке, если оно вылетит.

Если возникнет необходимость отловить сразу несколько типов исключений, то у нас есть возможность использовать так называемый multicatch: если наш код обработки совпадает для нескольких исключений, мы можем использовать в круглых скобках символ (|) и указать разные типы.

```
try {
    // code
}
catch (ExceptionType1 | Exceptiontype2 ex){
    // catch block
}
```

IT MENTOR

Следует учесть, что блоки catch анализируются по очереди. И если вы укажете родительское исключение раньше, чем его ребенка, то будет вызван код из первого совпавшего блока, а не из более узкоспециализированного, но написанного после

```
try {
            throw new FileNotFoundException();
        } catch (Exception e) {
// Этот блок будет выполнен, так как FileNotFoundException является «ребенком» класса Exception
        } catch (IOException e) {
// Этот код не будет выполнен, хотя FileNotFoundException является «ребенком» IOException, так как блок catch выполняется 1 раз, и если в предыдущем блоке тип уже совпал, все следующие не будут вызваны
        } catch (FileNotFoundException) {
// Этот код не будет выполнен, так как находится после тех, что уже попали по  типу под выброшенное исключение
        }
```

Для того чтобы гарантировать выполнение каких-то операций, используется блок **finally**

```
public static void checkFile(String path) {
    File file = new File(path);
    try {
        check(file);
    } catch (FileNotFoundException e) {
        System.out.println("Файл по пути "
                           + path + " не найден");
    } catch (IllegalArgumentException e) {
        System.out.println("Файл по пути "
                           + path + " является папкой");
    } finally {
        System.out.println("Проверка завершена");
    }
}
```

КОНСТРУКЦИЯ TRY-WITH-RESOURCES

При использовании **«try с ресурсами»** мы объявляем те ресурсы, с которыми планируем работать, например соединение с базой данных, открытие каких-то потоков данных и т. д. Объявление ресурсов происходит в круглых скобках после слова try. Все эти ресурсы по окончании работы требуют их закрытия, то есть вызова метода close(), а благодаря данной конструкции этого делать не нужно, так как Java в таком случае неявно создает блок finally, в котором вызывает на открытых ресурсах метод close().

В качестве ресурса можно использовать любой объект, класс которого реализует интерфейс java.lang.AutoCloseable или java.io.Closable.

```
public static String readFirstLineFromFileWithFinally-
Block(String path)
 throws IOException {
    BufferedReader br = new BufferedReader(new Fil-
eReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) {
            br.close();
        }
    }
}
```

IT MENTOR

Перепишем этот блок, используя конструкцию try-with-resources:

```
public static String readFirstLineFromFile(String path)
throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(-
path))) {
        return br.readLine();
    }
}
```

НАПИСАНИЕ СВОИХ ИСКЛЮЧЕНИЙ

Исключение представляет собой практически обычный класс. Единственная его особенность в том, что оно не должно иметь методов, так как все необходимые методы уже объявлены в родительских классах.

Чтобы классу стать исключением, ему нужно унаследоваться от **Exception** (если мы хотим сделать проверяемое исключение) или от **RuntimeException** (если мы хотим сделать непроверяемое исключение). Также наше исключение должно в своем конструкторе вызвать соответствующий конструктор родителя.

КОГДА СТОИТ ПИСАТЬ СВОИ ИСКЛЮЧЕНИЯ?

Написание своих исключений полезно, когда вы не хотите выбрасывать условный IllegalArgumentException в ситуации с папкой, указывая детали ошибки в сообщении. Вместо этого можно реализовать свое исключение FileIsDirectoryException и выбросить его в данной ситуации.

ВЫБРАСЫВАНИЕ СОБСТВЕННЫХ ИСКЛЮЧЕНИЙ

Чтобы в коде «выбросить» свое исключение, нужно написать следующую конструкцию:

```
throw new ИмяИсключения(параметры).
```

У исключений н**есколько конструкторов**. Обычно исключения принимают два параметра (в разных комбинациях):

5. сообщение, которое будет выведено в консоль в момент выброса исключения.
6. исключение, которое можно указать в качестве причины нашего исключения.

```
public class FooException extends Exception {
    public FooException(String message, Throwable
cause) {
        super(message, cause);
    }
}
```

В конструкции try with resources в качестве ресурса можно использовать любой объект, класс которого реализует интерфейс **java.lang.AutoCloseable** или **java.io.Closable.**

IT MENTOR

КОЛЛЕКЦИИ

Введение в коллекции. List и Queue

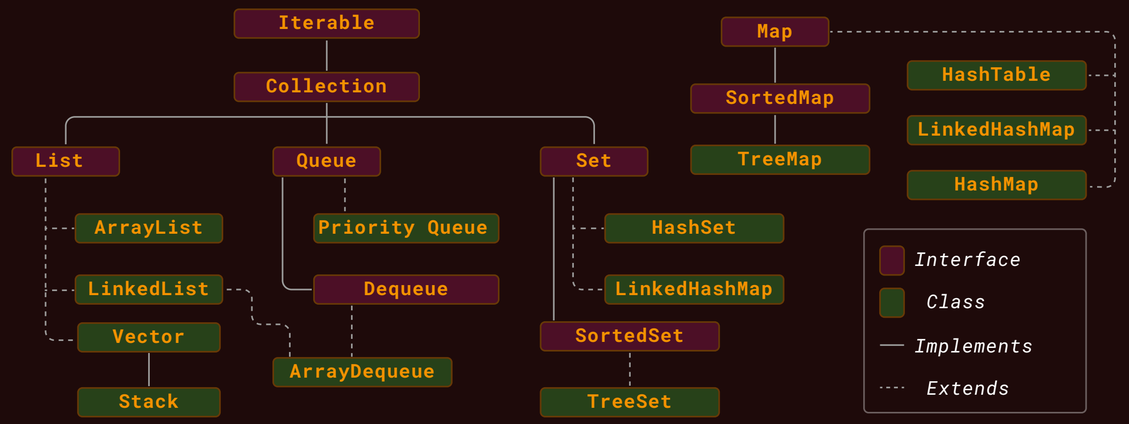
IT MENTOR

ВВЕДЕНИЕ В КОЛЛЕКЦИИ. LIST И QUEUE

Коллекция — это объект, способный хранить группу одинаковых элементов.

По сути, коллекции являются структурами данных. Разные структуры данных определяют разный подход к хранению данных.

ИЕРАРХИЯ КОЛЛЕКЦИЙ



Основными типами коллекций являются:

- List (списки),
- Queue (очереди),
- Set (множества),
- Map (словари/карты/мапы).

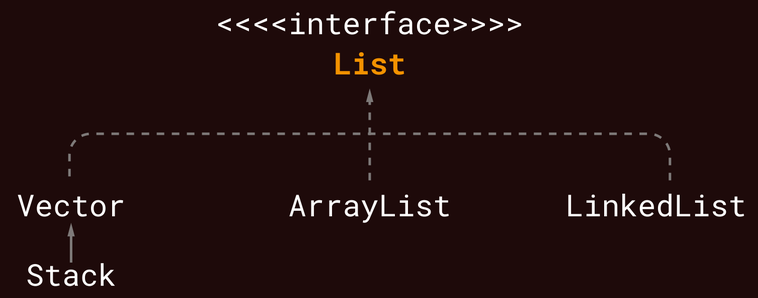
ЧЕМ КОЛЛЕКЦИИ ОТЛИЧАЮТСЯ ОТ МАССИВОВ:

- Коллекции являются более высокоуровневыми абстракциями массивов.
- В зависимости от задач они могут хранить свои данные в массиве (ArrayList) или связывать объекты друг с другом, создавая таким образом определенный порядок (LinkedList). Или вообще не иметь порядок, представляя собой что-то вроде «кучи» или «мешанины» из объектов (HashSet).
- Коллекции обычно сами следят за своим актуальным размером (количеством заполненных ячеек или связанных объектов), сами расширяются (в случае структур на базе массивов) при определенном проценте заполнения внутреннего хранилища и имеют множество других полезных функций.
- Также коллекции можно выводить в консоль без необходимости оборачивать во что-то, как было с Arrays.toString, ведь коллекции являются объектами в привычном нам понимании, т. е. имеют свои методы.

СПИСКИ (LIST)

Список в Java — это последовательность элементов в определенном порядке.

Интерфейс List пакета java.util реализует эту последовательность объектов, упорядоченных определенным образом, называемым List.



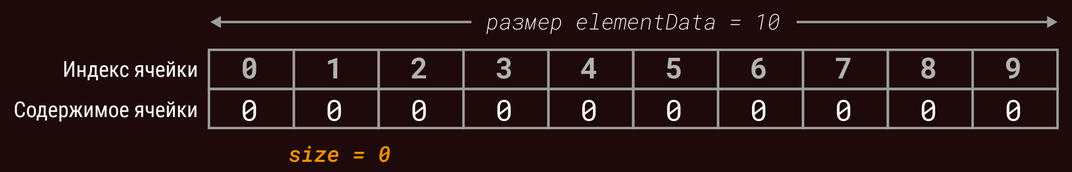
НЕКОТОРЫЕ ХАРАКТЕРИСТИКИ СПИСКОВ:

- Списки могут иметь повторяющиеся элементы.
- В списке также могут быть нулевые элементы.
- Благодаря полиморфизму в списке могут быть элементы разного типа, но одной иерархии.
- Списки всегда сохраняют порядок вставки и разрешают доступ по индексу.
- Аналогично массивам, к элементам списка также можно получить доступ, используя индексы, начиная с 0. Индекс указывает на конкретный элемент с указанным порядковым номером.

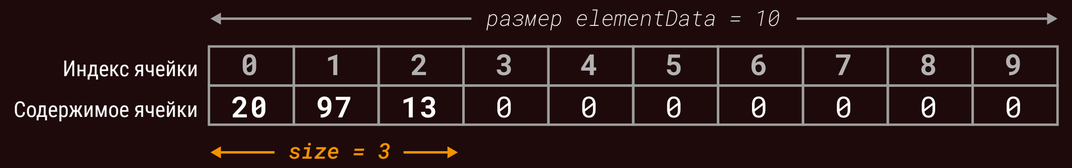
ARRAYLIST.

ArrayList - Этот класс в качестве хранилища использует массив, который в случае недостатка элементов расширяется.

При создании списка, используя конструктор по умолчанию:



После добавления трёх элементов в список:



При создании объекта ArrayList «за кулисами» внутри него создается массив размерностью 10. Так как массивы расширять нельзя, каждый раз, когда свободные ячейки заканчиваются, создается новый массив, равный 1,5 длины старого + 1 ячейка, а затем в него копируется содержимое старого массива. Изначальный размер такого списка (если мы не устанавливаем размер вручную) равен 10.

Мы создаем ArrayList, который будем заполнять объектами типа Integer.

```
List<Integer> nums = new ArrayList<>();
```

Теперь добавим в него элементы с помощью метода **add**.

```
int number = 5;
nums.add(number);
```

Если необходимо добавить элемент в начало, то используется другой набор параметров того же метода **add**.

```
nums.add(0, number);
```

IT MENTOR

Чтобы получить элемент из списка, нужно вызвать метод **get**.

```
int firstNumber = nums.get(0);
```

Если нам нужно заменить элемент по определенному индексу, на помощь приходит метод **set**.

```
nums.set(1, 65);
```

Получить количество элементов списка можно с помощью метода **size**.

```
int numsSize = nums.size();
```

С помощью метода **contains** можно определить, имеется ли элемент в списке.

```
if (nums.contains(5)) {
    System.out.println("В нашем списке имеется значение 5");
}
```

Удалить элемент из списка можно методом **remove**.

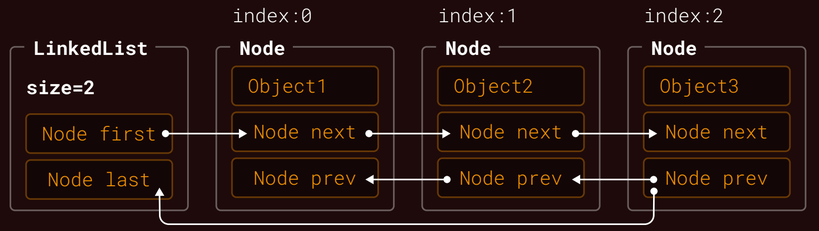
```
nums.remove(5);
```

Чтобы удалить непосредственно объект по содержанию, а не по номеру ячейки, использовать метод **remove** и нужно закастить.

```
nums.remove((Integer) 5);
```

LINKEDLIST

LinkedList в основе лежит не массив, а некий объект Node, который имеет поля, соответствующие следующему элементу, предыдущему элементу и значению. Сам объект LinkedList знает о существовании только первого и последнего элементов списка.



В момент добавления (вызова метода **add**) данный список:

1. создает объект Node,
2. укладывает в него ваш элемент
3. встраивает в цепочку из объектов списка, заполняя внутри себя поля ссылками на прошлый элемент и следующий (если добавляем не в конец).

Создается тем же способом, что и **ArrayList**.

```
List<Integer> linkedNums = new LinkedList<>();
```

LinkedList встречается значительно реже, чем ArrayList, так как ArrayList проще и быстрее в работе. Но в случае необходимости добавлять элементы, например в начало списка, LinkedList дает весомый прирост производительности.

IT MENTOR

КАК ХРАНЯТСЯ ЭЛЕМЕНТЫ В LINKEDLIST

Элементы при добавлении оборачиваются в сущность под названием Node, которая имеет три поля:

- Первое поле хранит в себе сам объект.
- Второе хранит ссылку на предыдущий node.
- Третье хранит ссылку на следующий node.

```
    private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }
```

Сам список хранит только ссылки на первый и последний **node**. Такой список называется **двусвязным**.

КАК ДОБАВЛЯЮТСЯ ЭЛЕМЕНТЫ LINKEDLIST

Метод **add** вызывает метод **linkLast**, передавая ему элемент, а затем возвращает true.

```
    void linkLast(E e) {
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;
        if (l == null)
            first = newNode;
        else
            l.next = newNode;
        size++;
        modCount++;
    }
```

При вызове метода **remove** в цикле ищется узел, чей объект соответствует полю item. Если такой узел найден, то его «отрезают» от тех узлов, что находятся у него в полях **prev** и **next**.

```
    public boolean remove(Object o) {
        if (o == null) {
            for (Node<E> x = first; x != null; x =
x.next) {
                if (x.item == null) {
                    unlink(x);
                    return true;
                }
            }
```

IT MENTOR

```
        } else {
            for (Node<E> x = first; x != null; x =
x.next) {
                if (o.equals(x.item)) {
                    unlink(x);
                    return true;
                }
            }
        }
        return false;
    }
```

При вызове метода **get** (с передачей индекса) метод проверяет, что индекс входит в допустимые значения (индекс >= 0 и индекс < size).

```
    public E get(int index) {
        checkElementIndex(index);
        return node(index).item;
    }
```

Поиск по индексу осуществляется через прохождение цикла по всем элементам цепи: в зависимости от индекса проход начинается с первого или последнего элемента.

В случае, когда нам необходимо получить первый или последний элемент, поиск не производится, происходит возврат объекта из полей списка (first или last).

ИНТЕРФЕЙС QUEUE И КЛАССЫ

Queue отвечает за **реализацию очередей.**

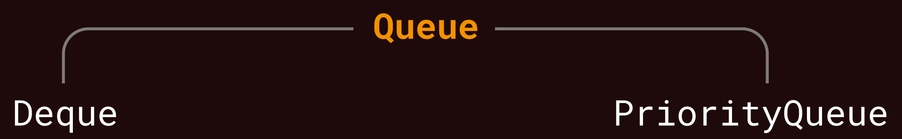
Сам по себе интерфейс Queue является отображением стандартных однонаправленных очередей с **принципом FIFO** (first in, first out — «первым пришёл — первым ушёл»), то есть первый добавленный в очередь элемент первым и будет получен из нее.

Важно запомнить, что очереди **не могут** хранить в себе null.

МЕТОДЫ ИНТЕРФЕЙСА QUEUE

	Выдаёт исключение	Возвращает специальное значение
Вставлять	`add()`	`offer(e)`
Удалять	`remove()`	`poll()`
Исследовать	`element()`	`peek()`

Есть много реализаций очередей: **Queue, Deque, PriorityQueue**



IT MENTOR

ИНТЕРФЕЙС DEQUE

Основным отличием Deque является то, что он представляет собой **двунаправленную** очередь, что обеспечивает возможность добавлять и получать элементы с обоих концов очереди.

То есть функционирует по обоим принципам:

4. FIFO (first in, first out — «первым пришёл — первым ушёл»)
5. LIFO (last in, first out, «последним пришёл — первым ушёл»).

МЕТОДЫ ИНТЕРФЕЙСА DEQUE

	First Element (Head)	
	Выдаёт исключение	Возвращает специальное значение
Вставлять	`addFirst(e)`	`offerFirst(e)`
Удалять	`removeFirst()`	`pollFirst()`
Исследовать	`getFirst()`	`peekFirst()`
	Last Element (Tail)	
Вставлять	`addLast(e)`	`offerLast(e)`
Удалять	`removeLast()`	`pollLast()`
Исследовать	`getLast()`	`peekLast()`

КЛАСС PRIORITYQUEUE

`PriorityQueue` — это класс очереди с приоритетами. Очередь с приоритетами хранит в себе элементы в соответствии с естественным порядком (по алфавиту или числа по возрастанию).

В случае, если несколько элементов имеют одинаковый приоритет, порядок определяется случайным образом. У нас есть возможность самостоятельно задать порядок хранения, передав в параметр конструктора `PriorityQueue Comparator` — интерфейс, который отвечает за сортировку.

КОНСТРУКТОРЫ КЛАССА PRIORITYQUEUE

Конструктор	Описание
`PriorityQueue`()	Создает объект PriorityQueuec начальной емкостью по умолчанию (11), который упорядочивает свои элементы в соответствии с их естественным порядком.
`PriorityQueue`(int initialCapacity)	Создает объект PriorityQueuec указанной начальной емкостью, который упорядочивает свои элементы в соответствии с их естественным порядком .
`PriorityQueue`(int initialCapacity, `Comparator`<? super `E`> comparator)	Создает объект PriorityQueuec указанной начальной емкостью, который упорядочивает свои элементы в соответствии с указанным компаратором.
`PriorityQueue`(`Collection`<? extends `E`> c)	Создает объект PriorityQueue, содержащий элементы указанной коллекции.
`PriorityQueue`(`Comparator`<? super `E`> comparator)	Создает объект PriorityQueuec начальной емкостью по умолчанию, элементы которого упорядочены в соответствии с указанным компаратором.
`PriorityQueue`(`PriorityQueue`<? extends `E`> c)	Создает объект PriorityQueue, содержащий элементы в очереди с указанным приоритетом.
`PriorityQueue`(`SortedSet`<? extends `E`> c)	Создает объект PriorityQueue, содержащий элементы указанного отсортированного набора.

IT MENTOR

МЕТОДЫ КЛАССА PRIORITYQUEUE

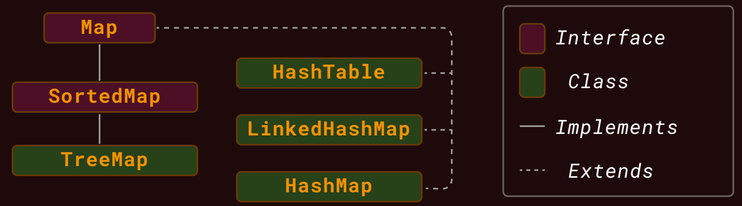
Модификатор и тип	Метод	Описание
boolean	**add**(E e)	Вставляет указанный элемент в эту приоритетную очередь.
void	**clear**()	Удаляет все элементы из этой приоритетной очереди.
Comparator<? super E>	**comparator**()	Возвращает компаратор, используемый для упорядочивания элементов в этой очереди, или nullесли эта очередь отсортирована в соответствии с естественным порядком ее элементов.
boolean	**contains**(Object o)	Возвращает значение true, если эта очередь содержит указанный элемент.
void	**forEach**(Consumer<? super E> action)	Выполняет заданное действие для каждого элемента до Iterable тех пор, пока все элементы не будут обработаны или пока действие не вызовет исключение.
Iterator<E>	**iterator**()	Возвращает итератор по элементам в этой очереди.
boolean	**offer**(E e)	Вставляет указанный элемент в эту приоритетную очередь.
E	**peek**()	Извлекает, но не удаляет заголовок этой очереди или возвращает значение, nullесли эта очередь пуста.
E	**poll**()	Извлекает и удаляет заголовок этой очереди или возвращает значение, nullесли эта очередь пуста.
boolean	**remove**(Object o)	Удаляет один экземпляр указанного элемента из этой очереди, если он присутствует.
boolean	**removeAll**(Collection<?> c)	Удаляет все элементы этой коллекции, которые также содержатся в указанной коллекции (необязательная операция).
boolean	**removeIf**(Predicate<? super E> filter)	Удаляет все элементы этой коллекции, удовлетворяющие заданному предикату.
boolean	**retainAll**(Collection<?> c)	Сохраняет только те элементы этой коллекции, которые содержатся в указанной коллекции (необязательная операция).
int	**size**()	Возвращает количество элементов в этой коллекции.
final Spliterator<E>	**spliterator**()	Создает позднюю привязку и быструю обработку сбоев Spliterator для элементов в этой очереди.
Object[]	**toArray**()	Возвращает массив, содержащий все элементы этой очереди.
<T> T[]	**toArray**(T[] a)	Возвращает массив, содержащий все элементы этой очереди; тип времени выполнения возвращаемого массива соответствует указанному массиву.

IT MENTOR

ВВЕДЕНИЕ В КОЛЛЕКЦИИ. MAP

Map не является потомком **Iterable** и, соответственно, **Collection**, у него нет общих для коллекций методов, поскольку Map — это другой вид объектов, поэтому методы у него свои.

Ассоциативный массив, или **map**, — это хранилище по принципу «ключ-значение», где ключи уникальны, а значения могут повторяться. Такие пары хранятся в ячейках-«корзинах», в которые укладываются по ключу, и создают в этих корзинах связные списки.



HASHMAP

HashMap – это базовая реализация Map, которая основывается на **hashCode** (уникальный числовой код объекта), это значит , что ключами должны быть неизменяемые объекты.

ХРАНЕНИЕ ДАННЫХ В HASHMAP

Данные хранятся элементы в массиве, но не в обычном

```
transient Node<K,V>[] table;
```

Используется уже знакомый нам класс Node, но для нашего мапа он работает иначе.

```
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;

        Node(int hash, K key, V value, Node<K,V> next)
{
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
```

Следует знать, что у массива **table** есть **loadFactor**, при превышении которого массив расширяется. У loadFactor есть значение, по умолчанию равное 0.75, то есть при заполнении 75% из всех доступных ячеек происходит расширение. Элементы в зависимости от реализации остаются в тех же ячейках или переносятся в заранее известные места.

IT MENTOR

БАЗОВЫЕ ОПЕРАЦИИ HASHMAP

Базовыми операциями мапа являются **добавление**, **получение** и **удаление** элемента.

Создадим мап Map<String, String>

```
Map<String, String> phoneBook = new HashMap<>();
```

КАК ДОБАВЛЯЮТСЯ ЭЛЕМЕНТЫ

```
phoneBook.put("Ivan", "122-12-12");
```

В тот момент, когда мы вызываем метод **put**, происходит следующее:

```
public V put(K key, V value){
            return putVal(hash: hash(key: key, value: value,
                                onliIfAbsent: false, evict: true);
```

Здесь происходит вычисление хеша на основе ключа (результата его метода **hashCode**), а затем вызов метода **putVal**.

АЛГОРИТМ РАБОТЫ PUTVAL ТАКОЙ

1. На основе полученного хеша и размера массива **table** рассчитать номер ячейки для хранения объекта «ключ-значение». Hash % table. size = номер ячейки.
2. Создать объект **Node**, обернув в него наш объект.
3. Если в ячейке уже лежит какой-то объект, то проверить, что новый объект и существующие объекты **не равны** по ключам.
4. Если равны, то **затереть** старый объект, заменив его новым.
5. Если не равны, то взять старый объект, положить ссылку на него в новый объект (поле **next**), а новый объект уже положить в эту ячейку. В результате в ячейке получится некий аналог связанного списка.
6. В тот момент, когда в одной ячейке накапливается 7 и более элементов И ячеек стало 64 и более,то превратить образованный в ячейке связанный список из Node в **дерево**. Если меньше 64, то увеличить размер массива провести **рехеш**.
7. В тот момент, когда элементов в ячейке становится 6 или менее, конвертировать дерево обратно в связанный список.

Рехеш — все имеющиеся ноды перераспределяются по бакетам по тем же правилам, но уже с учетом нового размера.

КАК ПРОИСХОДИТ ПОИСК И ПОЛУЧЕНИЕ ЭЛЕМЕНТА

Так же как и при работе метода **put**, всё строится на поиске ячейки на основе хеш-кода ключа + размерности массива **table**.

Метод **get** осуществляет тот же поиск по хешу и размерности, что и добавление.

```
System.out.println(phoneBook.get("Ivan"));
```

После нахождения ячейки ключи сравниваются и элемент или возвращается, или возвращается **true/false** в качестве маркера наличия элемента.

containsValue проходит линейно по всем ячейкам массива table, проверяя все узлы на предмет равенства поля **value** параметру метода.

КАК УДАЛЯЮТСЯ ЭЛЕМЕНТЫ

1. При получении в метод **remove** ключа происходит вычисление его хеша.

2. На основе хеша и размерности вычисляется ячейка.
3. Если в этой ячейке **один** элемент и ключи совпадают, он удаляется.
4. Если в этой ячейке **несколько** элементов, происходит поиск через сравнение ключей.
5. Если ключи **совпадают**, объект удаляется, а поля next остальных узлов **переписываются**, исключая объект из цепи.
6. Если после сравнения всех элементов совпадающий ключ **не найден**, удаления не происходит.

```
phoneBook.remove("Dmitriy");
```

ПРОВЕРКА НА НАЛИЧИЕ ЭЛЕМЕНТА

Для проверки, что элемент есть в мапе по ключу, используется метод **containsKey**.

```
if (phoneBook.containsKey("Dmitriy")) {
    System.out.println(phoneBook.get("Dmitriy"));
} else {
    System.out.println("Контакт не найден");
}
```

Если нам вдруг необходимо проверить наличие значения, в мапе реализован метод **containsValue**. Метод работает перебором всех значений мапы, потому использовать этот метод следует в крайнем случае.

ИТЕРАЦИЯ ПО АССОЦИАТИВНОМУ МАССИВУ

Так как мап не является реализацией интерфейса **Iterable**, итерироваться напрямую по мапу нельзя. Поэтому в мапе были реализованы методы, собирающие из определенной части мапа множество, а уже по этому **множеству** можно итерироваться.

```
for (Map.Entry<String, String> contact: phoneBook.en-
trySet()) {
    System.out.println("Контакт " + contact.getKey() +
": " + contact.getValue());
}
```

Если нам нужны только ключи или только значения, вместо entrySet используются методы keySet или values.

```
for (String name: phoneBook.keySet()) {
    System.out.println("В мапе присутствует контакт по
имени " + name);
}
for (String phoneNumber: phoneBook.values()) {
    System.out.println("В мапе присутствует номер
телефона  " + phoneNumber);
}
```

КЛАСС LINKEDHASHMAP

Класс **LinkedHashMap** - создает связный список элементов в карте, расположенных в том порядке, в котором **они вставлялись**. Это позволяет организовать итерацию по карте в порядке вставки.

LinkedHashMap основан как на хеш-таблице, так и на связанном списке, чтобы расширить функциональность **HashMap**.

Изначально HashMap — это, по сути, **массив с 16 ячейками**, а LinkedHashMap еще и **связывает** все эти ячейки.

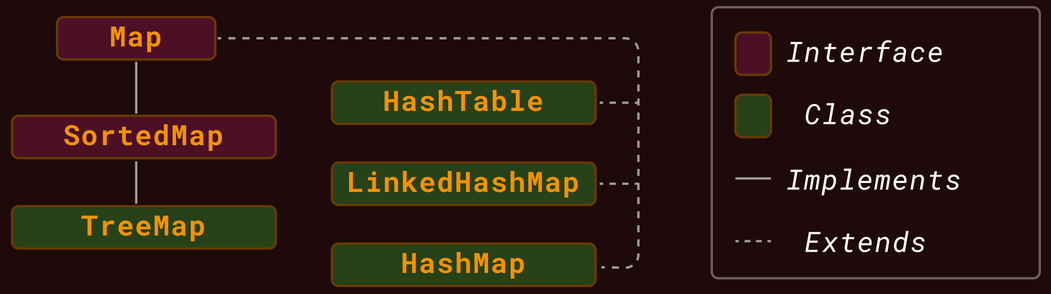
IT MENTOR

TREEMAP

TreeMap — это реализация карты, которая сортирует свои записи в соответствии с естественным порядком своих ключей или, что еще лучше, с использованием компаратора (объект, в который закладывается принцип сортировки), если он предоставляется разработчиком во время построения.

По умолчанию TreeMap сортирует все свои записи в соответствии с их **естественным** порядком. Для целого числа это будет означать возрастающий порядок, а для строк — алфавитный порядок

TreeMap, в отличие от хеш-карты и связанной хеш-карты, нигде **не использует** принцип хеширования, поскольку не использует массив для хранения своих записей.



IT MENTOR

КОЛЛЕКЦИИ. SET. ITERATOR

SET

Множества (sets) являются структурами, особенность которых в том, что все их элементы являются абсолютно уникальными.

По множеству можно проходить с помощью цикла, можно проверять наличие определенного элемента.

HASHSET

Самым популярным примером множества является **HashSet**.
Это множество, которое анализирует хеш-код каждого элемента, в зависимости от него выбирает нужную ячейку в массиве и укладывает туда элемент.

Создается с помощью интерфейса Set.

```
Set<Integer> numbers = new HashSet<>();
```

Множества, в отличие от списков, не имеют методов **get** и **set**, так как индексы в них отсутствуют.

В множество можно добавить элемент с помощью метода **add**.

```
numbers.add(5);
```

Проверить, что элемент есть в множестве, можно тем же методом **contains**.

```
if (numbers.contains(5)) {
    System.out.println("В нашем сете имеется значение 5");
}
```

Размер можно узнать с помощью метода **size**.

```
int numbersSize = numbers.size();
```

Удалить элемент из множества можно тем же методом **remove**

```
numbers.remove(5);
```

Стоит запомнить, что если в вашем классе не реализованы методы **equals** и **hashCode**, HashSet не сможет выбрать корректную ячейку для вашего элемента и/или найти дубликаты вашего элемента.

КАК ХРАНЯТСЯ ЭЛЕМЕНТЫ

Всю работу за HashSet выполняет внутреннее поле **map**.

IT MENTOR

КАК РАБОТАЕТ HASHSET?

В качестве ключа подаются наши с вами объекты, а в качестве значения — объект-заглушка PRESENT.

Модификатор и тип	Метод	Описание
boolean	**add**(E e)	Добавляет указанный элемент в этот набор, если он еще не присутствует.
void	**clear**()	Удаляет все элементы из этого набора.
Object	**clone**()	Возвращает неполную копию этого HashSetэкземпляра: сами элементы не клонируются.
boolean	**contains**(Object o)	Возвращает значение true, если этот набор содержит указанный элемент.
boolean	**isEmpty**()	Возвращает значение true, если этот набор не содержит элементов.
Iterator<E>	**iterator**()	Возвращает итератор по элементам в этом наборе.
boolean	**remove**(Object o)	Удаляет указанный элемент из этого набора, если он присутствует.
int	**size**()	Возвращает количество элементов в этом наборе (его мощность).
Spliterator<E>	**spliterator**()	Создает позднюю привязку и быструю обработку сбоев Spliterator для элементов в этом наборе.
Object[ ]	**toArray**()	Возвращает массив, содержащий все элементы этой коллекции.
<T> T[ ]	**toArray**(T[ ] a)	Возвращает массив, содержащий все элементы этой коллекции; тип времени выполнения возвращаемого массива соответствует указанному массиву.

LINKEDHASHSET И TREESET

По тому же принципу соответствия, как и у HashSet и HashMap, в интерфейсе Set реализованы LinkedHashSet и TreeSet. Они хранят данные в порядке в соответствии с добавлением элементов — в случае **LinkedHashSet**, и в естественном порядке хранения данных — в **TreeSet**.

ИТЕРАТОРЫ

Одним из ключевых методов интерфейса Collection является метод **Iterator**<E> iterator(), который возвращает **итератор** — то есть объект, реализующий интерфейс Iterator.

```
List<String> items = new ArrayList<>()
Iterator<String> iter = items.iterator();
```

Интерфейс Iterator имеет три основных метода:

boolean	**hasNext()** Возвращает true, если итерация имеет больше элементов
E	**next()** Возвращает следующий элемент в итерации
default void	**remove()** Удаляет из базовой коллекции последний возвращенный элемент этим итератором (дополнительная операция)

IT MENTOR

ИНТЕРФЕЙС LISTITERATOR

ListIterator — это расширение, добавляющее новые функции для перебора списков.

```
ListIterator<String> listIterator =
                    items.listIterator(items.size());
```

Обратите внимание, как мы можем указать начальную позицию, которая в данном случае является концом списка.

void	add(E e)	Вставляет указанный элемент в список (необязательная операция).
boolean	**hasNext**()	Возвращает значение true, если этот итератор списка имеет больше элементов при перемещении по списку в прямом направлении.
boolean	**hasPrevious**()	Возвращает значение true, если этот итератор списка имеет больше элементов при перемещении по списку в обратном направлении.
E	**next**()	Возвращает следующий элемент в списке и перемещает позицию курсора.
int	**nextIndex**()	Возвращает индекс элемента, который будет возвращен при последующем вызове next().
E	**previous**()	Возвращает предыдущий элемент в списке и перемещает позицию курсора назад.
int	**previousIndex**()	Возвращает индекс элемента, который будет возвращен при последующем вызове previous().
void	**remove**()	Удаляет из списка последний элемент, который был возвращен оператором next()или previous() (необязательная операция).
void	**set**(E e)	Заменяет последний элемент, возвращенный указанным элементом next()или previous() указанным элементом (необязательная операция).

ИТЕРАЦИЯ ПО АССОЦИАТИВНОМУ МАССИВУ

Так как мапа не является реализацией интерфейса **Iterable**, итерироваться напрямую по мапе нельзя. Поэтому в мапе были реализованы методы, собирающие из определенной части мапы множество, а уже по этому множеству можно итерироваться.

```
for (Map.Entry<String, String> contact: phoneBook.en-
trySet()) {
    System.out.println("Контакт " + pair.getKey() + ":
" + pair.getValue());
}
```

Если нам нужны только ключи или только значения, вместо entrySet используются методы **keySet**() или **values**()

```
for (String name: phoneBook.keySet()) {
    System.out.println("В мапе присутствует контакт по
имени " + name);
}
for (String phoneNumber: phoneBook.values()) {
    System.out.println("В мапе присутствует номер
телефона  " + phoneNumber);
}
```

IT MENTOR

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

StreamAPI, Optional

IT MENTOR

ФУНКЦИОНАЛЬНЫЕ ИНТЕРФЕЙСЫ И ЛЯМБДА-ВЫРАЖЕНИЯ

Декларативное программирование — это стиль, в котором описывается необходимый результат, а не порядок действий.

ФУНКЦИОНАЛЬНЫЕ ИНТЕРФЕЙСЫ (ПЕРВЫЙ ИНСТРУМЕНТ ДЕКЛАРАТИВНОГО СТИЛЯ)

Функциональный интерфейс — интерфейс, который содержит один абстрактный метод. При этом в нем, может быть, **неограниченное** количество **статических** и **дефолтных методов**.

Функциональные интерфейсы принято помечать аннотацией **@FunctionalInterface**, чтобы компилятор проверял, действительно ли интерфейс является функциональным

Predicate<T> Принимает на вход значение, проверяет состояние и возвращает boolean-значение в качестве результата:

```
public interface Predicate<T> {
    boolean test(T t);
}
```

Consumer<T> в методе accept выполняет некоторое действие над объектом типа T, при этом ничего не возвращая:

```
public interface Consumer<T> {
    void accept(T t);
}
```

Supplier<T> Ничего не принимает на вход, но возвращает какое-то значение:

```
public interface Supplier<T> {
    T get();
}
```

Function<T,R> Принимает значение в качестве аргумента одного типа и возвращает другое значение. Часто используется для преобразования одного значения в другое:

```
public interface Function<T, R> {
    R apply(T t);
}
```

UnaryOperator<T> Является частным случаем функции с той лишь разницей, что принимает значение в качестве аргумента одного типа и возвращает значение этого же типа:

```
public interface UnaryOperator<T> {
    T apply(T t);
}
```

BinaryOperator<T> Является частным случаем функции, хранит метод apply, в котором принимает в качестве параметра два объекта типа T, выполняет над ними бинарную операцию и возвращает ее результат так же в виде объекта типа T:

```
public interface BinaryOperator<T> {
    T apply(T t1, T t2);
}
```

СПОСОБЫ СОРТИРОВКИ

В Java есть 2 интерфейса, с помощью которых мы можем отсортировать объекты, которые хранятся в нашей структуре данных.

1. Comparable. Необходимо имплементировать его в классе, объекты которого мы хотим сортировать. Для реализации этого интерфейса мы должны переопределить в нашем классе метод compareTo(E item)(сравнивает текущий объект с другим объектом, переданным в качестве параметра), в котором указываем логику, которая определяет порядок или приоритет хранения объектов данного класса.

```
public class Person implements Comparable {
    private String name;
    private int age;
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    @Override
    public int compareTo(Person o) {
        if (this.getAge() > o.getAge()) {
            return 1;
        } else if (this.getAge() < o.getAge()) {
            return -1;
        }
        return 0;
    }
}
```

2. Comparator. Является внешним по отношению к сравниваемому типу элемента. Необходимо создать отдельный класс, в котором будут сравниваться объекты по какому-то конкретному параметру. В данном классе необходимо реализовать интерфейс Comparator и переопределить метод compare(T o1, T o2)

IT MENTOR

```
class PersonSortingByAge implements Comparator {
    @Override
    public int compare(Person o1, Person o2) {
        if (o1.getAge() > o2.getAge()) {
            return 1;
        } else if (o1.getAge() < o2.getAge()) {
            return -1;
        }
        return 0;
    }
}
```

АНОНИМНЫЕ КЛАССЫ

Список действий для реализации функциональных интерфейсов стандартным способом:

1. Создаем отдельный класс, который имплементирует интерфейс.
2. Реализовываем в классе абстрактные методы из интерфейса.
3. Создаем в другом классе объект нашего класса.
4. Через этот объект применяем реализованные методы.

Анонимный класс — это класс, который не имеет названия и объект которого создается в момент объявления этого самого класса.

Создаем анонимный класс, который имплементирует интерфейс SomeInterface

```
SomeInterface obj = new SomeInterface() {
    @Override
    public void someMethod() {
… ; }// логика метода
};
```

Лямбда-выражения представляют собой анонимную реализацию какого-то метода.

Синтаксис таких выражений:

1. Без аргумента.

```
() -> System.out.println("Hello")
```

2. Один аргумент.

```
s -> System.out.println(s)
```

3. Два аргумента.

```
(x, y) -> x + y
```

4. С открытым типом аргументов.

```
(Integer x, Integer y) -> x + y
```

5. Множественный оператор.

IT MENTOR

```
(x, y) -> {
    System.out.println(x);
    System.out.println(y);
    return (x + y);
}
```

Лямбда-выражения имеют два блока, которые разделены стрелкой (->). До стрелки указываются **параметры** метода, а после — **тело метода**.

Также **лямбда-выражения** являются более краткой формой записи анонимных классов, благодаря чему можно реализовать функциональный интерфейс еще более компактно.

```
SomeInterface obj = new SomeInterface(
                () -> …; // логика метода
)
```

ССЫЛКИ НА МЕТОДЫ

Ссылки на методы (Method References) — это компактные лямбда-выражения для методов, у которых уже есть имя.

К примеру, у нас есть следующее лямбда-выражение

```
Consumer<String> consumer = str ->
                            System.out.println(str);
```

Можно переписать с помощью **method references:**

```
Consumer<String> consumer = System.out::println;
```

IT MENTOR

ССЫЛКИ НА МЕТОДЫ БЫВАЮТ СЛЕДУЮЩИХ ТИПОВ:

1. Ссылка на нестатический метод любого объекта конкретного типа

```
SomeType::someMethod
Function<String, String> function = s ->
                                    s.toLowerCase();
System.out.println(function.apply("JAVA 8"));
Function<String, String> function =
                                  String::toLowerCase;
System.out.println(function.apply("JAVA 8"));
```

2. Ссылка на нестатический метод конкретного объекта

```
someObject::someMethod
Integer integer = 5;
Supplier<String> supplier = () -> integer.toString();
System.out.println(supplier.get());
Integer integer = 5;
Supplier<String> supplier = integer::toString;
System.out.println(supplier.get());
```

3. Ссылка на статический метод

```
SomeClass::someStaticMethod
Function<String, Boolean> function = e ->
                                    Boolean.valueOf(e);
System.out.println(function.apply("TRUE"));
Function<String, Boolean> function =
                                    Boolean::valueOf;
System.out.println(function.apply("TRUE"));
```

4. Ссылка на конструктор

```
SomeClass::new
Function<String, Integer> function = (d) ->
                                     new Integer(d);
System.out.println(function.apply("4"));
Function<String, Integer> function = Integer::new;
System.out.println(function.apply("4"));
```

IT MENTOR

STREAM API

Stream API — это инструмент Java, который позволяет работать с разными структурами данных в функциональном стиле.

Stream — представляет из себя последовательность объектов, над которой можно выполнить несколько операций.

СПОСОБЫ СОЗДАНИЯ СТРИМОВ

1. Из коллекции

```
collection.stream()
```

2. Задать значения явно

```
Stream.of(v1,… vN)
```

3. Также есть специальные стримы для примитивов

```
IntStream.of(1, … N)
DoubleStream.of(1.1, … N)
```

4. Из массива

```
Arrays.stream(arr)
```

5. Из файла, где каждая новая строка становится элементом

```
Files.lines(file_path)
```

6. С помощью стримбилдера

```
Stream.builder().add(...)....build().
```

7. Задать последовательность чисел от и до:

```
IntStream rangeS = IntStream.range(9, 91); // не включительно.
IntStream rangeS = IntStream.rangeClosed(9, 91); // включительно.
```

8. Можно создать параллельный стрим

```
collection.parallelStream()
```

IT MENTOR

9. Также у нас есть возможность создания бесконечных стримов:
   - В Stream.iterate мы задаем начальное значение, а также указываем, как будем получать следующее, используя предыдущий результат:

```
Stream<Integer> iterStream = Stream
                             .iterate(1, m -> m + 1)
```

   - Stream.generate позволяет бесконечно генерировать постоянные и случайные значения, которые соответствуют указанному выражению:

```
Stream<String> generateStream = Stream
                                .generate(() -> “f5”)
```

МЕТОДЫ СТРИМОВ

В Stream API разделяют 2 вида методов:

1. Промежуточные (конвейерные) операции. Может быть сколько угодно.
2. Терминальные (конечные) операции. Может быть только один. После его выполнения стрим завершается.

Промежуточные операции — это операции, которые каким-то образом модифицируют стрим перед вызовом терминального оператора.

Пока не вызван терминальный метод, ничего не происходит, так как промежуточные методы ленивые.

Ленивые методы — методы, которые не срабатывают до их непосредственного вызова. Это значит, что они обрабатывают данные и ждут команды, чтобы передать их терминальному методу.

1. Метод **filter** используется для фильтрации элементов стрима

```
collection.stream().filter(«java»::equals).count();
```

2. Метод **sorted** сортирует элементы в естественном порядке или на основе объекта типа Comparator.

```
collection.stream().sorted()
                   .collect(Collectors.toList())
```

3. Метод **limit** ограничивает стрим по указанному количеству.

```
collection.stream().limit(10)
                   .collect(Collectors.toList())
```

IT MENTOR

4. Метод **skip** пропустит указанное количество элементов стрима.

```
collection.stream().skip(3).findFirst().orElse("4")
```

5. Метод **distinct** сделает все элементы стрима уникальными (уберет повторы).

```
collection.stream().distinct()
                   .collect(Collectors.toList())
```

6. Метод **peek** выполнит заданное действие над каждым элементом, вернет стрим с исходными элементами (используется для проверок).

```
collection.stream()
                .map(String::toLowerCase)
                .peek((e) -> System.out.print("," + e))
                .collect(Collectors.toList())
```

7. Метод **map** используется для преобразования элемента стрима с помощью функции, которая принимает элемент стрима и возвращает новый элемент

```
Stream.of("3","4","5").map(Integer::parseInt)
                      .map(x -> x + 10)
                      .forEach(System.out::println);
```

8. Метод **flatMap** сработает как map, но преобразует один элемент в ноль, один или множество других. Используется для преобразования элемента стрима с помощью функции, которая принимает элемент стрима и возвращает новый стрим.

```
collection.stream().flatMap((p) -> Arrays
                      .asList(p.split(",")).stream())
                      .forEach(System.out::println);
```

IT MENTOR

Терминальные операции — это операции, которые завершают Stream и выдают конечный результат вычислений.

1. Метод **findFirst** вернет первый элемент, соответствующий заданному условию.

```
collection.stream().findFirst().orElse("10")
```

2. Метод **findAny** вернет любой элемент, соответствующий заданному условию.

```
collection.stream().findAny().orElse("10")
```

3. Метод **collect** собирает стрим в какую-то структуру данных.

```
collection.stream().filter((s) ->
   s.contains("10")).collect(Collectors.toList())
```

Collector — это специальный интерфейс, который может выполнять некоторые терминальные операции над стримами. Например, собирать их в список или в Map, группировать и даже объединять в одну строку.

4. Метод **count** подсчет элементов в стриме.

```
collection.stream().filter("f5"::equals).count()
```

5. Метод **anyMatch** возвращает true, когда хоть один элемент соответствует условиям.

```
collection.stream().anyMatch("f5"::equals)
```

6. Метод **noneMatch** возвращает true, когда ни один элемент не соответствует условиям.

```
collection.stream().noneMatch("b6"::equals)
```

7. Метод **allMatch** возвращает true, если все элементы стрима соответствуют условиям.

```
collection.stream().allMatch((s) -> s.contains("8"))
```

8. Метод **min** возвращает наименьший элемент стрима.

```
collection.stream().min(String::compareTo).get()
```

9. Метод **max** возвращает наибольший элемент стрима.

```
collection.stream().max(String::compareTo).get()
```

IT MENTOR

10. Метод **forEach** применит функцию ко всем элементам, но порядок выполнения гарантировать не может.

```
set.stream().forEach((p) -> p.append("_2"));
```

11. Метод **forEachOrdered** применит функцию ко всем элементам по очереди, порядок выполнения гарантирует.

```
list.stream().forEachOrdered((p) -> p.append("_nv"));
```

12. Метод **toArray** «упакует» стрим в массив.

```
collection.stream().map(String::toLowerCase)
                   .toArray(String[]::new);
```

13. Метод **reduce** преобразует все элементы в один объект.

```
collection.stream()
                   .reduce((c1, c2) -> c1 + c2)
                   .orElse(0)
```

OPTIONAL

Перед тем, как разбираться в Optional, давайте выясним какую проблему он решает. Представим, что у нас есть класс UserRepository, который работает с хранилищем наших пользователей, в данном примере это будет обычная HashMap.

Допустим, мы хотим найти пользователя в базе данных по идентификатору. После чего нам требуется вывести на консоль длину имени пользователя. На первый взгляд это простой и безобидный пример.

```
final Person person = personRepository.findById(1L);
final String firstName = person.getFirstName();
System.out.println("Длина твоего имени: " + firstName.
length());
```

Выполнив этот код, мы можем получить исключение:

```
Exception in thread "main" java.lang.NullPointerExcep-
tion: Cannot invoke "dev.struchkov.example.optional.
Person.getFirstName()" because "person" is null
    at dev.struchkov.example.optional.Main.test(Main.
java:18)
    at dev.struchkov.example.optional.Main.main(Main.
java:13)
```

Почему это могло произойти? Дело в том, что мы не учли того факта, что пользователя с идентификатором **1L** может не быть в нашей **HashMap**. И у нас нет варианта, кроме как вернуть **null**.

```
public class PersonRepository {
    private final Map<Long, Person> persons;
    public PersonRepository(Map<Long, Person> persons)
{
        this.persons = persons;
    }
    public Person findById(Long id) {
        return persons.get(id);
    }
}
```

Реализация **PersonRepository**

Во время вызова метода null-объекта вы получите **NullPointerException**. Это головная боль новичков в программировании. Но неважно, новичок ли вы в Java или за плечами у вас десять лет опыта — всегда есть вероятность, что вы столкнетесь с **NullPointerException**.

Тони Хоар, написал: “Изобретение null в 1965 году было мой ошибкой на миллиард долларов. Я не мог устоять перед искушением ввести нулевую ссылку просто потому, что ее было так легко реализовать”.

А мы теперь имеем то, что имеем. Так что же делать в таком случае? Можно проверять все возвращаемые значения на null, вот так:

```
final Person person = personRepository.findById(1L);
if (person != null) {
    final String firstName = person.getFirstName();
    System.out.println(“Длина твоего имени: “ + first-
Name.length());
}
```

Теперь если пользователя нет, то мы просто не выполняем код? Допустим, что пользователь с таким идентификатором все же есть, выполняем код:

```
Exception in thread “main” java.lang.NullPointerExcep-
tion: Cannot invoke “String.length()” because “first-
Name” is null
    at dev.struchkov.example.optional.Main.test(Main.
java:20)
    at dev.struchkov.example.optional.Main.main(Main.
java:13)
```

И мы снова получили NullPointerException, только на этот раз проблема оказалась в поле firstName, теперь оно null, и уже вызов метода length() выбросил исключение. Не беда, добавим еще одну проверку:

IT MENTOR

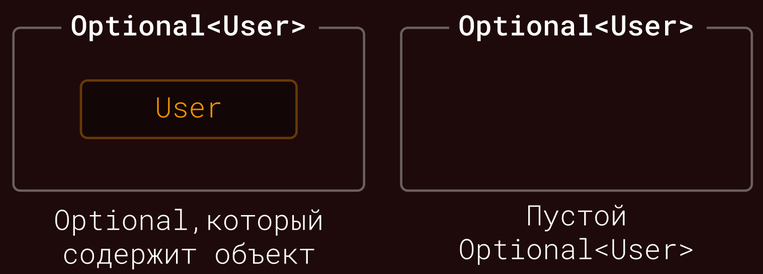
```
final User user = userRepository.findById(1L);
if (user != null) {
    final String firstName = user.getFirstName();
    if (firstName != null) {
        System.out.println("Длина твоего имени: " +
firstName.length());
    }
}
```

Хм, где-то мы явно свернули не туда. Эти проверки просто мешают бизнес-логике и раздражают. Они снижают общую читаемость нашей программы.

Так приходилось жить до появления Optional в Java. Что же имеем теперь?

ВВЕДЕНИЕ В OPTIONAL

Можно воспринимать Optional, как некую коробку, обертку, в которую кладется какой-либо объект. Optional всего лишь контейнер: он может содержать объект некоторого типа T, а может быть пустым.



Optional, который содержит объект

Пустой Optional<User>

Перепишем наш пример с использованием Optional и посмотрим, что изменится:

```
package dev.struchkov.example.optional;
import java.util.Map;
import java.util.Optional;
public class PersonRepository {
    private final Map<Long, Person> persons;
    public PersonRepository(Map<Long, Person> persons)
{
        this.persons = persons;
    }
    public Optional<Person> findById(Long id) {
        return Optional.ofNullable(persons.get(id));
    }
}


final Optional<Person> optPerson = personRepository.
findById(1L);
if (optPerson.isPresent()) {
    final Person person = optPerson.get();
    final String firstName = person.getFirstName();
    if (firstName != null) {

        System.out.println("Длина твоего имени: " +
firstName.length());
    }
}
```

Вы скажете: это не выглядит проще, даже более того, добавилась еще одна строка?!! Но на самом деле случилось концептуально важное событие, вы точно знаете что метод **findById**() возвращает контейнер, в котором объекта может не быть. Больше вы не ожидаете внезапного **null** значения, если только разработчик этого метода не сумасшедший, ведь ничто не мешает ему вернуть **null** вместо Optional

Здесь мы с вами увидели два основных метода: **isPresent**() возвращает **true**, если внутри есть объект, а метод get() возвращает этот объект.

Возвращаемся к нашему примеру, давайте упростим этот код с помощью других методов Optional, чтобы увидеть какие преимущества нам это даст. Обо всех этих методах мы еще подробнее поговорим ниже.

```
final Optional<Person> optPerson = personRepository.
findById(1L);
optUser.map(person -> person.getFirstName())
        .ifPresent(
                firstName -> System.out.println(“Длина
твоего имени: “ + firstName.length())
        );
```

Мы воспользовались методом **map**(), который преобразует наш Optional в Optional другого типа. В данном случае мы получили фамилию пользователя, то есть преобразовали **Optional**<User> в **Optional<String>.**

Вот такой вариант показывает это нагляднее:

```
final Optional<Person> optPerson = personReposito-
ry.findById(1L);
final Optional<String> optFirstName = optPerson.
map(user -> user.getFirstName());
optFirstName.ifPresent(
    firstName -> System.out.println(“Длина твоего
имени: “ + firstName.length())
);
```

А дальше мы вызвали метод **ifPresent**(), в котором мы вызвали вывод на консоль, можно еще чуть-чуть сократить код:

```
personRepository.findById(1L)
        .map(User::getFirstName)
        .ifPresent(
                firstName -> System.out.println(“Длина
твоего имени: “ + firstName.length())
        );
```

IT MENTOR

МЕТОДЫ OPTIONAL

Теперь, когда мы поняли зачем нам нужен Optional и разобрали пример использования, давайте рассмотрим остальные методы этого класса. Optional обладает большим количеством методов, правильное использование которых позволяет добиться более простого и понятного кода.

СОЗДАНИЕ OPTIONAL

У данного класса нет конструкторов, но есть три статических метода, которые и создают экземпляры класса,

МЕТОД OPTIONAL.OFNULLABLE(T T)

Optional.ofNullable принимает любой тип и создает контейнер. Если в параметры этого метода передать null, то будет создан пустой контейнер.

Используйте этот метод, когда есть вероятность, что упаковываемый объект может иметь null значение.

```
public class PersonRepository {
    private final Map<Long, Person> persons;
    public PersonRepository(Map<Long, Person> persons)
{
        this.persons = persons;
    }
    public Optional<Person> findById(Long id) {
        return Optional.ofNullable(persons.get(id));
    }
}
```

В этом примере мы используем **ofNullable**(), потому что метод **Map#get**() может вернуть нам **null**.

МЕТОД OPTIONAL.OF(T T)

Этот метод аналогичен **Optional.ofNullable**, но если передать в параметр **null** значение, то получим старый добрый **NullPointerException**.

Используйте этот метод, когда точно уверены, что упаковываемое значение не должно быть **null**.

```
public class PersonRepository {
    private final Map<Long, Person> persons;
    public PersonRepository(Map<Long, Person> persons)
{
        this.persons = persons;
    }
    public Optional<Person> findByLogin(String login) {
        for (Person person : persons.values()) {
            if (person.getLogin().equals(login)) {
                return Optional.of(person);
            }
        }
        return Optional.empty();
    }
}
```

Новый метод позволяет найти пользователя по логину. Для этого мы проходимся по значениям **Map** и сравниваем логины пользователей с переданным, как только находим совпадение вызываем метод **Optional.of**(), потому что мы уверены, что такой объект существует.

IT MENTOR

МЕТОД OPTIONAL.EMPTY()

Но что делать, если пользователь с переданным логином не был найден? Метод все равно должен вернуть Optional.

Можно вызвать Optional.ofNullable(null) и вернуть его, но лучше воспользоваться методом Optional.empty(), который возвращает пустой Optional.

Обратите внимание на последний пример, там мы как раз используем Optional.empty().

ПОЛУЧЕНИЕ СОДЕРЖИМОГО

Мы разобрали методы, которые позволяют нам создавать объект Optional. Теперь изучим методы, которые позволят нам доставать значения из контейнера.

МЕТОД ISPRESENT()

Прежде, чем достать что-то, неплохо убедиться, что это что-то там действительно есть. И с этим нам поможет метод isPresent(), который возвращает true, если в контейнере есть объект и false в противном случае.

```
Optional<Person> optPerson = personRepository.findBy-
Id(1L);
if(optPerson.isPresent()) {
    // если пользователь найден, то выполняем этот блок кода
}
```

Фактически это обычная проверка, как если бы мы сами написали if (value != null). И если зайти в реализацию метода isPresent(), то это мы там и увидим:

```
public boolean isPresent() {
    return value != null;
}
```

МЕТОД ISEMPTY()

Метод isEmpty() это противоположность методу isPresent(). Метод вернет true, если объекта внутри нет и false в противном случае. Думаю, вы уже догадались, как выглядит реализация этого метода:

```
public boolean isEmpty() {
    return value == null;
}
```

МЕТОД GET()

После того, как вы убедились в наличии объекта с помощью предыдущих методов, вы можете смело достать объект из контейнера с помощью метода get().

```
Optional<Person> optPerson = personRepository.findBy-
Id(1L);
if(optPerson.isPresent()) {
    final Person person = optPerson.get();
        // остальной ваш код
}
```

IT MENTOR

Конечно, вы можете вызвать метод get() и без проверки. Но если объекта там не окажется, то вы получите NoSuchElementException.

МЕТОД IFPRESENT()

Помимо метода **is**Present(), имеется метод **if**Present(), который принимает в качестве аргумента функциональный интерфейс Consumer. Это позволяет нам выполнить какую-то логику над объектом, если объект имеется.

Давайте с помощью этого метода выведем имя и фамилиюпользователя на консоль.

```
personRepository.findById(id).ifPresent(
        person -> System.out.println(person.getFirst-
Name() + “ “ + person.getLastName())
);
```

МЕТОД IFPRESENTORELSE()

Метод **ifPresent**() ничего не сделает, если у вас нет объекта, но если вам и в этом случае необходимо выполнить какой-то код, то используйте метод **ifPresentOrElse**(), который принимает в качестве параметра еще и функциональный интерфейс **Runnable**.

Возьмем наш пример выше и выведем в консоль Иван Иванов, если пользователь не был найден.

```
personRepository.findById(id).ifPresentOrElse(
        person -> System.out.println(person.getFirst-
Name() + “ “ + person.getLastName()),
        () -> System.out.println(“Иван Иванов”)
);
```

МЕТОД ORELSE()

Метод делает ровно то, что ожидается от его названия: возвращает значение в контейнере или значение по умолчанию, которое вы указали.

Например, мы можем вернуть пользователя по идентификатору, а если такого пользователя нет, то вернем анонимного пользователя.

```
public Person getPersonOrAnon(Long id) {
    return personRepository.findById(id)
            .orElse(new Person(-1L, “anon”, “anon”,
“anon”));
}
```

Используйте этот метод, когда хотите вернуть значение по умолчанию, если контейнер пустой.

Некоторым приходит в голову довольно странная конструкция: **objectOptional.orElse(null)**, которая лишает использование Optional всякого смысла. **Никогда так не делайте**, и бейте по рукам тем, кто так делает.

IT MENTOR

МЕТОД ORELSEGET()

Метод похож на предыдущий, но вместо возвращения значения, он выполнит функциональный интерфейс Supplier.

Возьмем пришлый пример и будем также выводить в консоль сообщение о том, что пользователь не был найден.

```
public Person getPersonOrAnonWithLog(Long id) {
    return personRepository.findById(id)
            .orElseGet(() -> {
                System.out.println("Пользователь не был найден, отправляем анонимного");
                return new Person(-1L, "anon", "anon", "anon");
            });
}
```

Используйте этот метод, когда вам недостаточно просто вернуть какое-то дефолтное значение, но и требуется выполнить какую-то более сложную логику.

МЕТОД ORELSETHROW()

Этот метод вернет объект если он есть, в противном случае выбросит стандартное исключение **NoSuchElementException("No value present")**.

МЕТОД ORELSETHROW(SUPPLIER S)

Этот метод позволяет вернуть любое исключение вместо стандартного **NoSuchElementException**, если объекта нет.

```
public Person getPersonOrThrow(Long id) {
    return personRepository.findById(id)
            .orElseThrow(() -> new NotFoundExcep-
tion("Пользователь не найден"));
}
```

ЭЛЕГАНТНЫЙ ORELSETHROW

Как вам такой вариант использования **orElseThrow**()

```
public Person getPersonOrElegantThrow(Long id) {
    return personRepository.findById(id)
            .orElseThrow(notFoundExcep-
tion("Пользователь {0} не найден", id));
}
```

На мой взгляд это самый эстетический вариант использования. Для этого вам нужно в классе исключения добавить метод, который будет возвращать **Supplier**.

IT MENTOR

Вот пример **NotFoundException**, мы используем два метода **notFoundException**: один позволяет передать просто строку, а другой использует **MessageFormat.format()**, чтобы формировать сообщение с параметрами.

```
import java.text.MessageFormat;
import java.util.function.Supplier;
public class NotFoundException extends RuntimeException
{
    public NotFoundException(String message) {
        super(message);
    }
    public NotFoundException(String message, Object...
args) {
        super(MessageFormat.format(message, args));
    }
    public static Supplier<NotFoundException> notFound-
Exception(String message, Object... args) {
        return () -> new NotFoundException(message,
args);
    }
    public static Supplier<NotFoundException> notFound-
Exception(String message) {
        return () -> new NotFoundException(message);

    }
}
```

ПРЕОБРАЗОВАНИЕ

Optional также имеет ряд методов, которые может быть знакомы вам по стримам: **map**, **filter** и **flatMap**. Они имеют такие же названия, и делают примерно то же самое.

МЕТОД FILTER()

Если внутри контейнера есть значение и оно удовлетворяет переданному условию в виде функционального интерфейса **Predicate**, то будет возвращен новый объект Optional с этим значением, иначе будет возвращен пустой Optional.

Например, мы сделаем метод, который возвращает только взрослых пользователей по **id**.

```
public Optional<Person> getAdultById(Long id) {
    return personRepository.findById(id)
            .filter(person -> person.getAge() > 18);
}
```

Используйте его, когда вам нужен контейнер, объект в котором удовлетворяет какому-то условию.

МЕТОД MAP()

Если внутри контейнера есть значение, то к значению применяется переданная функция, результат помещается в новый Optional и возвращается, в случае отсутствия значения будет возвращен пустой контейнер. Для преобразования используется функциональный интерфейс **Function**.

Сделаем метод, который по идентификатору пользователя возвращает **Optional<String>**, содержащий имя и фамилию этого пользователя.

```
public Optional<String> getFirstAndLastNames(Long id) {
    return personRepository.findById(id)
            .map(person -> person.getFirstName() + " "
+ person.getLastName());
}
```

Используйте этот метод, когда необходимо преобразовать объект внутри контейнера в другой объект.

МЕТОД FLATMAP()

Как уже было сказано, **map**() оборачивает возвращаемый результат лямбды Function. Но что, если эта лямбда у нас будет возвращать уже обернутый результат, у нас получится **Optional<Optional<T>>**. С таким дважды упакованным объектом будет сложно работать.

Для примера, мы можем запрашивать какие-то данные о пользователе из другого сервиса, и этих данных тоже может не быть, поэтому возникает второй контейнер. Вот как это будет выглядеть:

```
Optional<Optional<String>> optUserPhoneNumber = person-
Repository.findById(1L)
        .map(person -> {
            Optional<String> optPhoneNumber = phoneNum-
berRepository.findByPersonId(person.getId());
            return optPhoneNumber;
        });
```

В таком случае используйте **flatMap**(), он позволит вам избавиться от лишнего контейнера.

```
Optional<String> optUserPhoneNumber = personRepository.
findById(1L)
        .flatMap(person -> {
            Optional<String> optPhoneNumber = phoneNum-
berRepository.findByLogin(user.getLogin());
            return optPhoneNumber;
        });
```

МЕТОД STREAM()

В Java 9 в Optional был добавлен новый метод **stream**(), который позволяет удобно работать со стримом от коллекции Optional элементов.

Допустим, вы запрашивали множество пользователей по различным идентификаторам, а в ответ вам возвращался **Optional<Person>**. Вы сложили полученные объекты в коллекцию, и теперь с ними нужно как-то работать. Можно создать стрим от этой коллекции, и используя метод **flatMap**() стрима и метод **stream**() у **Optional**, чтобы получить стрим существующих пользователей. При этом все пустые контейнеры будут отброшены.

```
final List<Optional<Person>> optPeople = new Array-
List<>();
final List<Person> people = optPeople.stream()
        .flatMap(optItem -> optItem.stream())
        .toList();
```

IT MENTOR

МЕТОД OR()

Начиная с Java 11 добавили новый метод **or**(). Он позволяет изменить пустой Optional передав новый объект, раньше так сделать было нельзя.

Важно понимать, что этот метод не изменяет объект **Optional**, а создает и возвращает новый.

Например, мы запрашиваем пользователя по идентификатору, и если его нет в хранилище, то мы передаем Optional анонимного пользователя.

```
public Optional<Person> getPersonOrAnonOptional(Long
id) {
    return personRepository.findById(id)
            .or(() -> Optional.of(new Person(-1L,
“anon”, “anon”, “anon”, 0L)));
}
```

КОМБИНИРОВАНИЕ МЕТОДОВ

Все перечисленные методы возвращают в ответ **Optional**, поэтому вы можете составлять из них цепочки, прямо как у стримов.

Пример, оторванный от реальности, но иллюстрирующий цепочку методов:

```
final LocalDateTime now = LocalDateTime.now();
final DayOfWeek dayWeek = Optional.of(now)
        .map(LocalDateTime::getDayOfWeek)
        .filter(dayOfWeek -> DayOfWeek.SUNDAY.equals(-
dayOfWeek))
        .orElse(DayOfWeek.MONDAY);
```

ПРОЧИЕ НЮАНСЫ

Далее идут дополнительные советы и житейские мудрости, которые помогут вам еще лучше овладеть этим инструментом.

КОГДА СТОИТ ИСПОЛЬЗОВАТЬ OPTIONAL

Если открыть javadoc Optional, можно найти там прямой ответ на данный вопрос.

Optional в первую очередь предназначен для использования в качестве типа возвращаемого значения метода, когда существует явная необходимость представлять «отсутствие результата» и где использование null может вызвать ошибки.

IT MENTOR

КАК НЕ СТОИТ ИСПОЛЬЗОВАТЬ OPTIONAL

А теперь разберемся основные ошибки при использовании Optional.

- **Как параметр метода**

Не стоит использовать Optional, в качестве параметра метода. Если пользователь метода с параметром Optional не знает об этом, он может передать методу null вместо Optional.empty(). И обработка null приведет к исключению NullPointerException.

- **Как свойство класса**
- Также **не используйте Optional для объявления свойств класса.**

Во-первых, у вас могут возникнуть проблемы с такими популярными фремворками, как Spring Data/Hibernate. Hibernate не может замапить значения из БД на Optional напрямую, без кастомных конвертеров.

Хотя некоторые фреймворки, такие как Jackson отлично интегрируют Optional в свою экосиситему. Как раз с Jackson можно подумать об использовании Optional в качестве свойства класса, если создание объекта этого класса вы никика не контролируете. Например, при использовании Webhook.

Во-вторых, не стоит забывать, что Optional это объект, который обычно нужен на пару секунд, после чего он может быть безболезненно удален сборщиком мусора. Но на создание объекта нужно время, и если мы храним Optional в качестве поля, он может оставаться там вплоть до самой остановки программы. Вряд ли это приведет к проблемам в небольших приложениях, но все же учтите это.

В-третьих, использование таких полей будет неудобным Optional не имплементирует интерфейс Serializable. Проще говоря, любой объект, который содержит хотя бы одно Optional поле, нельзя сериализовать. Хотя с приходом микросервисов платформенная сериализация не является настолько важной, как раньше.

Решением в такой ситуации может быть использование Optional для геттеров класса. Однако у этого подхода есть один недостаток. Его нельзя полностью интегрировать с Lombok. `Optional` getters не поддерживаются библиотекой.

- **Как обертка коллекции**
- **Не оборачивайте коллекции в Optional.**

Любая коллекция является контейнером сама по себе. Чтобы вернуть пустую коллекцию, вместо null, можно воспользоваться следующими методами `Collections.emptyList()`, `Collections.emptySet()` и прочими.

- **Optional не должен равняться null**

Присваивание `null` вместо объекта `Optional` разрушает саму концепцию его использования. Никто из пользователей вашего метода не будет проверять Optional на эквивалентность с `null`. Вместо присваивания `null` следует использовать `Optional.empty()`.

Примитивы и Optional

Для работы с обертками примитивов есть `java.util.OptionalDouble, java.util.OptionalInt и java.util.OptionalLong`, которые позволяют избегать лишних автоупаковок и распаковок. Однако несмотря на это, на практике используются они крайне редко.

Все эти классы похожи на Optional, но не имеют методов преобразования. В них доступны только: `get, orElse, orElseGet, orElseThrow, ifPresent и isPresent.`

IT MENTOR

РЕЗЮМЕ

Класс **Optional** не решает проблему **NullPointerException** полностью, но при правильном применении позволяет снизить количество ошибок, сделать код более читабельным и компактным. Использование Optional не всегда уместно, но для возвращаемых значений из методов он подходит отлично.

Основные моменты, которые стоит запомнить

- Если методу требуется вернуть null, верните Optional;
- Не используйте Optional в качестве параметра методов и как свойство класса;
- Проверяйте Optional перед тем, как достать его содержимое;
- Никогда так не делайте: optional.orElse(null);

IT MENTOR

РАБОТА С ПАМЯТЬЮ В JAVA

Память в Java, Garbage Collector

IT MENTOR

ПАМЯТЬ В JAVA, GARBAGE COLLECTOR

Чтобы приложение работало максимально оптимизировано, JVM разделяет память на 2 части:

- Стек (Stack)
- Куча (Heap)

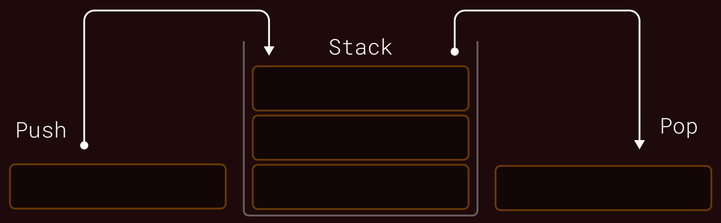
СТЕК

Стек (Stack) представляет собой небольшой (по сравнению с кучей) участок памяти, который хранит в себе примитивные типы данных, ссылки на объекты и методы.

Стек работает по схеме LIFO (Last In First Out). При вызове каждого нового метода из предыдущего метода на вершине стека создается отдельный блок памяти, который называется фрейм (**frame**). В этот фрейм и помещается последний вызванный метод.

В момент окончания работы метода программа выходит из него, и в этот момент блок памяти, в котором хранилось выполнение данного метода, очищается, и текущий фрейм удаляется с вершины стека.

При этом поток выполнения программы возвращается к тому методу, из которого был вызван последний, и на вершине стека опять оказывается фрейм, в котором содержится метод, к которому вернулась программа.



- При переполнении стека будет выброшено исключение java.lang. StackOverFlowError.
- Локальные переменные, созданные в методе, удаляются из стека во время окончания работы метода.
- Стек является быстрым участком памяти, так как процессы в нем проходят значительно быстрее, чем в куче.
- Стек заполняется фреймами и освобождается от них по мере вызова и завершения новых методов.

КУЧА

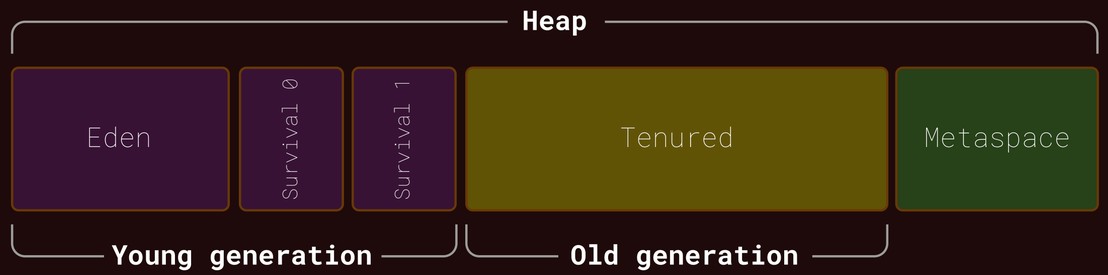
Куча, или **heap** — область памяти, которая создается для хранения всех объектов. Новые объекты всегда напрямую создаются в куче; в момент их создания там предварительно выделяется участок, в котором будет храниться новосозданный объект.

Куча разделена на несколько основных участков, каждый из которых исполняет свою конкретную роль:

1. **Young Generation** — область, в которой создаются объекты, а также хранятся в ней первое время. В момент ее полного заполнения происходит быстрая сборка мусора.
2. **Old (Tenured) Generation** — в этой области хранятся объекты-«долгожители», то есть объекты, которые пережили несколько сборок мусора. Объекты по достижении определенного «возраста» (количества сборок) перемещаются из Young Generation в Old Generation.

IT MENTOR

3. **Metaspace** — область памяти, которая появилась в 8-й версии Java вместо PermGeneration. В Metaspace виртуальная машина хранит метаданные загруженных классов. Также здесь находятся все статические элементы программы.



- В случае полного заполнения кучи выбрасывается ошибка java.lang. OutOfMemoryError.
- Если переменная примитивного типа является полем объекта, то она хранится в куче вместе с объектом, частью которого она является.
- Для того чтобы происходила очистка Heap, в Java существует сборщик мусора, в отличии от стека, в котором эти процессы происходят автоматически.
- Скорость работы кучи, а именно доступ к данным, которые в ней хранятся, медленнее, чем скорость работы стека.

СБОРЩИК МУСОРА

Мусором считается любой объект, до которого нельзя добраться по ссылкам от корня программы. Соответственно, такие объекты невозможно никак использовать в нашем приложении и от них можно избавиться.

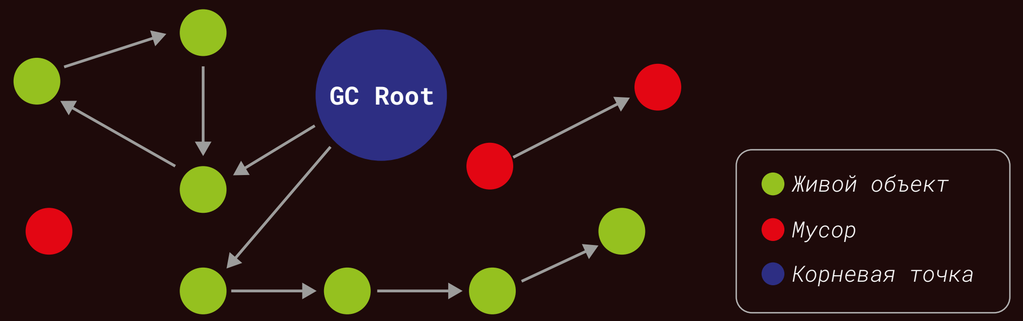
Сборка мусора — автоматический процесс, который осуществляет сборщик мусора (Garbage Collector). Способы определения, какие именно объекты подлежат удалению, и сами способы зачисток варьируются в зависимости от версии самого сборщика.

Garbage Collector должен делать всего две вещи:

Находить мусор, то есть неиспользуемые объекты.

Объект считается неиспользуемым, если ни одна из сущностей в коде, выполняемом в данный момент, не содержит ссылок на него либо цепочка ссылок, которая могла бы связать объект с некоторой сущностью приложения, обрывается.Освобождать память от мусора.

Tracing (трассировка)



Для **очистки памяти** от мусора существуют **два** основных метода:

1. **Copying collectors.**
2. **Mark-and-sweep.**

IT MENTOR

COPYING COLLECTORS

При copying-collectors-подходе память делится на две части, from-space и to-space, при этом сам принцип работы следующий:

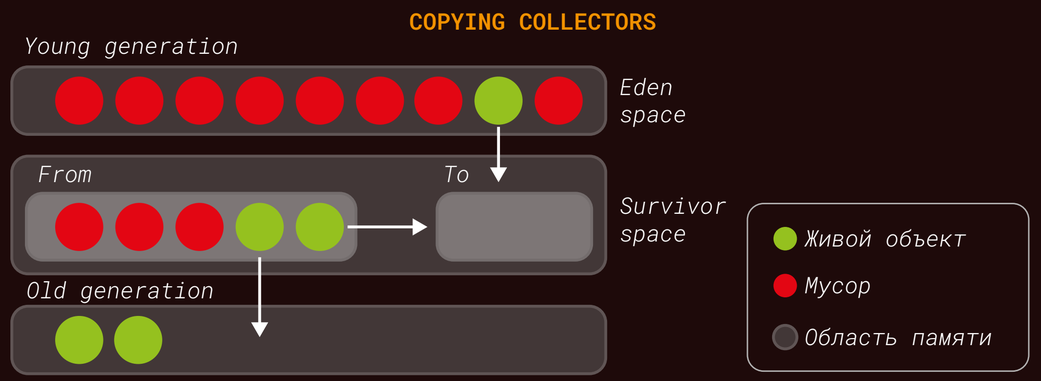
1. Объекты из eden и попадают в в from-space.
2. Когда from-space заполняется, приложение приостанавливается.
3. Запускается сборщик мусора. Находятся живые объекты в from-space и копируются в to-space.
4. Когда все объекты скопированы, from-space полностью очищается.
5. to-space и from-space меняются местами.

Плюсы:

- × Объекты плотно забивают память.

Минусы:

- × Приложение должно быть остановлено на время, необходимое для полного прохождения цикла сборки мусора.
- × В худшем случае (когда все объекты живые) form-space и to-space обязаны быть одинакового размера.



MARK-AND-SWEEP

1. Объекты создаются в памяти.
2. В момент, когда нужно запустить сборщик мусора, приложение приостанавливается.
3. Сборщик проходится по дереву объектов, помечая живые объекты.
4. Сборщик проходится по всей памяти, находя все неотмеченные куски памяти и сохраняя их в free list.
5. Когда новые объекты начинают создаваться, они создаются в памяти, доступной во free list.

Минусы:

- × Приложение не работает, пока происходит сборка мусора.
- × Время остановки напрямую зависит от размеров памяти и количества объектов.
- × Если не использовать сжатие объектов, память будет использоваться неэффективно

IT MENTOR

ПРОЦЕСС СБОРКИ МУСОРА

Механизм сборки мусора — это процесс освобождения места в куче для возможности добавления новых объектов.

Виртуальная машина Java, применяя механизм сборки мусора, гарантирует, что любой объект, обладающий ссылками, остается в памяти — все объекты, которые не достижимы из исполняемого кода, ввиду отсутствия ссылок на них удаляются с высвобождением отведенной для них памяти.

Программа может успешно завершить работу, не исчерпав ресурсов свободной памяти или даже не приблизившись к этой черте, и поэтому ей так и не потребуются «услуги» сборщика мусора.

Необходимость создания и удаления большого количества объектов существенным образом сказывается на производительности приложений, и если быстродействие программы является важным фактором, следует тщательно обдумывать решения, связанные с созданием объектов, — это, в свою очередь, уменьшит и объем мусора, подлежащего утилизации.

ТИПЫ СБОРЩИКОВ МУСОРА

Java HotSpot VM предоставляет разработчикам на выбор 7 различных сборщиков мусора:

- **Serial (последовательный)** — самый простой вариант для приложений с небольшим объемом данных, не требовательных к задержкам. На данный момент используется сравнительно редко, но на слабых компьютерах может быть выбран виртуальной машиной в качестве сборщика по умолчанию. -**XX:+UseSerialGC**.
- **Parallel (параллельный)** наследует подходы к сборке от последовательного сборщика, но добавляет параллелизм в некоторые операции, а также возможности по автоматической подстройке под требуемые параметры производительности. Параллельный сборщик включается опцией -**XX:+UseParallelGC**.
- **Concurrent Mark Sweep (CMS)** нацелен на снижение максимальных задержек путем выполнения части работ по сборке мусора параллельно с основными потоками приложения. Подходит для работы с относительно большими объемами данных в памяти. **XX:+UseConcMarkSweepGC**.
- **Garbage-First (G1)** создан для замены CMS, особенно в серверных приложениях, работающих на многопроцессорных серверах и оперирующих большими объемами данных. G1 включается опцией Java -XX:+UseG1GC.
- **Сборщик мусора Z (ZGC)** — это масштабируемый сборщик мусора с малой задержкой. ZGC выполняет всю дорогостоящую работу одновременно, не останавливая выполнение потоков приложения более чем на 10 мс, что делает его подходящим для приложений, требующих малой задержки и/или использующих очень большую кучу (несколько терабайт). -**XX:UseZGC**.**Epsilon GC** — сборщик мусора без операций. Занимается распределением памяти, но не реализует никакой фактический механизм восстановления памяти. Как только доступная куча Java будет исчерпана, JVM выключится. -**XX:+UnlockExperimentalVMOptions** и -**XX:+UseEpsilonGC**.
- **Shenandoah GC** — сборщик, который стремится поддерживать короткие паузы даже на кучах большого объема за счет выполнения как можно большего количества работы по сборке мусора в конкурентном режиме, то есть одновременно с работой основных потоков приложения. Использование Shenandoah включается опцией -**XX:+UseShenandoahGC**.

SQL

Structured Query Language

IT MENTOR

IT MENTOR

ВВЕДЕНИЕ В БАЗЫ ДАННЫХ

База данных — это такая же структура данных, как и массивы или коллекции, только с большим количеством своих преимуществ и хранением информации за пределами нашего приложения.

Основные преимущества:

- Базы позволяют настраивать доступы к информации.
- Базы данных отлично подходят для хранения огромного количества данных.
- Базы данных дают возможность работать с ними одновременно из разных точек приложения.
- Базу данных можно гибко настроить для различных проверок хранимой информации.

Система управления базами данных — это совокупность языковых и программных средств, которая осуществляет доступ к данным, позволяет их создавать, менять и удалять, обеспечивает безопасность данных и т.д.

Наиболее популярными системами управления реляционными базами данных являются: PostgreSQL, MySQL, Oracle DB.

Зачастую базы разделяют на 2 основных типа:

- **Реляционные**
- **Нереляционные**

IT MENTOR

РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ (SQL)

SQL — это язык программирования, предназначенный для работы с наборами фактов и отношениями между ними.

Данные в реляционных базах хранятся в таблицах, которые распределены по смыслу или по сущностям, которые в них хранятся. Каждому свойству сущности соответствует отдельная колонка.

Преимущества:

- × Можно легко и быстро манипулировать хранимыми данными
- × Можно легко и быстро структуировать хранимые данные

Таблицы в реляционной базе данных имеют возможность быть **связанными** между собой, то есть иметь отношение друг к другу (relation — отношение) . Эти связи настраиваются с помощью ключей, первичных и внешних.

Первичный ключ (primary key) — это колонка, которая отвечает за хранение уникальной информации для каждого объекта или каждой строки. Как правило, такой колонкой является поле id.

Внешний ключ (foreign key) — это колонка в таблице, которая содержит значения первичного ключа из другой таблицы, с которой необходимо поддерживать связь.

НЕРЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ (NOSQL)

Используют определенную модель, которая оптимизирована под конкретный тип хранимой информации.Типы баз NoSQL:

1. **Хранилище документов**. Хранит данные каждого объекта в полях, которые могут быть в различных форматах, например JSON или XML.
2. **Хранилище документов**. Тип хранения данных аналогичен реляционной модели.
3. **Хранилище внешних индексов**. Используется как посредник, который ускоряет доступ к другим базам данных. Часто используется для индексированных данных других хранилищ.
4. **Хранилище данных временных рядов**. Используется для информации, которая не является приоритетной и особо важной. Такой тип часто служит в роли своеобразного кэша.
5. **Хранилище объектов**. Используется для хранения и извлечения больших файлов и данных: изображения, видео, приложений и т.д.
6. **Хранение в виде «ключи-значения»**. Способ хранения похожий на структуру Map. Вся информация хранится в одной таблице, а кажый набор данных обладает уникальным ключом, по которому этот набор можно получить.

IT MENTOR

ВВЕДЕНИЕ В SQL

SQL (Structured Query Language) — это язык структурированных (т. е. создаваемых в соответствии с конкретным шаблоном) запросов, который используется в качестве эффективного способа сохранения данных, поиска их частей, обновления, извлечения из базы и удаления.

ГРУППЫ ОПЕРАТОРОВ SQL:

1. Операторы **определения данных** (Data Definition Language, DDL):
   - создать таблицу — **CREATE** TABLE table_name...
   - изменить таблицу — **ALTER** TABLE table_name...
   - удалить таблицу — **DROP** TABLE table_name...
2. Операторы **манипуляции данными** (Data Manipulation Language, DML):
   - осуществить выборку данных — **SELECT** FROM table_name...
   - положить данные в таблицу — **INSERT** INTO table_name...
   - обновить данные в таблице — **UPDATE** table_name...
   - удалить данные из таблицы — **DELETE** FROM table_name...
3. Операторы **определения доступа к данным** (Data Control Language, DCL):
   - для назначения привилегий пользователям — **GRANT** SELECT ON...
   - отмены привилегий — **REVOKE** SELECT ON...
   - запрета действий для пользователя — **DENY** SELECT ON...
4. Операторы **управления транзакциями** (Transaction Control Language, TCL):
   - **COMMIT** применяет транзакцию;
   - **ROLLBACK** откатывает все изменения, сделанные в контексте текущей транзакции;
   - **SAVEPOINT** разбивает транзакцию на более мелкие.

СОЗДАНИЕ БАЗЫ ДАННЫХ ЧЕРЕЗ SQL SHELL.

```
\l список всех созданных баз
```

Для того чтобы создать новую базу данных, необходимо прописать следующую команду:

```
CREATE DATABASE database_name;
```

Чтобы приступить к работе с ней, если работа производится из командной строки, нужно подключиться к ней с помощью команды

```
\c database_name
```

IT MENTOR

СОЗДАНИЕ ТАБЛИЦЫ ЧЕРЕЗ КОМАНДНУЮ СТРОКУ.

Таблица — основной объект базы данных, предназначенный для хранения данных в структурированном виде.

```
CREATE TABLE table_name;
CREATE TABLE book
(
    id     BIGSERIAL   NOT NULL PRIMARY KEY,
    title  VARCHAR(60) NOT NULL,
    author VARCHAR(60) NOT NULL,
    amount INT         NOT NULL
);
```

PRIMARY KEY — этот параметр нарекает текущую колонку первичным ключом, то есть является идентификатором для сущности, хранимой в таблице.

С помощью команды **DROP** можно удалить таблицу:

```
DROP TABLE book;
```

ТИПЫ ДАННЫХ POSTGRESQL.

- **BIGSERIAL** — целочисленный тип данных с автоматической инкрементацией при добавлении новых строк
- **VARCHAR** — строковый тип данных
- **INT**— целочисленный тип данных
- **BOOLEAN** — логическое значение (true/false)
- **DATE** — календарная дата (год,месяц,день).
- **MONEY** — сумма в валюте

Полный список типов данных:
https://www.postgresql.org/docs/current/datatype.html

IT MENTOR

ОСНОВНЫЕ CRUD-ОПЕРАЦИИ

ВСТАВКА ДАННЫХ.

Вставить новые данные в таблицу можно с помощью оператора **INSERT**.

```
INSERT INTO book (title, author, amount)
VALUES ('War and Peace', 'L.N.Tolstoy', 5);
```

- **INSERT INTO** — оператор вставки, без которого эта операция невозможна.
- **VALUES** — оператор, который указывает на информацию, подлежащую сохранению в таблице.

ПОЛУЧЕНИЕ ДАННЫХ.

```
SELECT *
FROM book;
```

«*» (звездочка) — указывает на то, что мы хотим получить все записи.

ОБНОВЛЕНИЕ ДАННЫХ

```
UPDATE book
SET amount = 10
WHERE id = 1;
```

- **SET** — сам оператор установки новых данных.
- **WHERE** — этот оператор переводится как «где», то есть дает нам понять, для каких сущностей проводить обновление.

Если у нас несколько колонок подлежат обновлению,
то они перечисляются через **запятую**.

УДАЛЕНИЕ ДАННЫХ

```
DELETE
FROM book
WHERE id = 1;
```

IT MENTOR

ВЫБОРКА ДАННЫХ И ВЛОЖЕННЫЕ ЗАПРОСЫ

ОПЕРАТОР WHERE.

Ситуация, когда требуется сделать выборку по определенному условию, встречается очень часто. Для этого в операторе **SELECT** существует параметр **WHERE**, после которого следует условие для ограничения строк. Если запись удовлетворяет этому условию, то попадает в результат, иначе отбрасывается.

```
SELECT поля_таблиц
FROM список_таблиц
WHERE условия_на_ограничения_строк
          [логический_оператор другое_условия_на_
ограничения_строк];
```

СПЕЦИАЛЬНЫЕ ОПЕРАТОРЫ.

1. **IS [NOT] NULL** — позволяет узнать равно ли проверяемое значение NULL.

```
SELECT *
FROM FamilyMembers
WHERE status IS NOT NULL;
```

2. **[NOT] BETWEEN** min **AND** max — позволяет узнать расположено ли проверяемое значение столбца в интервале между min и max.

```
SELECT *
FROM Payments
WHERE unit_price BETWEEN 100 AND 500;
```

3. **[NOT] IN** — позволяет узнать входит ли проверяемое значение столбца в список определённых значений.

```
SELECT member_name
FROM FamilyMembers
WHERE status IN (‘father’, ‘mother’);
```

4. **[NOT] LIKE** шаблон [ESCAPE символ]— позволяет узнать соответствует ли строка определённому шаблону.

```
SELECT member_name
FROM FamilyMembers
WHERE member_name LIKE ‘% Quincey’;
```

ТРАФАРЕТНЫЕ СИМВОЛЫ.

- символ подчеркивания (_), который можно применять вместо любого единичного символа в проверяемом значении
- символ процента (%) заменяет последовательность любых символов (число символов в последовательности может быть от 0 и более) в проверяемом значении.

IT MENTOR

Шаблон	Описание
never%	Сопоставляется любым строкам, начинающимся на «never»
%ing	Сопоставляется любым строкам, закачинающимся на «ing»
_ing	Сопоставляется любым строкам, имеющим длину 4 символа, при этом 3 последних обязательно должны быть «ing». Например, слова «sing» и «wing»

ESCAPE-СИМВОЛ.

ESCAPE-символ используется для экранирования трафаретных символов. В случае если вам нужно найти строки, содержащие проценты (а процент — это зарезервированный символ), вы можете использовать ESCAPE-символ.

```
SELECT job_id
FROM Jobs
WHERE progress LIKE '3!%' ESCAPE '!';
```

Если бы мы не экранировали трафаретный символ, то в выборку попало бы всё, что начинается на 3.

ЛОГИЧЕСКИЕ ОПЕРАТОРЫ.

- Оператор **NOT** — меняет значение специального оператора на противоположный
- Оператор **OR** — общее значение выражения истинно, если хотя бы одно из них истинно
- Оператор **AND** — общее значение выражения истинно, если они оба истинны
- Оператор **XOR** — общее значение выражения истинно, если один и только один аргумент является истинным

АГРЕГАТНЫЕ ФУНКЦИИ

Агрегатные функции — функции, которые берут группы значений и сводят их к одиночному значению.

- **SQL** предоставляет несколько агрегатных функций:
- **COUNT** — производит подсчет записей, удовлетворяющих условию запроса;
- **SUM** — вычисляет арифметическую сумму всех значений колонки;
- **AVG** — вычисляет среднее арифметическое всех значений;
- **MAX** — определяет наибольшее из всех выбранных значений;
- **MIN** — определяет наименьшее из всех выбранных значений.

Агрегатные функции применяются для значений, не равных **NULL**. Исключением является функция **COUNT**().

IT MENTOR

ОПЕРАТОР GROUP BY

Оператор **GROUP BY** группирует данные при выборке, имеющие одинаковые значения в какой-то из колонок.

```
SELECT [константы, агрегатные_функции, поля_
группировки]
FROM имя_таблицы
GROUP BY поля_группировки;
```

Следует иметь в виду, что для **GROUP BY** все значения **NULL** трактуются как **равные**, т.е. при группировке по полю, содержащему NULL-значения, все такие строки попадут в одну группу.

SQL-ПСЕВДОНИМЫ

Псевдонимы используются для представления столбцов или таблиц с именем отличным от оригинального. Это может быть полезно для улучшения читабельности имён и создания более короткого наименования столбца или таблицы.

AS — с помощью данного оператора мы можем в итоговой выборке создать новую колонку. Название новой колонки стоит справа от оператора, а информация, которая будет храниться, слева.

ОПЕРАТОР ORDER BY

Оператор **ORDER BY** упорядочивает вывод запроса согласно значениям в том или ином количестве выбранных столбцов.

```
SELECT поля_таблиц
FROM список_таблиц
ORDER BY столбец_1 [ASC | DESC][, столбец_n [ASC |
DESC]];
```

- **DESC** — сортировка по убыванию
- **ASC** (по умолчанию) — сортировка по возрастанию

NULL-элементы при сортировке по возрастанию всегда будут последними, а при порядке убывания — первыми.

При использовании **ORDER BY** можно задавать несколько условий порядка. Сначала данные сортируются, основываясь на их значениях для первого выражения. Строки с одинаковым значением для первого условия будут отсортированы по второму условию и т. д.

ОПЕРАТОР HAVING

Для фильтрации строк по значениям **агрегатных** функций используется оператор **HAVING**.

```
SELECT [константы, агрегатные_функции, поля_
группировки]
FROM имя_таблицы
GROUP BY поля_группировки
HAVING условие_на_ограничение_строк_после_группировки
```

IT MENTOR

Отличие **HAVING** от **WHERE**

- **WHERE** — сначала выбираются записи по условию, а затем могут быть сгруппированы, отсортированы и т.д.
- **HAVING** — сначала группируются записи, а затем выбираются по условию, при этом, в отличие от **WHERE**, в нём можно использовать значения агрегатных функций

ВЛОЖЕННЫЙ SQL ЗАПРОСЫ.

Вложенный запрос — это запрос на выборку, который используется внутри инструкции SELECT, INSERT, UPDATE или DELETE или внутри другого вложенного запроса. Подзапрос может быть использован везде, где разрешены выражения.

```
SELECT поля_таблиц
FROM список_таблиц
WHERE конкретное_поле IN (SELECT поле_таблицы
                          FROM таблица)
```

Подзапрос может возвращать скаляр (одно значение), одну строку, один столбец или таблицу (одну или несколько строк из одного или нескольких столбцов). Они называются скалярными, столбцовыми, строковыми и табличными подзапросами.

Вложенные подзапросы обрабатываются «снизу вверх». То есть сначала обрабатывается вложенный запрос самого нижнего уровня. Далее значения, полученные по результату его выполнения, передаются и используются при реализации подзапроса более высокого уровня и т.д.

СКАЛЯРНЫЙ ПОДЗАПРОС

Скалярный подзапрос — запрос, возвращающий единственное скалярное значение (строку, число и т.д.).

```
SELECT *
FROM FamilyMembers
WHERE birthday = (SELECT MAX(birthday) FROM FamilyMembers);
```

СТОЛБЦОВЫЕ ПОДЗАПРОСЫ С ANY,IN,ALL

ANY — ключевое слово, которое должно следовать за операцией сравнения (>, <, <>, = и т.д.), возвращающее TRUE, если хотя бы одно из значений столбца подзапроса удовлетворяет обозначенному условию.

```
SELECT поля_таблицы_1
FROM таблица_1
WHERE поле_таблицы_1 <= ANY (SELECT поле_таблицы_2 FROM
таблица_2);
```

ALL — ключевое слово, которое должно следовать за операцией сравнения, возвращающее TRUE, если все значения столбца подзапроса удовлетворяет обозначенному условию.

```
SELECT поля_таблицы_1
FROM таблица_1
WHERE поле_таблицы_1 > ALL (SELECT поле_таблицы_2 FROM
таблица_2);
```

IT MENTOR

IN — ключевое слово, являющееся псевдонимом ключевому слову ANY с оператором сравнения = (эквивалентность), либо <> ALL для NOT IN. Например, следующие запросы равнозначны.

```
WHERE поле_таблицы_1 IN (SELECT поле_таблицы_2 FROM
таблица_2);
```

СТРОКОВЫЕ ПОДЗАПРОСЫ

Строковый подзапрос — это подзапрос, возвращающий единственную строку с более чем одной колонкой.

```
SELECT поля_таблицы_1
FROM таблица_1
WHERE (первое_поле_таблицы_1, второе_поле_таблицы_1) =
      (SELECT первое_поле_таблицы_2, второе_поле_
таблицы_2
       FROM таблица_2
       WHERE id = 10);
```

СВЯЗАННЫЕ ПОДЗАПРОСЫ.

Связанным подзапросом является подзапрос, который содержит ссылку на таблицу, которая была объявлена во внешнем запросе.

```
SELECT поля_таблицы_1
FROM таблица_1
WHERE поле_таблицы_1 IN
      (SELECT поле_таблицы_2
       FROM таблица_2
       WHERE таблица_2.поле_таблицы_2 = таблица_1.поле_
таблицы_1);
```

ПОДЗАПРОСЫ КАК ПРОИЗВОДНЫЕ ТАБЛИЦЫ.

Производная таблица — выражение, которое генерирует временную таблицу в предложении FROM, которая работает так же, как и обычные таблицы, которые вы указываете через запятую.

```
SELECT поля_таблицы_1
FROM (подзапрос) [AS] псевдоним_производной_таблицы
```

Обратите внимание на то, что для производной таблицы обязательно должен указываться её псевдоним для того, чтобы имелась возможность обратиться к ней в других частях запроса.

IT MENTOR

ИНДЕКСЫ.

Индекс (index) — объект базы данных, создаваемый с целью повышения производительности выборки данных.

Индекс формируется из значений одного или нескольких полей и указателей на соответствующие записи набора данных. Таким образом достигается значительный прирост скорости выборки из этих данных.

Преимущества использования индексов:

- Ускорение поиска и сортировки по определенному полю или набору полей.
- Обеспечение уникальности данных.

Недостатки при использовании индексов:

- Требование дополнительного места на диске и в оперативной памяти. И чем больше и длиннее ключ, тем больше размер индекса.
- Замедление операций вставки, обновления и удаления записей, поскольку при этом приходится обновлять сами индексы.

Индексы предпочтительней:

- для поля-счетчика, чтобы в том числе избежать и повторения значений в этом поле;
- поля, по которому проводится сортировка данных;
- полей, по которым часто проводится соединение наборов данных. Поскольку в этом случае данные располагаются в порядке возрастания индекса и соединение происходит значительно быстрее.поля, которое объявлено первичным ключом (primary key);
- поля, в котором данные выбираются из некоторого диапазона. В этом случае, как только будет найдена первая запись с нужным значением, все последующие значения будут расположены рядом.

Использования индексов нецелесообразно:

- для полей, которые редко используются в запросах;
- полей, которые содержат всего два или три значения, например: мужской, женский пол или значения «да», «нет».

Как правило, индексы создаются на один столбец.

```
CREATE INDEX index_name (Ex: table_name_column_name_
idx)
    ON table_name (column_name);
```

Если нам необходимо удалить индекс, то это можно сделать с помощью команды:

```
DROP index_name;
```

IT MENTOR

РАБОТА С НЕСКОЛЬКИМИ ТАБЛИЦАМИ

Язык SQL позволяет нам получать данные и работать с ними не только из одной таблицы, но и из нескольких одновременно.

Первичный ключ (primary key) — это одно или несколько полей в таблице. Первичный ключ необходим для **уникальной идентификации** любой строки.

Первичный ключ накладывает некоторые ограничения на колонку:

- Все записи, относящиеся к первичному ключу, **должны быть уникальны**. Это означает, что если первичный ключ состоит из одного поля, то все записи в нем должны быть уникальными. А если первичный ключ состоит из нескольких полей, то комбинация этих записей должна быть уникальна, но в отдельных полях допускаются повторения.
- Записи в полях, относящихся к первичному ключу, **не могут быть пустыми**. Для этого при создании первичного ключа используется ограничение NOT NULL.
- У каждой таблицы может быть только **один primary key**.

Потенциальный ключ — простой или составной ключ, который уникально идентифицирует каждую запись набора данных.

При этом потенциальный ключ должен обладать критерием **неизбыточности**, то есть при удалении любого из полей набор полей перестает уникально идентифицировать запись. Из множества всех потенциальных ключей набора данных выбирают первичный ключ, все остальные ключи называют **альтернативными**.

Внешний ключ (foreign key) — это один или несколько столбцов, значения которых зависят от первичного ключа другой таблицы. Это ограничение используется для объединения двух таблиц.

При создании зависимой (дочерней) таблицы (таблицы, которая содержит внешние ключи) необходимо учитывать, что:

- каждый внешний ключ должен иметь такой же тип данных, как связанное поле главной таблицы;
- необходимо указать главную для нее таблицу и столбец, по которому осуществляется связь:

```
CREATE TABLE book
(
    id BIGSERIAL NOT NULL PRIMARY KEY,
    связанное_поле_зависимой_таблицы INT NOT NULL,
    FOREIGN KEY (связанное_поле_зависимой_таблицы)
REFERENCES главная_таблица (связанное_поле_главной_
таблицы)
);
```

IT MENTOR

СВЯЗЬ ONE TO MANY

Связь **One to Many** существует тогда, когда одной записи главной таблицы соответствуют несколько записей связанной таблицы, а каждой записи связанной таблицы соответствует только одна запись главной таблицы.

Например, один автор из главной таблицы может иметь несколько книг из связанной, при этом каждая книга могла быть написана только одним автором.

СВЯЗЬ MANY TO MANY

Связь **Many to Many** существует тогда, когда одной записи из первой таблицы соответствуют несколько записей во второй. И наоборот, каждой записи второй таблицы соответствуют несколько записей в первой.

Например, к созданию одной книги могут приложить руку сразу два автора. В таком случае не только у одного автора может быть несколько книг, но и у одной книги может быть сразу несколько авторов.

ОПЕРАТОР JOIN

JOIN — оператор языка SQL, который является реализацией операции соединения для нескольких таблиц. Предназначен для обеспечения выборки данных из двух таблиц и включения этих данных в один результирующий набор.

```
SELECT поля_таблиц
FROM таблица_1 [INNER] | [[LEFT | RIGHT | FULL][OUTER]]
JOIN таблица_2
ON условие_соединения
    [[INNER] | [[LEFT | RIGHT | FULL][OUTER]] JOIN
таблица_n
    ON условие_соединения]
```

Операции соединения имеют следующие особенности:

- в схему таблицы-результата входят столбцы обеих исходных таблиц (таблиц-операндов), то есть схема результата является «сцеплением» схем операндов;
- каждая строка таблицы-результата является «сцеплением» строки из одной таблицы-операнда со строкой второй таблицы-операнда;
- при необходимости соединения не двух, а нескольких таблиц операция соединения применяется несколько раз (последовательно).

INNER JOIN

Внутреннее соединение — соединение двух таблиц, при котором каждая запись из первой таблицы соединяется с каждой записью второй таблицы, создавая тем самым все возможные комбинации записей обеих таблиц (декартово произведение).

OUTER JOIN

Внешнее соединение может быть трёх типов: левое (LEFT), правое (RIGHT) и полное (FULL). По умолчанию оно является полным.

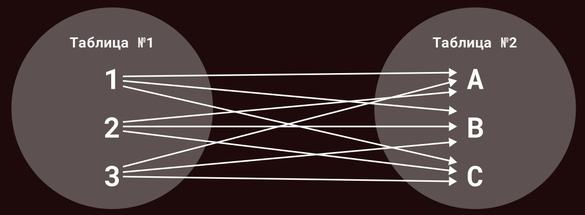
IT MENTOR

Главным отличием внешнего соединения от внутреннего является то, что оно обязательно возвращает все строки одной (LEFT, RIGHT) или двух таблиц (FULL).

- **LEFT (OUTER) JOIN** — производит выбор всех записей первой таблицы и соответствующих им записей второй таблицы. Если записи во второй таблице не найдены, то вместо них подставляется пустой результат (NULL). Порядок таблиц для оператора важен, поскольку оператор не является симметричным.
- **RIGHT (OUTER) JOIN** — по сути, является тем же самым LEFT JOIN, только с операндами, расставленными в обратном порядке. Порядок таблиц для оператора важен, поскольку оператор не является симметричным.
- **FULL (OUTER) JOIN** — результатом объединения таблиц являются все записи, которые присутствуют в таблицах. Порядок таблиц для оператора не важен, поскольку оператор является симметричным.

CROSS JOIN

CROSS JOIN реализует перекрестное соединение таблиц, или декартово произведение (все возможные пары из данных таблиц). При выборе каждая строка одной таблицы объединяется с каждой строкой второй таблицы, давая тем самым все возможные сочетания строк двух таблиц. Порядок таблиц для оператора не важен, поскольку оператор является симметричным.



БАЗОВЫЕ ЗАПРОСЫ ДЛЯ РАЗНЫХ ВАРИАНТОВ ОБЪЕДИНЕНИЯ ТАБЛИЦ.

Схема	Запрос с JOIN

	Получение всех данных из левой таблицы, соединённых с соответствующими данными из правой:
	`SELECT поля_таблиц`
	`FROM левая_таблица LEFT JOIN правая_таблица`
	`  ON правая_таблица.ключ = левая_таблица.ключ`

	Получение всех данных из правой таблицы, соединённых с соответствующими данными из левой:
	`SELECT поля_таблиц`
	`FROM левая_таблица RIGHT JOIN правая_таблица`
	`  ON правая_таблица.ключ = левая_таблица.ключ`

	Получение данных, относящихся только к левой таблице:
	`SELECT поля_таблиц`
	`FROM левая_таблица LEFT JOIN правая_таблица`
	`  ON правая_таблица.ключ = левая_таблица.ключ`
	`WHERE правая_таблица.ключ IS NULL`

	Получение данных, относящихся только к правой таблице:
	`SELECT поля_таблиц`
	`FROM левая_таблица RIGHT JOIN правая_таблица`
	`  ON правая_таблица.ключ = левая_таблица.ключ`
	`WHERE левая_таблица.ключ IS NULL`

	Получение данных, относящихся как к левой, так и к правой таблице:
	`SELECT поля_таблиц`
	`FROM левая_таблица INNER JOIN правая_таблица`
	`  ON правая_таблица.ключ = левая_таблица.ключ`

	Получение всех данных, относящихся к левой и правой таблицам, а также их внутреннему соединению:
	`SELECT поля_таблиц`
	`FROM левая_таблица`
	`  FULL OUTER JOIN правая_таблица`
	`  ON правая_таблица.ключ = левая_таблица.ключ`

	Получение данных, не относящихся к левой и правой таблицам одновременно (обратное INNER JOIN):
	`SELECT поля_таблиц`
	`FROM левая_таблица`
	`  FULL OUTER JOIN правая_таблица`
	`  ON правая_таблица.ключ = левая_таблица.ключ`
	`WHERE левая_таблица.ключ IS NULL`
	`  OR правая_таблица.ключ IS NULL`

IT MENTOR

НОРМАЛИЗАЦИЯ

Нормализация — это процесс преобразования таблицы базы данных к виду, отвечающему нормальным формам. Другими словами, это пошаговый обратимый процесс замены исходной таблицы другой таблицей, в которой наборы данных имеют более простую и логичную структуру.

Нормализация:

- предназначена для приведения структуры БД к виду, обеспечивающему минимальную логическую избыточность;
- не имеет целью уменьшение/увеличение производительности работы или уменьшение/увеличение физического объема БД;
- конечная цель — уменьшение потенциальной противоречивости хранимой в БД информации.

КАКИЕ СУЩЕСТВУЮТ НОРМАЛЬНЫЕ ФОРМЫ?

1. **Первая нормальная форма (1NF)** — таблица находится в 1NF, если значения всех ее атрибутов атомарны (неделимы).
2. **Вторая нормальная форма (2NF)** — таблица находится в 2NF, если она находится в 1NF и при этом все неключевые атрибуты зависят только от ключа целиком, а не от какой-то его части.
3. **Третья нормальная форма (3NF)** — таблица находится в 3NF, если она находится в 2NF и все неключевые атрибуты не зависят друг от друга.
4. **Четвертая нормальная форма (4NF)** — таблица находится в 4NF, если она находится в 3NF и если в ней не содержатся независимые группы атрибутов, между которыми существует отношение «многие ко многим».
5. **Пятая нормальная форма (5NF)** — таблица находится в 5NF, когда каждая нетривиальная (не единичная) зависимость соединения в ней определяется потенциальным ключом (ключами) этого отношения.
6. **Шестая нормальная форма (6NF)** — таблица находится в 6NF, когда она удовлетворяет всем нетривиальным зависимостям соединения, т. е. когда она неприводима, то есть не может быть подвергнута дальнейшей декомпозиции без потерь. Каждая переменная таблицы, которая находится в 6NF, также находится и в 5NF. Введена как обобщение пятой нормальной формы для хронологической базы данных.
7. **Нормальная форма Бойса - Кодда** — усиленная 3-я нормальная форма (BCNF) — таблица находится в BCNF, когда каждая ее нетривиальная и неприводимая слева функциональная зависимость имеет в качестве своего детерминанта некоторый потенциальный ключ.
8. **Доменно-ключевая нормальная форма (DKNF)** — таблица находится в DKNF, когда каждое наложенное на нее ограничение является логическим следствием ограничений доменов и ограничений ключей, наложенных на данную таблицу.

ДЕНОРМАЛИЗАЦИЯ

Денормализация базы данных — это процесс осознанного приведения базы данных к виду, в котором она не будет соответствовать правилам нормализации.

Обычно это необходимо для повышения производительности и скорости извлечения данных за счет увеличения избыточности данных.

IT MENTOR

ТРАНЗАКЦИИ В БД

ТРАНЗАКЦИИ И ПРИНЦИПЫ ACID

Транзакция — это воздействие на базу данных, переводящее ее из одного целостного состояния в другое и выражаемое в изменении данных, хранящихся в базе данных.

Транзакции обладают четырьмя основными свойствами ACID:

A — atomicity — Атомарность — гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все ее подоперации, либо не будет выполнено ни одной.

C — consistency — Согласованность — транзакция, достигающая своего нормального завершения и тем самым фиксирующая свои результаты, сохраняет согласованность базы данных.

I — isolation — Изолированность — Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на ее результат.

D — durability — Надежность — независимо от проблем на нижних уровнях (например, обесточивание системы или сбои в оборудовании) изменения, сделанные успешно завершенной транзакцией, должны остаться сохраненными после возвращения системы в работу.

УРОВНИ ИЗОЛИРОВАННОСТИ ТРАНЗАКЦИЙ

ПРОБЛЕМЫ ПРИ ПАРАЛЛЕЛЬНОМ ВЫПОЛНЕНИИ ТРАНЗАКЦИЙ

Одним из основных преимуществ баз данных является то, что базы данных дают возможность работать с ними одновременно из разных точек приложения.

Параллельный доступ — это одновременная работа с базой данных из разных точек.

При таком формате работы могут возникнуть некоторые проблемы:

- **Потерянное обновление** (lost update) — при одновременном изменении одного блока данных разными транзакциями одно из изменений теряется.
- **«Грязное» чтение** (dirty read) — чтение данных, добавленных или измененных транзакцией, которая впоследствии не подтвердится (откатится).
- **Неповторяющееся чтение** (non-repeatable read) — при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются измененными.
- **Фантомное чтение (phantom reads)** — одна транзакция в ходе своего выполнения несколько раз выбирает множество записей по одним и тем же критериям. Другая транзакция в интервалах между этими выборками добавляет или удаляет записи или изменяет столбцы некоторых записей, используемых в критериях выборки первой транзакции, и успешно заканчивается. В результате получится, что одни и те же выборки в первой транзакции дают разные множества записей.

Уровни изолированности транзакций — это мера степени успешной изоляции транзакций. Уровни изолированности определяются тем, какие из проблем параллельного доступа они решают.

1. **Чтение неподтвержденных данных** (read uncommitted, dirty read) — чтение незафиксированных изменений как своей транзакции, так и параллельных транзакций.
   - × Нет гарантии, что данные, измененные другими транзакциями, не будут в любой момент изменены в результате их отката, поэтому такое чтение является потенциальным источником ошибок.
   - × Невозможны потерянные обновления, возможны неповторяющееся чтение и фантомы.
2. **Чтение подтвержденных данных** (read committed) — чтение всех изменений своей транзакции и зафиксированных изменений параллельных транзакций.
   - × Потерянное обновление и грязное чтение не допускается, возможны неповторяющееся чтение и фантомы.
3. **Повторяемость чтения** (repeatable read, snapshot) — чтение всех изменений своей транзакции, любые изменения, внесенные параллельными транзакциями после начала своей, недоступны.
   - × Потерянное обновление, грязное и неповторяющееся чтение невозможны, возможны фантомы.
4. **Упорядочиваемость** (serializable) — результат параллельного выполнения сериализуемой транзакции с другими транзакциями должен быть логически эквивалентен результату их какого-либо последовательного выполнения.
   - × Проблемы синхронизации не возникают.

	Потерянные обновления	Грязное чтение	Неповторяемое чтение	Фантомы
Чтение неподтвержденных данных (read uncommitted, dirty read)	Невозможно	Возможно	Возможно	Возможно
Чтение подтвержденных данных (read committed)	Невозможно	Невозможно	Возможно	Возможно
Повторяемость чтения (repeatable read, snapshot)	Невозможно	Невозможно	Невозможно	Возможно
Упорядочиваемость (serializable)	Невозможно	Невозможно	Невозможно	Невозможно

JDBC

HIBERNATE

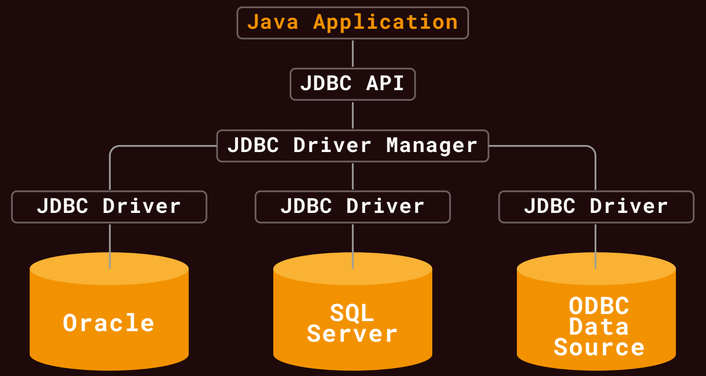
IT MENTOR

IT MENTOR

ВВЕДЕНИЕ В JDBC

JDBC (Java DataBase Connectivity — соединение с базами данных на Java) представляет собой стандартный API(описание способов взаимодействия одной компьютерной программы с другими), который предназначен для взаимодействия Java-приложения с различными системами управления базами данных (СУБД).

JDBC позволяет устанавливать соединение с базой данных согласно специально описанному URL с помощью драйверов, которые могут загружаться во время работы программы динамически.



MAVEN

maven - фреймворк, который отвечает за автоматизацию сборки проектов, а также позволяет легко управлять зависимостями для проекта.

maven создает локальный репозиторий в памяти машины и загружает туда все необходимые библиотеки для быстрого доступа к ним.

pom.xml(Project Object Model) - файл, который представляет собой декларативное описание проекта, то есть содержит конкретные спецификации, на основе которых и происходит сборка проекта, такие как зависимости и плагины.

Зачастую в качестве источника необходимых элементов используется ресурс **https://mvnrepository.com/**

Maven использует принцип **Maven-архетипов.**

Архетип — это инструмент шаблонов, каждый из которых определён паттерном(устоявшийся способ решения типовой задачи) или моделью. Благодаря применению того или иного архетипа мы получаем стандартную структуру каталогов(пакетов).

Maven также определяет **жизненный цикл проекта**.

Жизненный цикл maven-проекта — это список поименованных фаз, определяющий порядок действий при его построении.

Жизненный цикл Maven содержит три независимых порядка выполнения:

- **clean** — жизненный цикл для очистки проекта. Содержит следующие фазы:

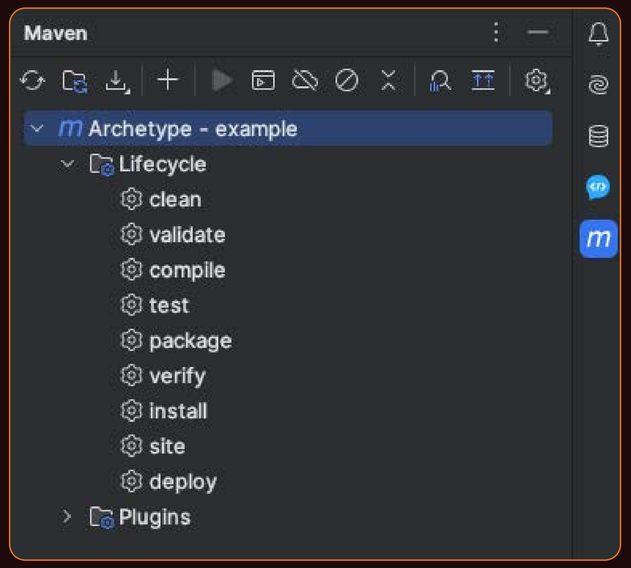
1. pre-clean
2. clean
3. post-clean

- **default** — основной жизненный цикл, содержащий следующие фазы:

4. validate — выполняется проверка, является ли структура проекта полной и правильной.

IT MENTOR

5. compile — компилируются исходные тексты.
6. test — собранный код тестируется заранее подготовленным набором тестов.
7. package — упаковка откомпилированных классов и прочих ресурсов. Например, в JAR-файл.
8. install — установка программного обеспечения в локальный Maven-репозиторий, чтобы сделать его доступным для других проектов текущего пользователя.
9. deploy — стабильная версия программного обеспечения распространяется на удаленный Maven-репозиторий, чтобы сделать его доступным для других пользователей.
   - `site` — жизненный цикл генерации проектной документации. Состоит из фаз:
10. pre-site
11. site
12. post-site
13. site-deploy



СОЕДИНЕНИЕ С БАЗОЙ И ТАБЛИЦЕЙ

Интерфейс `Connection` помогает установить соединение с необходимой базой данных Помимо установления соединения, он предоставляет управление транзакциями, поддержку сеансов базы данных и создание операторов SQL.

ИНТЕРФЕЙС CONNECTION

Основные методы интефейса Connection

- `void close()` — закрывает соединение с базой данных, связанное с объектом Connection, и освобождает ресурсы JDBC .
- `Statement createStatement()` — создает объект Statement, который используется для отправки запроса SQL в базу данных.
- `PreparedStatement prepareStatement (String sql)` — создает объект PreparedStatement, который используется для создания оператора SQL, указанного в строке sql. Обычно он используется, когда необходимо выполнить параметризованный оператор SQL.

КЛАСС DRIVERMANAGER

`Класс DriverManager` является компонентом JDBC API и находится в пакете `java.sql`. Класс `DriverManager` действует как посредник между приложением и драйверами. Он отслеживает доступные драйверы и устанавливает соединение между базой данных и соответствующим драйвером.

```
Connection connection = DriverManager
                    .getConnection(url, user, password);
```

IT MENTOR

ИНТЕРФЕЙС PREPAREDSTATEMENT

Интерфейс PreparedStatement является наследником интерфейса Statement. Он используется для выполнения параметризованного запроса.

Интерфейс PreparedStatement позволяет улучшить производительность приложения, за счет того, что запрос к базе данных компилируется только один раз.

ИНТЕРФЕЙС RESULTSET

Объект **ResultSet** представляет собой набор данных, который он получает из базы данных. **ResultSet** поддерживает курсор, указывающий на строку таблицы. Первоначально курсор указывает на место перед первой строкой.

Полный пример кода выглядит так:

```
import java.sql.*;

public class Application {

    public static void main(String[] args) throws SQL-
Exception {


        // Создаем переменные с данными для подключения к базе
        final String user = “postgres”;
        final String password = “5833118”;
        final String url = “jdbc:postgresql://local-
host:5432/skypro”;


        // Создаем соединение с базой с помощью Connection
        // Формируем запрос к базе с помощью PreparedStatement
        try (final Connection connection = DriverManag-
er.getConnection(url, user, password);
```

IT MENTOR

```
            PreparedStatement statement = connection.
prepareStatement("SELECT * FROM book WHERE book_id =
(?)")) {

            // Подставляем значение вместо wildcard
            statement.setInt(1, 6);

            // Делаем запрос к базе и результат кладем в ResultSet
            final ResultSet resultSet = statement.exe-
cuteQuery();

            // Методом next проверяем есть ли следующий элемент в re-
sultSet
            // и одновременно переходим к нему, если таковой есть
            while (resultSet.next()) {

                // С помощью методов getInt и getString получаем
данные из resultSet
                String titleOfBook = "Title: " + re-
sultSet.getString("title");
                String authorOfBook = "Author_id: " +
resultSet.getString("author_id");
                int amountOfBook = resultSet.getInt(4);

                // Выводим данные в консоль
                System.out.println(titleOfBook);
                System.out.println(authorOfBook);
                System.out.println("Amount: " +
amountOfBook);

            }
        }
    }
}


/*
Результатом в консоли будет:

Title: Uncle Vanya
Author_id: 4
Amount: 10
*/
```

IT MENTOR

ШАБЛОН DAO

DAO(Data Access Object) — абстрактный интерфейс к какому-либо типу базы данных или механизму хранения.

DAO описывает модуль взаимодействия с базой данных как прослойку между приложением и БД.

Для реализации DAO используется интерфейс с описанием общих методов, которые будут использоваться при взаимодействии с базой данных и класс, в котором реализованы все методы.

Реализация DAO на уровне класса подразумевает использование одного единственного коннекта для вызова всех методов унаследованного DAO класса. То есть нам требуется только единоразово в классе объявить поле класса Connection и добавить его в конструктор объекта класса реализации.

РЕАЛИЗАЦИЯ CRUD ОПЕРАЦИЯ ЧЕРЕЗ ШАБЛОН DAO

Создан интерфейс DAO и объявлены в нем методы.

```
public interface BookDAO {


    // Добавление объекта
    void create(Book book);
    // Получение объекта по id
    Book readById(int id);
    // Получение всех объектов
    List<Book> readAll();
    // Изменение объекта по id
    void updateAmountById(int id, int amount);
    // Удаление объекта по id
    void deleteById(int id);

}
```

IT MENTOR

МЕТОД ДОБАВЛЕНИЯ:

```
@Override
public void create(Book book) {

    // Формируем запрос к базе с помощью PreparedStatement
    try(PreparedStatement statement = connection.pre-
pareStatement(
            “INSERT INTO book (title, author_id,
amount) VALUES ((?), (?), (?))”)) {

        // Подставляем значение вместо wildcard
        // первым параметром указываем порядковый номер wildcard
        // вторым параметром передаем значение
        statement.setString(1, book.getTitle());
        statement.setInt(2, book.getAuthor().getId());
        statement.setInt(3, book.getAmount());

        // С помощью метода executeQuery отправляем запрос к базе
        statement.executeQuery();

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

МЕТОД ПОЛУЧЕНИЯ ОБЪЕКТА ПО ID:

```
@Override
public Book readById(int id) {

    Book book = new Book();
    // Формируем запрос к базе с помощью PreparedStatement
    try (PreparedStatement statement = connection.pre-
pareStatement(
            “SELECT * FROM book INNER JOIN author ON
book.author_id=author.author_id AND book_id=(?)”)) {

        // Подставляем значение вместо wildcard
        statement.setInt(1, id);

        // Делаем запрос к базе и результат кладем в ResultSet
        ResultSet resultSet = statement.executeQuery();

        // Методом next проверяем есть ли следующий элемент в resultSet
        // и одновременно переходим к нему, если таковой есть
        while(resultSet.next()) {
```

```
            // С помощью методов getInt и getString получаем данные из
resultSet
            // и присваиваем их полим объекта
            book.setId(Integer.parseInt(resultSet.get-
String("book_id")));
            book.setTitle(resultSet.getString("ti-
tle"));
            book.setAuthor(new Author(resultSet.
getInt("author_id"),
                    resultSet.getString("name_au-
thor")));
            book.setAmount(Integer.parseInt(resultSet.
getString("amount")));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return book;
}
```

ПОЛУЧЕНИЕ ВСЕХ ОБЪЕКТОВ ИЗ БАЗЫ:

```
@Override
public List<Book> readAll() {


    // Создаем список, в который будем укладывать объекты
    List<Book> bookList = new ArrayList<>();

    try(PreparedStatement statement = connection.pre-
pareStatement(
            "SELECT * FROM book INNER JOIN author ON
book.author_id=author.author_id")) {

        ResultSet resultSet = statement.executeQuery();
        while (resultSet.next()) {

            int id = Integer.parseInt(resultSet.get-
String("book_id"));
            String title = resultSet.getString("ti-
tle");
```

IT MENTOR

```
            Author author = new Author(resultSet.
getInt("author_id"),
                    resultSet.getString("name_au-
thor"));
            int amount = Integer.parseInt(resultSet.
getString("amount"));

            // Создаем объекты на основе полученных данных
            // и укладываем их в итоговый список
            bookList.add(new Book(id, title, author,
amount));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return bookList;
}
```

МЕТОД ОБНОВЛЕНИЯ ДАННЫХ В БАЗЕ:

```
@Override
public void updateAmountById(int id, int amount) {

    try(PreparedStatement statement = connection.pre-
pareStatement(
            "UPDATE book SET amount=(?) WHERE book_
id=(?)")) {

        statement.setInt(1, amount);
        statement.setInt(2, id);

        statement.executeQuery();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

IT MENTOR

МЕТОД УДАЛЕНИЯ ДАННЫХ ИЗ БАЗЫ:

```
@Override
public void deleteById(int id) {

    try(PreparedStatement statement = connection.pre-
pareStatement(
            "DELETE FROM book WHERE book_id=(?)")) {

        statement.setInt(1, id);
        statement.executeQuery();

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Для того, чтобы применить эти методы, необходимо в классе Application создать объект класса реализации DAO, передав ему в конструкторе объект Connection.

```
public class Application {

    public static void main(String[] args) throws SQL-
Exception {

        final String user = "postgres";
        final String password = "5833118";
        final String url = "jdbc:postgresql://local-
host:5432/skypro";
        try (Connection connection = DriverManager.get-
Connection(url, user, password)) {

            // Создаем объект класса BookDAOImpl
            BookDAO bookDAO = new BookDAOImpl(connec-
tion);
            Author author = new Author(1, "L.N.Tol-
stoy");              Book book1 = new Book("Anna Kareni-
na", author, 6);

            // Вызываем метод добавления объекта
            bookDAO.create(book1);

            // Создаем список наполняя его объектами, которые получаем
            // путем вызова метода для получения всех элементов
таблицы
            List<Book> list = new ArrayList<>(bookDAO.
readAll());

           // Выведем список в консоль
            for (Book book : list) {
                System.out.println(book);
            }

        }
    }
}
```

IT MENTOR

HIBERNATE: ВВЕДЕНИЕ, ENTITY

ORM, JPA, HIBERNATE

Object Relational Mapping (ORM) — это концепция или процесс отображения данных из объектно-ориентированного языка в информацию для реляционных баз данных и наоборот.

Java Persistence API (JPA) — это спецификация или стандарт обеспечивающий объектно-реляционное отображение простых PO-JO-объектов (Plain Old Java Object) и предоставляющий универсальный набор классов, интерфейсов и методов для сохранения, получения и управления такими объектами.

Сам JPA не умеет ни сохранять, ни управлять объектами, JPA только определяет правила игры: как должен действовать каждый провайдер (Hibernate, EclipseLink, OJB, Torque и т.д.), реализующий стандарт JPA. Для этого JPA определяет интерфейсы, которые должны быть реализованы провайдерами.

Также JPA определяет правила, как должны описываться метаданные отображения и как должны работать провайдеры. Каждый провайдер обязан реализовывать всё из JPA, определяя стандартное получение, сохранение и управление объектами. Помимо этого, провайдеры могут добавлять свои личные классы и интерфейсы, расширяя функционал JPA.

JAVA-код, написанный только с использованием интерфейсов и классов JPA, позволяет разработчику гибко менять одного провайдера на другого. Например, если приложение использует Hibernate как провайдера, то ничего не меняя в коде можно поменять провайдера на любой другой. Но, если мы в коде использовали интерфейсы, классы или аннотации, например, из Hibernate, то поменяв провайдера на EclipseLink, эти интерфейсы, классы или аннотации уже работать не будут.

Hibernate — это провайдер, реализующий спецификацию JPA. Hibernate полностью реализует JPA, плюс добавляет функционал в виде своих классов и интерфейсов, расширяя свои возможности по работе с сущностями и БД.

ENTITY MANAGER

EntityManager — это интерфейс JPA, используемый для взаимодействия с персистентным контекстом. Описывает API для всех основных операций над Entity, а также для получения данных и других сущностей JPA

Персистентный контекст — это набор экземпляров сущностей, загруженных из БД или только что созданных.

Персистентный контекст является своего рода кэшем данных в рамках транзакции. Внутри контекста персистентности происходит управление экземплярами сущностей и их жизненным циклом.

EntityManager автоматически сохраняет в БД все изменения, сделанные в его персистентном контексте в момент коммита транзакции, либо при явном вызове метода.

flush() Один или несколько EntityManager образуют или могут образовать persistence context.

IT MENTOR

ОСНОВНЫЕ ФУНКЦИИ ENTITYMANAGER

Операции над Entity:

1. **persist** (добавление Entity под управление JPA),

merge (изменение),

remove (удаление),

refresh (обновление данных),

detach (удаление из-под управления контекста персистентности).

2. Получение данных: **find** (поиск и получение Entity).

3. Получение других сущностей JPA: **getTransaction**, **getEntityManagerFactory**.

EntityManagerFactory — это фабрика, которая содержит connection к базе, всю конфигурацию, кэш объекты и т.д. Суть фабрики — создавать EntityManager.

Интерфейс Persistence — по сути это класс. В нем практически нет методов, кроме пары статических. Один из них **CreateEntityManagerFactory**, то есть он создает EntityManagerFactory.

ENTITY

Entity (Сущность) — это объект персистентной области. Как правило, сущность представляет таблицу в реляционной базе данных, и каждый экземпляр сущности соответствует строке в этой таблице.

Основным программным представлением entity является **класс сущности**. Класс сущности может использовать другие классы, которые служат вспомогательными классами или используются для представления состояния сущности.

Персистентное состояние сущности представлено персистентными **полями** или персистентными **свойствами**.

Персистентное поле — поле сущности, которое отражается в БД в виде столбца таблицы.

Персистентное свойство — это методы, которые аннотированы вместо полей для доступа провайдера к полям.

Требования к Entity:

- Entity класс должен быть помечен аннотацией **@Entity** или описан в XML-файле конфигурации JPA.
- Entity класс должен содержать **публичный конструктор без параметров** (может иметь дополнительно конструкторы с аргументами).
- Entity класс должен быть классом **верхнего уровня**.
- Перечисление или интерфейс не могут быть определены как сущность (Entity).
- Entity класс **не может** быть финальным. Entity класс не может содержать финальные поля или методы, если они участвуют в маппинге (являются элементами таблицы БД).
- Как обычный так и абстрактный класс может быть Entity. Entity могут наследоваться как от не Entity классов, так и от Entity классов. А не Entity классы могут наследоваться от Entity классов.
- Поля Entity классов должны быть объявлены как **private**, **protected** или **default**.
- Entity класс должен содержать **первичный ключ**, то есть атрибут или группу атрибутов, которые уникально определяют запись этого Entity класса в базе данных.

IT MENTOR

ЖИЗНЕННЫЙ ЦИКЛ ENTITY

Согласно JPA объект сущности может иметь один из четырех статусов жизненного цикла:

1. **new** — объект создан, не имеет primary key, не является частью контекста персистентности (не управляется JPA);
2. **managed** — объект создан, имеет primary key, является частью контекста персистентности (управляется JPA);
3. **detached** — объект создан, имеет primary key, не является (или больше не является) частью контекста персистентности (не управляется JPA);
4. **removed** — объект создан, является частью контекста персистентности (управляется JPA), будет удален при commit-е транзакции.

АННОТАЦИИ ID, TABLE, COLUMN

АННОТАЦИЯ @ID

@Id определяет простой (не составной) первичный ключ, состоящий из одного поля.

В соответствии с JPA, допустимые типы атрибутов для первичного ключа:

1. Примитивные типы и их обертки;
2. Строки;
3. BigDecimal и BigInteger;
4. java.util.Date и java.sql.Date.

СТРАТЕГИИ ГЕНЕРАЦИИ ID

Если мы хотим, чтобы назначение первичного ключа генерировалось для нас автоматически, мы можем добавить первичному ключу, отмеченному аннотацией @Id аннотацию **@GeneratedValue**.

Согласно спецификации JPA возможно 4 различных варианта:

AUTO — (тип генерации по умолчанию) указывает, что Hibernate должен выбрать подходящую стратегию для конкретной базы данных, учитывая её диалект, так как у разных БД разные способы по умолчанию. То, как провайдер должен реализовывать тип генерации AUTO, оставлено на усмотрение провайдера. Поведение по умолчанию — исходить из типа поля идентификатора.

IDENTITY — указывает, что для генерации значения первичного ключа будет использоваться столбец IDENTITY, имеющийся в базе данных. Значения в столбце автоматически увеличиваются, что позволяет базе данных генерировать новое значение при каждой операции вставки.

SEQUENCE — указывает, что для получения значений первичного ключа Hibernate должен использовать имеющиеся в базе данных механизмы генерации последовательных значений (Sequence). Но если наша БД не поддерживает тип SEQUENCE, то Hibernate автоматически переключится на тип TABLE.

TABLE — в настоящее время используется редко. Hibernate должен получать первичные ключи для сущностей из специально создаваемой для этих целей таблицы, способной содержать несколько именованных значений сегментов значений для любого количества сущностей.

IT MENTOR

АННТАЦИЯ @COLUMN

@Column сопоставляет поле класса столбцу таблицы, а ее атрибуты определяют поведение в этом столбце, используется для генерации схемы таблицы базы данных.

@Column позволяет нам указать имя столбца в таблице и ряд других свойств, например:

- insertable/updatable — можно ли добавлять изменять данные в колонке, по умолчанию true
- length — длина, для строковых типов данных, по умолчанию 255
- nullable — может ли соответствующий столбец быть null

АННОТАЦИЯ @TABLE

С помощью этой аннотации мы говорим Hibernate, с какой именно таблицей необходимо связать (map) данный класс. Аннотация @Table имеет различные аттрибуты, с помощью которых мы можем указать имя таблицы, каталог, БД и уникальность столбцов в таблице БД.

РАБОТА С БД С ПОМОЩЬЮ HIBERNATE

Для того, чтобы использовать при работе с БД Hibernate необходимо сначала добавить соответствующую зависимость в pom.xml:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.14.Final</version>
</dependency>
```

Наша задача скорректировать класс Book для работы с Hibernate по критериям Entity.

```
import javax.persistence.*;
import java.util.Objects;

@Entity
@Table(name = “book”)
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = “book_id”)
    private int id;
    @Column(name = “title”)
    private String title;
    @Column(name = “author_id”)
    private int author;
    @Column(name = “amount”)
    private int amount;
    public Book() {
    }
}
```

IT MENTOR

Для работы с БД необходимо создать конфигурационный класс, благодаря которому мы сможем получать фабрику сессий – SessionFactory:

```
package config;

import model.Employee;
import org.hibernate.HibernateException;
import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceReg-
istryBuilder;
import org.hibernate.cfg.Configuration;

public class HibernateSessionFactoryUtil {
    private static SessionFactory sessionFactory;

    public HibernateSessionFactoryUtil() {
    }

    public static SessionFactory getSessionFactory() {
        if (sessionFactory == null) {
            try {
               Configuration configuration = new
Configuration().configure();
                configuration.addAnnotatedClass(Employ-
ee.class);
                StandardServiceRegistryBuilder builder
= new StandardServiceRegistryBuilder()
                        .applySettings(configuration.
getProperties());
                sessionFactory = configuration
                        .buildSessionFactory(builder.
build());
            } catch (HibernateException e) {
                e.printStackTrace();
            }
        }
        return sessionFactory;
    }
}
```

Данный класс отвечает за создание сессий для транзакций БД. Это стандартный набор настроек, который обеспечивает связь с базой и возможность обращаться к ней в рамках отдельных сессий.

`Configuration configuration = new Configuration().configure();` — в данной строке происходит считывание конфигураций базы данных для подключения к ней.

IT MENTOR

Чтобы эта строчка работала, необходимо в папке resources создать файл **hibernate.cfg.xml**

А также конфигурационный файл с названием **hibernate.cfg.xml**, который создается в папке **resources**. В данном файле прописываются все необходимые настройки для подключения к БД.

```
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configura-
tion-3.0.dtd">

<hibernate-configuration>
    <session-factory>

        <property name="connection.url">jdbc:post-
gresql://localhost/skypro</property>
<!-- BD Name -->
        <property name="connection.driver_class">org.
postgresql.Driver</property>
<!-- DB Driver -->
        <property name="connection.username">postgres</
property>
<!-- DB User -->
        <property name="connection.password">Hf,jn-
f,kz1</property>
<!-- DB Password -->

        <property name="hibernate.dialect"> org.hiber-
nate.dialect.PostgreSQL82Dialect</property>
<!-- DB Dialect -->
        <property name="hbm2ddl.auto">update</property>
<!-- create / create-drop / update -->

        <property name="show_sql">true</property>
<!-- Show SQL in console -->
        <property name="format_sql">true</property>
<!-- Show SQL formatted -->

    </session-factory>
</hibernate-configuration>
```

IT MENTOR

Теперь у нас обеспечено соединение с БД, осталось только заново реализовать методы в классе **BookDAOImpl**. **Hibernate** позволяет не прописывать запросы к базе в реализующем классе **DAO**, а только вызывать соответствующие методы в рамках транзакции или без нее.

Например, метод **добавление данных** в таблицу будет выглядеть так:

```
@Override
public void create(Book book) {
    // В ресурсах блока try создаем объект сессии с помощью нашего конфиг-файла
    // И открываем сессию
    try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession();) {
        // Создаем транзакцию и начинаем ее
        Transaction transaction = session.beginTransaction();
        // вызываем на объекте сессии метод save
        // данный метод внутри себя содержит необходимый запрос к базе
        // для создания новой строки
        session.save(book);
        // Выполняем коммит, то есть сохраняем изменения,
        // которые совершили в рамках транзакции
        transaction.commit();
    }
}
```

Метод **получения объекта по id** будет выглядеть так:

```
@Override
public Employee getById(int id) {
// В ресурсах блока try создаем объект сессии с помощью нашего конфиг-файла и открываем сессию
    try (Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()) {
        // и через метод get получаем объект В параметре метода get нужно указать объект какого класса нам нужен и его id
        return session.get(Book.class, id);
    }
}
```

Метод **получения всех элементов** в лист:

```
@Override
public List<Employee> readAll() {
    try (Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()) {
// метод createQuery, в параметр которого нам достаточно передать строку “From Book”, то есть указать таблицу из которой нам нужны объекты.
        return session.createQuery(“From Book”).list();
    }
}
```

IT MENTOR

Метод **обновления данных** в строке выглядит так:

```
@Override
public void updateAmount(Book book) {
    try (Session session = HibernateSessionFactoryUtil.
getSessionFactory().openSession()){
        Transaction transaction = session.beginTransac-
tion();
        // Для обновления данных нужно передать в конструктор
        // объект с актуальными данными
        session.update(book);
        transaction.commit();
    }
}
```

Метод **удаления объекта** из таблицы выглядит так:

```
@Override
public void delete(Book book) {
    try(Session session = HibernateSessionFactoryUtil.
getSessionFactory().openSession()) {
        Transaction transaction = session.beginTransac-
tion();
        // Для удаления объекта из таблицы нужно передать его в метод
delete
        session.delete(book);
        transaction.commit();
    }
}
```

IT MENTOR

HIBERNATE: РАБОТА С НЕСКОЛЬКИМИ ТАБЛИЦАМИ

НАСТРОЙКА СВЯЗЕЙ МЕЖДУ ТАБЛИЦАМИ

ТИПЫ СВЯЗЕЙ МЕЖДУ СУЩНОСТЯМИ

Существуют четыре типа связей между сущностями (Entity):

1. **OneToOne** — когда один экземпляр Entity может быть связан не больше чем с одним экземпляром другого Entity.
2. **OneToMany** — когда один экземпляр Entity может быть связан с несколькими экземплярами других Entity.
3. **ManyToOne** — обратная связь для OneToMany. Несколько экземпляров Entity могут быть связаны с одним экземпляром другого Entity.
4. **ManyToMany** — экземпляры Entity могут быть связаны с несколькими экземплярами друг друга.

НАПРАВЛЕНИЕ В ОТНОШЕНИЯХ СУЩНОСТЕЙ

Направление отношений делятся на:

Двунаправленные отношения имеют сторону-владельца и владеемую сторону.

Однонаправленные отношения имеют только сторону-владельца.

Двунаправленные отношения

В двунаправленном отношении каждая сущность имеет поле, которое ссылается на другую сущность. Через это поле код первой сущности может получить доступ ко второй сущности, находящейся на другой стороне отношений.

Двунаправленные отношения должны следовать следующим **правилам**:

1. Владеемая сторона в двунаправленных отношениях должна ссылаться на владеющую сторону используя элемент **mapped-By** аннотаций **@OneToOne**, **@OneToMany** или **@ManyToMany**. Элемент **mappedBy** определяет поле в объекте, который является владельцем отношения. Если применить атрибут **mappedBy** на одной стороне связи, то **Hibernate** не станет создавать сводную таблицу:

```
@Entity
@Table(name=”cart”)
public class Cart {
    //...
    @OneToMany(mappedBy=”cart”)
    private Set<Items> items; // getters and setters
}
@Entity
@Table(name=”ITEMS”)
public class Items {
    //...
    @ManyToOne
    @JoinColumn(name=”cart_id”, nullable=false)
    private Cart cart;
    public Items() {}

    // getters and setters
}
```

2. Сторона **Many** в отношениях **ManyToOne** всегда является владельцем отношений и не может определять элемент **mappedBy** (такого параметра у аннотации **@ManyToOne** нет).
3. Для двунаправленных отношений **OneToOne**, сторона-владелец — это та сторона, чья таблица имеет столбец с внешним ключом на другую таблицу. Если не указан параметр **mappedBy**, то колонки с айдишниками появляются у каждой таблицы.
4. Для двунаправленных отношений **ManyToMany**, любая сторона может быть стороной-владельцем.

Однонаправленные отношения

В однонаправленных отношениях только одна сущность имеет поле, которое ссылается на вторую сущность. Вторая сущность (сторона) не имеет поля первой сущности и не знает об отношениях.

ЗАПРОСЫ И НАПРАВЛЕНИЕ ОТНОШЕНИЙ

Запросы языка **Java Persistence** часто перемещаются между отношениями. Направление отношений определяет, может ли запрос перемещаться от одной сущности к другой. Например, в двунаправленных отношениях запрос может перемещаться как от первой сущности ко второй, так и обратно. В однонаправленных отношениях запрос может перемещаться только в одну сторону — от владеющей сущности к владеемой.

ВЛАДЕЛЕЦ СВЯЗИ

По сути, у кого есть внешний ключ на другую сущность — тот и владелец связи. То есть, если в таблице одной сущности есть колонка, содержавшая внешние ключи от другой сущности, то первая сущность признается владельцем связи, вторая сущность — владеемой.

В однонаправленных отношениях сторона, которая имеет поле с типом другой сущности, является владельцем этой связи по умолчанию, например:

```
@Entity
public class LineItem {
    @Id
    private Long id
    @OneToOne
    private Product product;
}

@Entity
public class Product {
    @Id
    private Long id
    private String name;
    private Double price;
}
```

Получается, что в таблице LineItem есть колонка с внешним ключом на таблицу Product, а в таблице Product колонки LineItem нет.

IT MENTOR

Двунаправленные отношения имеют как сторону-владельца, так и владеемую сторону:

```
@Entity
public class CustomerOrder {
    @Id
    private Long id

    @OneToMany(mappedBy = “customerOrder”)
    private Set<LineItem> lineItems = new HashSet<>();
}

@Entity
public class LineItem {
    @Id
    private Long id

    @OneToOne
    private Product product;
    @ManyToOne
    private CustomerOrder customerOrder;
}
```

В данном примере владельцем связи является сторона **LineItem**, потому что в таблице LineItem есть колонка с внешними ключами на таблицу **CustomerOrder**. По правилам Java, сторона **@ManyToOne** всегда является владельцем связи, а элемент **mappedBy** определяет поле в объекте, который является владельцем отношения. Тут так и получается: сторона **@ManyToOne** — **LineItem** элемент **mappedBy** определяет поле “**customerOrder**” в объекте **LineItem**. **LineItem** в своей таблице владеет ключами от **CustomerOrder**, поэтому **LineItem** и является владельцем.

Владельца связи и владеемого легко спутать с родителем отношения и ребенком. В нашем случае **CustomerOrder** является владеемым и одновременно родителем в отношениях, а **LineItem** является владеющим и одновременно ребенком в отношениях.

Родительская сущность (таблица) — это сущность (таблица), на которую ссылается внешний ключ из дочерней сущности (таблицы).

Дочерняя сущность (таблица) — это сущность (таблица), в которой есть колонка с внешним ключом, ссылающимся на родительскую сущность (таблицу).

КАСКАДНЫЕ ОПЕРАЦИИ

КАСКАДНЫЕ ОПЕРАЦИИ И ОТНОШЕНИЯ

Каскадные операции — это операции, которые при выполнении на одной сущности распространяются и на зависимую сущность.

JPA позволяет распространять операции с сущностями на связанные сущности. Это означает, что при включенном каскадировании, если сущность A сохраняется или удаляется, тогда сущность B также будет сохраняться или удаляться без явных команд сохранения или удаления.

Каскадирования можно добиться, указав у любой из аннотаций **@OneToOne**, **@ManyToOne**, **@OneToMany**, **@ManyToMany** элемент cascade и присвоив ему одно или несколько значений из перечисления **CascadeType** (**ALL, DETACH, MERGE, PERSIST, REFRESH, REMOVE**).

IT MENTOR

УДАЛЕНИЕ СИРОТ В ОТНОШЕНИЯХ (ORPHAN REMOVAL)

Представим, что у нас есть класс Customer, у которого есть коллекция Order:

```
@Entity
public class Customer {
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval
= true)
    private List<Order> orders = new ArrayList<>();
    // other mappings, getters and setters
}
```

Пусть у нас есть один объект Customer — родитель, в коллекции которого есть 4 объекта **Order** — дети. Мы установили атрибут **orphanRemoval = true** над этой коллекцией. В нашей базе данных в таблице Customer будет одна строка, а в таблице **Order** будет четыре строки. Также в таблице Order будет колонка с внешними ключами на таблицу Customer. В каждой из четырех ячеек этой колонки будут ссылки на один и тот же первичный ключ объекта Customer. Например, мы удалим из коллекции **orders** один объект **Order** — любой из четырех, в результате чего у объекта Customer останется три объекта **Order**:

```
Customer customer = entityManager
                    .find(Customer.class, 1L);

Order order = customer
                    .getOrders().get(0);

customer.getOrders().remove(order); flushAndClear();
```

После запуска метода **flushAndClear()** — обновления объекта Customer отправятся в базу данных, и произойдет следующее:

1. **Hibernate** заметит, что у объекта Customer уже не 4, а 3 связанных дочерних объекта Order;
2. в связи с этим **Hibernate** найдёт в таблице **Order** строку с удаленным объектом из коллекции **Order**;
3. очистит в этой строке ячейку с внешним ключом на Customer;
4. после чего удалит саму эту строку, как осиротевшую (более не ссылающуюся на родителя).

Если не будет атрибута **orphanRemoval = true**, то пункт 4 не выполнится, и в таблице **Order** останется сущность **Order**, не связанная ни с одной сущностью Customer, то есть ее ячейка с внешним ключом будет пустой. Такая сущность будет считаться осиротевшей.

ВЛИЯНИЕ КАСКАДНЫХ ОПЕРАЦИЙ НА ОБЪЕКТЫ ENTITY.

1. **new** — объект создан, не имеет primary key, не является частью контекста персистентности (не управляется JPA);
2. **managed** — объект создан, имеет primary key, является частью контекста персистентности (управляется JPA);
3. **detached** — объект создан, имеет primary key, не является (или больше не является) частью контекста персистентности (не управляется JPA);
4. **removed** — объект создан, является частью контекста персистентности (управляется JPA), будет удален при commit-е транзакции.

IT MENTOR

CASCADETYPE.PERSIST

Распространяет операцию сохранения от родительского объекта к дочернему.

- **new - managed**, объект будет сохранен в базу при commit-е транзакции или в результате flush-операции.
- **managed** - операция игнорируется, однако связанные entity могут поменять статус на managed, если у них есть аннотации каскадных изменений.
- **removed - managed.**
- **detached - exception** сразу или на этапе commit-а транзакции (так как у detached уже есть первичный ключ).

CASCADETYPE.REMOVE

Распростаняет **операцию удаления от родительского к дочернему объекту**.

- **new** - операция игнорируется, однако связанные entity могут поменять статус на **removed**, если у них есть аннотации каскадных изменений и они имели статус **managed**.
- **managed - removed**, и запись в базе данных будет удалена при commit-е транзакции (также произойдут операции remove для всех каскадно зависимых объектов).
- **removed** - операция игнорируется.
- **detached - exception** сразу или на этапе commit-а транзакции.

CASCADETYPE.MERGE

Распространяет операцию слияния от родительского к дочернему объекту.

new - будет создана новая **managed** entity, в которую будут скопированы данные объекта.

managed - операция игнорируется, однако операция merge сработает на каскадно зависимых entity, если их статус не managed.

removed - exception сразу или на этапе commit-а транзакции.

detached - либо данные будут скопированы в существующую в БД managed entity с тем же первичным ключом, либо будет создана новая managed entity, в которую скопируются данные.

CASCADETYPE.REFRESH

Операции обновления **повторно считывают значение данного экземпляра из базы данных**. В некоторых случаях мы можем изменить экземпляр после сохранения в базе данных, но позже нам нужно будет отменить эти изменения.

В таком сценарии это может быть полезно. **Когда мы используем эту операцию с каскадным типом REFRESH , дочерняя сущность также перезагружается из базы данных всякий раз, когда обновляется родительская сущность**.

managed - будут восстановлены все изменения из базы данных данного entity, также произойдет **refresh** всех каскадно зависимых объектов.

new, removed, detached - exception.

CASCADETYPE.DETACH

Когда мы используем CascadeType.DETACH , дочерняя сущность также будет удалена из постоянного контекста.

managed, removed - detached.

new, detached - операция игнорируется.

IT MENTOR

FETCH СТРАТЕГИИ

FetchType в JPA говорит, когда мы хотим, чтоб связанная сущность или коллекция была загружена.

С помощью функции **fetch**() можно отправлять сетевые запросы на сервер — как получать, так и отправлять данные.

В **JPA** описаны два типа **fetch** стратегий:

1. **LAZY** — данные поля сущности будут загружены только во время первого обращения к этому полю.
2. **EAGER** — данные поля будут загружены немедленно вместе с сущностью.

FetchType.EAGER: Hibernate должен **сразу** загрузить соответствующее аннотированное поле или свойство. Это поведение работает по умолчанию для полей аннотированных @ManyToOne и @ OneToOne.

FetchType.LAZY: Hibernate может загружать данные **не сразу**, а при первом обращении к ним, но так как это необязательное требование, то Hibernate имеет право изменить это поведение и загружать их сразу. Это поведение работает по умолчанию для полей, аннотированных **@OneToMany и @ManyToMany**.

АННОТАЦИЯ @JOINCOLUMN

@JoinColumn используется для указания столбца FOREIGN KEY, используемого при установлении связей между сущностями.
Мы помним, что только сущность-владелец связи может иметь внешние ключи от другой сущности (владеемой). Однако, мы можем указать аннотацию **@JoinColumn** как во владеющей таблице, так и во владеемой, но столбец с внешними ключами все равно появится во владеющей таблице.

Особенности использования:

- **@OneToOne** — означает, что появится столбец addressId в таблице сущности-владельца связи Office. Этот столбец будет содержать внешний ключ, ссылающийся на первичный ключ владеемой сущности Address.

```
@Entity
public class Office {
    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = “addressId”)
    private Address address;
}
```

IT MENTOR

@OneToMany/@ManyToOne — в данном случае мы можем использовать параметр **mappedBy** для того, чтобы столбец с внешними ключами находился на владеющей стороне **ManyToOne** — то есть в таблице Email:

```
@Entity
public class Employee {
    @Id
    private Long id;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = “em-
ployee”)
    private List<Email> emails;
}
@Entity
public class Email {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = “employee_id”)
    private Employee employee;
}
```

@JOINCOLUMNS

@JoinColumns используется для группировки нескольких аннотаций @JoinColumn, которые используются при установлении связей между сущностями, у которых составной первичный ключ и требуется несколько колонок для указания внешнего ключа. В каждой аннотации @JoinColumn должны быть указаны элементы **name** и **referenced-ColumnName**:

```
@ManyToOne
@JoinColumns({
        @JoinColumn(name = “ADDR_ID”, referencedColumn-
Name = “ID”),
        @JoinColumn(name = “ADDR_ZIP”, referenced-
ColumnName = “ZIP”)
})

public Address getAddress() {
    return address;
}
```

IT MENTOR

HIBERNATE: КЭШИ, АННОТАЦИИ, РЕШЕНИЕ ПРОБЛЕМЫ N+1

КЭШИ

Кэширование является одним из способов оптимизации работы приложения, ключевой задачей которого является уменьшить количество прямых обращений к базе данных.

В JPA существует два вида кэшей (cache):

- **first-level cache** (кэш первого уровня) — кэширует данные одной транзакции;
- **second-level cache** (кэш второго уровня) — кэширует данные транзакций от одной фабрики сессий.

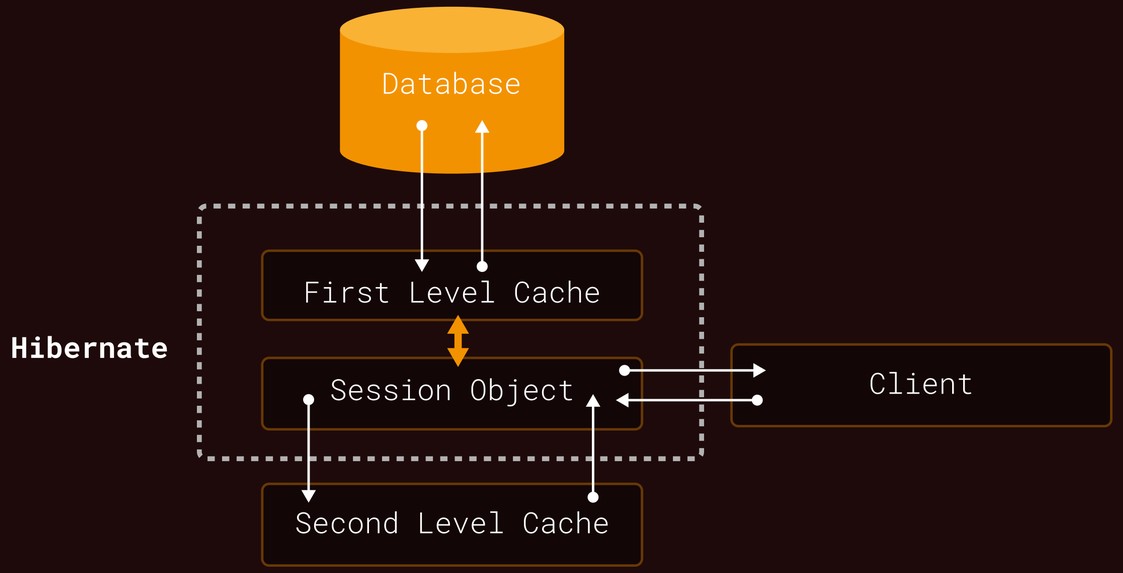
Провайдер JPA может, но не обязан реализовывать работу с кэшем второго уровня. Такой вид кэша позволяет сэкономить время доступа и улучшить производительность, однако оборотной стороной является возможность получить устаревшие данные.

КЭШ ПЕРВОГО УРОВНЯ (FIRST-LEVEL CACHE)

Кэш первого уровня – это кэш сессии (Session), который является обязательным. Это и есть PersistenceContext, через него проходят все запросы.

В случае, если мы выполняем несколько обновлений объекта, Hibernate старается отсрочить (насколько это возможно) обновление этого объекта для того, чтобы сократить количество выполненных запросов в БД.

Например, при пяти подряд одинаковых запросах на получение одного и того же объекта из БД в рамках одного persistence context, запрос в БД будет выполнен один раз, а остальные четыре загрузки будут выполнены из кэша. Если мы закроем сессию, то все объекты, находящиеся в кэше, теряются, а далее – либо сохраняются в БД, либо обновляются.



IT MENTOR

Особенности кэша первого уровня:

- Включен **по умолчанию**, его нельзя отключить;
- **Связан с сессией** (контекстом персистентности), то есть разные сессии видят только объекты из своего кэша, и не видят объекты, находящиеся в кэшах других сессий;
- При **закрытии сессии** контекст персистентности **очищается** — кэшированные объекты, находившиеся в нем, удаляются;
- При **первом запросе** сущности из БД, она **загружается** в кэш, связанный с этой сессией;
- Если в рамках этой же сессии мы снова запросим эту же сущность из БД, то она будет загружена из кэша, и никакого второго SQL-запроса в БД сделано не будет;
- Сущность можно **удалить** из кэша сессии методом **evict**(), после чего следующая попытка получить эту же сущность повлечет обращение к базе данных;
- Метод **clear() очищает** весь кэш сессии.

КЭШ ВТОРОГО УРОВНЯ

Если кэш первого уровня привязан к объекту сессии, то кэш второго уровня привязан к **объекту-фабрике сессий** (Session Factory object). Следовательно, кэш второго уровня доступен одновременно в нескольких сессиях, или контекстах персистентности.

Особенности кэша второго уровня:

- Кэш второго уровня требует настройки и поэтому **не включен** по умолчанию.
- Настройка кэша заключается в **конфигурировании реализации кэша** и **разрешения** сущностям быть **закэшированными**.
- Чтение из кэша второго уровня происходит только в том случае, если нужный объект **не был найден** в кэше первого уровня

Hibernate не реализует сам никакого **in-memory cache**, а использует существующие реализации кэшей.

Hibernate поставляется со встроенной поддержкой стандарта кэширования **Java JCache**, а также двух популярных библиотек кэширования: **Ehcache** и **Infinispan**.

Чтобы использовать кэш второго уровня нужно:

Добавить **соответствующую зависимость** (например ehcache):

```
<dependency>
<groupId>org.hibernate</groupId>
<rtifactId>hibernate-ehcache</artifactId>
<version>5.4.21.Final</version>
</dependency>
```

IT MENTOR

Далее нужно **включить кэш** второго уровня и определить конкретного провайдера:

```
hibernate.cache.use_second_level_cache=true
      hibernate.cache.region.factory_class=org.hiber-
nate.cache.ehcache.EhCacheRegionFactory
```

И установить у нужных сущностей JPA-аннотацию **@Cacheable**, обозначающую, что сущность нужно кэшировать, и Hibernate аннотацию **@Cache**, настраивающую детали кэширования, у которой в качестве параметра указать **стратегию параллельного доступа**.

```
@Entity
@Table(name = “shared_doc”)
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class SharedDoc{
    private Set<User> users;
}
```

СТРАТЕГИЯ ПАРАЛЛЕЛЬНОГО ДОСТУПА К ОБЪЕКТАМ

Проблема: кэш второго уровня доступен из нескольких сессий сразу и несколько потоков программы могут **одновременно** в разных транзакциях работать с **одним и тем же** объектом. В связи с этим необходимо обеспечивать их **одинаковым представлением** этого объекта.

В Hibernate существует **четыре стратегии одновременного доступа** к объектам в кэше:

- **READ_ONLY** — используется только для сущностей, которые никогда не изменяются (будет выброшено исключение, если попытаться обновить такую сущность).
- **NONSTRICT_READ_WRITE** — кэш обновляется после совершения транзакции, которая изменила данные в БД и закоммитила их.
- **READ_WRITE** — эта стратегия гарантирует строгую согласованность, которую она достигает, используя «мягкие» блокировки: когда обновляется кэшированная сущность, на нее накладывается мягкая блокировка, которая снимается после коммита транзакции. (**Ehcache** использует эту стратегию по умолчанию.)
- **TRANSACTIONAL** — полноценное разделение транзакций. Каждая сессия и каждая транзакция видят объекты, как если бы только они с ним работали последовательно одна транзакция за другой.

IT MENTOR

КЭШ ЗАПРОСОВ (QUERY CACHE)

Результаты запросов также могут быть кэшированы. Это полезно, если мы часто выполняем запрос к объектам, которые редко меняются. Чтобы включить кэш запросов, установите для свойства **hibernate.cache.use_query_cache** значение **true**.

Затем для **каждого** запроса мы должны явно указать, что запрос кэшируется через подсказку в запросе **setHint("org.hibernate.cacheable", true)**:

```
entityManager.createQuery("select f from Foo f")
        .setHint("org.hibernate.cacheable", true)
        .getResultList();
```

Кэш запросов похож на кэш второго уровня. Но в отличии от него, ключом к данным кэша выступает не идентификатор объекта, а **совокупность параметров запроса**. А сами данные — это **идентификаторы объектов**, соответствующих критериям запроса, а не значения полей сущностей. И, чтобы получить закэшированный объект, мы должны по данному идентификатору его найти в этом же кэше. Именно поэтому кэш запросов рационально использовать с кэшем второго уровня.

У кэша запросов есть и своя цена — Hibernate будет вынужден отслеживать сущности закэшированные с определённым запросом и выкидывать запрос из кэша, если кто-то поменяет значение сущности. То есть для кэша запросов стратегия параллельного доступа всегда read-only.

АННОТАЦИИ

@IDCLASS

@IdClass используется в случаях, когда необходимо сформировать составной первичный ключ. Указывает на класс в котором определены поля, которые формируют **составной ключ**.

```
@Embeddable
public class BookId {
    private String title;
    private String language;

    // default constructor
    public BookId(String title, String language) {
        this.title = title;
        this.language = language;
    }

    // getters, equals() and hashCode() methods
}
```

Нужно аннотировать сущность **Account** аннотацией **@IdClass**. Мы также должны объявить поля из класса **AccountId** в сущности **Account с такими же именами** и аннотировать их с помощью **@Id**:

```
@Entity
@IdClass(AccountId.class)
public class Account {
    @Id
    private String accountNumber;
    @Id
    private String accountType;
    // other fields, getters and setters
}
```

@EMBEDDEDID

@EmbeddedId используется в случаях, когда необходимо сформировать составной первичный ключ. Помечает поле - объект, который является **составным первичным ключом**.

```
@Embeddable
public class BookId {
    private String title;
    private String language;
    // default constructor
    public BookId(String title, String language) {
        this.title = title;
        this.language = language;
    }
    // getters, equals() and hashCode() methods
}
```

Нужно встроить этот класс в сущность Book, используя **@EmbeddedId**:

```
@Entity
public class Book {
    @EmbeddedId
    private BookId bookId;
    // constructors, other fields, getters and setters

}
```

@ORDERBY

@OrderBy используется для сортировки данных. **Указывает порядок**, в соответствии с которым должны располагаться элементы коллекций сущностей, базовых или встраиваемых типов при их извлечении из БД. Эта аннотация может использоваться с аннотациями @ElementCollection, @OneToMany, @ManyToMany.

При использовании с коллекциями **базовых типов**, которые имеют аннотацию @ElementCollection, элементы этой коллекции будут отсортированы в натуральном порядке, по значению базовых типов:

```
@ElementCollection
@OrderBy
private List<String> phoneNumbers;
```

В данном примере коллекция строк phoneNumbers, будет отсортирована в натуральном порядке, по значениям базового типа String.

IT MENTOR

@OrderColumn

@OrderColumn используется для сортировки данных. **Создает столбец** в таблице, который используется для поддержания постоянного порядка в списке, но этот столбец **не считается** частью состояния сущности.

Hibernate отвечает за поддержание порядка как в базе данных при помощи столбца, так и при получении сущностей и элементов из БД. Hibernate отвечает за обновление порядка при записи в базу данных, чтобы отразить любое добавление, удаление или иное изменение порядка, влияющее на список в таблице.

Аннотация @OrderColumn может использоваться с аннотациями @ElementCollection, @OneToMany, @ManyToMany и указывается на стороне отношения, ссылающегося на коллекцию, которая должна быть упорядочена. В примере ниже Hibernate добавил в таблицу Employee_PhoneNumbers третий столбец PHONENUMBERS_ORDER, который и является результатом работы @OrderColumn.

```
‘Show Columns from Employee_PhoneNumbers’

[EMPLOYEE_ID, BIGINT(19), NO, PRI, NULL]
[PHONENUMBERS, VARCHAR(255), YES, , NULL]
[PHONENUMBERS_ORDER, INTEGER(10), NO, PRI, NULL]

‘Select * FROM Employee_PhoneNumbers’

[1, 111-111,111, 0]
[1, 222-222,222, 1]
[2, 333-333-333, 0]
[2, 444-444-444, 1]
[2, 666-666-666, 2]
[3, 555-555-555, 0]
```

РЕШЕНИЕ ПРОБЛЕМЫ N+1

Проблема N+1 запросов возникает, когда получение данных из БД выполняется за **N дополнительных SQL-запросов** для извлечения тех же данных, которые могли быть получены при выполнении основного SQL-запроса.

Допустим, у нас есть две сущности Post и PostComment. В БД имеется 4 Post и у каждого из них по одному PostComment:

```
@Entity
@Table(name = “post”)
public class Post {
    @Id
    private Long id;
    private String title;

    //Getters and setters omitted for brevity

}


@Entity
@Table(name = “post_comment”)
public class PostComment {

    @Id
    private Long id;
```

IT MENTOR

```
@ManyToOne
    private Post post;
    private String review;

    //Getters and setters omitted for brevity

}
```

N+1 при FetchType.EAGER:

Так как у @ManyToOne план извлечения по умолчанию — EAGER, то при получении из БД сущности PostComment немедленно будет загружена связанная с ней сущность Post:

```
List<PostComment> comments = entityManager.createQuery(
“select pc from PostComment pc”, PostComment.class).ge-
tResultList();
```

Этот SQL-запрос приведет к проблеме N+1 и выполнит больше запросов, чем нужно:

```
SELECT
    pc.id AS id1_1_,
    pc.post_id AS post_id3_1_,
    pc.review AS review2_1_
FROM
    post_comment pc
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM
post p WHERE p.id=1
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM
post p WHERE p.id=2
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM
post p WHERE p.id=3
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM
post p WHERE p.id=4
```

Обратите внимание на дополнительные 4 оператора SELECT, которые выполняются, потому что 4 сущности Post должны быть извлечены из БД до возврата списка из 4 сущностей PostComment с инициализированными полями Post.

N+1 при FetchType.LAZY

Даже если мы явно переключимся на использование FetchType.LAZY для всех ассоциаций, мы всё равно можем столкнуться с проблемой N+1.

```
@ManyToOne(fetch = FetchType.LAZY)
private Post post;
```

IT MENTOR

Когда мы выбираем все сущности PostComment, Hibernate выполнит одну инструкцию SQL:

```
SELECT
    pc.id AS id1_1_,
    pc.post_id AS post_id3_1_,
    pc.review AS review2_1_
FROM
    post_comment pc
```

Но, если в этом же контексте персистентности, мы обратимся к сущностям Post в PostComment:

```
for(PostComment comment : comments) {
        “The Post ‘{}’ got this review ‘{}’”,
        comment.getPost().getTitle(),
        comment.getReview()
        }
```

То мы опять получим проблему N+1:

```
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM
post p WHERE p.id=1
-- The Post ‘High-Performance Java Persistence - Part
1’ got this review
-- ‘Excellent book to understand Java Persistence’
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM
post p WHERE p.id=2
-- The Post ‘High-Performance Java Persistence - Part
2’ got this review
-- ‘Must-read for Java developers’
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM
post p WHERE p.id=3
-- The Post ‘High-Performance Java Persistence - Part
3’ got this review
-- ‘Five Stars’
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM
post p WHERE p.id=4
-- The Post ‘High-Performance Java Persistence - Part
4’ got this review

-- ‘A great reference book’
```

Поскольку Post извлекается **лениво**, вторая группа SQL-запросов, состоящая в нашем случае из 4 штук, будет выполняться при обращении к ленивым полям Post у каждого PostComment.

IT MENTOR

РЕШЕНИЯ ПРОБЛЕМЫ N+1

JOIN FETCH

При FetchType.EAGER и FetchType.LAZY нам поможет запрос с **JOIN FETCH**. Опцию “**FETCH**” можно использовать в **JOIN**(INNER JOIN или LEFT JOIN) для выборки связанных объектов в одном запросе вместо дополнительных запросов для каждого доступа к ленивым полям объекта.

```
List<PostComment> comments = entityManager.createQ-
uery(“
        select pc
        from PostComment pc
        join fetch pc.post p
        “, PostComment.class).getResultList();

for(PostComment comment : comments) {
        “The Post ‘{}’ got this review ‘{}’”,
        comment.getPost().getTitle(),
        comment.getReview()
        }
```

На этот раз Hibernate выполнит одну инструкцию SQL:

```
SELECT
    pc.id as id1_1_0_,
    pc.post_id as post_id3_1_0_,
    pc.review as review2_1_0_,
    p.id as id1_0_1_,
    p.title as title2_0_1_
FROM
    post_comment pc
        INNER JOIN
    post p ON pc.post_id = p.id
```

IT MENTOR

Использование **LEFT JOIN FETCH** аналогично JOIN FETCH, только будут загружены **все** сущности из таблицы PostComment, даже те, у которых нет связанной сущности Post

@Fetch(FetchMode.SUBSELECT) Это аннотация **Hibernate**, в **JPA** её нет. Можно использовать **только с коллекциями**. Будет сделан **один sql-запрос** для получения корневых сущностей и, если в контексте персистентности будет обращение к ленивым полям-коллекциям, то выполнится еще один запрос для получения связанных коллекций:

```
// Первый запрос:
select ...
 from customer customer0_
// Второй запрос:
select ...
     from
order order0_
where
    order0_.customer_id in (
    select
    customer0_.id
    from
    customer customer0_
    )
```