ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Современные технологии программирования

Пальчевский Евгений Владимирович
Кандидат технических наук,
старший преподаватель кафедры информационных технологий

О курсе

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

<u>Прикладная информатика.</u> Что ждёт в осеннем семестре 2024/25 учебного года?

1. **Лекции (8 штук)**
2. **Семинарские занятия (9 штук), исходя из нагрузки.**
3. **Выполнение индивидуальных заданий (решение задач) для семинарских занятий (в том числе и дома).**
4. **Контрольные работы.**
5. **За всё вы получаете баллы в соответствии с балльно-рейтинговой системой (БРС).**
6. **Зачет.**

Темы лекционных и семинарских занятий

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Основные темы лекций (некоторые видеолекции включают две и более тем):

1. Обзор платформы и языка программирования Java.
2. Основные языковые конструкции Java: лекция №1, лекция №2.
3. Базовые принципы ООП и работа с базами данных. Лекция №3: (часть 1, часть 2, часть 3, часть 4).
4. Классы, методы и объекты Java. Лекция №3: (часть 1, часть 2, часть 3, часть 4).
5. Интерфейсы. Исключения. Лекция №4: (часть 1, часть 2, часть 3, часть 4, часть 5, часть 6, часть 7, часть 8).
6. Коллекции. Лекция №4: (часть 1, часть 2, часть 3, часть 4, часть 5, часть 6, часть 7, часть 8).

Плейлист YouTube/плейлист ВК

Балльно-рейтинговая система (БРС)

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Вид поощрения	Баллы	Максимум	Возможно набрать согласно БРС
Посещение семинарских занятий и лекций	0,59 за каждое посещение	**10**	**40**
Подготовка к семинарским и практическим занятиям (в том числе и выступления, онлайн-курсы)	0,765 за семинар (онлайн-курсы, решения задач)	**13**	
Контрольные работы	4,25 за каждую (4 контрольных)	**17**	
Зачет/экзамен	60	**60**	**60**
ИТОГО	**Зачтено** идет от 50 баллов и выше	**100**	**100**

Ваши рейтинги с комментариями будут доступны в личном кабинете студента на org.fa.ru

Баллы за задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

По факту за работу в семестре можно набрать не 40 баллов, а гораздо больше, что будет плюсом на экзамене

Вид задачи	Баллы
Сложный вариант (Консольное меню + MySQL + Excel)*	5
Контрольная работа	6
Суммарно	79

Баллы за посещение лекций и семинарских занятий входят в баллы за задачи и контрольные работы.
Как считать? 11 задач * 5 баллов = 55 баллов, 4 к/р * 6 = 24.
ИТОГО: 55+24 = 79.
Набираете 70+ баллов, получаете зачет автоматом.

Ваши рейтинги с комментариями будут доступны в личном кабинете студента на org.fa.ru

Даты контрольных работ

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Дата	Группы
28.02.2025 31.03.2025 30.04.2025 30.05.2025	Все группы, кроме БИ21-3 и БИ21-4

Ваши рейтинги с комментариями будут доступны в личном кабинете студента на **org.fa.ru**

Контакты с преподавателем

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Социальная сеть / мессенджер / почта	Контакт
Электронная почта	teelxp@inbox.ru, evpalchevskij@fa.ru
VK	https://vk.com/teelxp
WhatsApp	+7-937-485-80-48
YouTube YouTube-канал	https://www.youtube.com/@teelxp
Лекции в ВК (альтернатива ютубчику)	https://vk.com/finkablog

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Требования к лекциям

1. Посещение лекций.
2. Ноутбук.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Материал на весенний семестр 2024/25 учебного года

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Какое ПО можно использовать?

Логотип	Название	Группы
	IntelliJ IDEA Ultimate	https://palchevsky.ru/materials.php
	Visual Studio Code	https://code.visualstudio.com/
	MySQL	https://dev.mysql.com/downloads/installer/
	PostgreSQL	https://www.postgresql.org/download/
	MongoDB	https://www.mongodb.com/try/download/community-edition

Курсы для дополнительного освоения материала

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

https://stepik.org/course/82867/promo - базовый, но очень широкий курс
https://stepik.org/course/53650/promo - JavaFX, Swing **(самая база, поэтому придется еще много чего самим прогуглить)**
https://stepik.org/course/146/promo - ORM и Hibernate, паттерны, MySQL
https://stepik.org/course/63054/promo - SQL
https://stepik.org/course/1240/promo - Введение в базы данных

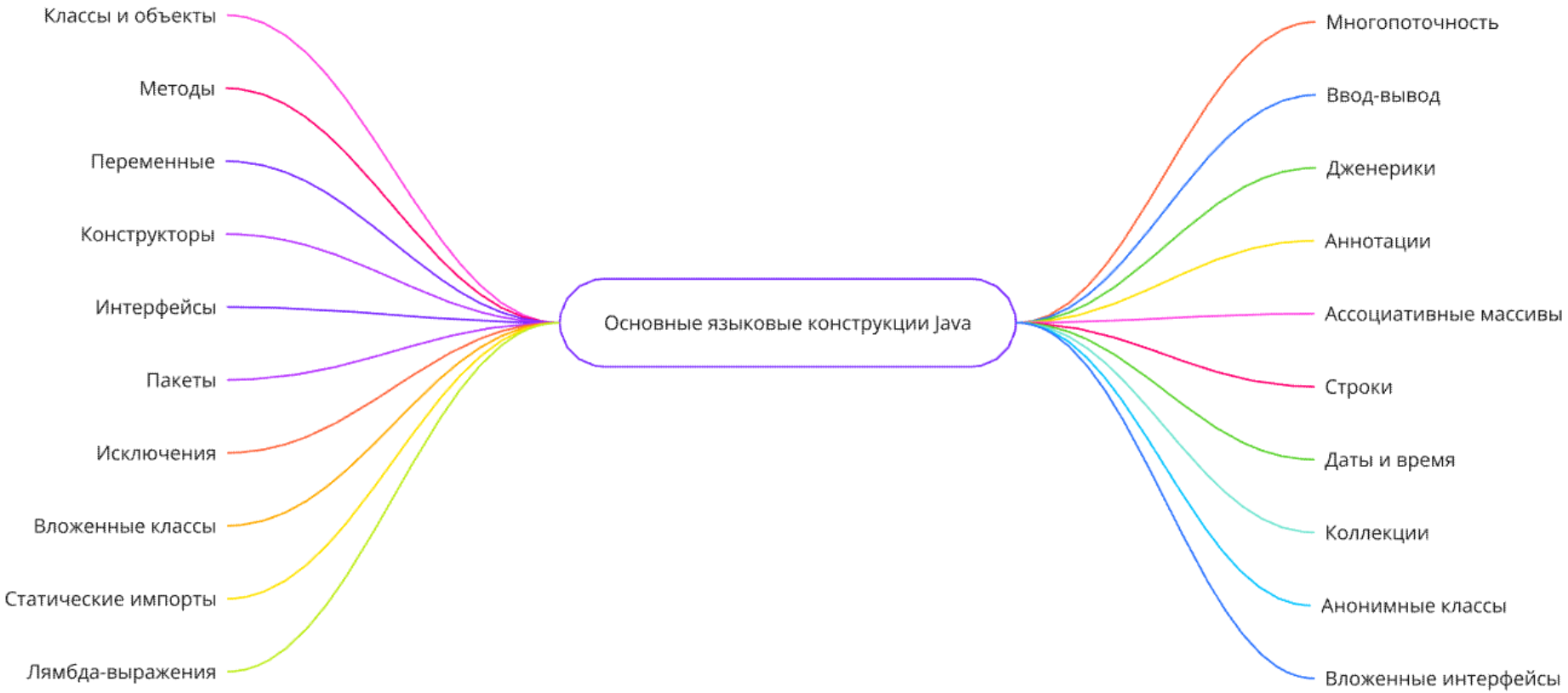
ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Дедлайн

# задач	Дедлайн	Группы
1-4	28.02.2025	Все группы
5-8	31.03.2025	
9-11	30.04.2025	
12-15	31.05.2025	

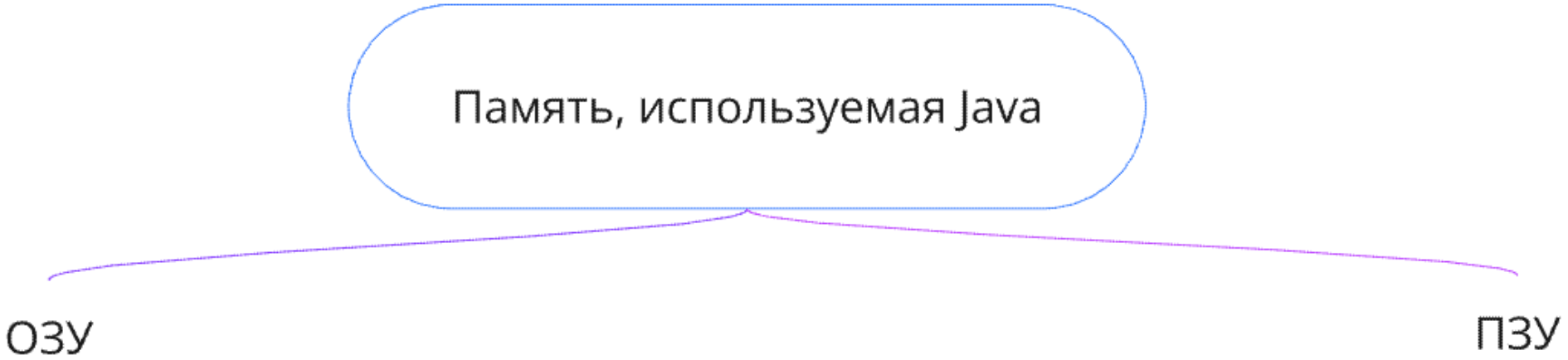
ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.



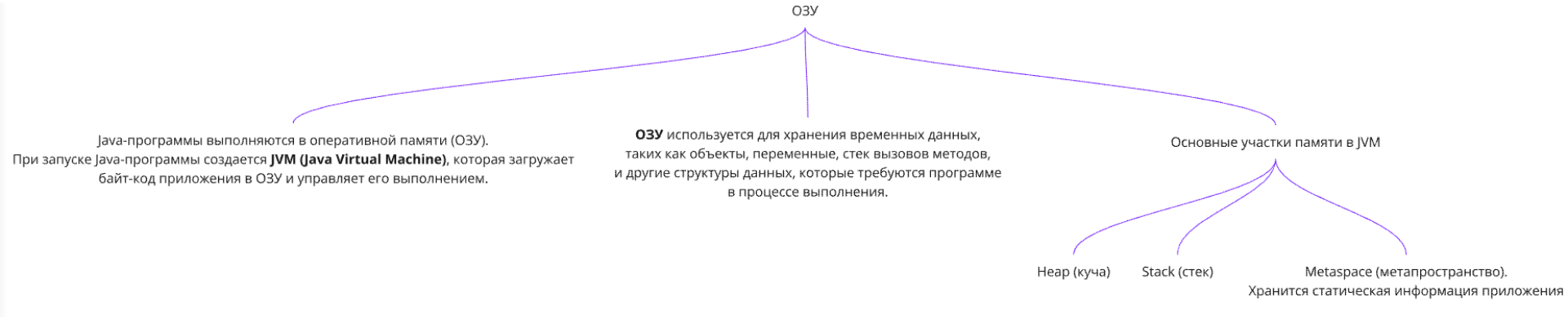
ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Какой тип памяти использует Java для выполнения своих программ?



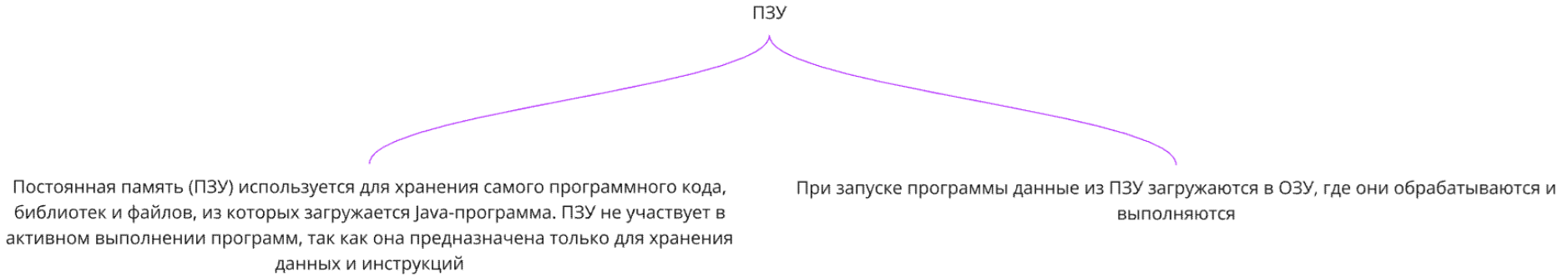
ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Какой тип памяти использует Java для выполнения своих программ?



ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Какой тип памяти использует Java для выполнения своих программ?



Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Почему ОЗУ?

1. **Быстродействие**. ОЗУ является гораздо более быстрой памятью по сравнению с ПЗУ, что позволяет JVM быстро выполнять операции и обрабатывать данные.
2. **Временные данные и переменные.** В ОЗУ хранятся временные данные, такие как объекты, переменные, вызовы методов, которые изменяются и обновляются во время выполнения программы.
3. **Гибкость управления памятью.** ОЗУ позволяет динамически выделять и освобождать память, что важно для создания и удаления объектов в Java.
4. **Garbage Collection (Сборка мусора)**. Java использует механизм сборки мусора для автоматического управления памятью в куче, удаляя неиспользуемые объекты, чтобы освободить память для новых данных.

Java активно использует **оперативную память (ОЗУ)** для выполнения своих программ, так как она предоставляет необходимые возможности для динамического управления данными и обеспечивает высокую производительность. Постоянная память (ПЗУ) используется только для хранения исходного кода и библиотек, которые загружаются в ОЗУ при выполнении программ.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Классы, методы и объекты.

Класс – основной строительный блок в Java. Он определяет структуру и поведение объектов. Объекты создаются на основе классов и содержат данные (поля) и методы (функции).

```
package test;

public class main1 {
```

Метод – совокупность команд, выполняющих определенные действия над объектами или возвращают значения. Они могут быть статическими (доступными без создания экземпляра класса) или нестатическими (доступными через объект).

```
package test;

public class main1 {
   //Основной метод Java является точкой входа любой программы Java . Его синтаксис всегда public static void main(String[] args).
   //По большому счету, в нем мы можем изменить только имя аргумента args массива входных данных string. Например, давайте изменим его на test.
   public static void main(String[] test) {
```

Объект – переменная внутри класса или метода.

```
int a = 10;
```

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

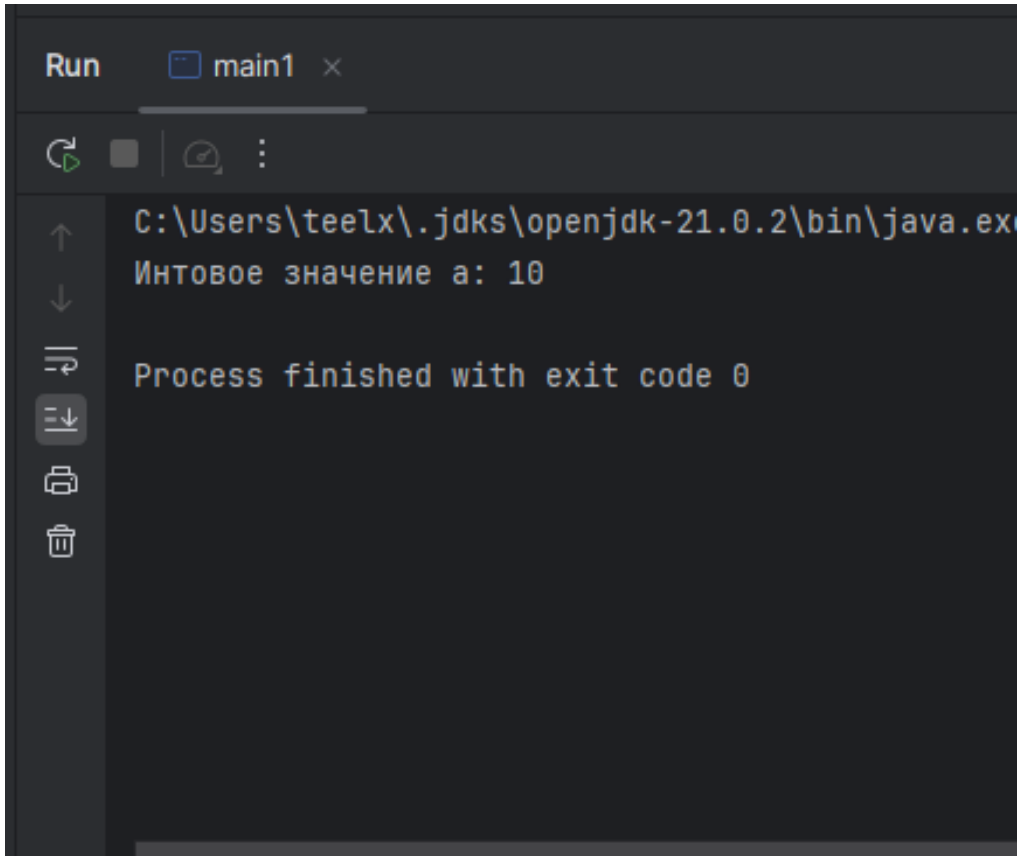
Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Классы и объекты.

Полный исходный код

```
package test;

public class main1 {
   //Основной метод Java является точкой входа любой программы Java . Его синтаксис всегда public static void main(String[] args).
   //По большому счету, в нем мы можем изменить только имя аргумента args массива входных данных string. Например, давайте изменим его на test.
   public static void main(String[] test) {

//инициализируем переменные значениями. Отрицательные значения в типе данных byte использовать нельзя
      int a = 10;

//выводим данные из переменных с помощью методов вывода
//System - сущность (т.е. класс), выполняющая роль некого посредника (ну моста), соединяющая программу и нашу среду разработки (IDE)
//Out - сущность (т.е. класс), хранящаяся внутри сущности (класса) System
//println - метод, который вызывается у класса out для вывода данных в нашу консоль
      System.out.println("Интовое значение a: " + a); // объединяем строку – конкатенация

   }
}
```

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Классы и объекты.

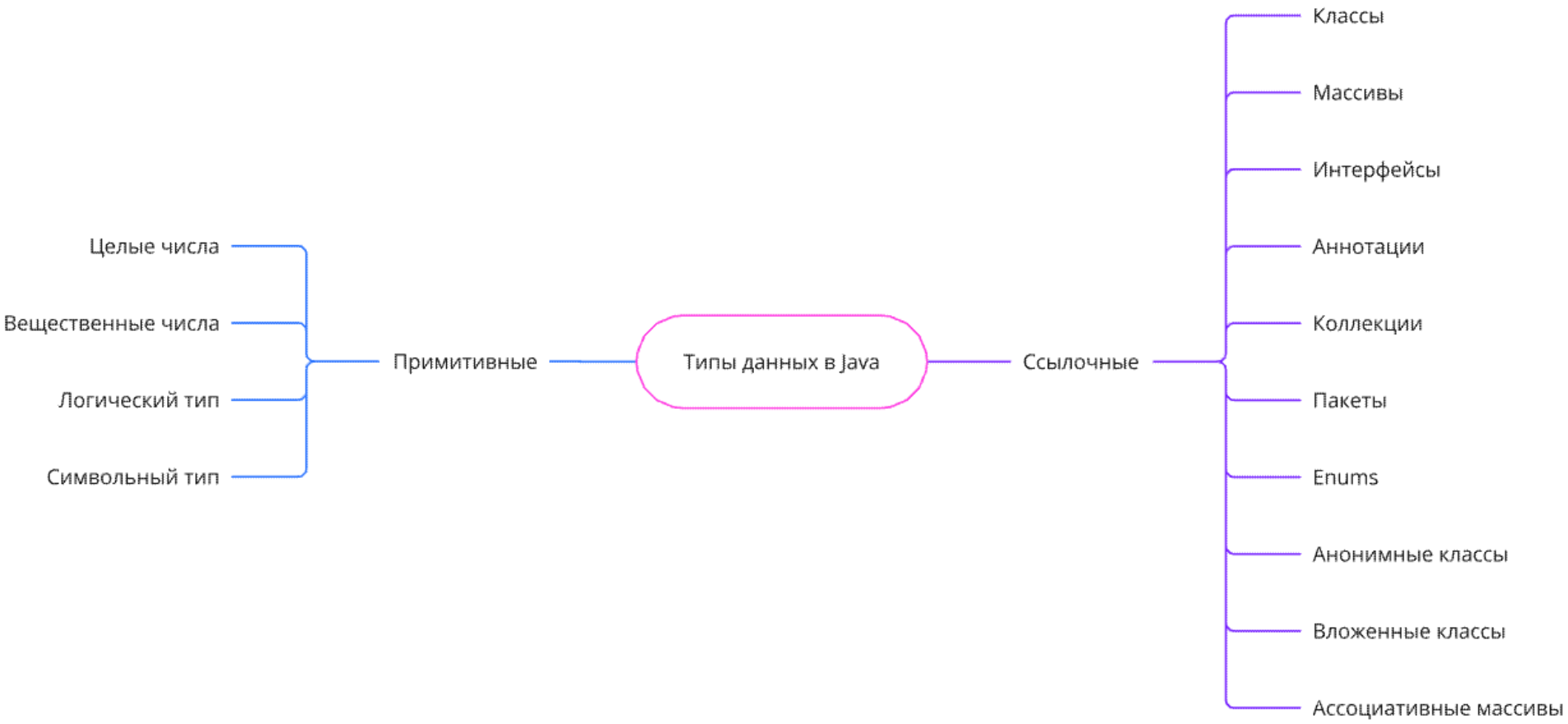
Результат в консоли



ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.

Типы данных



ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

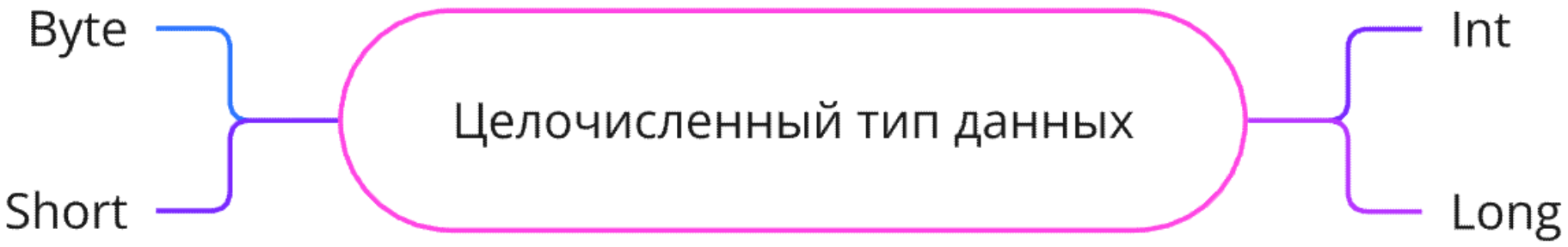
Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.

Отличие типов данных

Примитивные переменные (типы данных)	Ссылочные переменные (типы данных)
Хранят значение	Хранят адрес объекта в памяти, на который ссылаются (отсюда и название). Используются для доступа к объектам (его нельзя получить, если на объект нет ссылки)
Создаются присваиванием значения	Создаются через конструкторы классов (присваивание только создаёт вторую ссылку на существующий объект)
Имеют строго заданный диапазон допустимых значений	По умолчанию их значение — null
В аргументы методов попадают копии значения переменной (это передача по значению)	В методы передаётся значение ссылки — операция выполняется над оригинальным объектом, на который ссылается переменная
	Могут использоваться для ссылки на любой объект объявленного или совместимого типа

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Целочисленный тип данных



ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

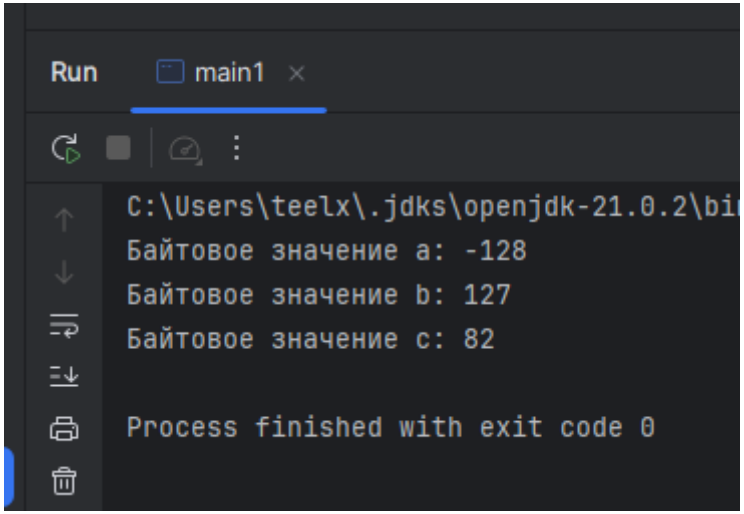
Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Целочисленный тип (подтип) данных Byte

Byte. 8-битное целое число со знаком в диапазоне от -128 до 127.
Далеко не самый популярный подвид целочисленного типа данных. Как правило, используется при работе с потоками больших данных, который получили либо из файла, либо по сети.

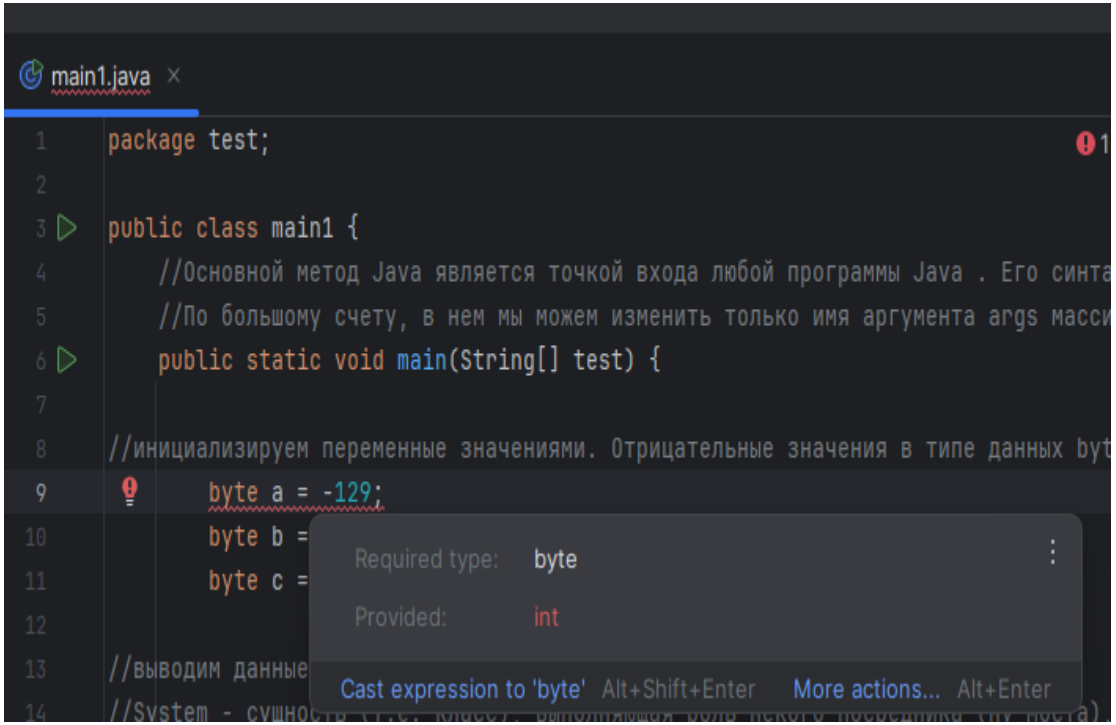
```
package test;

public class main1 {
   //Основной метод Java является точкой входа любой программы Java . Его синтаксис всегда public static void main(String[] args).
   //По большому счету, в нем мы можем изменить только имя аргумента args массива входных данных string. Например, давайте изменим его на test.
   public static void main(String[] test) {

//инициализируем переменные значениями. Отрицательные значения в типе данных byte использовать нельзя
      byte a = -128;
      byte b = 127;
      byte c = 82;

//выводим данные из переменных с помощью методов вывода
//System - сущность (т.е. класс), выполняющая роль некого посредника (ну моста), соединяющая программу и нашу среду разработки (IDE)
//Out - сущность (т.е. класс), хранящаяся внутри сущности (класса) System
//println - метод, который вызывается у класса out для вывода данных в нашу консоль
      System.out.println("Байтовое значение a: " + a); // объединяем строку
      System.out.println("Байтовое значение b: " + b);
      System.out.println("Байтовое значение c: " + c);

   }
}
```

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Целочисленный тип (подтип) данных Byte

Результат



Если введем значение больше, то ошибка



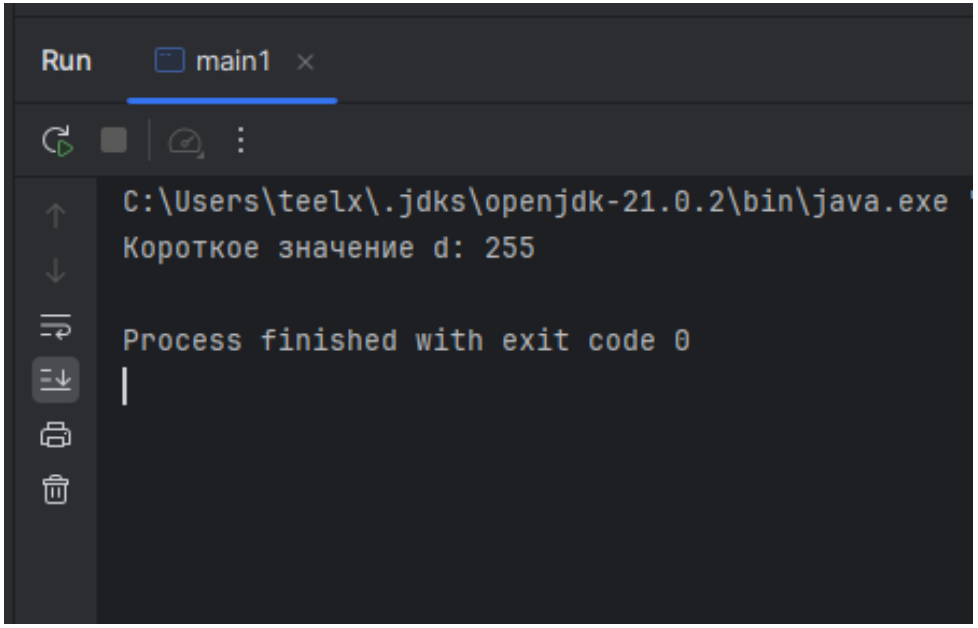
ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Целочисленный тип (подтип) данных Short

Short. 16-битное целое число со знаком в диапазоне от -32768 до 32767.

По сравнению с подвидом byte у него увеличенный, но всё же ограниченный диапазон значений. Используется, когда требуется хранить небольшие целые числа, что способствует экономии оперативной памяти.

```
package test;

public class main1 {

    public static void main(String[] test) {

//Инициализируем переменные значениями
        short d = 255;
        System.out.println("Короткое значение d: " + d);

    }
}
```

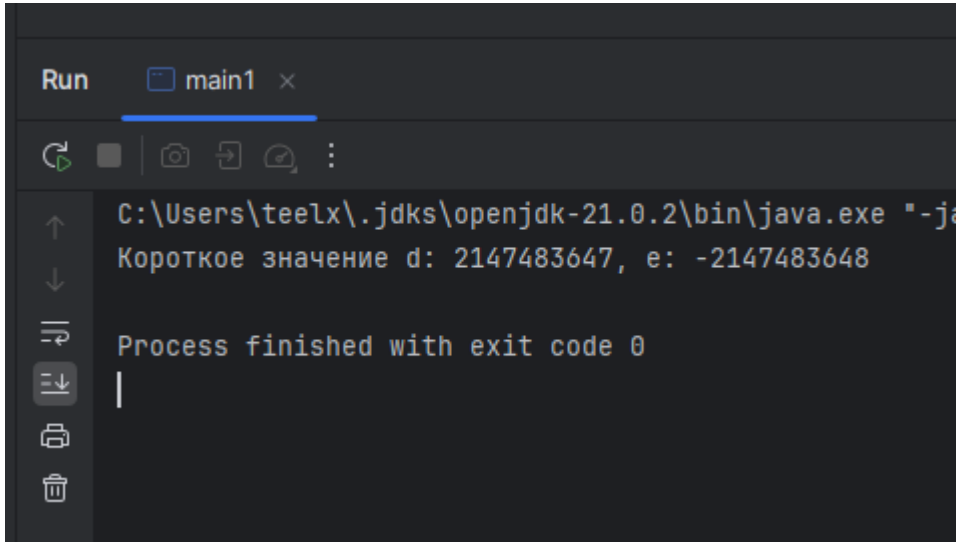


ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Целочисленный тип (подтип) данных Int

Int (Integer). 32-битное целое число со знаком в диапазоне от -2^31 до 2^31-1. Является самым популярным целочисленным типом данных. При вычислениях в виртуальной машине остальные целочисленные типы (byte, short) занимают столько же памяти, сколько int. В основном мы будем работать с этим подвидом. Минимальное значение: **-2,147,483,648**. Максимальное значение: **2 147 483 647**.

```
package test;

public class main1 {

    public static void main(String[] test) {

        int d = 2147483647;
        int e = -2147483648;
        System.out.println("Короткое
значение d: " + d + ", e: " + e);

    }
}
```

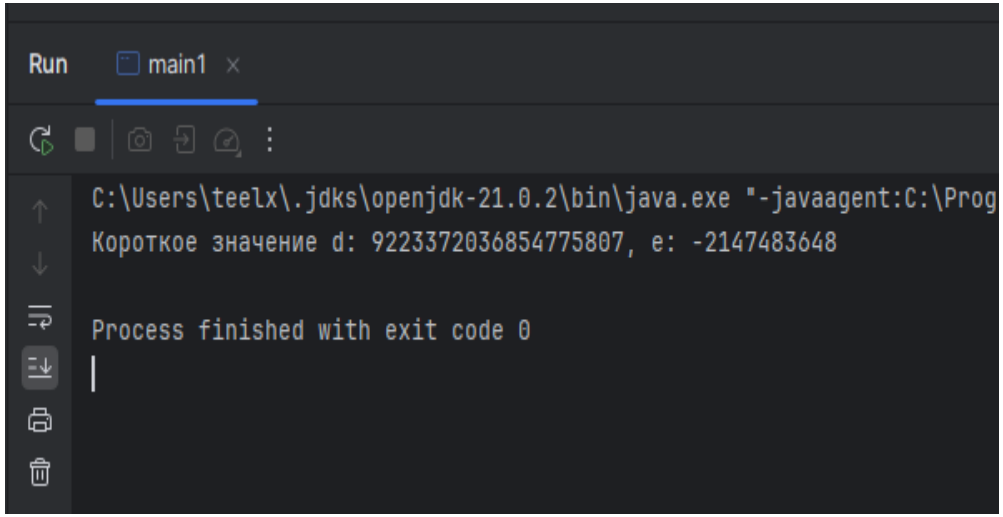


ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Целочисленный тип (подтип) данных Long

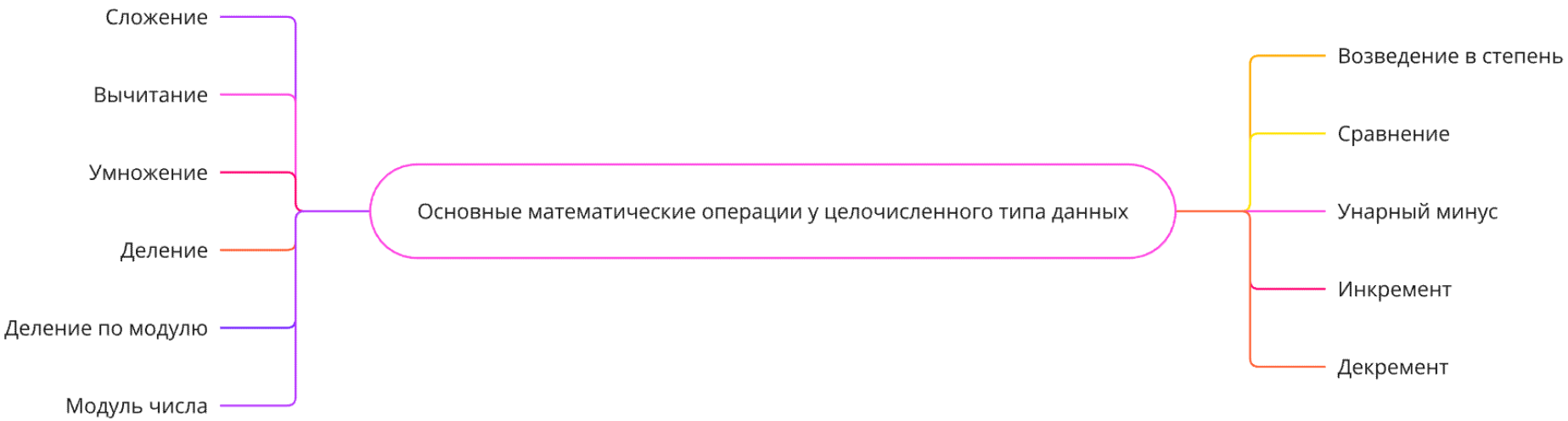
Long. 64-битное целое число со знаком в диапазоне от -2^63 до 2^63-1.
Тип данных long в Java используется для представления целых чисел со знаком, которые занимают 64 бита (8 байт) в памяти. Этот тип данных может хранить числа в диапазоне от -2^63 до 2^63-1. Тип long обычно используется, когда требуется хранить очень большие целые числа или когда требуется большая точность, чем может предоставить тип int.
Минимум: -9 223 372 036 854 775 808. Максимум: 9 223 372 036 854 775 807.

```
package test;

public class main1 {

    public static void main(String[] test) {

        long d = 9223372036854775807L;
        long e = -2147483648;
        System.out.println("Длинное значение d: " + d + ", e: " + e);

    }
}
```



ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Основные математические операции у целочисленного типа данных



ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Основные математические операции у целочисленного типа данных

```
package test;

public class main1 {

    public static void main(String[] test) {

        byte a = 20, b = 30, c = 35;
        byte x = (byte)(a+b+c); // Тут мы обозначаем, что
складываем значения целочисленного типа данных byte
        System.out.println("Сумма: " + x);
        byte x1 = (byte)(a-b-c);
        System.out.println("Разность: " + x1);
        byte x2 = (byte)(a*b*c);
        System.out.println("Произведение: " + x2);
        byte x3 = (byte)(a / b);
        System.out.println("Частное: " + x3);
        byte x4 = (byte)(a % b); // например, a % 10
        System.out.println("Частное (деление по модулю): " + x4);
        byte x5 = (byte)(-a % 10);
        System.out.println("Остаток: " + x5);
        byte x6 = (byte)(Math.abs(a));
        System.out.println("Число по модулю: " + x6);
        byte x7 = (byte)(Math.pow(b, 2));
        System.out.println("Число " + b + " в степени 2 равно " + x7);
        System.out.println("Результат операции сравнения: " +
(a==b));

    }
}
```

```
int n = 90;
n = -n;
System.out.println("Результат после первого минуса: " + n);
n = -n;
System.out.println("Результат после второго минуса: " + n);
n++;
System.out.println("Итоговый результат №1: " + n);
n--;
System.out.println("итоговый результат №2: " + n);
```

Run main1

```
C:\Users\teelx\.jdks\openjdk-21.0.2\bin\
Сумма: 85
Разность: -45
Произведение: 8
Частное: 0
Частное (деление по модулю): 20
Остаток: 0
Число по модулю: 20
Число 30 в степени 2 равно -124
Результат операции сравнения: false
Результат после первого минуса: -90
Результат после второго минуса: 90
Итоговый результат №1: 91
итоговый результат №2: 90

Process finished with exit code 0
```

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.

Примитивные типы данных. Вещественный тип данных

Float. 32-битное вещественное число с плавающей точкой в диапазоне от 1.4E-45 до 3.4028235E+38. Занимают 4 байта в памяти.

Тип float используют как экономичный вариант хранения больших массивов данных с плавающей точкой. Соответственно, когда переменной присваивают тип float, язык Java воспринимает её как тип данных double. Чтобы этого не происходило, нужно добавлять в конце переменной символ f или F. Т.е. аналогично целочисленному типу данных Long. Тип float обычно используется, когда требуется хранить числа с плавающей точкой, такие как десятичные дроби или научные числа, и когда точность типа double (64 бита) не требуется.

Double. 64-битное вещественное число с плавающей точкой в диапазоне от 4.9E-324 до 1.7976931348623157E+308.

Тип данных double в Java используется для представления вещественных чисел с плавающей точкой, которые занимают 64 бита (8 байт) в памяти. Этот тип данных может хранить числа в диапазоне от 4.9E-324 до 1.7976931348623157E+308. Тип double обычно используется, когда требуется хранить числа с плавающей точкой с высокой точностью, такие как научные числа или числа с большим количеством десятичных знаков.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Вещественный тип данных

Отличие Float от Double?

Float	Double
32 бита. Хранит меньше данных, точность меньше	64 бита. Хранит больше чисел, точность выше
Производительный тип данных	Медленный тип данных
Базовое представление IEEE 754	Расширенное представление IEEE 754
В среднем 5–9 знаков после запятой*	В среднем 10–17 знаков после запятой*

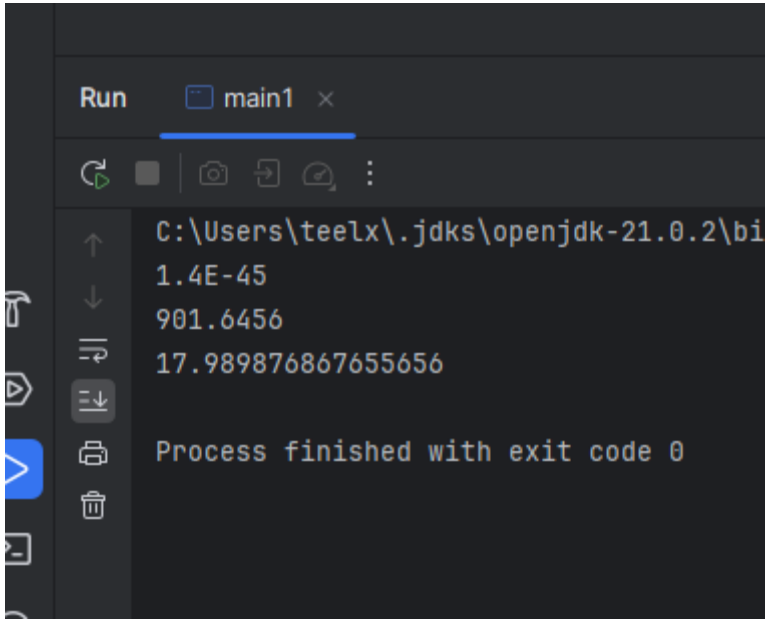
IEEE 754 – стандарт IEEE (*Institute of Electrical and Electronics Engineers,* Институт инженеров электротехники и электроники), описывающий формат представления чисел с плавающей точкой. Используется практически повсеместно.

*Важно отметить, что хотя эти типы могут хранить произвольное количество десятичных знаков, они ограничены в точности из-за использования представления с плавающей точкой. Это означает, что при некоторых операциях, таких как деление двух очень близких чисел, результат может быть округлен до ближайшего представимого значения, что может привести к потере точности. Т.е. с точки зрения синтаксиса все будет хорошо, но по расчетам – не факт.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Вещественный тип данных

```
package test;

public class main1 {

    public static void main(String[] test) {

//инициализация переменных типа float
        float test1 =
0.00000000000000000000000000000000000000000000000014f; //Тип данных float
        float test2 = (float) 901.645656;
//другой вариант инициализации переменной типа float во избежание путаницы с double
        double test3 =
17.989876867655656565;
//инициализация типа данных double
        System.out.println(test1 + "\n" + test2 +
"\n" + test3);
    }
}
```



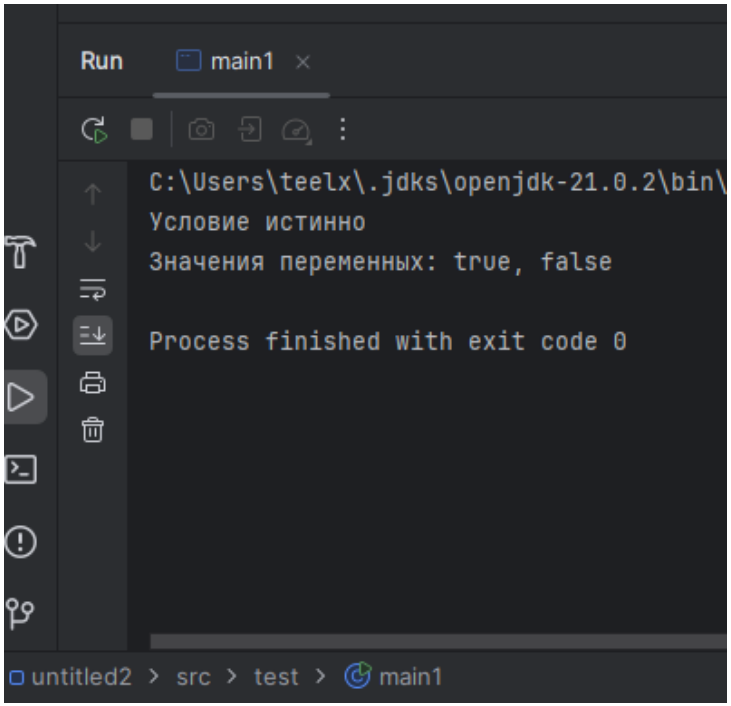
ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Логический тип данных

Boolean. True/False

В Java существует логический тип данных, который называется boolean. Этот тип данных может хранить только два значения: true (истина) и false (ложь). Логические переменные часто используются в условиях (if, while, for и т.д.) для проверки условий и принятия решений в программе.

```
package test;

public class main1 {

    public static void main(String[] test) {

        boolean isValid = true; // логическая переменная, инициализированная значением `true`

        if (isValid) {
            System.out.println("Условие истинно");
        } else {
            System.out.println("Условие ложно");
        }
        boolean test1 = true;
        boolean test2 = false;
        System.out.println("Значения переменных: " + test1 + ", " + test2);

    }
}
```



ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

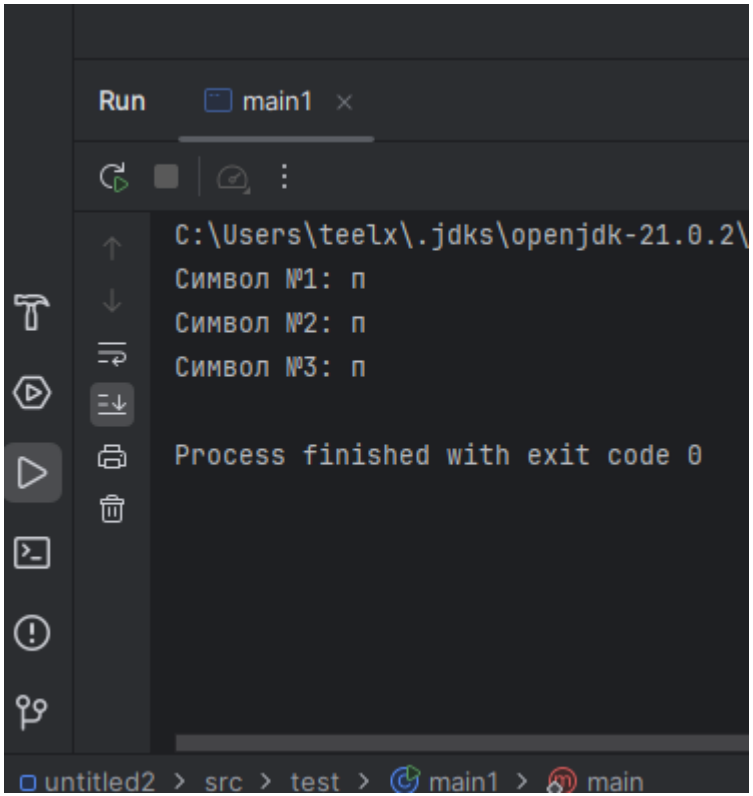
Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Символьный тип данных

Char. 16-битный символ Unicode в диапазоне от U+0000 до U+FFFF.
U+0000 - это шестнадцатеричный код (код в 16-тиричной системе счисления), который обозначает символ NUL (нулевой символ) в Юникоде. Символ NUL не отображается и обычно используется для обозначения конца строки или кадра в некоторых форматах данных.
U+FFFF - это шестнадцатеричный код, который обозначает символ FFFF (шестнадцатеричная запись числа 65535) в Юникоде. Этот символ не имеет специального назначения и обычно используется для обозначения конца блока данных или кадра в некоторых форматах.
Тип данных char используется для хранения и обработки текста, а также для работы с символами, которые не могут быть представлены в виде ASCII-символов.
Unicode предоставляет стандартную кодировку для представления символов из различных языков и систем письма, что делает его идеальным выбором для международного программирования.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Символьный тип данных

```
package test;

public class main1 {

    public static void main(String[] test) {

//инициализируем переменные типа char
        char symbol1 = 1087; //по индексу символа в таблице UTF-8 char  - поиск символов (строка DEC)
        char symbol2 = 'п'; //по значению символа
        char symbol3 = '\u043F'; //через шестнадцатеричную форму Unicode (это всё ещё «п») - https://symbl.cc/ru/unicode-table/ - поиск символов

//вызываем вывод информации
        System.out.println("Символ №1: " + symbol1);
        System.out.println("Символ №2: " + symbol2);
        System.out.println("Символ №3: " + symbol3);

//во всех случаях будет выдан один и тот же символ — «п»


    }
}
```



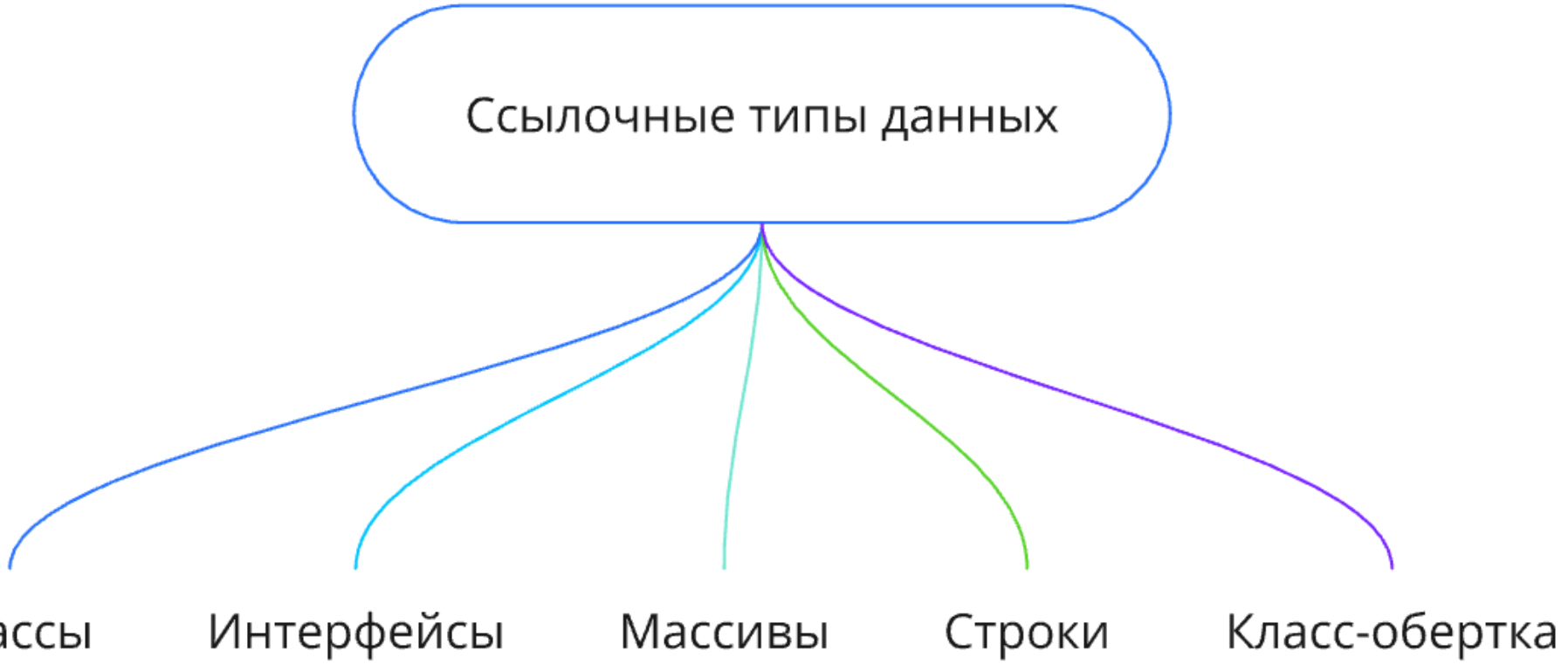
ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java.
Примитивные типы данных. Символьный тип данных
Char vs String?

Критерий	Char	String
Размер	Представляет один символ и занимает 16 бит (2 байта) в памяти	Представляет последовательность символов и может занимать любое количество памяти в зависимости от длины строки
Представление	Представляет один символ	Представляет последовательность символов, которые могут быть разной длины
Методы	Не предоставляет методов для работы со строками	Имеет множество методов для работы со строками, таких как конкатенация, поиск подстрок, преобразование в нижний/верхний регистр и т.д.
Инициализация	Инициализируется одним символом	Инициализируется последовательностью символов в двойных кавычках
Переменные	Используется для хранения одного символа	Хранит строку/строки
Безопасность	Полностью безопасен	Может быть подвержен SQL-инъекции
Доступ к символам	Прямой доступ в память к каждому символу	Необходимы методы для извлечения отдельных символов
Преобразование	Может быть преобразован в String с использованием оператора new String(char)	Может быть преобразован в char с использованием метода charAt()

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных

Ссылочные типы данных в Java представляют собой типы, значения которых являются ссылками (указателями) на объекты в памяти. В отличие от примитивных типов данных, которые хранят сами значения (например, числа или логические значения), ссылочные типы хранят адреса объектов, созданных в динамической памяти (куче).



ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – классы

Ссылочные типы данных в Java представляют собой типы, значения которых являются ссылками (указателями) на объекты в памяти. В отличие от примитивных типов данных, которые хранят сами значения (например, числа или логические значения), ссылочные типы хранят адреса объектов, созданных в динамической памяти (куче).

Создание экземпляра класса:

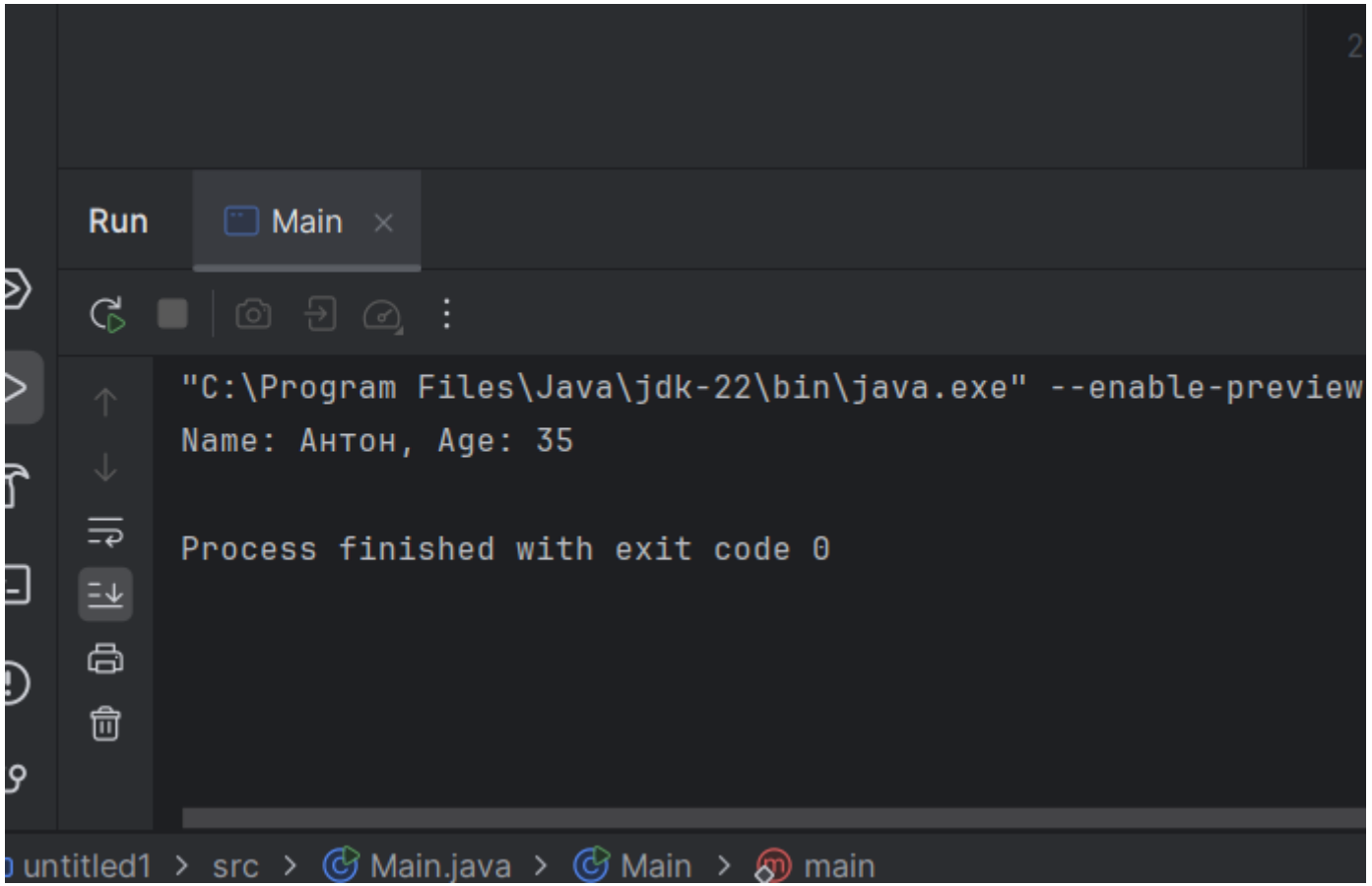
```
// Создание класса Person
class Person {
    String name;  // Поле для хранения имени
    int age;      // Поле для хранения возраста

    // Конструктор класса Person
    Person(String name, int age) {
        this.name = name;  // Инициализация поля name
        this.age = age;    // Инициализация поля age
    }

    // Метод для отображения информации о человеке
    void displayInfo() {
        System.out.println(STR."Name: \{name}, Age: \{age}");
    }
}
```

```
// Основной класс с методом main
public class Main {
    public static void main(String[] args) {
        // Создание экземпляра класса Person
        Person person1 = new Person("Антон", 35); // Вызов
конструктора и создание объекта

        // Вызов метода объекта
        person1.displayInfo();
    }
}
```

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – классы

Результат создания экземпляра класса



Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – интерфейсы

Интерфейсы в Java — это типы данных, которые определяют набор абстрактных методов (без реализации) и могут содержать константы (поля, объявленные как public static final). Интерфейсы используются для создания контрактов, которые определяют, что класс должен делать, но не как именно это делать. Класс, который реализует интерфейс, обязуется реализовать все его методы.

Основные характеристики интерфейсов в Java:

1.Абстрактные методы: Интерфейсы содержат методы без реализации. Методы в интерфейсе по умолчанию являются public и abstract.

2.Константы: Интерфейсы могут содержать константы — поля, которые по умолчанию являются public static final.

3.Поддержка множественного наследования: Класс может реализовать несколько интерфейсов, что позволяет обойти ограничения одиночного наследования в Java.

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – интерфейсы

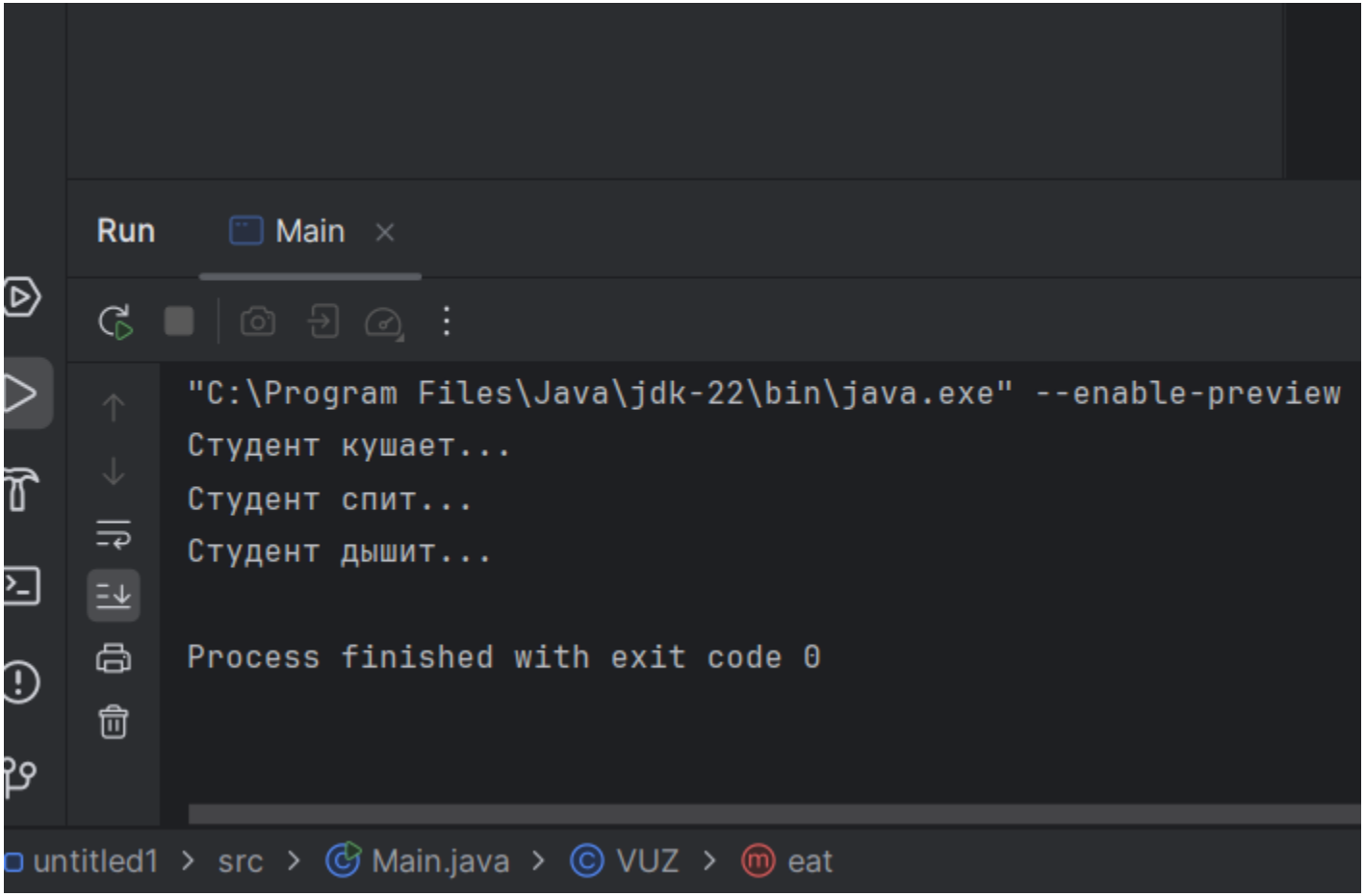
```java
// Объявление интерфейса Студент. В интерфейсе мы реализовываем поведение объектов класса
interface Student {
    // Абстрактный метод (должен быть реализован в классе)
    void eat();

    // Метод по умолчанию (с реализацией)
    default void sleep() {
        System.out.println("Студент спит...");
    }

    // Статический метод (можно вызывать без создания объекта)
    static void breathe() {
        System.out.println("Студент дышит...");
    }
}
```

```java
// Класс ВУЗ реализует интерфейс Студент
class VUZ implements Student {
    // Реализация абстрактного метода eat
    @Override // Переопределяем созданный метод за счет аннотации @Override
    public void eat() {
        System.out.println("Студент кушает...");
    }
}

public class Main {
    public static void main(String[] args) {
        VUZ student = new VUZ();
        student.eat();          // Вывод метода eat
        student.sleep();        // Вывод метода sleep
        Student.breathe();      // Вывод метода breathe
    }
}
```

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – интерфейсы

Результат расширения класса за счет интерфейса



Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – массивы

Массивы в Java — это структура данных, которая позволяет хранить фиксированное количество элементов одного типа. Они предоставляют удобный способ работы с группами данных и поддерживают быстрый доступ к элементам по индексу.

Основные характеристики массивов в Java:

1.Однородность данных: Все элементы массива должны быть одного типа (например, все элементы — целые числа, строки, объекты одного класса и т.д.).

2.Фиксированный размер: Размер массива задается при его создании и не может быть изменен. При необходимости изменения размера нужно создавать новый массив.

3.Индексация: Элементы массива индексируются, начиная с 0. Это означает, что первый элемент имеет индекс 0, второй — 1, и так далее.

4.Прямой доступ: Массивы обеспечивают быстрый доступ к элементам, так как элементы хранятся в непрерывных областях памяти.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – массивы

```
public class Main {
    public static void main(String[] args) {
        // Создание и инициализация массива
        int[] numbers = {10, 20, 30, 40, 50};

        // Доступ к элементам массива
        System.out.println("Первый элемент: " + numbers[0]);  // Вывод: Первый элемент: 10
        System.out.println("Размер массива: " + numbers.length);  // Вывод: Размер массива: 5

        // Изменение значения элемента массива
        numbers[2] = 99;
        System.out.println("Измененный элемент: " + numbers[2]);  // Вывод: Измененный элемент: 99

        // Перебор элементов массива с помощью цикла
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Элемент " + i + ": " + numbers[i]); // Вывод элементов массива на новой
строке
        }
    }
}
```

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – одномерные массивы (ввод с клавиатуры)

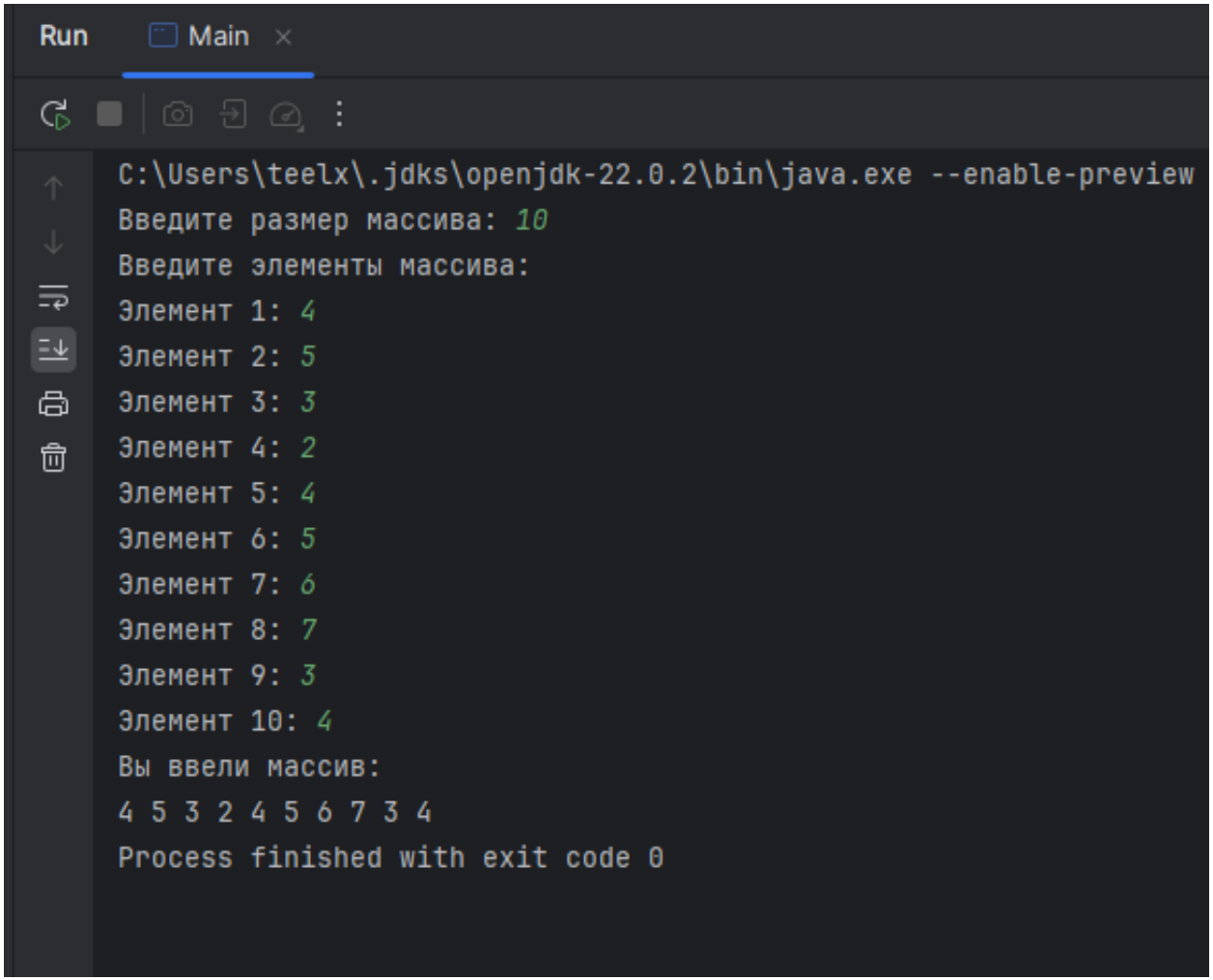
```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // Создание объекта Scanner для ввода с клавиатуры

        System.out.print("Введите размер массива: ");
        int size = scanner.nextInt(); // Считывание размера массива

        int[] array = new int[size]; // Создание пустого массива указанного размера

        System.out.println("Введите элементы массива:");
```

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – одномерные массивы (ввод с клавиатуры)

```
        // int i = 0
        // Эта часть выполняется один раз перед началом цикла.
        // В данном случае создается переменная i и инициализируется значением 0.
        // Эта переменная обычно используется как счетчик для отслеживания текущей итерации цикла.
        // i < size
        // Это условие проверяется перед каждой итерацией цикла.
        // Если условие истинно (возвращает true), выполняется тело цикла.
        // В противном случае цикл завершает выполнение.
        // В данном случае цикл будет продолжать выполняться, пока значение i меньше size.
        // Это условие позволяет ограничить количество итераций.
        // i++ - счетчик.
        // Эта операция выполняется после каждого выполнения тела цикла.
        // В данном случае i++ увеличивает значение переменной i на единицу после каждой итерации (инкремент).
        // Это важно для того, чтобы цикл постепенно приближался к своему завершению.
        for (int i = 0; i < size; i++) { // Цикл для ввода элементов массива
            System.out.print("Элемент " + (i + 1) + ": ");
            array[i] = scanner.nextInt(); // Считывание каждого элемента массива
        }

        // Вывод элементов массива
        System.out.println("Вы ввели массив:");
        for (int i = 0; i < size; i++) {
            System.out.print(STR."\{array[i]} ");
        }
    }
}
```

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – одномерные массивы (ввод с клавиатуры). Результат



ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

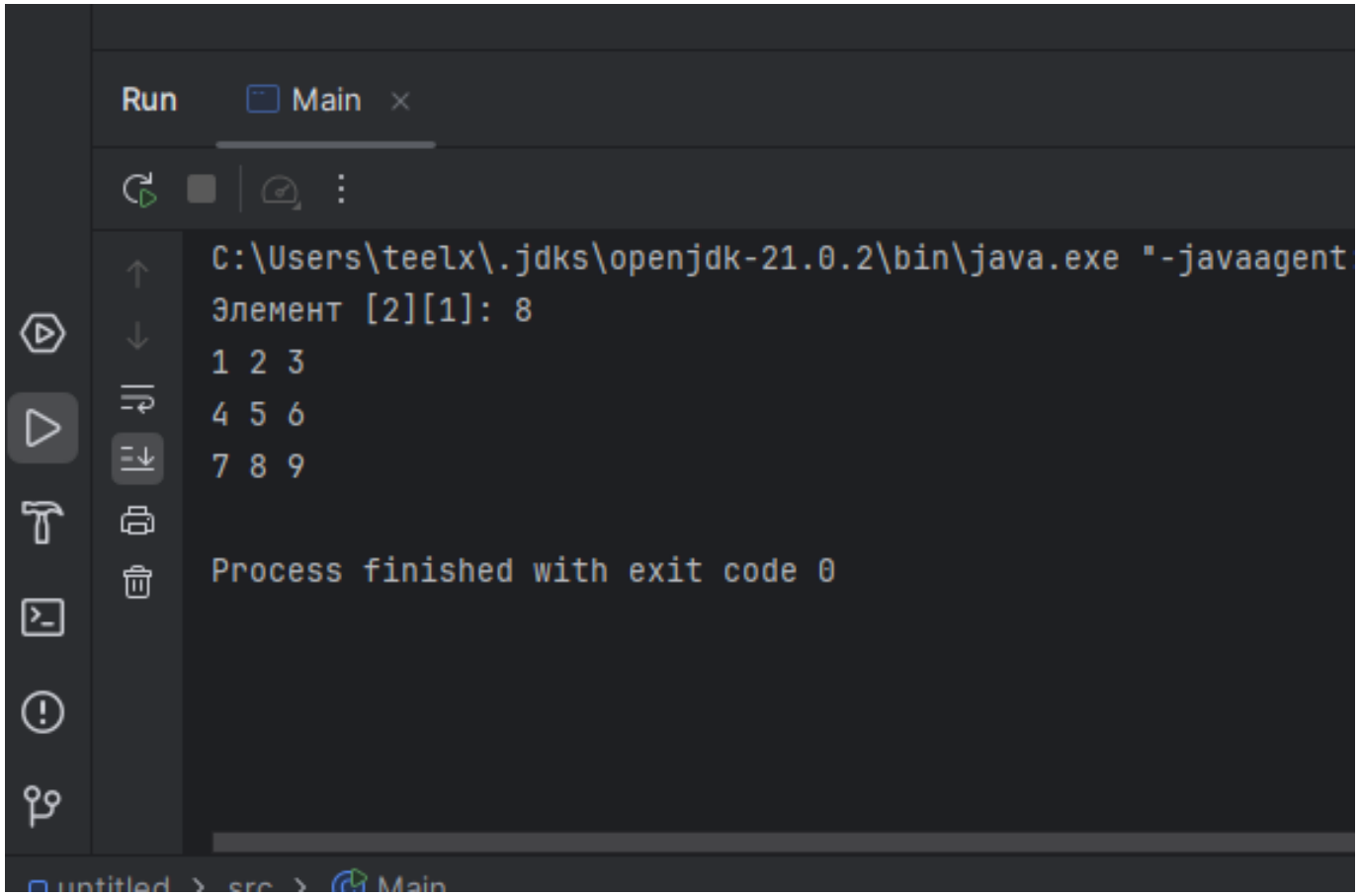
Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – двумерные массивы

Двумерный массив — это массив массивов, который можно представить как таблицу (матрицу) с строками и столбцами. Двумерные массивы используются для хранения данных в виде сетки/таблицы.

```
public class Main {
    public static void main(String[] args) {
        // Создание двумерного массива
        int[][] matrix = {
                {1, 2, 3},
                {4, 5, 6},
                {7, 8, 9}
        };

        // Доступ к элементам двумерного массива
        System.out.println("Элемент [2][1]: " + matrix[2][1]); // Вывод: Элемент [2][1]: 5

        // Перебор двумерного массива с помощью вложенных циклов
        for (int i = 0; i < matrix.length; i++) { // Проход по строкам
            for (int j = 0; j < matrix[i].length; j++) { // Проход по столбцам
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Ссылочные типы данных – двумерные массивы

Результат вывода двумерного массива:

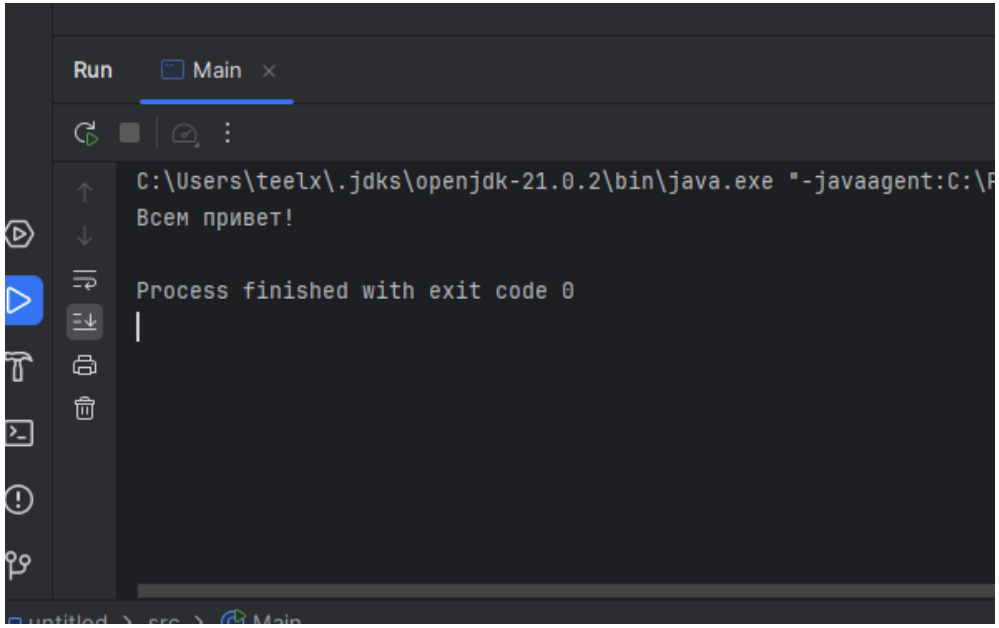


ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Строки

Строки в Java — это объекты класса String, являющиеся ссылочным типом данных.

```
public class Main {
    public static void main(String[] args) {
        String message = "Всем привет!"; // message — это ссылка (ссылочная переменная) на строковый объект.
        System.out.println(message);
    }
}
```

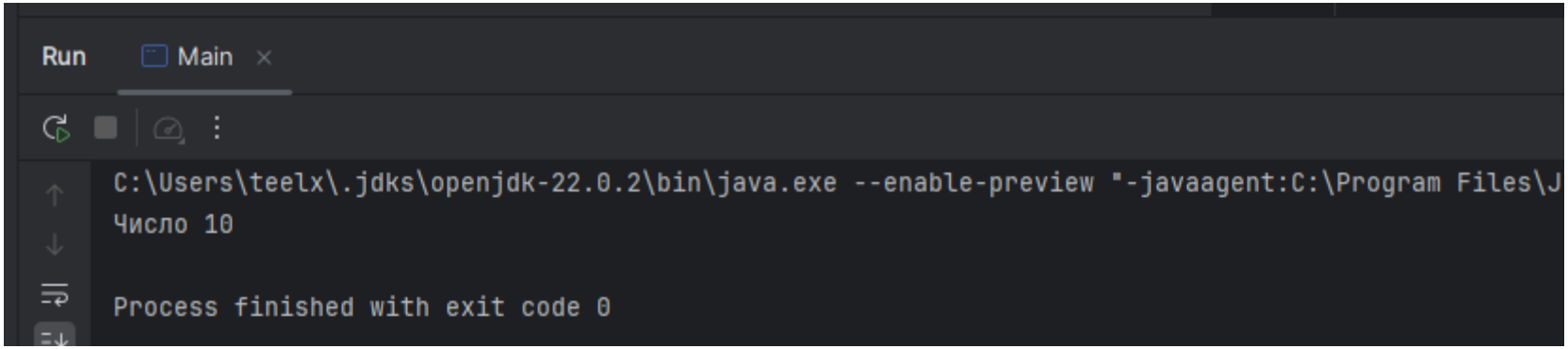


ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Оберточные классы (обертки)

Java содержит классы-обертки для всех примитивных типов данных (Integer, Double, Boolean и т.д.), которые позволяют работать с примитивами как с объектами.

```
public class Main {
    public static void main(String[] args) {
        Integer number = 10; // number — это ссылка (ссылочная переменная) на объект класса Integer.
        System.out.println("Число " + number);

    }
}
```



Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Другие особенности ссылочных типов данных

1. Null значение. Ссылочные типы могут иметь значение null, что означает отсутствие ссылки на какой-либо объект. Примитивные – не могут.
2. Сравнение. Сравнение ссылочных типов через оператор == проверяет равенство ссылок, а не содержимого объектов. Для сравнения содержимого объектов используется метод .equals().
3. Автоматическая сборка мусора. Память, занятую объектами, на которые больше нет ссылок, автоматически освобождается сборщиком мусора (Garbage Collector).
4. Передача по ссылке. В методах Java ссылочные типы передаются по ссылке, что позволяет изменять состояние объекта, на который ссылается переменная.
Ссылочные типы данных позволяют работать с более сложными структурами данных и объектно-ориентированными концепциями в Java, такими как наследование, полиморфизм и инкапсуляция.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Циклы

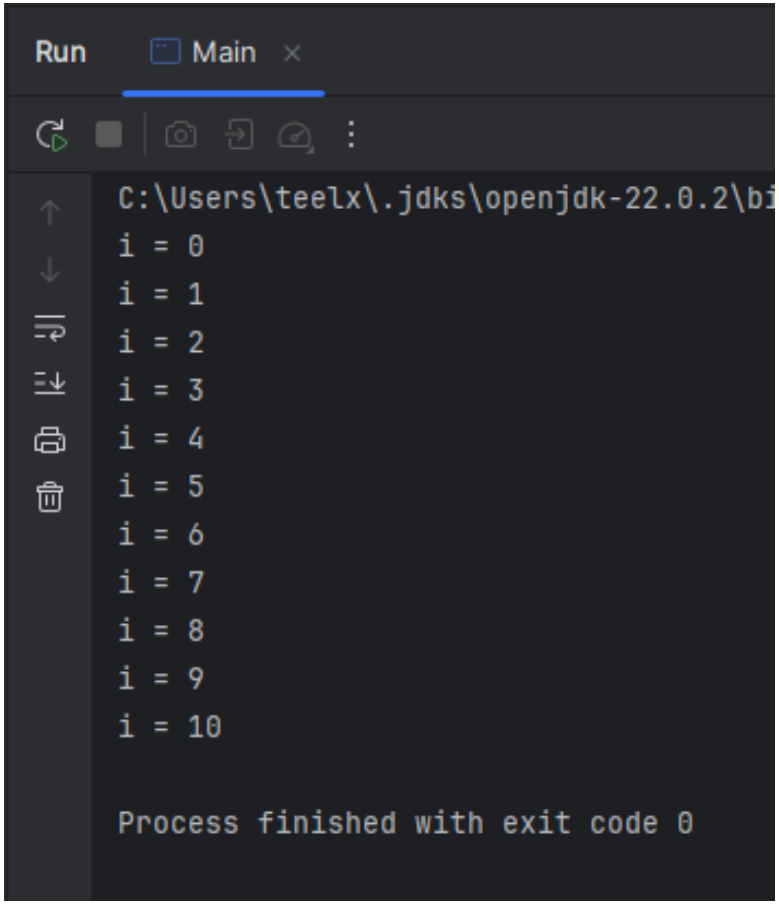
ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Цикл for

Цикл **for** используется, когда известно, сколько раз нужно повторить блок кода. Является одним из самых удобных и часто используемых циклов в Java.

```
public class Main {
    public static void main(String[] args) {
        for (инициализация; условие; обновление) {
            // блок кода, который выполняется на каждой
итерации
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Цикл for, который выводит числа от 0 до 10
        for (int i = 0; i < 11; i++) {
            System.out.println("i = " + i);
        }
    }
}
```



ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Цикл while

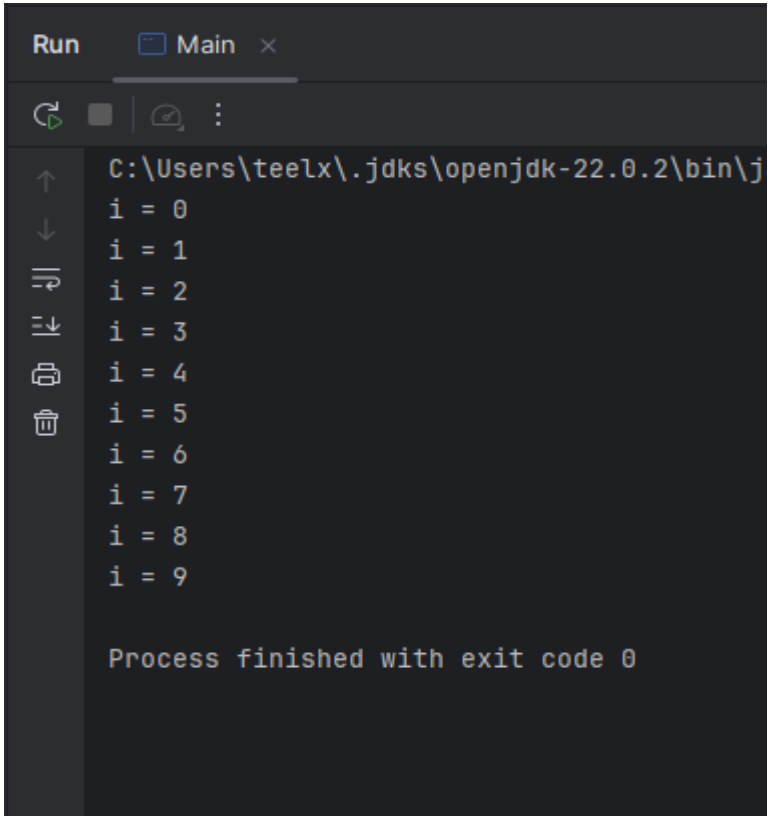
Цикл **while** используется, когда количество повторений заранее неизвестно, и цикл должен выполняться до тех пор, пока условие истинно.

1. Условие проверяется перед каждой итерацией. Если оно истинно, выполняется тело цикла. Если ложное, цикл завершается.

2. Тело цикла выполняется только если условие истинно.

```
public class Main {
    public static void main(String[] args) {
        while (условие) {
            // блок кода, который выполняется, пока условие истинно
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        int i = 0;
// Цикл while, который выводит числа от 0 до 9
        while (i < 10) {
            System.out.println("i = " + i);
            i++;
        }

    }
}
```



Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Цикл while

Пример со строкой

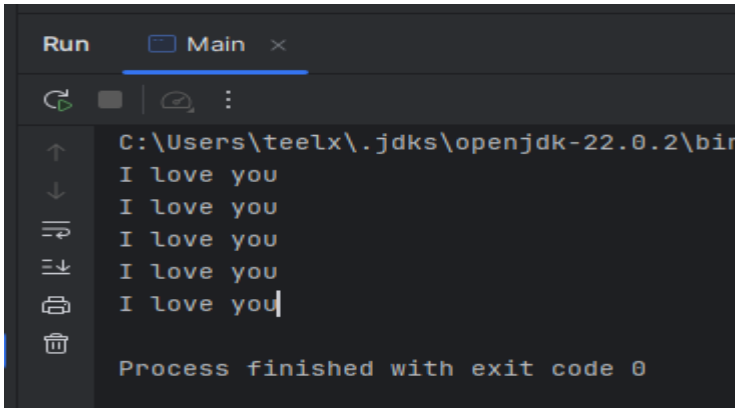
```
public class Main {
    public static void main(String[] args) {
        String test = "I love you";
        int i = 0;
// Цикл while, который выводит символы из строки
        while (i < test.length()) {
            System.out.println(test.charAt(i));
            i++;
        }

    }
}
```

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Цикл do-while

Цикл **do-while** гарантирует выполнение тела цикла хотя бы один раз, даже если условие изначально ложно.

Условие истинно:

```
public class Main {
    public static void main(String[] args) {
        String test = "I love you";
        int i = 0;
        do {
            System.out.println(test);
            i++;
            //Выводим нашу строку 5 раз
        } while (i < 5);

    }
}
```



Условие ложно:

```
public class Main {
    public static void main(String[] args) {
        String test = "I love you";
        int i = 0;
        do {
            System.out.println(test);
            i++;
            //Выводим нашу строку 1 раз, так как условие ложно
        } while (i > 5);

    }
}
```

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Вложенные циклы

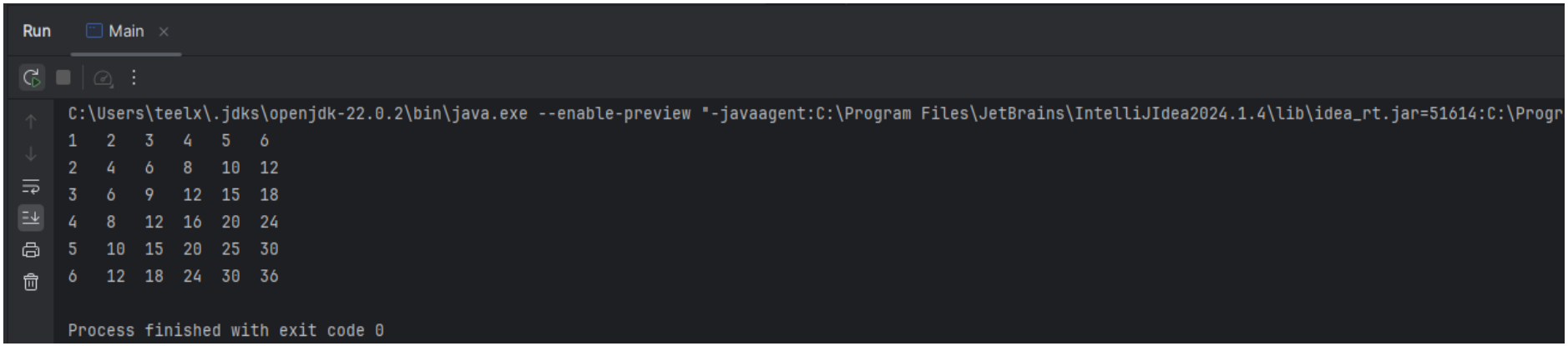
Вложенные циклы в Java — это циклы, которые находятся внутри других циклов. Они позволяют выполнять многократные итерации, особенно полезны при работе с многомерными структурами данных, например, при обработке двумерных массивов или создании сложных таблиц.

Основная идея вложенных циклов заключается в том, что внутренний цикл завершает все свои итерации для каждой итерации внешнего цикла. Это означает, что внешний цикл управляет количеством повторений внутреннего цикла.

```
public class Main {
    public static void main(String[] args) {
        for (инициализация_внешнего цикла; условие_внешнего цикла; обновление_внешнего цикла) {
            for (инициализация_внутреннего цикла; условие_внутреннего цикла; обновление_внутреннего цикла) {
                // Тело внутреннего цикла
            }
            // Тело внешнего цикла
        }
    }
}
```

Лекция №1. Обзор платформы и языка программирования Java. Основные языковые конструкции Java. Вложенные циклы

```
public class Main {
    public static void main(String[] args) {
// Внешний цикл отвечает за строки
        for (int i = 1; i <= 15; i++) {
            // Внутренний цикл отвечает за столбцы
            for (int j = 1; j <= 15; j++) {
                // Вывод произведения текущих значений i и j
                System.out.print(i * j + "\t"); // \t - пробел (табуляция) после каждого значения
            }
            // Переход на новую строку после завершения внутреннего цикла
            System.out.println();
        }
    }
}
```



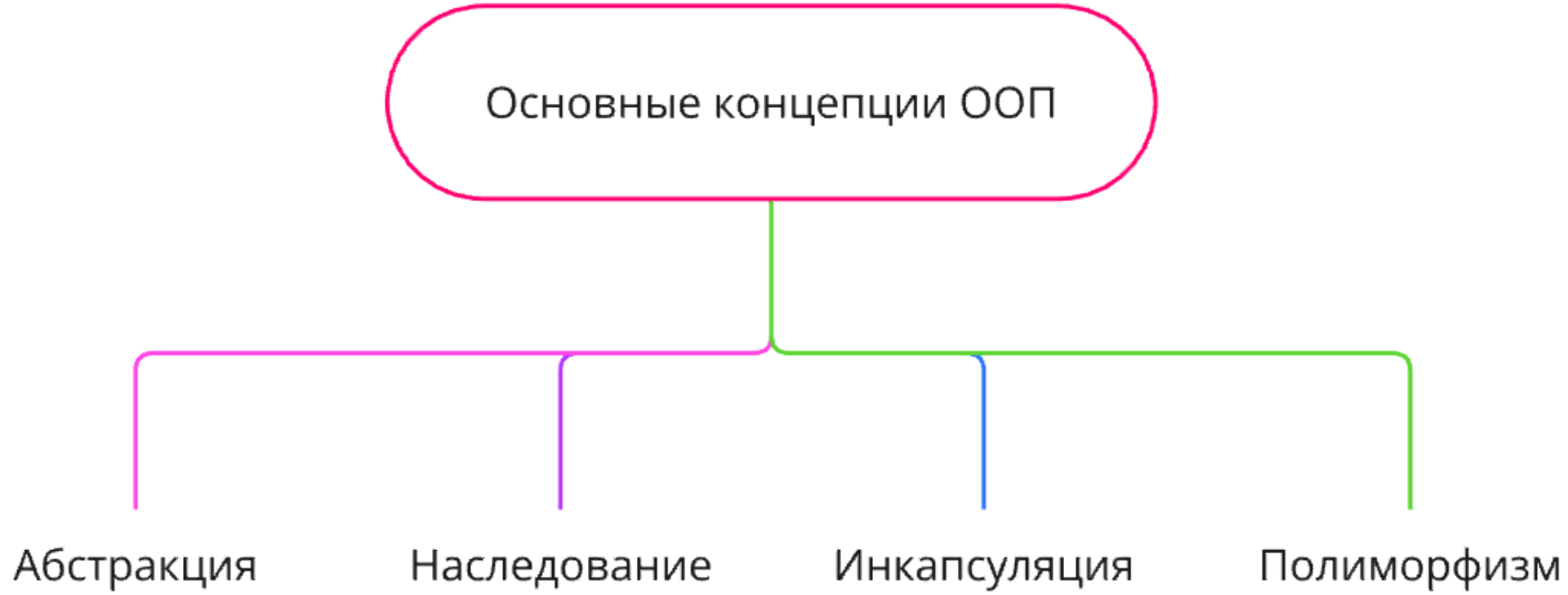
Лекция №2. Базовые принципы ООП и работа с базами данных

Объектно-ориентированное программирование (ООП) — это парадигма программирования, используемая для создания программного обеспечения, построенного на концепции объектов. В Java ООП является основой языка, которая позволяет разрабатывать более структурированные, понятные и легко поддерживаемые приложения. Основные элементы ООП — это классы и объекты, которые позволяют моделировать реальные сущности в коде.

В рамках ООП:

1. **Класс** — это шаблон или "чертеж" для создания объектов. Класс описывает, какие свойства (поля) и методы (поведение) будут у объектов, созданных на его основе.
2. **Объект** — это экземпляр класса, содержащий реальные данные и поведение, описанные в классе.

Лекция №2. Базовые принципы ООП и работа с базами данных



Лекция №2. Базовые принципы ООП и работа с базами данных. Абстрактные классы

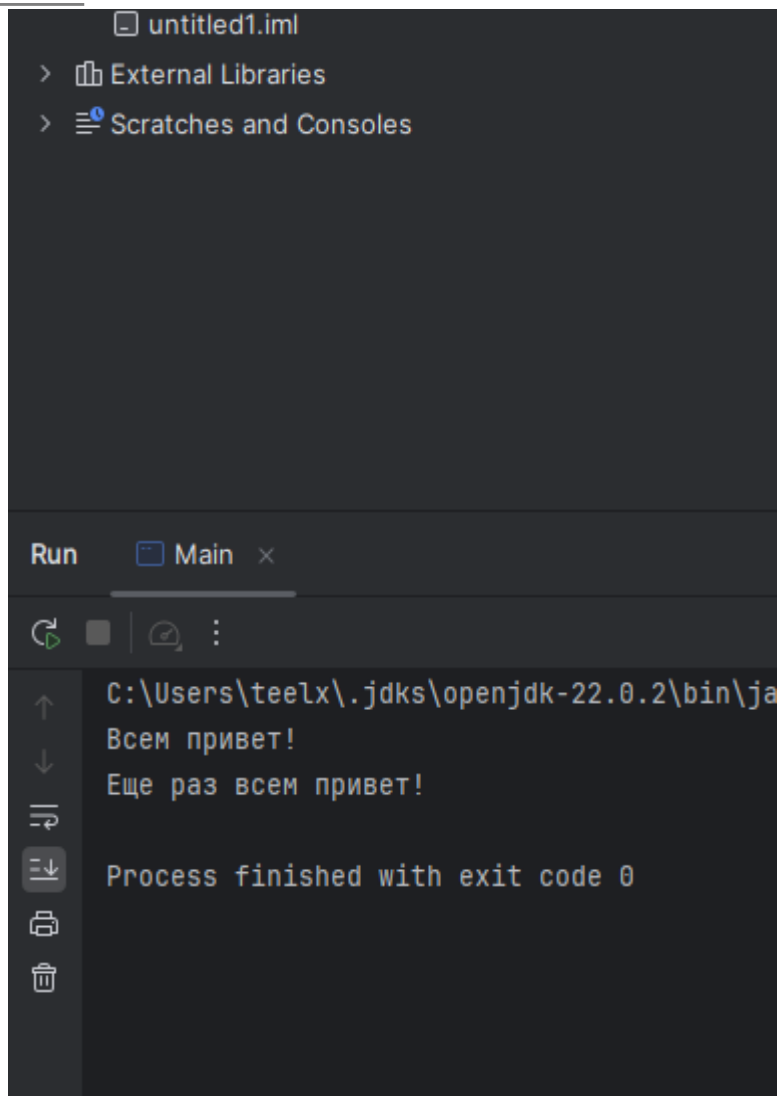
Абстракция заключается в том, чтобы скрывать сложные детали реализации и предоставлять пользователю только необходимые аспекты объекта. Это помогает сосредоточиться на решении конкретных задач, не вникая в детали реализации.

Например, мы можем реализовать метод подключения к БД и использовать его далее путем вызова из класса.

Абстракция достигается с помощью абстрактных классов и интерфейсов, которые описывают, какие действия можно совершать с объектом, но не как они реализованы.

Лекция №2. Базовые принципы ООП и работа с базами данных. Абстрактные классы

```java
// Абстрактный класс Test
abstract class Test {
    abstract void draw(); // Абстрактный метод без реализации
}

// Переопределяем метод draw
class Test1 extends Test {
    @Override
    void draw() {
        System.out.println("Всем привет!");
    }
}

// Еще раз переопределяем метод draw. Таким образом, его можно переопределять сколько угодно раз
class Test2 extends Test {
    @Override
    void draw() {
        System.out.println("Еще раз всем привет!");
    }
}

public class Main {
    public static void main(String[] args) {
        Test one = new Test1();
        one.draw(); // Вывод переопределенного метода из класса Test1
        Test one1 = new Test2();
        one1.draw(); // Вывод переопределенного метода из класса Test2
    }
}
```



Лекция №2. Базовые принципы ООП и работа с базами данных. Наследование

Наследование — один из ключевых принципов объектно-ориентированного программирования (ООП), позволяющий создать новый класс на основе функциональности существующего. Это помогает избегать дублирования кода, обеспечивая повторное использование, упрощает разработку и улучшает структуру программы.

Основные моменты наследования в Java:

1. **Базовый класс (суперкласс, родительский класс)** — это класс, который передает свои свойства и методы другому классу.
2. **Подкласс (сабкласс)** — это класс, который наследует свойства и методы базового класса и может добавлять свои собственные.
3. **Ключевое слово extends** используется для расширения подкласса, т.е. для создания наследования в Java.
4. **Доступ к унаследованным членам** можно получить напрямую, если они не имеют модификатора доступа private.
5. **Ключевое слово super** используется для вызова методов и конструкторов суперкласса.
6. **Наследование от final класса.** Невозможно наследовать класс, помеченный модификатором класса final.

Лекция №2. Базовые принципы ООП и работа с базами данных. Наследование

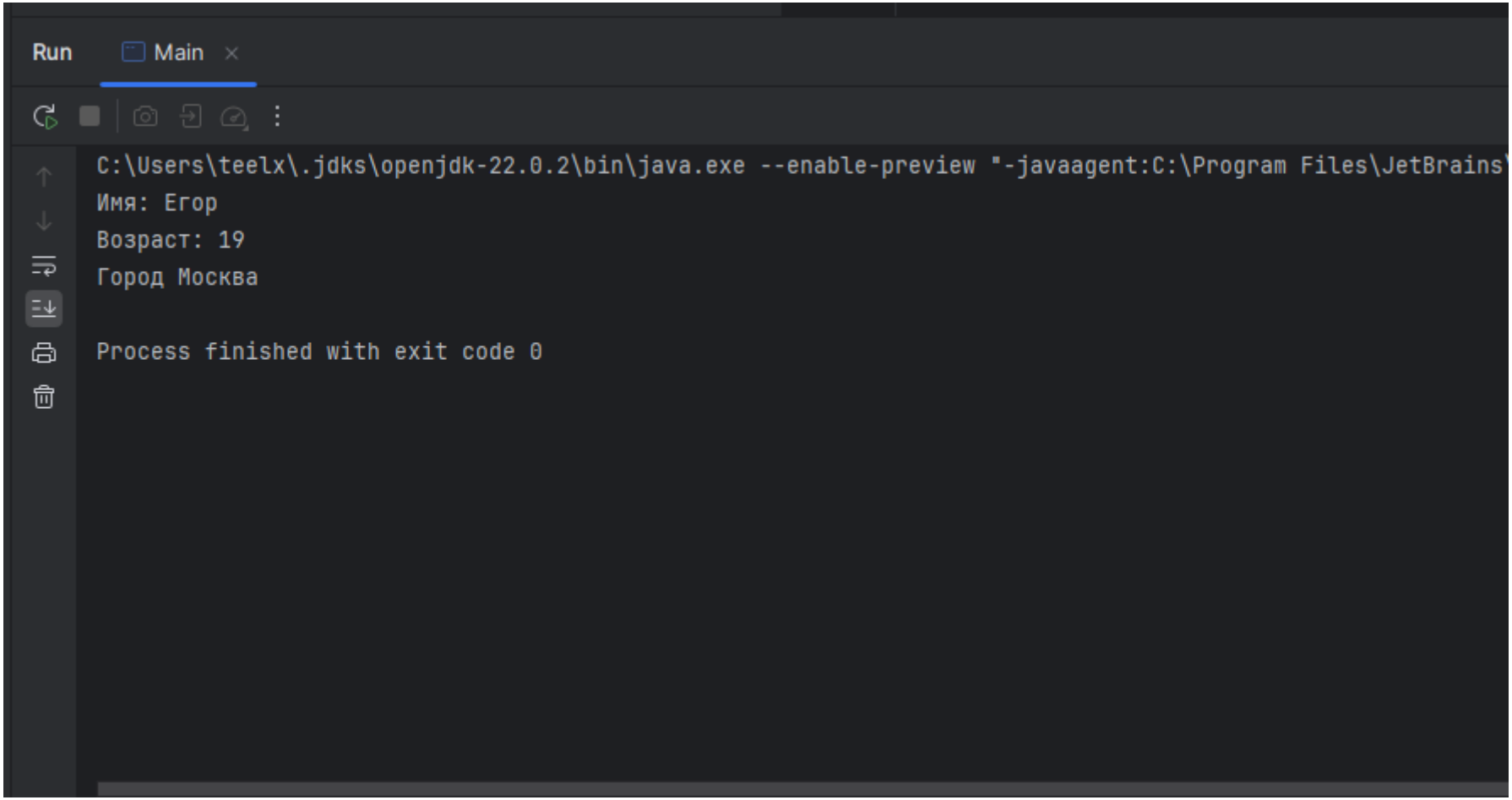
```java
// Базовый класс Student
class Student {
    // Поля (переменные) базового класса
    String name;
    int age;

    // Конструктор базового класса, необходим для работы ключевого слова "Super"
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Метод базового класса
    void displayInfo() {
        System.out.println("Имя: " + name);
        System.out.println("Возраст: " + age);
    }
}
```

Лекция №2. Базовые принципы ООП и работа с базами данных. Наследование

```
// Подкласс Prepod, наследующий родительский класс Student
class Prepod extends Student {
    // Дополнительное поле подкласса
    String city;

    // Конструктор подкласса, который вызывает конструктор родительского класса
    Prepod(String name, int age, String city) {
        super(name, age); // Вызов конструктора родительского класса Student
        this.city = city;
    }

    @Override
    // Метод подкласса, который добавляет дополнительное поведение
    void displayInfo() {
        // Вызов метода базового класса Student для отображения общих свойств
        super.displayInfo();
        // Отображение дополнительного свойства подкласса
        System.out.println("Город " + city);
    }
}

// Основной класс для запуска программы
public class Main {
    public static void main(String[] args) {
        // Создание объекта подкласса Prepod
        Prepod student = new Prepod("Егор", 19, "Москва");
        // Вызов метода displayInfo, который показывает все свойства
        student.displayInfo();
    }
}
```

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

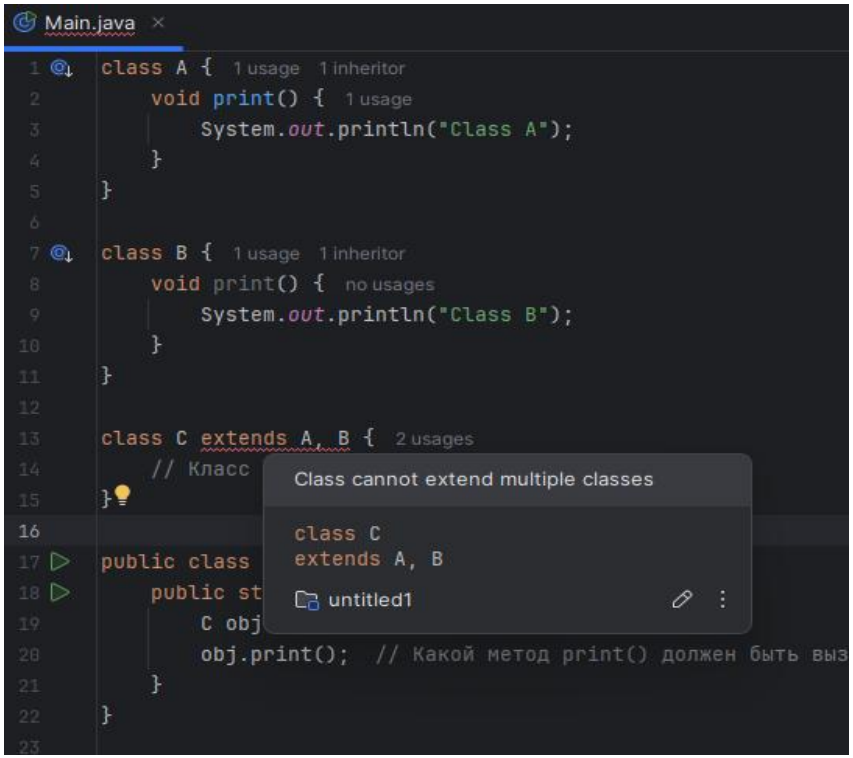
Лекция №2. Базовые принципы ООП и работа с базами данных. Результат



Лекция №2. Базовые принципы ООП и работа с базами данных. Множественное наследование

В Java множественное наследование классов не поддерживается. Это означает, что подкласс не может наследовать более одного класса. Это ограничение введено для избегания неоднозначностей, которые могут возникнуть при наследовании одинаковых методов или полей от нескольких родительских классов. Например, если бы класс мог наследовать два родительских класса, и оба родительских класса имели бы метод с одинаковым названием, Java не смогла бы однозначно определить, какой именно метод нужно использовать. Можно одновременно наследовать функциональность от интерфейсов и классов.

```
class A {
    void print() {
        System.out.println("Class A");
    }
}

class B {
    void print() {
        System.out.println("Class B");
    }
}

class C extends A, B {
    // Класс C наследует классы A и B
}

public class Main {
    public static void main(String[] args) {
        C obj = new C();
        obj.print();  // Какой метод print() должен быть вызван? Из класса A или из класса B?
    }
}
```



Лекция №2. Базовые принципы ООП и работа с базами данных. Инкапсуляция

Инкапсуляция — один из ключевых принципов объектно-ориентированного программирования (ООП), наряду с наследованием, полиморфизмом и абстракцией. Инкапсуляция заключается в объединении данных (переменных) и методов, работающих с этими данными, в одном классе, а также в ограничении доступа к этим данным извне, чтобы защитить их от нежелательных изменений.

Основные аспекты инкапсуляции:

1. **Сокрытие данных**. Основная цель инкапсуляции — скрыть внутреннее состояние объекта и предоставить доступ к данным только через методы класса (геттеры и сеттеры).
2. **Контроль над доступом**. Использование модификаторов доступа (private, protected, public) для контроля видимости полей и методов класса.
3. **Повышение безопасности**. Защищает данные от некорректных или случайных изменений, обеспечивая целостность данных.
4. **Упрощение изменений**. Позволяет изменять реализацию класса без необходимости менять код, который использует этот класс.

Реализация инкапсуляции в Java.

Для реализации инкапсуляции в Java поля класса обычно объявляются как private, а доступ к ним обеспечивается через публичные методы get и set.

Лекция №2. Базовые принципы ООП и работа с базами данных. Инкапсуляция

```
import lombok.Getter;

// Класс Account с инкапсуляцией данных
@Getter
class Account {
    // Геттер для получения текущего баланса
    // Приватное поле класса, доступное только внутри этого класса
    private double balance;

    // Конструктор для инициализации счета
    public Account(double initialBalance) {
        setBalance(initialBalance); // Используем сеттер для инициализации с проверкой
    }

    // Сеттер для изменения значения баланса с проверкой на отрицательное значение
    public void setBalance(double balance) {
        if (balance >= 0) {
            this.balance = balance;
        } else {
            System.out.println("Баланс не может быть отрицательным. Установка значения по умолчанию: 0");
            this.balance = 0;
        }
    }
```

Лекция №2. Базовые принципы ООП и работа с базами данных. Инкапсуляция

```
    // Метод для пополнения баланса
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Счет пополнен на: " + amount);
        } else {
            System.out.println("Сумма пополнения должна быть положительной");
        }
    }

    // Метод для снятия средств с баланса
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Снято со счета: " + amount);
        } else {
            System.out.println("Недостаточно средств или некорректная сумма");
        }
    }
}
```

Лекция №2. Базовые принципы ООП и работа с базами данных. Инкапсуляция

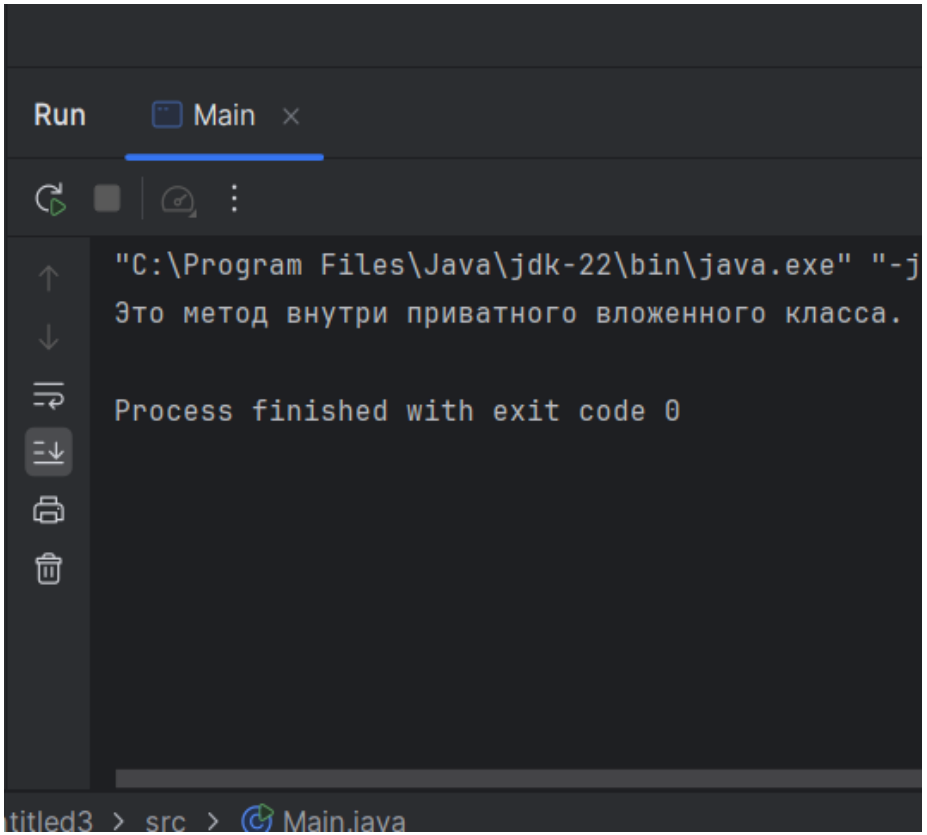
```
// Основной класс для тестирования инкапсуляции
public class Main {
    public static void main(String[] args) {
        // Создаем объект класса Account с начальным балансом
        Account myAccount = new Account(100.0);
        System.out.println("Текущий баланс: " + myAccount.getBalance());

        // Пополняем счет
        myAccount.deposit(50.0);
        System.out.println("Текущий баланс: " + myAccount.getBalance());

        // Пытаемся снять сумму
        myAccount.withdraw(30.0);
        System.out.println("Текущий баланс: " + myAccount.getBalance());

        // Пытаемся установить отрицательный баланс
        myAccount.setBalance(-10.0);
        System.out.println("Текущий баланс: " + myAccount.getBalance());
    }
}
```

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №2. Базовые принципы ООП и работа с базами данных. Инкапсуляция – модификатор доступа private в сочетании с классом

В Java использование модификатора доступа private с классом возможно, но только для **вложенных классов** (inner classes). Класс, объявленный с модификатором private, может быть доступен только внутри другого класса, в котором он объявлен. Такой класс не может быть использован вне его внешнего класса. Т.е. вложенный класс.

```
class OuterClass { // по умолчанию модификатор доступа "Protected"
    // Приватный вложенный класс
    private class InnerClass {
        void display() {
            System.out.println("Это метод внутри приватного вложенного класса.");
        }
    }

    // Метод, который создает объект вложенного класса и вызывает его метод
    void showInner() {
        InnerClass inner = new InnerClass();
        inner.display();
    }
}

public class Main {
    public static void main(String[] args) {
        // Создаем объект внешнего класса
        OuterClass outer = new OuterClass();

        // Вызов метода, который демонстрирует использование приватного
вложенного класса
        outer.showInner();
    }
}
```



Лекция №2. Базовые принципы ООП и работа с базами данных. Полиморфизм

Полиморфизм — один из ключевых принципов объектно-ориентированного программирования (ООП), наряду с инкапсуляцией, наследованием и абстракцией. В Java полиморфизм позволяет использовать единый интерфейс для объектов разных типов, что делает код более гибким и расширяемым.

Основные виды полиморфизма:

1.Компиляционный полиморфизм (Ad-hoc полиморфизм):

1. **Перегрузка методов (Method Overloading)**: позволяет иметь несколько методов с одинаковым именем, но разными параметрами.
2. **Перегрузка операторов**: в Java не поддерживается напрямую, но доступна через перегрузку методов.

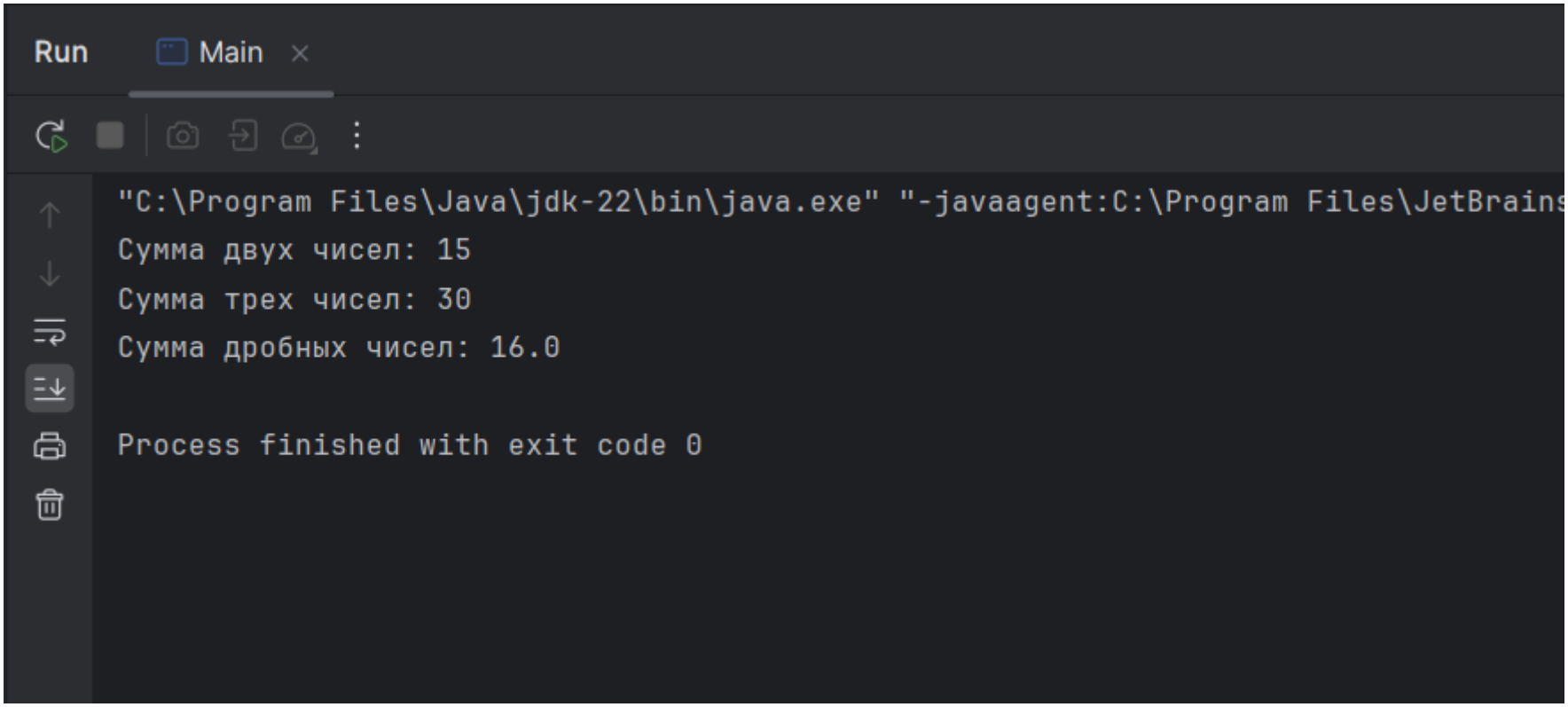
2.Полиморфизм во время выполнения (Runtime полиморфизм):

1. **Переопределение методов (Method Overriding)**: дочерний класс переопределяет метод родительского класса, предоставляя свою реализацию.
2. **Использование интерфейсов и абстрактных классов**: позволяет объектам разных классов обрабатывать запросы одинаково.

Лекция №2. Базовые принципы ООП и работа с базами данных. Полиморфизм – Перегрузка методов

```
class Calculator {
    // Метод для сложения двух целых чисел
    public int add(int a, int b) {
        return a + b;
    }

    // Перегруженный метод для сложения трех целых чисел
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Перегруженный метод для сложения двух чисел с плавающей точкой
    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Сумма двух чисел: " + calc.add(5, 10)); // Вызов метода add(int, int)
        System.out.println("Сумма трех чисел: " + calc.add(5, 10, 15)); // Вызов метода add(int, int, int)
        System.out.println("Сумма дробных чисел: " + calc.add(5.5, 10.5)); // Вызов метода add(double, double)
    }
}
```

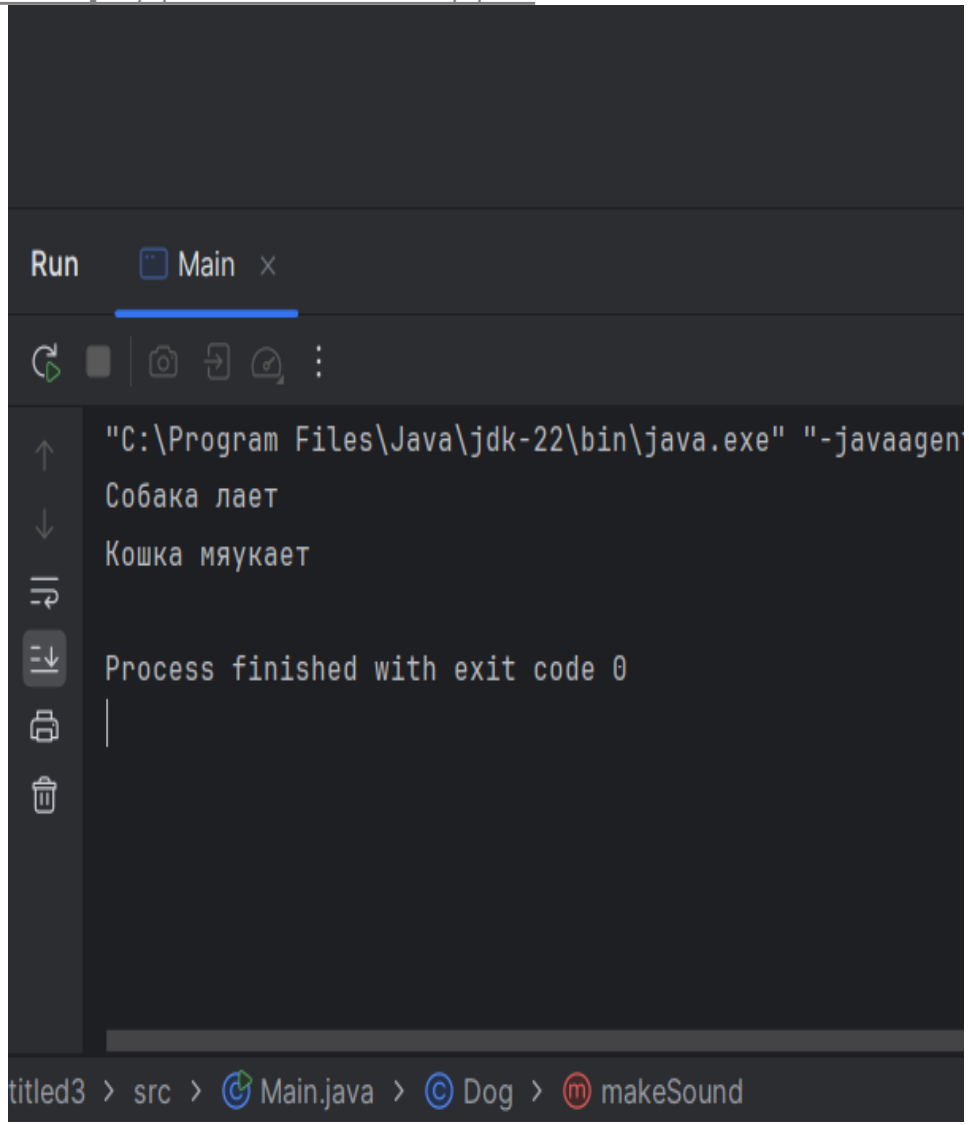
Лекция №2. Базовые принципы ООП и работа с базами данных. Полиморфизм – Перегрузка методов



ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Лекция №2. Базовые принципы ООП и работа с базами данных. Полиморфизм – Переопределение методов

```java
// Родительский класс Animal
class Animal {
    // Метод, который будет переопределяться
    public void makeSound() {
        System.out.println("Животное издает звук");
    }
}

// Класс Dog, наследующий Animal и переопределяющий метод makeSound
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Собака лает");
    }
}

// Класс Cat, наследующий Animal и переопределяющий метод makeSound
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Кошка мяукает");
    }
}

public class Main {
    public static void main(String[] args) {
        // Полиморфизм во время выполнения
        Animal myDog = new Dog(); // Объект типа Dog, но ссылается на Animal
        Animal myCat = new Cat(); // Объект типа Cat, но ссылается на Animal

        myDog.makeSound(); // Вызовет переопределенный метод Dog
        myCat.makeSound(); // Вызовет переопределенный метод Cat
    }
}
```



Лекция №2. Базовые принципы ООП и работа с базами данных.

Заключение

1. Инкапсуляция

Суть: объединяет данные и методы, работающие с этими данными, в единый класс и скрывает детали реализации от внешнего мира.

Вывод: инкапсуляция позволяет защитить внутреннее состояние объектов от некорректного использования, что делает программы более безопасными и управляемыми.

2. Наследование

Суть: позволяет создавать новые классы на основе существующих, унаследовав их свойства и методы.

Вывод: наследование способствует повторному использованию кода и упрощает расширение функциональности программы, что снижает количество дублирующегося кода и упрощает сопровождение.

3. Полиморфизм

Суть: позволяет использовать объекты разных типов через единый интерфейс, что упрощает взаимодействие между объектами.

Вывод: полиморфизм делает системы более гибкими и расширяемыми, позволяет легко добавлять новые функции без изменения существующего кода, поддерживая принцип открытости/закрытости (Open/Closed Principle).

4. Абстракция

Суть: скрывает сложность реализации, представляя только значимые для пользователя детали.

Вывод: абстракция помогает сосредоточиться на ключевых аспектах задачи, упрощает использование объектов, скрывая ненужные детали реализации.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

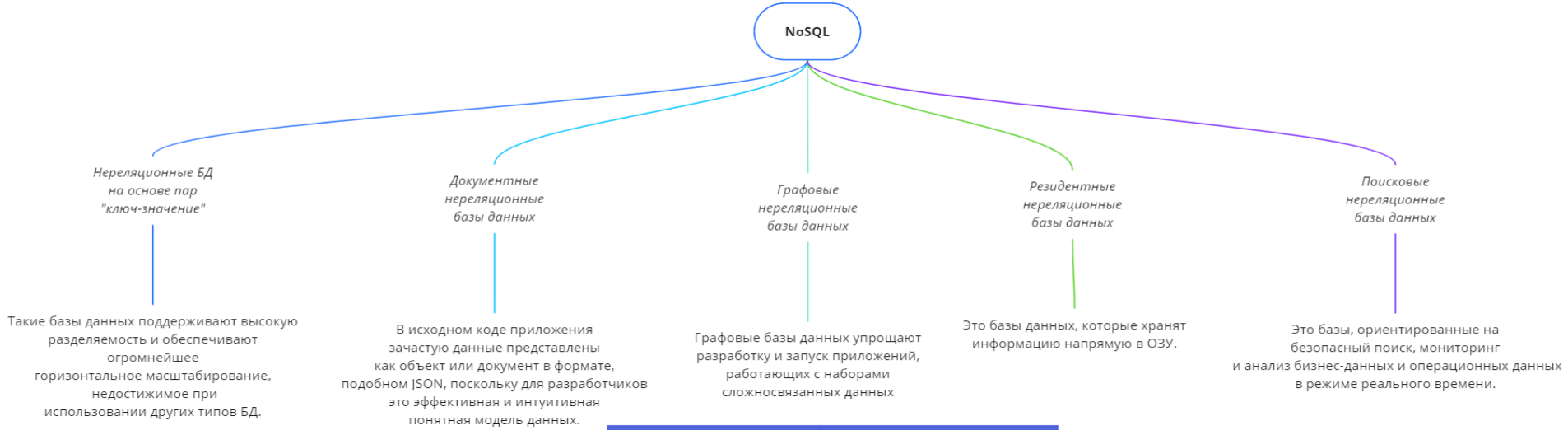
Лекция №3. Базовые принципы ООП. Работа с базами данных в Java. Основы объектно-реляционного отображения. Разница SQL и NoSQL?

Особенность (свойства, функциональность)	Реляционные базы данных (SQL)	Нереляционные базы данных (NoSQL)
Подходящие рабочие нагрузки	Предназначены для ***OLTP*** и ***OLAP***.	Такие базы данных работают по создаваемым шаблонам. Можем создавать собственные структуры данных.
Модель данных	Табличная	Ключ-значение, документы и графы.
Свойства ACID	SQL СУБД обеспечивают полный набор требований к транзакционной системе, обеспечивающий наиболее надёжную и предсказуемую её работу, иначе говоря ACID.	Базы данных NoSQL зачастую предлагают компромисс, смягчая жесткие требования свойств ACID ради более гибкой модели данных, которая допускает горизонтальное масштабирование.
Производительность	Зависит от дисковой подсистемы. Максимальная производительность требует определенной оптимизации структуры таблицы, запросов и индексов.	Зависит от: 1. Аппаратного обеспечения и пропускной способности сети сервера/кластера. 2. Задержки сети приложения, с которым работает NoSQL база данных.
Масштабирование	Масштабируются путем увеличения вычислительных возможностей аппаратного обеспечения или добавления отдельных копий для рабочих нагрузок чтения.	Базы данных NoSQL обычно поддерживают высокую разделяемость благодаря шаблонам доступа с возможностью масштабирования на основе распределенной архитектуры.
API	Запросы на запись и извлечение данных составляются на языке SQL. Эти запросы анализирует и выполняет реляционная база данных.	Объектно-ориентированные API позволяют разработчикам приложений без труда осуществлять запись и извлечение структур данных.

Лекция №3. Базовые принципы ООП.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Основные отличия в понятиях SQL и NoSQL?



SQL	NoSQL
База данных	База данных
Таблица	Коллекция
Ряд	Документ
Столбец	Поле
Первичный ключ	ObjectID
Индекс	Индекс
Представление	Представление
Вложенная таблица	Встроенный документ
Массив	Массив

Синтаксис SQL (повтор). Основные понятия

База данных – это есть хранилище хранимой нами информации. Т.е. туда мы будем добавлять данные в табличном виде. Создается база данных командой CREATE DATABASE <название таблицы>.

Таблица. Таблицы являются объектами, которые содержат все данные в базах данных. В таблицах данные логически организованы в виде строк и столбцов по аналогии с электронной таблицей. Каждая строка представляет собой уникальную запись, а каждый столбец — поле записи.

Число таблиц в базе данных ограничено только числом объектов доступных в базе данных (2 147 483 647). Стандартная определяемая пользователем таблица может содержать до 1 024 столбцов. Число строк и общий размер таблицы ограничиваются только емкостью SSD-накопителей или жестких дисков на сервере.

Можно также устанавливать свойства для таблицы и каждого столбца в таблице для управления допустимыми данными и другими свойствами. Например, можно задать ограничения на столбец, чтобы в нем не допускались значения NULL, или указать значение по умолчанию, если оно не задано. Также можно присвоить ограничения ключа на таблицу, который обеспечивает уникальность, или установить связи между таблицами.

Данные в таблице могут быть сжаты либо по строкам, либо по страницам. Сжатие данных может позволить отображать больше строк на странице.

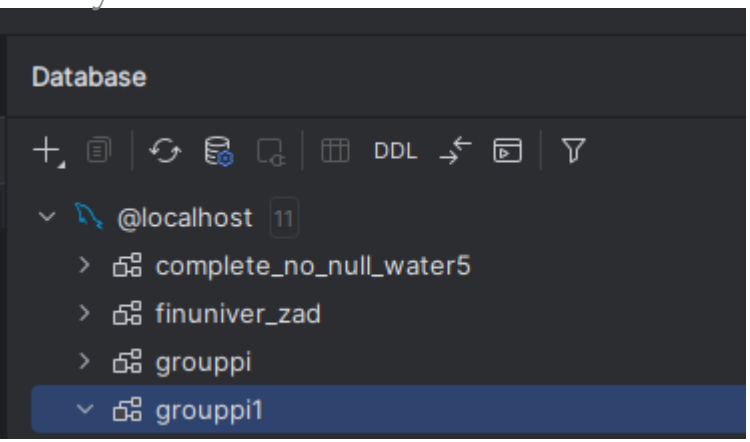
Синтаксис SQL (повтор). Практика

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

1. Создание базы данных

```
CREATE DATABASE
GroupPI1;
```

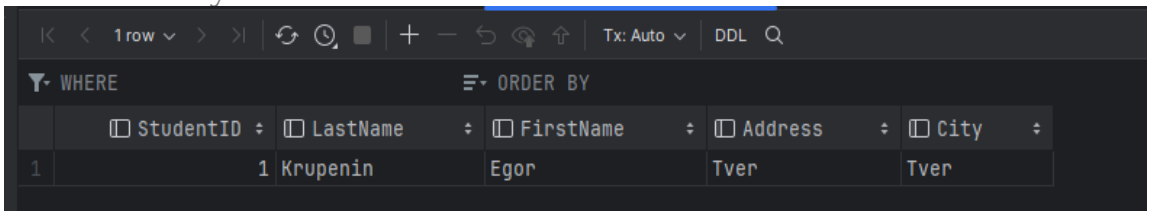
Результат:



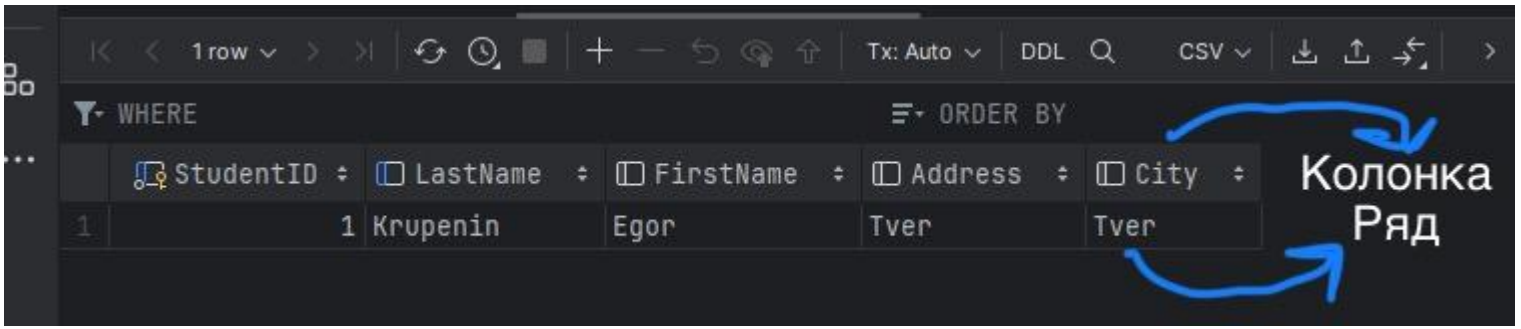
2. Создание таблицы

```
use GroupPI1;
CREATE TABLE Students (
    StudentID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
INSERT INTO Students (StudentID, LastName, FirstName, Address, City)
VALUES ('1', 'Krupenin', 'Egor', 'Tver', 'Tver')
```

Результат:



3. Ряд и столбец



Синтаксис SQL (повтор). Основные понятия

Ряд – это строка, в которую заносим наши значения. Формируется только в том случае, если добавляем какие-либо данные. Строка может быть пустой в том случае, если там будет значение «NULL».

Столбец – это колонка, в которую также заносим значения. В каждой колонке есть ряды.

Первичный ключ – это поле, позволяющее идентифицировать каждую запись. Каждая таблица может содержать только один первичный ключ. За счет первичного ключа мы можем соединять сущности друг с другом, а также присваиваем каждому ряду уникальность и целостность данных.

ALTER TABLE – это команда, служащая для изменения макета таблицы после того, как она была создана с помощью директивы (или инструкции, команды) CREATE TABLE.

Синтаксис SQL (повтор). Основные понятия

Индексы в базе данных используются для ускорения операций поиска и сортировки. Они представляют собой специальные структуры данных, которые позволяют быстро находить записи в таблицах базы данных.

Когда вы выполняете запрос к базе данных, который включает в себя условие WHERE, сервер базы данных должен найти все строки, соответствующие этому условию. Если таблица большая, это может занять много времени. Индексы помогают сократить это время, так как они хранят информацию о расположении строк в таблице, которая соответствует определенным условиям.

Кроме того, индексы могут использоваться для оптимизации операций сортировки. Когда вы выполняете запрос, который требует сортировки данных, сервер базы данных может использовать индекс для определения порядка строк перед тем, как начать их физическое перемещение.

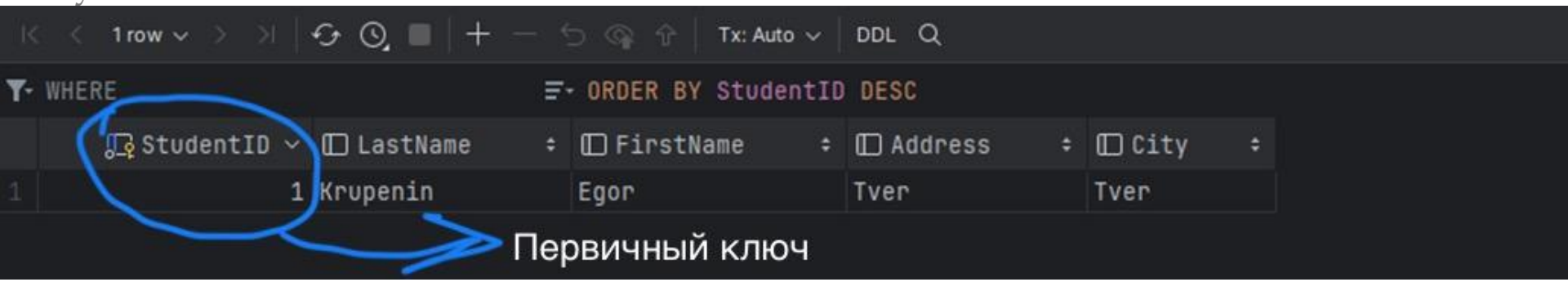
В целом, индексы являются важным инструментом для повышения производительности базы данных и ускорения выполнения запросов.

4. Первичный ключ

```
ALTER TABLE grouppi1.students
ADD PRIMARY KEY (StudentID)
```

За счет команды **ADD PRIMARY KEY** присваиваем столбцу **StudentID** уникальный ключ. Т.е. этот столбец теперь является уникальным ключом.

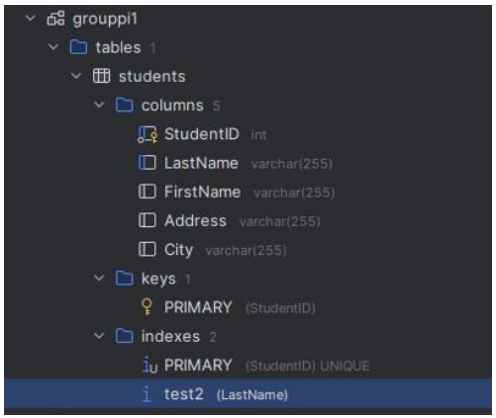
Результат:



5. Индексы

```
CREATE INDEX test2
ON grouppi1.students (LastName);
```

Результат:



Индексы – специальные таблицы, которые могут быть использованы поисковым двигателем базы данных (далее – БД), для ускорения получения данных.

Создаются такие таблицы с помощью команды **CREATE INDEX**.

Синтаксис SQL (повтор). Основные понятия

Представление. Представления в SQL являются особыми таблицами, которые содержат данные, полученные запросом SELECT из обычных таблиц. Это виртуальная таблица, к которой можно обратиться как к обычным таблицам и получить хранимые данные. Грубо говоря, мы извлекаем необходимые данные из нескольких таблиц и помещаем их в одну. Это намного проще, чем заново создавать таблицы и выполнять кучу других запросов. Более того, это можно сравнить с инкапсуляцией в ООП (объектно-ориентированном языке программирования): там мы можем создавать переменные с различными модификаторами доступа, а тут создаем таблицу с теми данными, которые хотим, чтобы видел пользователь в веб-интерфейсе реализованной системы.

Вложенные таблицы. Вложенную таблицу можно рассматривать как одномерный массив, в котором индексами служат значения целочисленного типа. Грубо говоря, небольшая таблица в виде коллекции внутри таблицы. Вложенная таблица может иметь пустые элементы, которые появляются после их удаления встроенной процедурой DELETE. Количество элементов может динамически увеличиваться. Максимальный размер – 2 ГБ. Очень удобно использовать при работе с большими однородными или темпоральными однородными данными. Однородные данные – это данные одного типа, а темпоральные – данные, привязанные к дате и времени.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

6. Представление

6.1. Создаем еще одну таблицу в БД:

```
use GroupPI1;
CREATE TABLE Students1 (
    StudentID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
INSERT INTO Students1 (StudentID, LastName, FirstName, Address, City)
VALUES ('1', 'Titov', 'Semen', 'Moscow', 'Moscow')
```

6.2. Присваиваем первичный ключ:

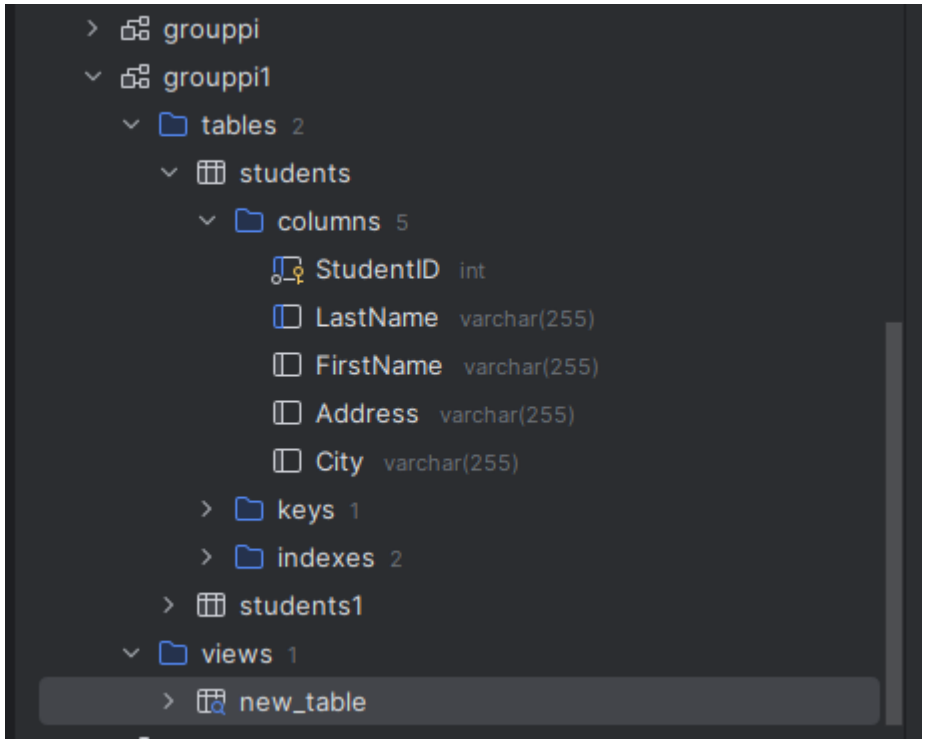
```
ALTER TABLE grouppi1.students
ADD PRIMARY KEY (StudentID)
```

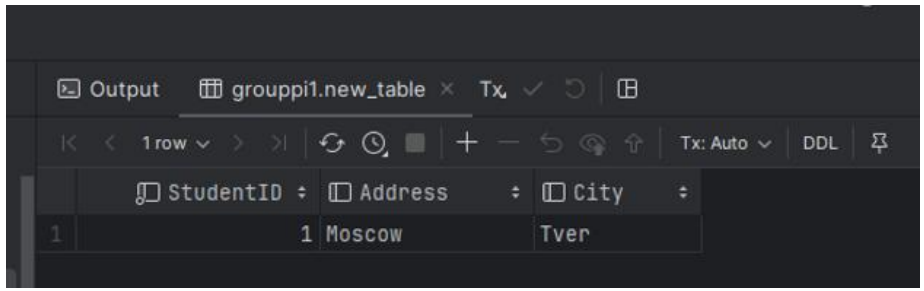
6.3. Создаем представление:

```
CREATE VIEW new_table
AS SELECT grouppi1.students.StudentID, grouppi1.students1.Address,
grouppi1.students.City
FROM students, students1
WHERE students.StudentID = students1.StudentID
```

```
SELECT * FROM new_table
```

6.4. Результат:





7. Вложенные таблицы (Nested Table)

Вложенную таблицу можно рассматривать как одномерный массив, в котором индексами служат значения целочисленного типа. Грубо говоря, небольшая таблица в виде коллекции внутри таблицы. Вложенная таблица может иметь пустые элементы, которые появляются после их удаления встроенной процедурой DELETE. Количество элементов может динамически увеличиваться. Максимальный размер – 2 ГБ. Очень удобно использовать при работе с большими однородными или темпоральными однородными данными. Однородные данные – это данные одного типа, а темпоральные – данные, привязанные к дате и времени.

8. Массивы

```
CREATE TABLE group2 (
    FirstName       text,
    LastName        text[],
    GroupHistory    text[][]
);
```

Задача

Разработка информационной системы «Кафе».

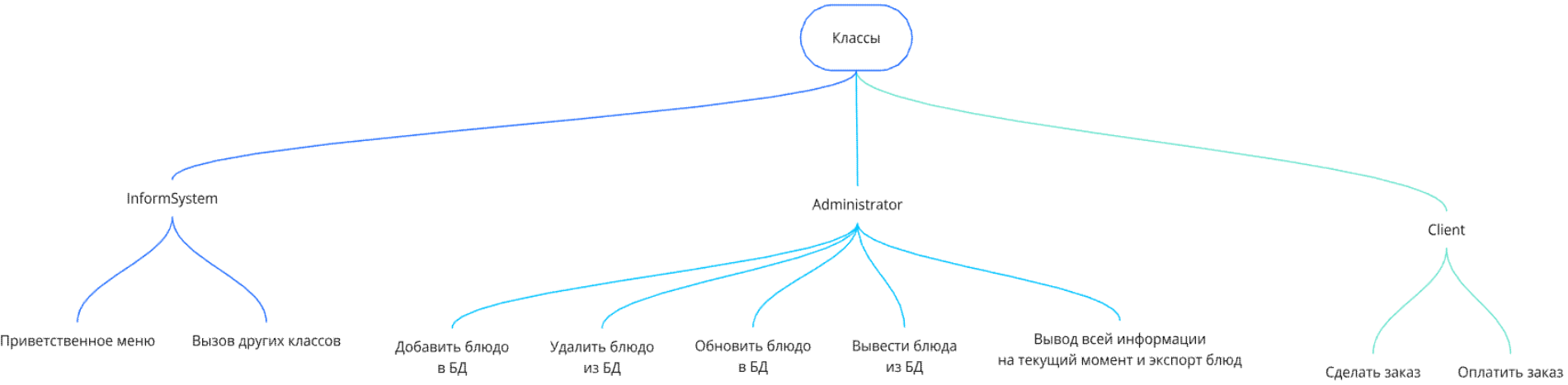


Рисунок 1 – Пример бизнес-логики программы

Лекция №3. Базовые принципы ООП.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Решение задачи. Класс InformSystem

```
package test;

import java.sql.*;
import java.util.Scanner;

public class InformSystem {
   protected static Scanner scan = new Scanner(System.in);
   protected static String tablename = scan.nextLine();
   protected static String tablename1 = scan.nextLine();
   protected static String mysqlUrl =
"jdbc:mysql://localhost/test1000";
   protected static Connection con;

   static {
      try {
         con = DriverManager.getConnection(mysqlUrl, "root",
"root");
      } catch (SQLException e) {
         throw new RuntimeException(e);
      }
   }
```

Одна база для администратора, вторая – для клиента

```
public static void main(String[] args) throws SQLException {
      int x = 0;
      String s = "";
      while (!"3".equals(s)) {
         System.out.println("Добро пожаловать в наше кафе! Вы -
администратор или клиент?");
         System.out.println("1. Администратор.");
         System.out.println("2. Клиент.");
         System.out.println("3. Выход");
         s = scan.next();

         try {
            x = Integer.parseInt(s);
         } catch (NumberFormatException e) {
            System.out.println("Неверный формат ввода");
         }
         switch (x) {
            case 1 -> Administrator.main();
            case 2 -> Client.main();
         }
      }
      System.out.println("Пока! :)");
   }
}
```

Решение задачи. Класс Administrator

Начало

```
package test;

import java.sql.*;

public class Administrator extends InformSystem {
   protected static void main() throws SQLException {
      int x = 0;
      String s = "";
      String s1;
      Statement stmt = con.createStatement();
      String query = "CREATE TABLE IF NOT EXISTS " + tablename + " (ID int, Dish varchar(255), Tableintheres int, Quantity int, Price float,
Ingredients varchar(1000))";
      stmt.executeUpdate(query);
      while (!"8".equals(s)) {
         System.out.println("Привет, администратор! Выбери свое действие: ");
         System.out.println("1. Добавить блюдо в базу данных.");
         System.out.println("2. Обновить блюдо в базе данных.");
         System.out.println("3. Вывести блюда из базы данных.");
         System.out.println("4. Удалить блюдо из базы данных.");
         System.out.println("5. Добавить количество свободных столиков на текущий момент.");
         System.out.println("6. Вывести полную информацию на текущий момент.");
         System.out.println("7. Экспортировать все блюда и количество свободных столиков в локальное хранилище.");
         System.out.println("8. Выход в главное меню.");
         s = scan.next();
```

Лекция №3. Базовые принципы ООП.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Решение задачи. Класс Administrator
Продолжение

```
try {
    x = Integer.parseInt(s);
} catch (NumberFormatException e) {
    System.out.println("Неверный формат ввода");
}
if (x == 1) {
    System.out.println("Добавляем блюдо в базу данных! Продолжить?");
    while (scan.hasNext()) {
        s1 = scan.next();
        if (s1.equals("Выход")) {
            break;
        }
        System.out.print("Введите ID блюда: ");
        int ID = scan.nextInt();
        scan.nextLine();
        System.out.print("Введите название блюда: ");
        String Dish = scan.nextLine();
        System.out.print("Введите количество блюд в наличии на сегодня: ");
        int Quantity = scan.nextInt();
        System.out.print("Введите цену на это блюдо: ");
        float Price = scan.nextFloat();
        scan.nextLine();
        System.out.print("Введите ингредиенты блюда: ");
        String Ingredients = scan.nextLine();
        String query1 = "INSERT INTO " + tablename + " (ID, Dish, Quantity, Price, Ingredients) VALUES (?, ?, ?, ?, ?)";
        PreparedStatement stmt1 = con.prepareStatement(query1);
        stmt1.setInt(1, ID);
        stmt1.setString(2, Dish);
        stmt1.setInt(3, Quantity);
        stmt1.setFloat(4, Price);
        stmt1.setString(5, Ingredients);
        stmt1.executeUpdate();
        System.out.println("Блюдо '" + Dish + "' успешно внесено в базу данных!");
        System.out.println("Продолжить ввод? Если нет, то введите 'Выход' для выхода в меню администратора.");
    }
}
```

Решение задачи. Класс Administrator
Продолжение

```
if (x == 2) {
    System.out.println("Изменяем блюдо в базе данных! Продолжить?");
    while (scan.hasNext()) {
        s1 = scan.next();
        if (s1.equals("Выход")) {
            break;
        }
        System.out.print("Введите ID изменяемого блюда: ");
        int ID = scan.nextInt();
        scan.nextLine();
        System.out.print("Введите новое название блюда: ");
        String Dish = scan.nextLine();
        System.out.print("Введите новое количество блюд на сегодня: ");
        int Quantity = scan.nextInt();
        System.out.print("Введите новую цену блюда: ");
        float Price = scan.nextFloat();
        System.out.print("Введите новые ингредиенты блюда: ");
        scan.nextLine();
        String Ingredients = scan.nextLine();
        String query2 = "UPDATE " + tablename + " SET Dish = '" + Dish + "', Quantity = '" + Quantity + "', Price = '" + Price + "', Ingredients = '" + Ingredients + "' WHERE ID = " +
ID + "";
        PreparedStatement stmt1 = con.prepareStatement(query2);
        stmt1.executeUpdate();
        System.out.println("Блюдо '" + Dish + "' добавлено. Хотите еще обновить блюдо? Если нет, то введите 'Выход' для выхода в меню администратора.");
    }
}
```

Решение задачи. Класс Administrator

Продолжение

```
if (x == 3) {
    Statement stmt1 = con.createStatement();
    ResultSet rs = stmt1.executeQuery("SELECT * FROM " + tablename + "");
    System.out.printf("%1s | %-20s | %-5s | %-6s | %-5s \n", "ID", "Название", "Количество", "Цена", "Ингредиенты");
    while (rs.next()) {

        int ID = rs.getInt("ID");
        String Dish = rs.getString("Dish");
        int Quantity = rs.getInt("Quantity");
        float Price = rs.getFloat("Price");
        String Ingredients = rs.getString("Ingredients");

        System.out.printf("%1d. | %-20s | %-10s | %.2f | %-5s \n", ID, Dish, Quantity, Price, Ingredients);
    }
}
if (x == 4) {
    System.out.println("Удаляем блюдо из базы данных! Продолжить?");
    while (scan.hasNext()) {
        s1 = scan.next();
        if (s1.equals("Выход")) {
            break;
        }
        System.out.print("Введите ID удаляемого блюда: ");
        int ID = scan.nextInt();
        String query2 = "SET SQL_SAFE_UPDATES = 0";
        String query3 = "DELETE FROM " + tablename + " WHERE ID = '" + ID + "'";
        String query4 = "SET SQL_SAFE_UPDATES = 1";
        PreparedStatement stmt1 = con.prepareStatement(query2);
        PreparedStatement stmt2 = con.prepareStatement(query3);
        PreparedStatement stmt3 = con.prepareStatement(query4);
        stmt1.executeUpdate();
        stmt2.executeUpdate();
        stmt3.executeUpdate();
        System.out.println("Блюдо с ID '" + ID + "' успешно удалено из базы данных!");
        System.out.println("Хотите еще удалить блюдо? Если нет, то введите 'Выход' для выхода в меню администратора.");
    }
}
```

Лекция №3. Базовые принципы ООП.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Решение задачи. Класс Administrator
Продолжение

```
if (x == 5) {
    System.out.print("Добавьте количество свободных столиков на текущий момент: ");
    int Tableintheres = scan.nextInt();
    String query1 = "SET SQL_SAFE_UPDATES = 0";
    String query2 = "UPDATE " + tablename + " SET Tableintheres = '" + Tableintheres + "' WHERE Tableintheres IS NULL";
    String query3 = "SET SQL_SAFE_UPDATES = 1";
    PreparedStatement stmt1 = con.prepareStatement(query1);
    PreparedStatement stmt2 = con.prepareStatement(query2);
    PreparedStatement stmt3 = con.prepareStatement(query3);
    stmt1.executeUpdate();
    stmt2.executeUpdate();
    stmt3.executeUpdate();
    System.out.println("Столики успешно добавлены!");
}
if (x == 6) {
    Statement stmt1 = con.createStatement();
    ResultSet rs = stmt1.executeQuery("SELECT * FROM " + tablename + "");
    System.out.printf("%1s | %-20s | %-5s | %-6s | %-20s | %-5s \n", "ID", "Название", "Количество", "Цена", "Количество свободных столиков", "Ингредиенты");
    while (rs.next()) {

        int ID = rs.getInt("ID");
        String Dish = rs.getString("Dish");
        int Quantity = rs.getInt("Quantity");
        float Price = rs.getFloat("Price");
        int Tableintheres = rs.getInt("Tableintheres");
        String Ingredients = rs.getString("Ingredients");

        System.out.printf("%1d. | %-20s | %-10s | %.2f | %-29s | %-5s \n", ID, Dish, Quantity, Price, Tableintheres, Ingredients); // % - спецификатор формата, после
которого указываются аргументы, тире и число - количество пробелов, d (десятичное целое), f (флоат), s (стринг) - типы данных.
    }
}
```

Решение задачи. Класс Administrator
Окончание

```
        if (x == 7) {
            scan.nextLine();
            System.out.print("Введите название файла с расширением: ");
            String file1 = scan.nextLine();
            String query1 = "SET SQL_SAFE_UPDATES = 0";
            String query2 = "UPDATE " + tablename + " SET Price = ROUND(Price, 2)";
            String query3 = "SET SQL_SAFE_UPDATES = 1";
            String query4 = "SELECT 'ID', 'Dish', 'Tableintheres', 'Quantity', 'Price', 'Ingredients' UNION ALL SELECT * FROM " + tablename + " INTO OUTFILE
'C:/Users/teelx/Desktop/mysql/" + file1 + "' CHARACTER SET cp1251";
            PreparedStatement stmt1 = con.prepareStatement(query1);
            PreparedStatement stmt2 = con.prepareStatement(query2);
            PreparedStatement stmt3 = con.prepareStatement(query3);
            PreparedStatement stmt4 = con.prepareStatement(query4);
            stmt1.executeUpdate();
            stmt2.executeUpdate();
            stmt3.executeUpdate();
            stmt4.executeQuery(); // Когда SELECT, тогда всегда пишем executeQuery, а не executeUpdate.
            System.out.println("Данные успешно экспортированы!");
        }
    }
  }
}
```

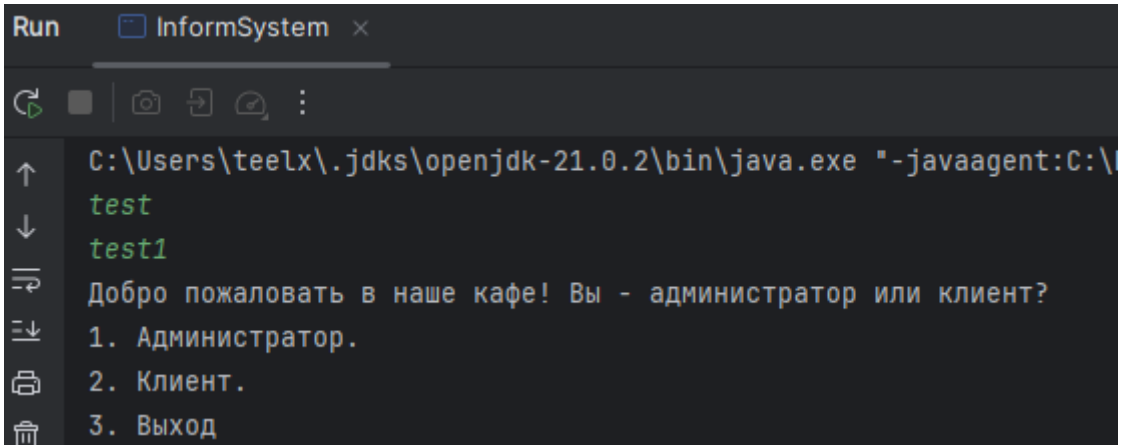
Лекция №3. Базовые принципы ООП.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Решение задачи. Класс Client

```
package test;
import java.sql.*;
import java.util.Random;

public class Client extends InformSystem {
   public static void main() throws SQLException {
      Random random = new Random();
      int x = 0;
      String s = "";
      String s1;
      Statement stmt = con.createStatement();
      String query = "CREATE TABLE IF NOT EXISTS " + tablename1 + " (ID int, Dish varchar(255),
Tableintheres int)";
      stmt.executeUpdate(query);
      while (!"3".equals(s)) {
         System.out.println("Привет, клиент! Выбери свое действие: ");
         System.out.println("1. Сделать заказ.");
         System.out.println("2. Оплатить.");
         System.out.println("3. Выйти в главное меню.");
         s = scan.next();

         try {
            x = Integer.parseInt(s);
         } catch (NumberFormatException e) {
            System.out.println("Неверный формат ввода");
         }
         if (x == 1) {
            System.out.println("Вы хотите сделать заказ! Продолжить?");
            while (scan.hasNext()) {
               s1 = scan.next();
               if (s1.equals("Выход")) {
                  break;
               }
               System.out.println("Доступные на текущий момент блюда:");
               Statement stmt1 = con.createStatement();
               ResultSet rs = stmt1.executeQuery("SELECT * FROM " + tablename
+ "");
               System.out.printf("%1s | %-20s | %-5s | %-6s | %-5s \n", "ID",
"Название", "Количество", "Цена", "Ингредиенты");
               while (rs.next()) {

                  int ID = rs.getInt("ID");
                  String Dish = rs.getString("Dish");
                  int Quantity = rs.getInt("Quantity");
                  float Price = rs.getFloat("Price");
                  String Ingredients = rs.getString("Ingredients");

                  System.out.printf("%1d. | %-20s | %-10s | %.2f | %-5s \n", ID,
Dish, Quantity, Price, Ingredients);
               }
               scan.nextLine();
               System.out.print("Выберите название блюда: ");
               String Dish = scan.nextLine();
               System.out.println("Вы выбрали блюдо " + Dish);
               int a = random.nextInt();
               System.out.println("Ваш заказ № " + a + ". Оплатите его и
присаживайтесь за любой свободный столик.");
               System.out.println("Введите 'Выход' для выхода в меню с целью
оплаты заказа.");
            }
         }
         if (x == 2) {
            System.out.println("Спасибо за оплату заказа!");
         }
      }
   }
}
```

Решение задачи. Результат. Вывод результатов выполнения функций (методов) класса «InformSystem»

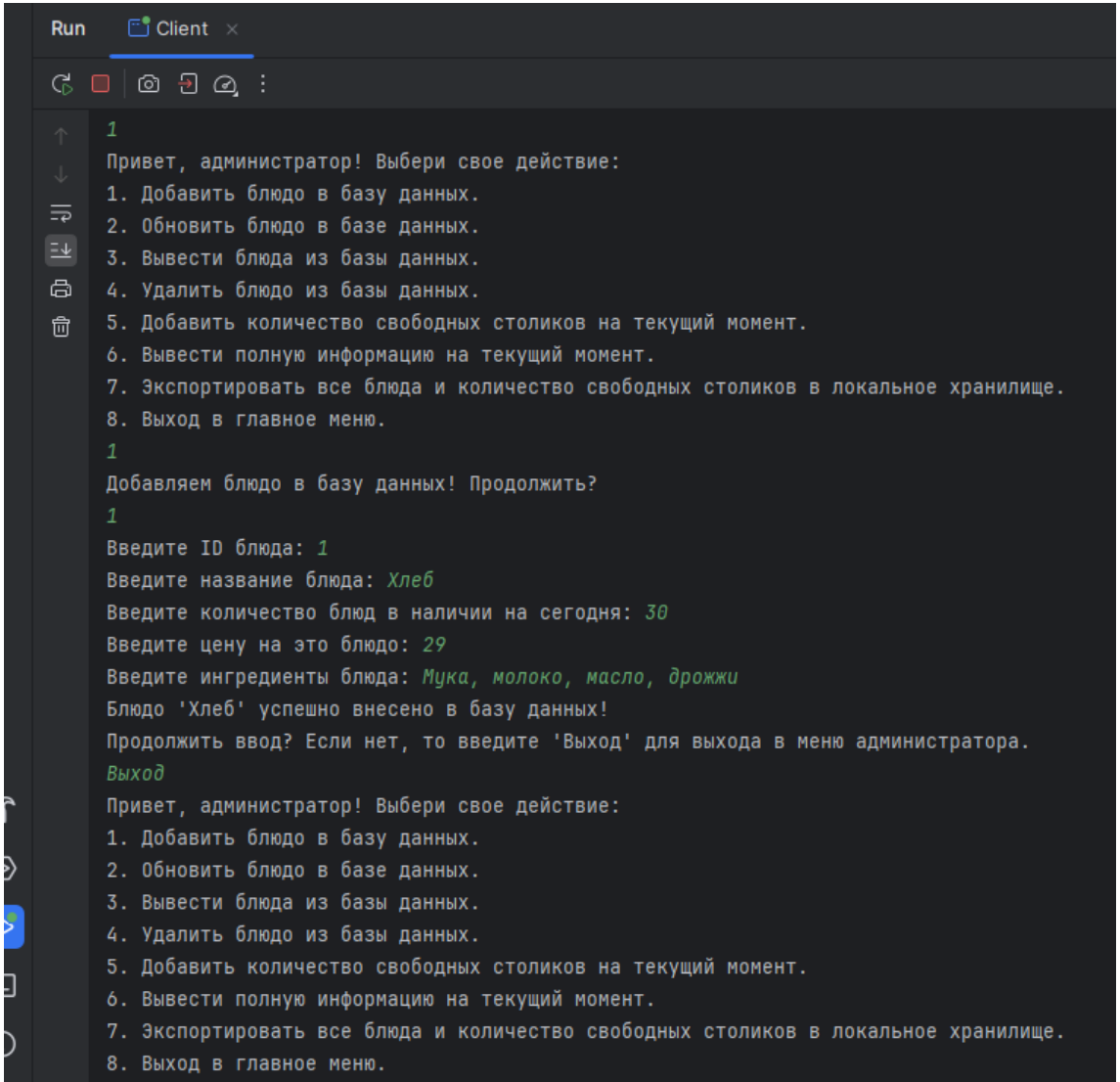


Нажимаем «1»

Лекция №3. Базовые принципы ООП.

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

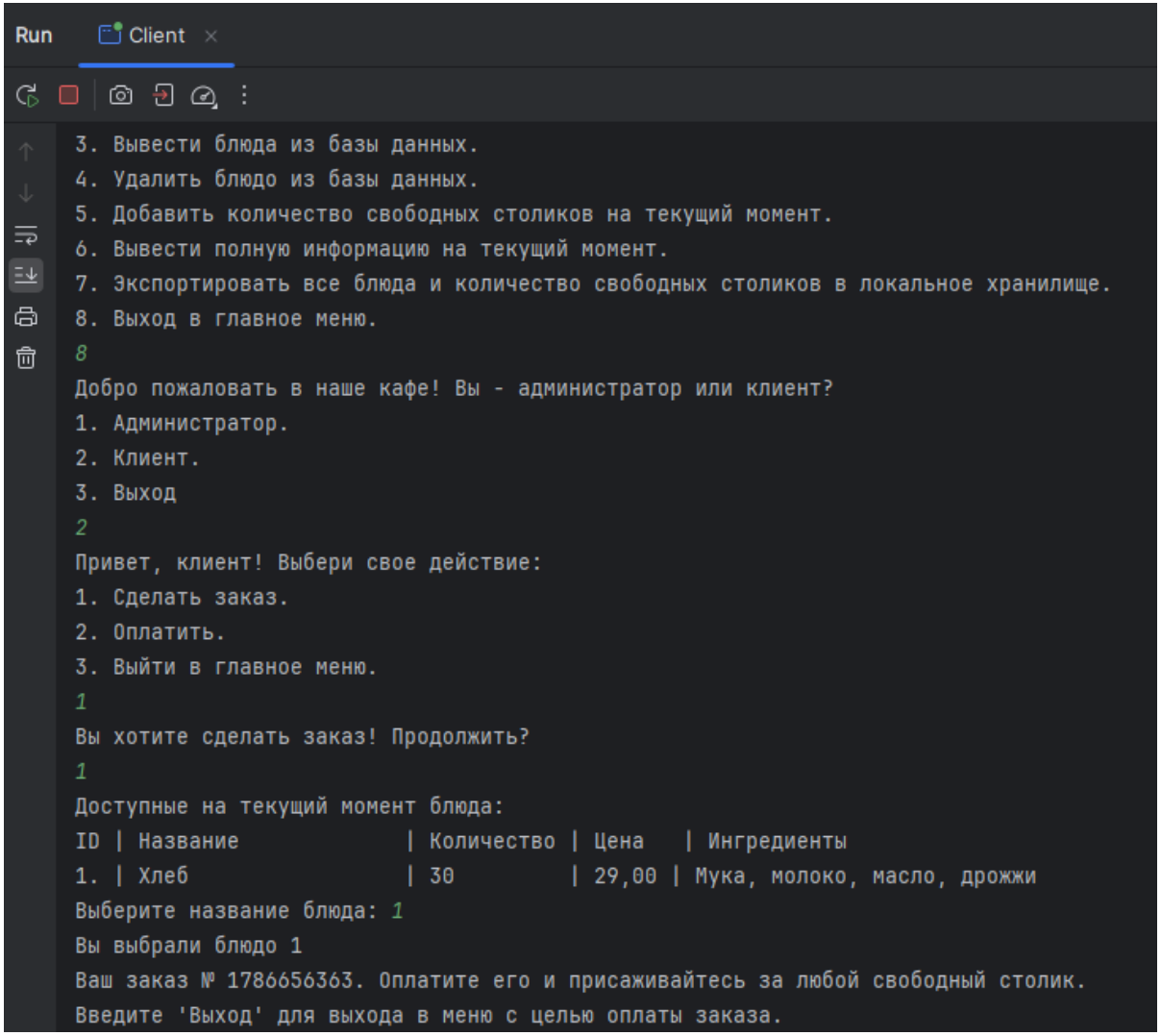
Решение задачи. Результат. Вывод результатов выполнения функций (методов) класса «Administrator»



Решение задачи. Результат. Вывод результатов выполнения функций (методов) класса «Administrator»

```
Привет, администратор! Выбери свое действие:
1. Добавить блюдо в базу данных.
2. Обновить блюдо в базе данных.
3. Вывести блюда из базы данных.
4. Удалить блюдо из базы данных.
5. Добавить количество свободных столиков на текущий момент.
6. Вывести полную информацию на текущий момент.
7. Экспортировать все блюда и количество свободных столиков в локальное хранилище.
8. Выход в главное меню.
6
ID | Название            | Количество | Цена   | Количество свободных столиков | Ингредиенты
1. | Хлеб                | 30         | 29,00 | 0                             | Мука, молоко, масло, дрожжи
Привет, администратор! Выбери свое действие:
1. Добавить блюдо в базу данных.
2. Обновить блюдо в базе данных.
3. Вывести блюда из базы данных.
4. Удалить блюдо из базы данных.
5. Добавить количество свободных столиков на текущий момент.
6. Вывести полную информацию на текущий момент.
7. Экспортировать все блюда и количество свободных столиков в локальное хранилище.
8. Выход в главное меню.
```

Решение задачи. Результат. Вывод результатов выполнения функций (методов) класса «Client»



Лекция №3. Базовые принципы ООП и работа с базами данных. Заключение

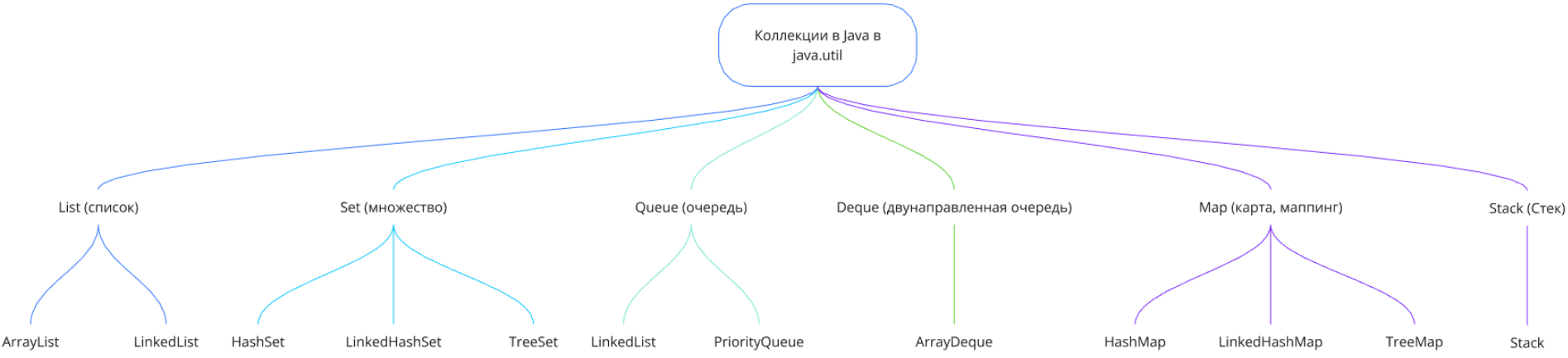
Преимущества ООП:

1. **Модульность.** Код разбит на отдельные объекты (классы), что упрощает разработку, тестирование и отладку.
2. **Повторное использование кода.** Благодаря наследованию и интерфейсам код можно использовать повторно, сокращая время разработки.
3. **Легкость в сопровождении и расширении.** За счет четкой структуры и принципов ООП новые функции легко добавлять, а ошибки — исправлять.
4. **Снижение сложности.** Инкапсуляция и абстракция уменьшают сложность, делая программы более управляемыми.
5. **Безопасность.** Скрытие внутреннего состояния объектов и управление доступом к ним повышает безопасность данных.

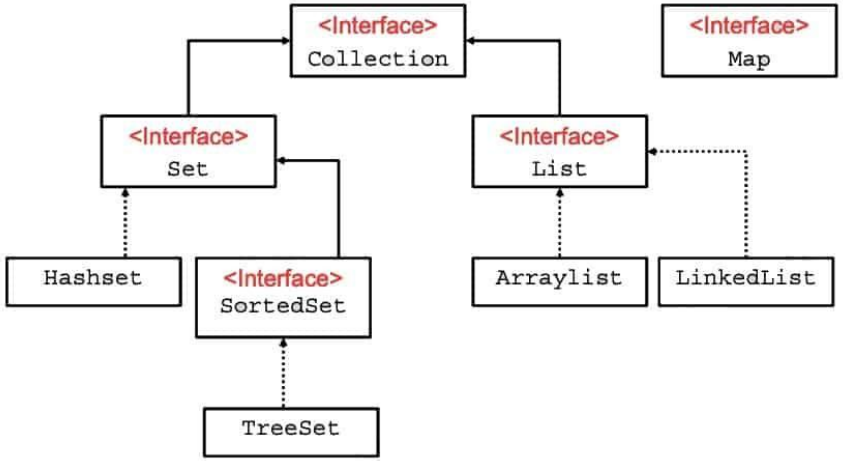
Таким образом, ООП значительно улучшает структуру программ, делает их более гибкими и управляемыми. Использование ООП позволяет создавать сложные системы, которые легко поддерживать и расширять. Оно помогает разрабатывать надежный и понятный код, который легко адаптируется под изменяющиеся требования, делая разработку эффективной и безопасной.

Лекция №4. Коллекции. Определение и виды

Коллекции в Java — набор классов и интерфейсов, которые обобщенно можно назвать фреймворком, предназначенные для хранения и управления группами объектов. Коллекции позволяют легко манипулировать данными, такими как добавление, удаление, поиск и упорядочивание элементов. Коллекции входят в состав стандартной библиотеки java.util и обеспечивают удобные структуры данных, которые упрощают написание кода для работы с наборами данных. В целом, коллекции позволяют разработчику работать с группами объектов гибко и эффективно.



Лекция №4. Коллекции. Основные преимущества



Коллекции в Java позволяют:

1. Хранить динамически изменяемые наборы данных.
2. Упростить добавление, удаление и поиск элементов внутри самой коллекции.
3. Выполнять стандартные алгоритмы для сортировки, поиска и других операций с данными.

Коллекции. List (список). *ArrayList*: определение, основные операции + реализация

ArrayList — это структура данных в языке Java, представляющая собой динамический массив и предназначенная для хранения множества значений. Это усовершенствованный массив, в котором можно изменять количество элементов и с легкостью выполнять с ними различные операции.

Более того, ***ArrayList*** является реализацией изменяемого массива интерфейса List и частью Collection Framework, который отвечает за список (или динамический массив), расположенный в пакете java.utils. Этот класс реализует все необязательные операции со списком и предоставляет методы управления размером массива, который используется для хранения списка.

ArrayList хранит только ссылочные типы, любые объекты, включая сторонние классы. Строки, потоки вывода, другие коллекции. Для хранения примитивных типов данных используются классы-обертки (например, для Int – это Integer).

Коллекции. List (список). *List* + *ArrayList*: основные операции + реализация. Часть 1

1. Создание списка (динамического массива):

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Создание списка (динамического массива)
        List<String> list = new ArrayList<>();

        // 1. Добавление элементов
        list.add("Евгений");        // Добавление элемента в конец
        list.add("Вадим");
        list.add("Давид");
        list.add(1, "Олег");      // Вставка элемента на определенную позицию (индекс)
        System.out.println("Список после добавления: " + list);
```

Мы объявили переменную list типа List<String> и инициализировали ее экземпляром ArrayList<>();. Это означает, что мы используем интерфейс List для работы со списком, а ArrayList является его конкретной реализацией.

2. Получение элемента по индексу:

```
// 2. Получение элемента
String elementAtIndex2 = list.get(2);   // Получение элемента по индексу
System.out.println("Элемент с индексом 2: " + elementAtIndex2);
```

3. Изменение элемента по индексу:

```
// 3. Изменение элемента
list.set(2, "Антон");   // Замена элемента по индексу
System.out.println("Список после изменения: " + list);
```

4. Удаление элементов:

```
list.remove(1);      // Удаление по индексу
System.out.println("Список после удаления по индексу: " + list);
list.remove("Олег"); // Удаление по значению
System.out.println("Список после удаления Олега: " + list);
list.removeAll(Arrays.asList("Антон", "Давид")); // Удаление нескольких элементов
System.out.println("Список после удаления нескольких элементов: " + list);
list.add("Давид");
System.out.println("Получившийся список после добавления элементов: " + list);
list.removeIf(Name -> Name.startsWith("E")); // Удаление элемента по условию,
//при этом Name ... - Лямбда-выражение, где Name — это каждый элемент списка (предполагается, что это строка), и мы проверяем, начинается ли он с буквы "E".
System.out.println("Список после удаления элементов, начинающихся с 'E': " + list);
list.clear(); // Очистка списка (удаление всех элементов)
System.out.println("Список после очистки: " + list);
```

Коллекции. List (список). *List* + *ArrayList*: основные операции + реализация. Часть 2

5. Проверка на наличие элемента:

```
List<String> names = Arrays.asList("Самвел", "Игорь", "Иван", "Евгений");
//Добавление нескольких элементов в список
list.addAll(names);
System.out.println("Получившийся список 'list': " + list);
boolean contains = list.contains("Евгений"); // Проверка наличия элемента
System.out.println("Список содержит элемент 'Евгений': " + contains);
int indexOfEvgeny = list.indexOf("Евгений"); // Индекс первого вхождения элемента
System.out.println("Индекс первого вхождения элемента 'Евгений': " + indexOfEvgeny);
list.add("Евгений");
int lastIndexOfEvgeny = list.lastIndexOf("Евгений"); // Индекс последнего вхождения элемента
System.out.println("Индекс первого вхождения элемента 'Евгений': " + lastIndexOfEvgeny);
```

6. Размер списка:

```
int size = list.size();  // Получение размера списка
System.out.println("Размер списка: " + size);
```

7. Проверка списка на наличие элементов:

```
boolean isEmpty = list.isEmpty();  // Проверка, пуст ли список
System.out.println("Список пуст: " + isEmpty);
list.clear();
boolean isEmpty1 = list.isEmpty();  // Проверка, пуст ли список
System.out.println("Список пуст: " + isEmpty1);
```

8. Итерация по элементам списка:

```
List<String> names1 = Arrays.asList("Самвел", "Игорь", "Иван", "Евгений");
//Добавление нескольких элементов в список
list.addAll(names1);
System.out.println("Итерация через цикл for:");
int counter = 1;
for (String name : list) {
    System.out.println(counter + ". " + name);
    counter++;
}

int counter1 = 1;
System.out.println("Итерация через for с индексами:");
for (int i = 0; i < list.size(); i++) {
    System.out.println(counter1 + ". " + list.get(i));
    counter1++;
}

System.out.println("Итерация с использованием Iterator:");
int counter2 = 1;
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    System.out.println(counter2 + ". " + iterator.next());
    counter2++;
}
```

Коллекции. List (список). *List + ArrayList*: основные операции + реализация. Часть 3

9. Преобразование в массив:

```
String[] array = list.toArray(new String[0]); // Преобразование списка в массив
System.out.println("Список, преобразованный в массив: " + Arrays.toString(array));
System.out.println("Вывод элементов массива через цикл:");
int counter3 = 1;
for (String name : array) {
    System.out.println(counter3 + ". " + name);
    counter3++;
}
```

10. Сортировка списка:

```
Collections.sort(list);  // Сортировка в естественном порядке (по алфавиту)
System.out.println("Список после сортировки в алфавитном порядке: " + list);
Collections.sort(list, Comparator.reverseOrder()); // Сортировка в обратном порядке
// Компаратор - это интерфейс в Java, который используется для определения порядка элементов в коллекции или массиве
System.out.println("Список после сортировки в обратном порядке: " + list);
```

11. Получение подсписка:

```
List<String> subList = list.subList(1, 3); // Подсписок с элементами от индекса 1 до 3 (не включая 3)
System.out.println("Подсписок по индексам: " + subList);
// Создаем подсписок, содержащий только элементы, которые начинаются с буквы 'E' или 'Д'
List<String> subList1 = list.stream()
        .filter(s -> s.startsWith("E") || s.startsWith("C"))
        .toList().reversed(); // Делаем список в алфавитном порядке
System.out.println("Получившийся подсписок, начинающийся с буквы 'E' или 'C': " + subList1);
```

12. Добавление элементов из другой коллекции:

```
List<String> anotherList = new ArrayList<>();
anotherList.add("Захар");
anotherList.add("Ирина");
list.addAll(anotherList);
System.out.println("Список после добавления всех элементов другой коллекции 'anotherList': " + list);
```

Коллекции. List (список). *List* + *ArrayList*: основные операции + реализация. Часть 4

13. Создание размерности массива:

```
ArrayList<String> list1 = new ArrayList<>();
list1.ensureCapacity(100); // емкость нашего массива, можем и больше
List<String> names2 = Arrays.asList("Самвел", "Игорь", "Иван", "Евгений"); //Добавление нескольких элементов
list1.addAll(names2);
System.out.println("Получившийся массив: " + list1);
//Как работает ensureCapacity:
//Проверка текущей емкости: метод проверяет, превышает ли minCapacity текущую емкость внутреннего массива.
//Расчет новой емкости: если minCapacity больше текущей емкости, вычисляется новая емкость (обычно с некоторым запасом).
//Создание нового массива: внутренний массив расширяется до новой емкости, и существующие элементы копируются в него.
list1.trimToSize(); //Обрезка емкости до текущего размера
System.out.println("Емкость списка обрезана до текущего размера.");
```

14. Потоковая обработка:

```
    // Stream не изменяет исходную коллекцию, а возвращает новую с результатом операции, за счет чего повышается производительность
    // Есть возможность использовать многопоточность, что ускоряет работу программы
    Stream<String> stream = list.stream();
    List<String> names3 = Arrays.asList("Евлампия", "Егор", "Евкакий", "Евгений"); //Добавление нескольких элементов
    list.addAll(names3);
//      System.out.println("Потоковая обработка элементов и их вывод через цикл без счетчика:");
//      stream.filter(s -> s.contains("E")).forEach(System.out::println);
//      System.out.println("Потоковая обработка элементов и их вывод через цикл со счетчиком:");
    AtomicInteger element = new AtomicInteger(0);
//      stream.filter(s -> s.contains("E")).forEach(s -> System.out.println("Элемент " + element.getAndIncrement() + ": " + s));
    //Вывод без условия
    stream.forEach(s -> System.out.println("Элемент " + element.getAndIncrement() + ": " + s));
    // Использование многопотока
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

    int sum = numbers.parallelStream()
            .reduce(0, Integer::sum);

    System.out.println("Сумма значений: " + sum);
```

Коллекции. List (список). *List* + *ArrayList*: основные операции + реализация. Часть 5

15. Клонирование списка:

```
ArrayList<String> clonedList = (ArrayList<String>) list1.clone();
System.out.println("Клонированный список: " + clonedList);
// Можем убрать предупреждение @SuppressWarnings("unchecked")
//Более правильный метод: ArrayList<String> clonedList = new ArrayList<>(list1);
```

16. Сравнение списков:

```
boolean isEqual = list1.equals(clonedList);
System.out.println("Списки равны: " + isEqual);
```

17. Получение хэш-кода списка:

```
// Хэш-код — это целочисленное значение фиксированного размера, которое вычисляется на основе данных объекта.
// Хэш-код используется для ускорения поиска и сравнения объектов в коллекциях, таких как хэш-таблицы
int hashCode = list.hashCode();
System.out.println("Хэш-код списка: " + hashCode);
```

Коллекции. List (список). *LinkedList*: определение и основные характеристики. Часть 1

LinkedList —класс из стандартной библиотеки Java, который реализует структуру данных **двусвязного списка**. Он является частью коллекций Java и находится в пакете java.util. Т.е. **LinkedList – двусвязный список**.

Основные особенности (характеристики).

1. *Двусвязный список, который содержит в себе **элементы**.*

1.1. Каждый **элемент** (узел) содержит:
1.2. Данные (т.е. сам элемент).
1.3. Ссылку на предыдущий узел.
1.4. Ссылку на следующий узел.

Это позволяет эффективно вставлять и удалять элементы в любой позиции списка без необходимости сдвига других элементов.

*2. Реализует **интерфейсы**:*

2.1. **List**: позволяет работать с LinkedList как со списком, предоставляя методы для доступа по индексу, добавления, удаления и поиска элементов.
2.2. **Deque** (Double-Ended Queue): позволяет использовать LinkedList как двустороннюю очередь, поддерживая операции добавления и удаления элементов с обеих сторон.
2.3. **Queue**: позволяет использовать LinkedList как обычную очередь (FIFO — First-In-First-Out).

Коллекции. List (список). *LinkedList*: определение и основные характеристики. Часть 2

3. Особенности работы:

3.1. Быстрая вставка и удаление элементов в начале и конце списка (O(1)), так как требуется лишь переназначить ссылки узлов.

3.2. Медленный доступ по индексу (O(n)), поскольку для доступа к элементу требуется последовательный обход списка от начала или конца до нужной позиции.

3.3. Более высокая нагрузка на память по сравнению с ArrayList, так как каждый узел хранит дополнительные ссылки на соседние узлы.

Когда использовать LinkedList?

1. Необходимы частые вставки и удаления элементов в середине, начале или конце списка.
2. Когда не требуется быстрый случайный доступ к элементам по индексу.
3. Для реализации очередей и стеков, где важна производительность операций добавления и удаления с обеих сторон.

Преимущества:

1. Эффективные вставки и удаления в любом месте списка.
2. Гибкость благодаря реализации нескольких интерфейсов (List, Deque, Queue).
3. Подходит для реализации стеков и очередей.

Недостатки:

1. Медленный доступ по индексу, так как требуется последовательный обход.
2. Более высокое потребление памяти из-за хранения ссылок на соседние узлы.
3. Низкая эффективность при частом доступе по индексу, по сравнению с ArrayList.

Коллекции. List (список). *LinkedList vs ArrayList*: основные отличия. Часть 1

1. Структура данных

1.1. ***ArrayList***
1.1.1. Основан на динамическом массиве.
1.1.2. Элементы хранятся в непрерывном блоке памяти.
1.1.3. При добавлении элементов, если массив заполнен, происходит резервирование нового массива большего размера и копирование элементов.

1.2. ***LinkedList***
1.2.1. Реализован как двусвязный список.
1.2.2. Каждый элемент является узлом, содержащим данные и ссылки на предыдущий и следующий узлы.
1.2.3. Элементы распределены по памяти, нет необходимости в непрерывном блоке.

Коллекции. List (список). *LinkedList vs ArrayList*: основные отличия. Часть 2

2. Производительность операций (с точки зрения быстроты)

2.1. Доступ по индексу (get(int index), set(int index, E element))

ArrayList:

1. Быстрый доступ по индексу за константное время O(1) из-за того, что элементы расположены подряд в массиве.

LinkedList:

1. Доступ по индексу требует последовательного обхода от начала или конца списка.
2. Временная сложность O(n), где n — позиция элемента.

2.2. Вставка и удаление элементов

ArrayList:

1. Добавление в конец — быстрое, т.е. (O(1)), если не происходит расширение массива.
2. Вставка или удаление в середине требует сдвига элементов, что занимает O(n) времени.

LinkedList:

1. Добавление или удаление в начале или конце списка — O(1).
2. Вставка или удаление в середине также может быть O(1), если есть ссылка на узел. Но поиск позиции занимает O(n).

2.3. Итерация

1. Оба списка обеспечивают схожую производительность при последовательном обходе.

Коллекции. List (список). *LinkedList vs ArrayList*: основные отличия. Часть 3

3. Память

3.1. ArrayList

1. Требует меньше памяти на элемент, так как хранит только данные.
2. Возможен перерасход памяти из-за резервирования дополнительного пространства в массиве.

3.2. LinkedList

1. На каждый элемент приходится дополнительная память для хранения ссылок на соседние узлы.
2. Нет перерасхода памяти из-за динамического выделения памяти под каждый узел.

4. Интерфейсы и функциональность

4.1. ArrayList

1. Реализует интерфейс List.
2. Оптимизирован для случайного доступа по индексу.

4.2. LinkedList

1. Реализует интерфейсы List, Deque, Queue.
2. Может использоваться как очередь, стек или двусторонняя очередь.
3. Предоставляет дополнительные методы для работы с началом и концом списка (addFirst, addLast, removeFirst, removeLast и т.д.).

Коллекции. List (список). *LinkedList vs ArrayList*: основные отличия. Часть 4

5. Использование и применение

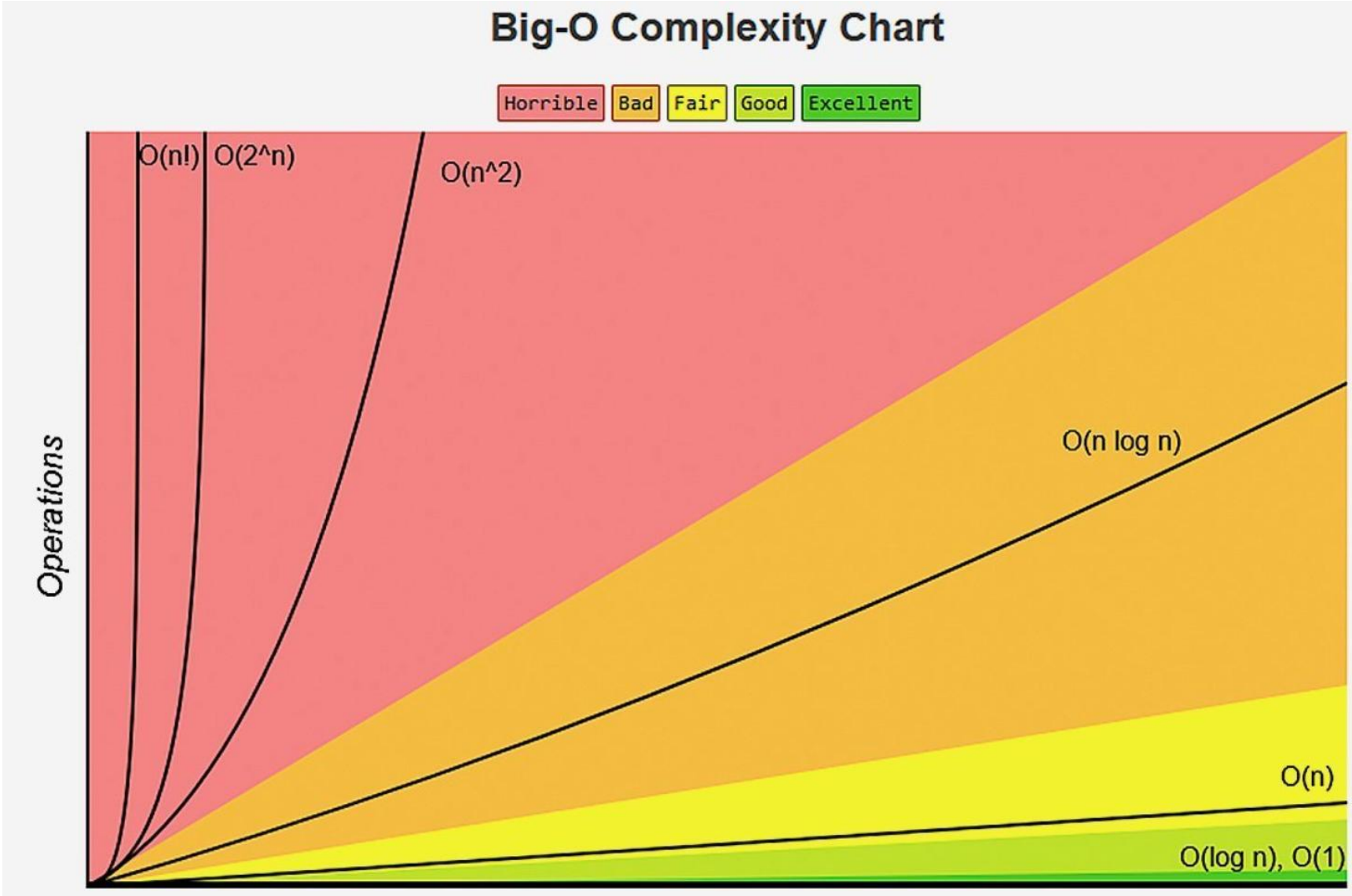
*5.1. Используйте **ArrayList**, если:*

1. Необходим быстрый случайный доступ к элементам.
2. Необходимо часто выполнять считывание данных, при этом в массиве происходят редкие вставки и удаления.
3. Операции добавления и удаления происходят в основном в конце списка.

*5.2. Используйте **LinkedList**, если:*

1. Необходимы частые вставки и удаления элементов, особенно в начале списка (массива).
2. Необходимо реализовать очередь или стек.
3. Не требуется быстрый доступ по индексу.

Коллекции. List (список). *График сложностей алгоритмов*



Коллекции. List (список). *LinkedList vs ArrayList*: основные отличия. Часть 5

6. Примеры операций и их сложность

#	Операция	ArrayList (сложность)	LinkedList (сложность)
1	add() – в конец списка	O(1)* – легко	O(1) – легко
2	add() – по индексу	O(n) – чуть сложнее	O(n)** – чуть сложнее
3	get() – по индексу	O(1) – легко	O(n) – чуть сложнее
4	remove () – по индексу	O(n) – чуть сложнее	O(n)** – чуть сложнее
5	Итерирование	O(n) – чуть сложнее	O(n) – чуть сложнее

*O(1) при условии, что нет расширения массива.
**Если позиция известна или используется итератор, операции могут быть быстрее.

Коллекции. List (список). *LinkedList vs ArrayList*: основные отличия. Часть 6

7. Основные преимущества и недостатки

7.1. ArrayList

Преимущества:
1. Быстрый доступ по индексу.
2. Менее затратен по памяти.

Недостатки:
1. Медленные вставки и удаления в середине списка.
2. Возможны затраты времени на расширение массива.

7.2. LinkedList

Преимущества:
1. Быстрые вставки и удаления при работе с началом и концом.
2. Нет лишних затрат (переизбытка ресурсов) на перераспределение памяти.

Недостатки:
1. Медленный доступ по индексу, т.е. на данную операцию расходуется больше ресурсов.
2. Больше затрачивается память на хранение ссылок.

Коллекции. List (список). *LinkedList*: реализация. Часть 1

1. Создание списка и добавление элементов:

```
// 1. Создание списка
LinkedList<String> list = new LinkedList<>();

// 2. Добавление элементов списка
list.add("Евгений"); // Добавление элемента в конец списка
list.add("Антон");
list.add(1, "Давид"); // Добавление элемента по индексу
list.addFirst("Вадим"); // Добавление элемента в начало списка
list.addLast("Петр"); // Добавление элемента в конец списка
list.offer("Захар"); // Добавляет элемент в конец списка (как в двусторонней очереди)
list.offerFirst("Жанна"); // добавляет элемент в начало списка
list.offerLast("Антонина"); // Добавление элемента в конец списка
// Вывод списка после добавления элементов
System.out.println("После добавления элементов №1: " + list);
String[] elements = {"Элемент X", "Элемент Y", "Элемент Z"};
Collections.addAll(list, elements); // Добавление нескольких элементов в конец списка
System.out.println("После добавления элементов №2: " + list);
String[] elements1 = {"Элемент A", "Элемент B", "Элемент C"};
list.addAll(0, Arrays.asList(elements1)); // Добавление нескольких элементов в начало списка
System.out.println("После добавления элементов №3: " + list);
```

2. Удаление элементов:

```
// 2. Удаление элементов
String removedElement = list.remove(); // remove(): удаляет и возвращает первый элемент
System.out.println("Удален элемент: " + removedElement);
list.remove(2); // remove(int index): удаляет элемент по индексу
list.remove("Евгений"); // remove(Object o): удаляет первое вхождение элемента
String firstElement = list.removeFirst(); // removeFirst(): удаляет и возвращает первый элемент
String lastElement = list.removeLast(); // removeLast(): удаляет и возвращает последний элемент
String polledElement = list.poll(); // poll(): удаляет и возвращает первый элемент (как в очереди)
String polledFirst = list.pollFirst(); // pollFirst(): удаляет и возвращает первый элемент
String polledLast = list.pollLast(); // pollLast(): удаляет и возвращает последний элемент
// list.clear(); // clear(): очищает список
list.subList(1, 3).clear(); // Удаление элементов по индексам
System.out.println("Обновленный список: " + list);
// pop(): удаляет и возвращает первый элемент (как в стеке)
// Пример создания и использования стека
LinkedList<String> stack = new LinkedList<>();
stack.push("Володенька");
stack.push("Егорка");
String poppedElement = stack.pop();
System.out.println("Извлечен из стека: " + poppedElement);
System.out.println("Элементы стека: " + stack);
```

Коллекции. List (список). *LinkedList*: реализация. Часть 2

3. Извлечение элементов:

```
String elementAtIndex = list.get(1); // Получение элемента по индексу
System.out.println("Элемент по индексу 1: " + elementAtIndex);
String first = list.getFirst(); // Получаем первый элемент списка
String last = list.getLast(); // Получаем последний элемент
String peekElement = list.peek(); // Возвращаем первый элемент без удаления
String peekFirst = list.peekFirst(); // Возвращаем первый элемент без удаления (на примере очереди)
String peekLast = list.peekLast(); // Возвращает последний элемент без удаления
String firstElementQueue = list.element(); // Возвращает первый элемент без удаления (как в очереди)
```

4. Итерация и поиск:

```
// 4. Итерация и поиск
boolean containsB = list.contains("Евгений"); // Проверяем наличие элемента
int indexOfC = list.indexOf("Евгений"); // Индекс первого вхождения
int lastIndexOfA = list.lastIndexOf("A"); // Индекс последнего вхождения
Iterator<String> iterator = list.iterator(); // Итератор по элементам
int element = 1;
while (iterator.hasNext()) {
    System.out.println("Элемент " + element + ": " + iterator.next());
    element++;
}
Iterator<String> descendingIterator = list.descendingIterator(); // Обратный итератор
int element1 = 1;
while (descendingIterator.hasNext()) {
    System.out.println("Элемент " + element1 + ": " + descendingIterator.next());
    element1++;
}
```

5. Замена элементов и сортировка:

```
list.set(2, "Харитон"); // Заменяет элемент по индексу
Collections.sort(list); // Сортировка в алфавитном порядке
Collections.reverse(list); // Сортировка в обратном порядке
System.out.println("Сортировка в обратном порядке" + list);
```

6. Проверка на пустоту и размер списка:

```
int size = list.size(); // Размер списка
boolean isEmpty = list.isEmpty(); // проверка на пустоту
```

Коллекции. List (список). *LinkedList*: реализация. Часть 3

7. Дополнительные методы:

```
// toArray(): преобразует в массив
Object[] array = list.toArray();

// clone(): создает поверхностную копию
LinkedList<String> clonedList = (LinkedList<String>) list.clone();

// push(E e): добавляет элемент в начало (как в стеке)
list.push("Георгис");

// listIterator(): двунаправленный итератор
ListIterator<String> listIterator = list.listIterator(); // Итератор вперед
int el = 1;
while (listIterator.hasNext()) {
    System.out.println(el + ". " + listIterator.next());
    el++;
}
int el1 = 1;
while (listIterator.hasPrevious()) { // Итератор назад
    System.out.println(el1 + ". " + listIterator.previous());
    el1++;
}
```

Коллекции. Set (Множество). *HashSet*: теория. Часть 1

HashSet — класс в Java, реализующий интерфейс Set и использующий хеш-таблицу для хранения элементов. Он предназначен для хранения уникальных элементов и не гарантирует порядок их расположения.

Особенности:

1. Содержится внутри пакета «java.util»;
2. Может наследоваться от «AbstractSet<E>», а также реализует интерфейсы «Set<E>», Cloneable и Serializable.
3. Хранит только уникальные элементы, так как является множеством.
4. Допускается хранение только одного null-значения.
5. Хаотичен, т.е. неупорядочен.
6. Не синхронизирован, т.е. не является потокобезопасным без дополнительной синхронизации.

Коллекции. Set (Множество). *HashSet*: теория. Часть 2

Конструкторы создания (методы):

1. HashSet() — создает пустой набор с начальной емкостью по умолчанию (16) и коэффициентом загрузки (0.75 – 75%). **Коэффициент загрузки** – показатель того, насколько заполненным может быть **HashSet** до того момента, когда его емкость автоматически увеличится.
2. HashSet(Collection<? extends E> c) — создает набор, содержащий элементы указанной коллекции.
3. HashSet(int initialCapacity) — создает пустой набор с указанной начальной емкостью и коэффициентом загрузки по умолчанию.
4. HashSet(int initialCapacity, float loadFactor) — создает пустой набор с указанной начальной емкостью и коэффициентом загрузки.

Коллекции. Set (Множество). *HashSet*: реализация. Основные команды. Часть 1

1. Создание множества и добавление элементов

```
// Создание HashSet
HashSet<String> set = new HashSet<>();

// Добавление элементов
set.add("Женек");
set.add("Санек");
set.add("Антоха");
set.add("Женек"); // Дубликат, не будет добавлен
List<String> elements = Arrays.asList("Гена", "Аркадий", "Ислам");
set.addAll(elements);
System.out.println("После добавления элементов: " + set);
```

2. Размерность и наличие элемента

```
// Проверка размера
System.out.println("Размер набора: " + set.size());

// Проверка наличия элемента
if (set.contains("Женек")) {
    System.out.println("Набор содержит элемент Женек");
}
```

3. Удаление по элементу

```
// Удаление элемента по индексу не поддерживается, так как это множество.
// У множества нет индексов из-за неупорядоченности
set.remove("Санек");
System.out.println("После удаления: " + set);
```

4. Добавление нескольких элементов (другая разновидность добавления)

```
// Добавление коллекции элементов
Set<String> newSet = new HashSet<>();
newSet.add("Катя");
newSet.add("Танюха");
set.addAll(newSet);
System.out.println("После добавления новой коллекции: " + set);
```

5. Удаление коллекции элементов

```
// Удаление коллекции элементов
set.removeAll(newSet);
System.out.println("После удаления новой коллекции: " + set);
```

6. Проверка на пустоту и клонирование набора

```
// Проверка на пустоту
System.out.println("Набор пустой? " + set.isEmpty());

// Клонирование набора
HashSet<String> clonedSet = (HashSet<String>) set.clone();
System.out.println("Клонированный набор: " + clonedSet);
```

7. Очистка множества

```
// Очистка набора
set.clear();
System.out.println("После очистки набора: " + set);
```

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Коллекции. Set (Множество). *HashSet*: реализация. Основные команды. Часть 2

8. Клонирование множества

```
// Использование клонирования
System.out.println("Использование клонирования и вывод через цикл:");
clonedSet.forEach(System.out::println);
```

9. Проверка наличия нескольких элементов

```
// Проверка containsAll()
boolean containsAll = clonedSet.containsAll(newSet);
System.out.println("Клонированный набор содержит все элементы newSet? " + containsAll);
```

10. Сохранение общих элементов

```
// Использование retainAll()
// используется для удаления из текущего набора всех элементов, которые не содержатся в указанной коллекции.
// Иными словами, он сохраняет только те элементы, которые присутствуют в обеих коллекциях.
set.add("Катя");
set.add("Танюха");
clonedSet.retainAll(set);
System.out.println("Сохраненные элементы: " + clonedSet);
System.out.println("После удаления: " + set);
```

11. Добавление null (значение, обозначающие отсутствие объекта)

```
// Пример работы с null
set.add(null);
System.out.println("После добавления null: " + set);
set.remove(null);
System.out.println("После удаления null: " + set);
```

12. Синхронизация

```
// Синхронизация HashSet
// Создание синхронизированного набора позволяет безопасно использовать его в многопоточной среде.
// Если несколько потоков одновременно обращаются к HashSet и хотя бы один из них модифицирует его,
// это может привести к непредсказуемому поведению, таким как вызову исключений ConcurrentModificationException или несогласованию данных.
Set<String> synchronizedSet = synchronizedSet(new HashSet<>());
synchronizedSet.add("Synchronized Item");
synchronizedSet.add("Всем привет!");
System.out.println("Синхронизированный набор: " + synchronizedSet);
```

13. Сортировка множества

```
// Сортировка при помощи Stream
// Stream в Java — это последовательность элементов, поддерживающая последовательные и параллельные операции над ними.
// Потоки были введены в Java 8 как часть пакета java.util.stream и предоставляют функциональный подход к обработке коллекций и массивов.
set.add("Женек");
set.add("Санек");
set.add("Антоха");
set.add("Женек"); // Дубликат, не будет добавлен
List<String> sortedList = set.stream()
        .sorted()
        .toList().reversed();

// Вывод отсортированного списка
System.out.println("Отсортированный List с использованием Stream API: " + sortedList);
```

Коллекции. Set (Множество). *HashSet*: реализация. Основные команды. Часть 3

14. Добавление множества в MySQL при помощи лямбда-выражений с последующим сохранением в файл

```
public class ClonedSet {
    public static void main(String[] args) {
        // Параметры подключения к базе данных
        String url = "jdbc:mysql://localhost:3306/test1000";
        String username = "root";
        String password = "root";

        // Путь к выходному файлу
        String outputFile = "output.txt"; // файл будет создан в рабочем каталоге

        // Создаем HashSet и добавляем элементы
        HashSet<String> set = new HashSet<>();
        set.add("Саня");
        set.add("Гена");
        set.add("Антоха");

        // Клонируем набор
        HashSet<String> clonedSet = (HashSet<String>) set.clone();
```

Коллекции. Set (Множество). *HashSet*: реализация. Основные команды. Часть 4

14. Добавление множества в MySQL при помощи лямбда-выражений с последующим сохранением в файл

```
// Подключение к базе данных и вставка данных
try (Connection connection = DriverManager.getConnection(url, username, password)) {
    // Создаем таблицу, если ее нет
    String createTableSQL = "CREATE TABLE IF NOT EXISTS hashset_elements ("
            + "id INT AUTO_INCREMENT PRIMARY KEY, "
            + "element VARCHAR(255) NOT NULL"
            + ")";
    try (PreparedStatement createTableStmt = connection.prepareStatement(createTableSQL)) {
        createTableStmt.executeUpdate();
    }

    // Подготавливаем SQL-запрос для вставки
    String insertSQL = "INSERT INTO hashset_elements (element) VALUES (?)";
    try (PreparedStatement insertStmt = connection.prepareStatement(insertSQL)) {
        clonedSet.forEach(element -> {
            try {
                insertStmt.setString(1, element);
                insertStmt.executeUpdate();
                System.out.println("Вставлен элемент: " + element);
            } catch (SQLException e) {
                e.printStackTrace();
            }
        });
    }

    System.out.println("Все элементы успешно вставлены в базу данных.");
```

Коллекции. Set (Множество). *TreeSet*: теория

TreeSet – класс в Java, относящийся к пакету java.util и реализующий структуру данных множество. В TreeSet элементы хранятся в отсортированном порядке. Само множество создано на основе структуры красно-черного дерева (Red-Black tree). Все элементы в TreeSet уникальны, что делает его полезным для хранения данных, которые должны быть как уникальными, так и отсортированными. Соответственно, все данные хранятся внутри дерева.

Основные особенности:

1. Множество можно сортировать.

1.1. Элементы в TreeSet всегда хранятся в отсортированном порядке (т.е. в порядке «по возрастанию».

2. Элементы могут быть только уникальными, так как реализует интерфейс Set.

3. Данные хранятся внутри бинарного дерева.

Когда использовать HashSet?

Есть необходимость быстрого добавления данных и нет необходимости в сортировке и упорядочивании элементов.

Когда использовать TreeSet?

Есть необходимость в сортировке и упорядочивании элементов.

Коллекции. Set (Множество). *TreeSet*: реализация. Часть 1

1. Создание множества и добавление элементов

```
// Создание TreeSet
TreeSet<Integer> treeSet = new TreeSet<>(); // Пустое множество, по умолчанию сортирует элементы по возрастанию
TreeSet<Integer> treeSetWithComparator = new TreeSet<>(Comparator.reverseOrder()); // Создает пустое множество TreeSet, используя заданный компаратор для определения порядка элементов (реверс).
TreeSet<Integer> treeSetFromCollection = new TreeSet<>(Arrays.asList(14, 2, 3)); // Создает TreeSet, содержащий элементы из указанной коллекции, отсортированные по возрастанию.

// Добавление элементов в treeSet
treeSet.add(25);
treeSet.add(20);
treeSet.add(30);
treeSet.add(40);
treeSet.add(50);

System.out.println("Множество 'treeSet': " + treeSet);

// Добавление элементов в treeSetWithComparator
treeSetWithComparator.add(10);
treeSetWithComparator.add(20);
treeSetWithComparator.add(30);

System.out.println("Множество 'treeSetWithComparator': " + treeSetWithComparator);

// Добавление в treeSetFromCollection
treeSetFromCollection.add(10);
treeSetFromCollection.add(20);
treeSetFromCollection.add(30);

System.out.println("Множество 'treeSetFromCollection': " + treeSetFromCollection);
```

Коллекции. Set (Множество). *TreeSet*: реализация. Часть 2

2. Удаление элементов, проверка наличия элементов, получение первого и последнего элементов, извлечение первого и последнего элементов множества

```
// Удаление элементов из множества
treeSet.remove(20);

System.out.println("Множество после удаления элемента: " + treeSet);

// Проверка наличия элементов в множестве
boolean contains10 = treeSet.contains(10);
System.out.println("Элемент '10' есть в множестве? Ответ: " + contains10);

// Получение первого и последнего элемента
int firstElement = treeSet.first();
int lastElement = treeSet.last();
System.out.println("Первый элемент множества 'treeSet': " + firstElement + "\nПоследний элемент множества
'treeSet': " + lastElement);

// Извлечение и удаление первого и последнего элемента
int firstRemoved = treeSet.pollFirst();
int lastRemoved = treeSet.pollLast();
System.out.println("Первый удаленный элемент множества: " + firstRemoved + "\nПоследний удаленный элемент
множества: " + lastRemoved);
```

Коллекции. Set (Множество). *TreeSet*: реализация. Часть 3

3. Навигационные методы, получение подмножества, получение меньшего и бОльшего подмножества, итерация по элементам.

```
// Навигационные методы
treeSet.addAll(Arrays.asList(10, 15, 20, 25, 30));
System.out.println("Текущее множество: " + treeSet);
Integer higher = treeSet.higher(15); // Возвращает наименьший элемент, при этом, бОльший задаваемого числа
Integer lower = treeSet.lower(15); // Возвращает наибольший элемент, но меньший, чем задаваемое число
Integer ceiling = treeSet.ceiling(15); // Возвращает наименьший элемент, который больше или равен задаваемому числу
Integer floor = treeSet.floor(15); // Возвращает наибольший элемент, который меньше или равен задаваемому числу
System.out.println("Наименьший элемент множества среди чисел, которые больше 15: " + higher);
System.out.println("Наибольший элемент множества среди чисел, которые меньше 15: " + lower);
System.out.println("Наименьший элемент множества среди чисел, которые больше или равны 15: " + ceiling);
System.out.println("Наибольший элемент множества среди чисел, которые меньше или равны 15: " + floor);

// Получение подмножества
NavigableSet<Integer> subSet = treeSet.subSet(10, true, 25, false);
System.out.println("Получившееся подмножество: " + subSet);

// Получение меньшего и бОльшего подмножества
SortedSet<Integer> headSet = treeSet.headSet(20); // подмножество из начальных элементов множества
NavigableSet<Integer> tailSet = treeSet.tailSet(15, true); // подмножество из остальных элементов множества
System.out.println("Подмножество (< 20): " + headSet);
System.out.println("Подмножество (>= 15): " + tailSet);

// Итерация по элементам
System.out.println("Итерация элементов (ASC):");
Iterator<Integer> iterator = treeSet.iterator();
while (iterator.hasNext()) {
    System.out.print(iterator.next() + " ");
}
System.out.println();

System.out.println("Итерация элементов (DESC):");
Iterator<Integer> descendingIterator = treeSet.descendingIterator();
while (descendingIterator.hasNext()) {
    System.out.print(descendingIterator.next() + " ");
}
System.out.println();
```

Коллекции. Set (Множество). *TreeSet*: реализация. Часть 4

4. Проверка на пустоту, размер множества, очистка множества, сортировка множества, проверка на сортировку через компаратор, преобразование множества в массив.

```
// Проверка на пустоту и получение размера
boolean isEmpty = treeSet.isEmpty();
int size = treeSet.size();
System.out.println("Пустое множество? Ответ: " + isEmpty);
System.out.println("Размер множества: " + size);

// Очистка множества
treeSet.clear();
System.out.println("Пустое множество (после очистки): " + treeSet.isEmpty());

// Получение представления множества в обратном порядке
treeSet.addAll(Arrays.asList(5, 10, 15, 20, 25));
NavigableSet<Integer> descendingSet = treeSet.descendingSet();
System.out.println("Отсортированное в обратом порядке множество: " + descendingSet);

// Получение компаратора. Т.е. проверка на сортировку
Comparator<? super Integer> comparator = treeSet.comparator();
if (comparator == null) {
    System.out.println("TreeSet использует упорядочивание по умолчанию.");
} else {
    System.out.println("TreeSet использует кастомную сортивку.");
}

// Преобразование множества в массив
Object[] array = treeSet.toArray();
System.out.println("Преобразованное в массив множество: " + Arrays.toString(array));

Integer[] integerArray = treeSet.toArray(new Integer[0]);
System.out.println("Преобразованное в целочисленный массив множество: " + Arrays.toString(integerArray));
```

Коллекции. Очереди. *FIFO*: теория. Интерфейс Queue

Очередь — одна из фундаментальных структур данных, представляющая собой коллекцию элементов, организованных по принципу **FIFO.**

FIFO (First-In, First-Out) — принцип организации и управления данными, при котором первый элемент, поступивший в систему, обрабатывается первым, что напоминает очередь в реальной жизни: первый пришёл — первый обслужен. Реализовывается при помощи интерфейса **Queue**.

Основные особенности:

1. ***Порядок обработки элементов.*** Т.е. элементы обрабатываются в последовательности их поступления.
2. ***Добавление элементов в конец очереди.*** Т.е. новые элементы добавляются в конец структуры данных.
3. ***Удаление и извлечение элементов из начала очереди.*** Т.е. элементы удаляются и извлекаются по умолчанию из начала очереди.
4. **Отсутствие произвольного доступа.** Т.е. в классической реализации очереди нет возможности доступа к элементам по индексу или изменения порядка элементов.

Коллекции. Очереди. *FIFO*: теория. Интерфейс Queue. Где используется (основные моменты)?

1. Многопоточность

1.1. Интерфейс BlockingQueue. Используется для передачи данных между потоками с возможностью блокировки при добавлении или извлечении элементов. Необходим для избегания выброса исключений при одновременном использовании очереди несколькими потоками.

1.2. Класс ThreadPoolExecutor. Использует очереди для хранения задач, ожидающих выполнения потоками из пула.

2. Обработка событий и сообщений

2.1. Java Message Service (JMS). Использует очереди для асинхронной передачи сообщений между различными компонентами распределённых систем.

2.2. Обработка событий в GUI (Graphical User Interface). В **Swing** и **JavaFX** события пользователя ставятся в очередь **Event Dispatch Thread (EDT)** для последовательной обработки.

3. Планирование задач

3.1. Планировщики задач. Интерфес **ScheduledExecutorService** использует очереди с приоритетом для планирования выполнения задач в определённое время или с определённой задержкой.

3.2. Batch-обработка. В приложениях, где необходимо обработать большое количество задач пакетно, задачи ставятся в очередь и обрабатываются последовательно или параллельно.

4. **Machine learning & Deep learning**, большие данные.

4.1. Параллельная обработка задач — при обработке больших объёмов данных задачи могут ставиться в очередь для распределённой обработки на кластере.

4.2. Потоковая обработка данных — в таких фреймворках, как **Apache Flink** или **Apache Storm**, написанные на Java, очереди используются для управления потоком данных.

Коллекции. Очереди. *FIFO*. Интерфейс Queue. Реализация. Часть 1

1. Создание очереди.

```
// Создаём очередь
Queue<String> queue = new LinkedList<>();
```

2. Добавление элемента в конец очереди.

```
// Метод add(). Добавляет элемент в очередь. Вызывает исключение, если не удалось добавить элемент.
queue.add("Евгений");
System.out.println("Очередь после добавления первого элемента: " + queue);
```

3. Добавление элемента в конец очереди.

```
// offer(E). Добавляет элемент в конец очереди. Возвращает false, если не удалось добавить.
boolean offerResult = queue.offer("Антон");
System.out.println("Очередь после добавления еще одного элемента: " + queue + ". Элемент добавлен? Ответ: " + offerResult);
```

4. Возвращение (вывод) первого элемента.

```
// element(). Возвращает первый элемент очереди без удаления. Вызывает исключение, если очередь пуста.
String headElement = queue.element();
System.out.println("Первый элемент очереди element(): " + headElement);
```

5. Возвращение (вывод) первого элемента. Возвращает null, если очередь пуста.

```
// peek(). Возвращает первый элемент очереди без удаления или null, если очередь пуста.
String peekElement = queue.peek();
System.out.println("Первый элемент очереди peek(): " + peekElement);
```

6. Возвращение (вывод) первого элемента. Вызывает исключение, если очередь пуста.

```
// remove(): Удаляет и возвращает первый элемент очереди. Вызывает исключение, если очередь пуста.
String removedElement = queue.remove();
System.out.println("После удаления (remove()) элемента '" + removedElement + "', очередь стала следующей: " + queue);
```

7. Возвращение (вывод) первого элемента. Возвращает null, если очередь пуста.

```
// poll(): Удаляет и возвращает первый элемент очереди или null, если очередь пуста.
String polledElement = queue.poll();
System.out.println("После удаления (poll()) элемента '" + polledElement + "', очередь стала следующей: " + queue);
```

8. Вызываем исключения.

```
// Проверяем поведение методов при пустой очереди
try {
    queue.remove(); // Вызываем NoSuchElementException
} catch (Exception e) {
    System.out.println("Ошибка (исключение) при удалении (remove()) элемента из пустой очереди: " + e);
}
```

Коллекции. Очереди. *FIFO*. Интерфейс Queue. Реализация. Часть 2

9. Удаление в пустой очереди.

```
String nullPoll = queue.poll(); // Вернёт null
System.out.println("Удаление элемента методом 'poll()' в пустой очереди вернуло '" + nullPoll + "'");
```

10. Вызов исключения.

```
try {
    queue.element(); // Вызываем NoSuchElementException
} catch (Exception e) {
    System.out.println("Ошибка (исключение) при вызове элемента методом 'element()' в пустой очереди: " + e);
}
```

11. Удаление в пустой очереди.

```
String nullPeek = queue.peek(); // Вернёт null
System.out.println("Ошибка (исключение) при вызове элемента методом 'peek()' в пустой очереди: " + nullPeek);
```

12. Добавление нескольких элементов в очередь, проверка наличия элемента, вызов элемента очереди по индексу, очистка очереди.

```
//Добавление нескольких элементов в очередь
queue.addAll(Arrays.asList("Герман", "Валентин"));
System.out.println("После добавления элементов в очередь: " + queue);

//Проверка наличия элемента
boolean containsThird = queue.contains("Герман");
System.out.println("Очередь содержит имя " + queue.peek() + "? Ответ: " + containsThird);

// Вызов элемента по индексу напрямую невозможен
List<String> list = (List<String>) queue; // Преобразовываем очередь в список
String elementAtIndex = list.get(1); // Получаем элемент по индексу
System.out.println("Элемент с индексом 1: " + elementAtIndex);

//Очистка очереди
queue.clear();
System.out.println("Пустая очередь: " + queue);
```

Коллекции. Стек. *LIFO*: теория. Интерфейс Deque

Стек — абстрактная структура данных, которая работает по принципу **LIFO** (Last In, First Out), что переводится как «последним пришёл — первым вышел». Это означает, что последний добавленный элемент будет извлечён первым. Стек можно представить как стопку книг: вы кладёте книги одну на другую, и чтобы взять книгу, нужно снять верхнюю.

LIFO (англ. **Last In, First Out** — «последним пришёл, первым вышел») —принцип организации и управления данными, при котором последний добавленный элемент будет извлечён первым. Реализовывается при помощи интерфейса **Deque**.

Основная особенность: ***порядок обработки элементов.*** Т.е. последний вошел, первый ушел: последний добавленный элемент извлекается первым.

Применяется при управлении памятью и вызове методов, при «обходе» графов и деревьев, а также в браузерах при реализации навигации и при вызове исключений.

Коллекции. Стек. *LIFO*. Интерфейс Deque. Реализация. Часть 1

1. Добавление элементов.

```
// Используем метод push()
stack.push("Евгений");
stack.push("Давид");
stack.push("Антон");
System.out.println("Стек после добавления: " + stack);

// Используем метод addFirst()
stack.addFirst("Александр");
System.out.println("После добавления: " + stack);

// Используем метод offerFirst()
stack.offerFirst("Егор");
System.out.println("После добавления: " + stack);

/* Разница между данными методами
 * push - генерирует IllegalStateException, если Deque заполнен
 * addFirst - генерирует IllegalStateException, если Deque заполнен
 * offerFirst - возвращает false, если элемент не может быть
добавлен из-за переполнения.
 */
```

2. Извлечение элементов верхнего уровня.

```
// Используем peek()
System.out.println("peek(): " + stack.peek());

// Используем peekFirst()
System.out.println("peekFirst(): " + stack.peekFirst());

// Используем getFirst()
try {
    System.out.println("Вызов элемента методом getFirst(): " + stack.getFirst());
} catch (NoSuchElementException e) {
    System.out.println("Стек пустой при вызове getFirst()");
}
```

3. Проверка состояния стека.

```
// Проверяем, содержит ли стек определенный элемент
System.out.println("Содержит ли стек элемент 'Евгений'? " + stack.contains("Евгений"));

// Размер стека
System.out.println("Размер стека: " + stack.size());

// Проверка, пуст ли стек
System.out.println("Стек пуст? " + stack.isEmpty());
```

4. Итерация стека.

```
// Перебор элементов стека от вершины к основанию
System.out.println("Итерация от вершины к основанию:");
Iterator<String> iterator = stack.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
// Перебор элементов стека от основания к вершине
System.out.println("Итерация от основания к вершине:");
Iterator<String> descendingIterator = stack.descendingIterator();
while (descendingIterator.hasNext()) {
    System.out.println(descendingIterator.next());
}
```

Коллекции. Стек. *LIFO*. Интерфейс Deque. Реализация. Часть 2

5. Удаление и получение удаленных элементов из стека.

```
// Используем метод pop()
System.out.println("Удаленный элемент методом pop(): " + stack.pop());
System.out.println("Стек после удаления: " + stack);
// Используем метод removeLast()
System.out.println("Удаленный элемент методом removeLast(): " + stack.removeLast());
System.out.println("Стек после удаления: " + stack);

// Используем метод removeFirst()
try {
    System.out.println("Удаленный элемент методом removeFirst(): " + stack.removeFirst());
} catch (NoSuchElementException e) {
    System.out.println("Стек пустой при вызове removeFirst()");
}
System.out.println("Стек после удаления: " + stack);

// Используем метод pollFirst()
String polledElement = stack.pollFirst();
if (polledElement != null) {
    System.out.println("Удаленный элемент методом pollFirst(): " + polledElement);
} else {
    System.out.println("Стек пустой при вызове pollFirst()");
}
System.out.println("Стек после удаления: " + stack);

// Используем для очистки метод clear()
stack.clear();
System.out.println("После clear: " + stack);

// Проверяем, пуст ли стек после очистки
System.out.println("Стек пуст после clear()? " + stack.isEmpty());
```

Коллекции. Стек. *LIFO*. Интерфейс Deque. Реализация. Часть 3

6. Получение элементов из пустого стека.

```
// Используем метод peek() на пустом стеке
System.out.println("Получение элемента методом peek() на пустом стеке: " + stack.peek());

// Используем метод pop() на пустом стеке
try {
    System.out.println("Получение элемента с удалением методом pop() на пустом стеке: " + stack.pop());
} catch (NoSuchElementException e) {
    System.out.println("Исключение при вызове pop() на пустом стеке: " + e);
}

// Используем метод getFirst() на пустом стеке
try {
    System.out.println("Получение элемента методом getFirst() на пустом стеке: " + stack.getFirst());
} catch (NoSuchElementException e) {
    System.out.println("Исключение при вызове getFirst() на пустом стеке: " + e);
}
```

Коллекции. Приоритетная очередь. Класс PriorityQueue. Теория

Очередь с приоритетом — это абстрактная структура данных, в которой каждый элемент имеет определённый приоритет. В отличие от обычной очереди (FIFO — First In, First Out), где элементы обрабатываются в порядке их поступления, очередь с приоритетом организует элементы таким образом, что элемент с наивысшим приоритетом извлекается первым, независимо от времени его добавления.

Основные особенности:

1. ***Естественный порядок или компаратор.*** Т.е. элементы должны реализовывать интерфейс Comparable или должен быть предоставлен Comparator для определения порядка приоритетов. По умолчанию приоритет выставляется на возрастание.
2. ***Не допускает null элементов.*** Т.е. попытка добавить null вызовет исключение NullPointerException.
3. ***Не гарантирует порядок для одинаковых приоритетов.*** Т.е. если элементы имеют одинаковый приоритет, порядок их извлечения не гарантируется.
4. ***Непотокобезопасна.*** Т.е. для многопоточных приложений рекомендуется использовать PriorityBlockingQueue из пакета java.util.concurrent.

Коллекции. Приоритетная очередь. Класс PriorityQueue. Реализация. Часть 1

1. Общая реализация.

```
import java.util.Arrays;
import java.util.PriorityQueue;

public class Main {
    public static void main(String[] args) {
        // Создание очереди с приоритетом
        PriorityQueue<Integer> queue = new PriorityQueue<>();

        // Добавление элементов
        queue.offer(10);
        queue.offer(20);
        queue.offer(15);

        // Просмотр начала очереди
        System.out.println("Начало очереди (peek): " + queue.peek()); // 10

        // Проверка содержимого
        System.out.println("Содержит элемент 20? " + queue.contains(20)); //
true

        // Размер очереди
        System.out.println("Размер очереди: " + queue.size()); // 3

        // Итерация по очереди
        System.out.println("Элементы очереди:");
        for (Integer num : queue) {
            System.out.println(num);
        }
        // Удаление элементов
        System.out.println("Удалено: " + queue.poll()); // 10
        System.out.println("Удалено: " + queue.remove()); // 15

        // Добавление коллекции элементов
        queue.addAll(Arrays.asList(5, 25, 30));

        // Проверка, содержит ли очередь все элементы из другой коллекции
        System.out.println("Содержит все элементы [5, 25]? " +
queue.containsAll(Arrays.asList(5, 25))); // true

        // Удаление всех элементов из указанной коллекции
        queue.removeAll(Arrays.asList(5, 25));
        System.out.println("После removeAll: " + queue);

        // Удаление всех элементов, кроме указанных в коллекции
        queue.retainAll(Arrays.asList(20, 30));
        System.out.println("После retainAll: " + queue);

        // Очистка очереди
        queue.clear();
        System.out.println("Очередь пуста? " + queue.isEmpty()); // true
    }
}
```

Коллекции. Приоритетная очередь. Класс PriorityQueue. Реализация. Часть 2

2. Очередь с приоритетом на основе чисел.

```
import java.util.PriorityQueue;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Integer> queue = new PriorityQueue<>();

        // Добавление элементов
        queue.add(50);
        queue.add(1);
        queue.add(3);
        queue.add(2);
        queue.add(4);

        // Извлечение элементов
        while (!queue.isEmpty()) {
            System.out.println(queue.poll());
        }
    }
}
```

3. Очередь с приоритетом на основе длины строки.

```
import java.util.Comparator;
import java.util.PriorityQueue;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<String> queue = new
PriorityQueue<>(Comparator.comparingInt(String::length).thenComparing(C
omparator.naturalOrder())); // первичное условие comparingInt -
сортируем по длине строк; вторичное условие naturalOrder - сортировка
элементов по возрастанию в том случае, если элементы одинаковой
длины

        // Добавление элементов
        queue.add("Евгений");
        queue.add("Давид");
        queue.add("Игорь");
        queue.add("Антон");
        queue.add("Александр");

        // Извлечение элементов
        while (!queue.isEmpty()) {
            System.out.println(queue.poll());
        }
    }
}
```

Коллекции. Приоритетная очередь. Класс PriorityQueue. Реализация. Часть 3

4. Очередь с выставлением собственного приоритета.

```
import java.util.PriorityQueue;
import java.util.Comparator;

/**
 * @param priority Чем меньше значение, тем выше приоритет
 * Record - это специальный тип классов, предназначенный для хранения неизменяемых данных, например, в виде подобия конструктора класса.
 * Основное преимущество - не нужно писать геттеры и сеттеры, а также поля по умолчанию имеют модификатор доступа private и модификатор класса final, что обеспечивает неизменность объектов после создания.
 */
record Task(String name, int priority) {
}

public class Main {
    public static void main(String[] args) {
        // Создаем компаратор для сравнения задач по приоритету
        Comparator<Task> taskComparator = Comparator.comparingInt(Task::priority);

        // Создаем очередь с приоритетом с использованием компаратора
        PriorityQueue<Task> taskQueue = new PriorityQueue<>(taskComparator);

        // Добавляем задачи
        taskQueue.add(new Task("Написать отчет", 2));
        taskQueue.add(new Task("Проверить почту", 1));
        taskQueue.add(new Task("Подготовить презентацию", 3));

        // Извлекаем задачи в порядке приоритета
        while (!taskQueue.isEmpty()) {
            Task task = taskQueue.poll(); // poll() используется для извлечения и удаления задачи с наивысшим приоритетом из очереди
            System.out.println("Выполняется задача '" + task.name() + "' с приоритетом " + task.priority());
        }
    }
}
```

Потоки (многопоточность). Теория. Часть 1. Основные понятия и определения

Многопоточность представляет собой механизм выполнения нескольких потоков одновременно в рамках одного процесса. **Многопоточность** используется для улучшения производительности и параллельной обработки задач.

Основные понятия:

1. Процесс (Process) представляет собой экземпляр программы, который выполняется в операционной системе. Каждый процесс имеет свое собственное адресное пространство. Процессы изолированы друг от друга.
2. Поток (Thread) представляет собой подпроцесс, который выполняется в рамках одного процесса. Потоки одного процесса делят общее адресное пространство, что позволяет им эффективно взаимодействовать, но требует синхронизации для предотвращения конфликтов.
3. Многопоточность (Multithreading). Используется для выполнения параллельных задач, таких как обработка ввода/вывода, вычисления, обработка данных, взаимодействие с пользователем и т. д.

Потоки (многопоточность). Теория. Часть 2. Основные преимущества многопоточности

1. Параллелизм. Потоки позволяют выполнять несколько задач одновременно, что особенно полезно, например, при обработке больших данных.

2. Повышение производительности. Разделение задач между потоками помогает максимально эффективно использовать ресурсы процессора. Например, один поток можно распараллелить на два ядра процессора.

3. Быстрый отклик. Многопоточность делает приложения более отзывчивыми, например, в графическом интерфейсе пользователя (GUI).

4. Эффективное использование ресурсов. Потоки могут работать в режиме ожидания при выполнении операций ввода/вывода, позволяя другим потокам использовать процессор.

Потоки (многопоточность). Теория. Часть 3. Модель многопоточности в Java

1. Класс Thread. В Java многопоточность реализована через класс Thread, который представляет поток выполнения подпроцесса. Для создания нового потока необходимо либо создать класс наследник от класса Thread, либо использовать интерфейс Runnable.

2. Интерфейс Runnable. Интерфейс Runnable задает метод run(), который содержит код, который необходимо выполнить в потоке. Этот подход позволяет избежать ограничений множественного наследования, так как Java не поддерживает наследование более одного класса.

3. Пул потоков (Thread Pool). В Java предоставлен механизм для управления потоками через пулы потоков. Это снижает ресурсы на создание и уничтожение потоков, в любых проектах, но особенно в высоконагруженных.

Потоки (многопоточность). Теория. Часть 4. Жизненный цикл потока (1)

1. Создание потока – начальная стадия жизненного цикла подразумевает, что поток создан за счет инициализации нового экземпляра класса и существует как объект класса Thread(). Метод «start()» еще не вызван, как следствие, поток не выполняет никаких операций.

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Поток выполняется.");
        });
        System.out.println("Поток создан, но не запущен.");

    }
}
```

2. Запуск потока – вторая стадия жизненного цикла подразумевает, что после вызова метода «start()» поток переходит в состояние выполнения («Runnable»). Т.е. поток готов к выполнению, но само выполнение зависит от планировщика потоков и от загруженности процессора.

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Поток выполняется."); // Задача внутри потока выполняется после того, как мы запустим сам поток
        });
        thread.start(); // Запуск потока
        System.out.println("Поток в состоянии Runnable."); // Сообщение об успешном запуске потока
    }
}
```

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Потоки (многопоточность). Теория. Часть 4. Жизненный цикл потока (2)

3. Блокировка или «заморозка» потока – третья стадия жизненного цикла подразумевает, что поток может быть как заблокирован («заморожен»). Это значит, что поток временно приостановлен или не выполняется. Основные причины:

- ожидание завершения другого потока (метод *join()*);
- ожидание сигнала (методы wait() и notify());
- выполнение метода sleep() для приостановки на заданное время.

Далее поток возвращается в рабочее состояние, т.е. в состояние «Runnable», выполняет заданный код.

```
public class Main {

    // Когда программа запускается, JVM создает главный поток для выполнения метода main(). Этот поток называется main.
    public static void main(String[] args) throws InterruptedException {
        // Главный поток начинает выполнение программы
        System.out.println("Главный поток main(): запуск Потока 1."); // Вывод из главного потока

        // Пример с join()
        // Создаем поток 1
        Thread thread1 = new Thread(() -> System.out.println("Поток 1 выполнился."));
        thread1.start(); // Поток 1 запускается и выполняет свой код
        thread1.join(); // Главный поток останавливается и ждет завершения потока 1
        System.out.println("Главный поток main(): дождался завершения потока 1."); // После завершения Потока 1 главный поток продолжает выполнение

        // Создаем поток 2
        // Пример с sleep()
        System.out.println("Главный поток main(): запуск потока 2."); // Сообщаем, что запускаем Поток 2
        Thread thread2 = new Thread(() -> {
            System.out.println("Поток 2 'замораживается (приостанавливается)' на 1 секунду."); // Поток 2 сообщает, что засыпает
            try {
                Thread.sleep(1000); // Поток 2 приостанавливается на 1 секунду
            } catch (InterruptedException e) {
                e.printStackTrace(); // Обрабатываем исключение, если поток прерывается
            }
            System.out.println("Поток 2 снова работает."); // Поток 2 просыпается и продолжает выполнение
        });
        thread2.start(); // Запускаем поток 2
```

Потоки (многопоточность). Теория. Часть 4. Жизненный цикл потока (3)

```
    // Создаем общий объект lock класса Object, который используется как средство синхронизации между потоками.
    final Object lock = new Object();
    System.out.println("Главный поток: запуск потоков 3 и 4."); // Сообщаем, что запускаем потоки 3 и 4

    // Поток 3 - ожидающий сигнал
    Thread thread3 = new Thread(() -> {
        synchronized (lock) {
            try {
                System.out.println("Поток 3: Ждет сигнала."); // Поток 3 сообщает, что ждет
                lock.wait(); // Поток 3 приостанавливается и ждет сигнала на объекте lock
                System.out.println("Поток 3: Получил сигнал."); // После получения сигнала поток 3 продолжает выполнение
            } catch (InterruptedException e) {
                e.printStackTrace(); // Обрабатываем исключение
            }
        }
    });

    // Поток 4 - отправляющий сигнал
    Thread thread4 = new Thread(() -> {
        synchronized (lock) {
            System.out.println("Поток 4: отправляет сигнал Потоку 3."); // Поток 4 сообщает, что отправляет сигнал
            lock.notify(); // Поток 4 отправляет сигнал для разблокировки потока 3
        }
    });

    thread3.start(); // Запускаем Поток 3
    Thread.sleep(500); // Главный поток приостанавливается на 0.5 секунды, чтобы поток 3 успел начать ожидание
    thread4.start(); // Запускаем Поток 4, который отправит сигнал Потоку 3

    // Завершение программы
    // Главный поток завершает выполнение своих инструкций, но программа не завершится, пока все дочерние потоки (thread3 и thread4) не завершат
выполнение.
    System.out.println("Главный поток завершает работу."); // Сообщаем, что главный поток завершает свою работу, так как основной код выполнен
}
```

Потоки (многопоточность). Теория. Часть 4. Жизненный цикл потока (4)

4. «Уничтожение» или завершение потока – финальная стадия жизненного цикла подразумевает, что поток завершает выполнение кода, помещенного внутрь потока. Соответственно, поток больше не может быть перезапущен после завершения. После завершения потока создается новый.

```
public class Main {

    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Поток выполняется.");
        });

        thread.start();

        try {
            thread.join(); // Ожидание завершения потока
            System.out.println("Поток завершен.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

Потоки (многопоточность). Теория. Часть 4. Жизненный цикл потока (5)

Пример вывода состояний потока

```
public class Main {

    public static void main(String[] args) {
        // Создаем поток (состояние NEW)
        Thread thread = new Thread(() -> {
            System.out.println("Поток выполняется (RUNNING).");
            try {
                // Приостанавливаем поток на 2 секунды (BLOCKED)
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Поток завершен (TERMINATED).");
        });

        // Поток находится в состоянии NEW
        System.out.println("Состояние потока: " + thread.getState()); // NEW

        // Запускаем поток (переход в RUNNABLE)
        thread.start();
        System.out.println("Состояние потока после запуска: " +
thread.getState()); // RUNNABLE
        // Проверяем состояние потока в процессе выполнения
        try {
            Thread.sleep(500); // Даем время потоку начать выполнение
            System.out.println("Состояние потока в процессе
выполнения: " + thread.getState());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Ожидаем завершения потока
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Поток завершен
        System.out.println("Состояние потока после завершения: " +
thread.getState()); // TERMINATED
    }
}
```

Потоки (многопоточность). Практика. Часть 1. Жизненный цикл потока

Пример применения многопоточности при внесении больших данных в MySQL (1)

```
import java.io.BufferedReader; // Для построчного чтения файла
import java.io.FileReader; // Для открытия текстового файла
import java.sql.Connection; // Для работы с соединением к базе данных
import java.sql.DriverManager; // Для управления соединением к базе данных
import java.sql.PreparedStatement; // Для выполнения SQL-запросов с параметрами
import java.sql.Statement; // Для выполнения SQL-запросов без параметров
import java.util.ArrayList; // Для использования динамических массивов
import java.util.List; // Для работы со списками
import java.util.concurrent.atomic.AtomicInteger; // Потокобезопасный счетчик

public class Main {

    private static final String DB_URL = "jdbc:mysql://localhost:3306/newdata"; // URL для подключения к базе данных
    private static final String DB_USER = "root"; // Имя пользователя базы данных
    private static final String DB_PASSWORD = "root"; // Пароль пользователя базы данных
    private static final int THREAD_COUNT = 14; // Количество потоков для обработки данных
    private static final AtomicInteger progressCounter = new AtomicInteger(0); // Потокобезопасный счетчик прогресса

    public static void main(String[] args) throws Exception {
        // Создаем таблицу, если она не существует
        createTableIfNotExists();

        // Указываем путь к CSV-файлу
        String filePath = "C:\\Users\\teelx\\IdeaProjects\\untitled10\\src\\updated_data_for_lstm.csv";
```

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Потоки (многопоточность). Практика. Часть 2. Жизненный цикл потока

Пример применения многопоточности при внесении больших данных в MySQL (2)

```
// Читаем данные из файла в список строковых массивов
List<String[]> data = readDataFromFile(filePath);

// Определяем общее количество строк
int totalRows = data.size();

// Выводим общее количество строк в консоль
System.out.printf("Всего строк для обработки: %d%n",
totalRows);

// Рассчитываем размер партиции (количество строк на
поток)
int partitionSize = (int) Math.ceil((double) totalRows /
THREAD_COUNT);

// Создаем список для хранения партиций данных
List<List<String[]>> partitions = new ArrayList<>();

// Разделяем данные на партиции и выводим диапазоны
строк для каждого потока
for (int i = 0; i < THREAD_COUNT; i++) {
    int start = i * partitionSize; // Начальная строка для
текущей партиции
    int end = Math.min(start + partitionSize, totalRows); //
Конечная строка для текущей партиции
    partitions.add(data.subList(start, end)); // Добавляем
партицию в список
    System.out.printf("Поток %d обрабатывает строки с %d до
%d%n", i + 1, start, end); // Выводим диапазон
}
```

(3)

```
    // Фиксируем время начала вставки данных
    long startTime = System.currentTimeMillis();

    // Создаем список потоков
    List<Thread> threads = new ArrayList<>();

    // Создаем и запускаем потоки для вставки данных
    for (List<String[]> partition : partitions) {
        Thread thread = new Thread(() -> {
            try {
                insertDataIntoDB(partition, totalRows); // Вставляем данные из текущей партиции
            } catch (Exception e) {
                e.printStackTrace(); // Выводим ошибку, если она возникла
            }
        });
        threads.add(thread); // Добавляем поток в список
        thread.start(); // Запускаем поток
    }

    // Ожидаем завершения всех потоков
    for (Thread thread : threads) {
        thread.join();
    }

    // Фиксируем время окончания вставки данных
    long endTime = System.currentTimeMillis();

    // Сообщаем об успешной вставке данных
    System.out.println("Данные успешно добавлены в базу данных.");

    // Выводим общее время выполнения
    System.out.printf("Общее время выполнения: %.2f секунд.%n", (endTime - startTime) / 1000.0);
}
```

Потоки (многопоточность). Практика. Часть 3. Жизненный цикл потока

Пример применения многопоточности при внесении больших данных в MySQL (4)

```
// Метод для создания таблицы в базе данных, если она отсутствует
private static void createTableIfNotExists() throws Exception {
    String createTableSQL = """
        CREATE TABLE IF NOT EXISTS datagas (
            id INT AUTO_INCREMENT PRIMARY KEY,
            pressure DOUBLE,
            temperature DOUBLE,
            flow_rate DOUBLE,
            external_temp DOUBLE,
            humidity DOUBLE,
            target_pressure DOUBLE,
            mass_in DOUBLE,
            date_and_time VARCHAR(255)
        );
    """;

    try (Connection connection = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD); // Устанавливаем соединение
         Statement statement = connection.createStatement()) { // Создаем объект для выполнения SQL-запроса
        statement.execute(createTableSQL); // Выполняем SQL-запрос
        System.out.println("Таблица 'datagas' проверена или создана.");
    }
}
```

Потоки (многопоточность). Практика. Часть 4. Жизненный цикл потока

Пример применения многопоточности при внесении больших данных в MySQL (5)

```
// Метод для чтения данных из CSV-файла
private static List<String[]> readDataFromFile(String filePath) throws Exception {
    List<String[]> data = new ArrayList<>(); // Список для хранения данных
    try (BufferedReader br = new BufferedReader(new FileReader(filePath))) { // Открываем файл для чтения
        String line;
        br.readLine(); // Пропускаем первую строку (заголовок)
        while ((line = br.readLine()) != null) { // Читаем файл построчно
            data.add(line.split(",")); // Разделяем строки на столбцы и добавляем в список
        }
    }
    return data; // Возвращаем список данных
}
```

(6)

```
// Метод для вставки данных в базу данных
private static void insertDataIntoDB(List<String[]> partition, int totalRows) throws Exception {
    try (Connection connection = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD)) { // Устанавливаем соединение
        String sql = "INSERT INTO datagas (pressure, temperature, flow_rate, external_temp, humidity, target_pressure, mass_in, date_and_time) VALUES (?, ?, ?, ?, ?, ?, ?, ?)"; // SQL-запрос для вставки данных
        try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) { // Подготавливаем запрос
            for (String[] row : partition) {
                preparedStatement.setDouble(1, Double.parseDouble(row[0])); // Устанавливаем значение для pressure
                preparedStatement.setDouble(2, Double.parseDouble(row[1])); // Устанавливаем значение для temperature
                preparedStatement.setDouble(3, Double.parseDouble(row[2])); // Устанавливаем значение для flow_rate
                preparedStatement.setDouble(4, Double.parseDouble(row[3])); // Устанавливаем значение для external_temp
                preparedStatement.setDouble(5, Double.parseDouble(row[4])); // Устанавливаем значение для humidity
                preparedStatement.setDouble(6, Double.parseDouble(row[5])); // Устанавливаем значение для target_pressure
                preparedStatement.setDouble(7, Double.parseDouble(row[6])); // Устанавливаем значение для mass_in
                preparedStatement.setString(8, row[7]); // Устанавливаем значение для date_and_time
                preparedStatement.addBatch(); // Добавляем запрос в пакет для ускоренного вставления данных в БД

                // Обновляем прогресс вставки
                int progress = progressCounter.incrementAndGet();
                if (progress % 1000 == 0 || progress == totalRows) { // Каждые 1000 строк или в конце
                    System.out.printf("Прогресс: %.2f%%%n", (progress / (double) totalRows) * 100);
                }
            }
            int[] results = preparedStatement.executeBatch(); // Выполняем пакет запросов
            System.out.printf("Поток обработал %d строк.%n", results.length); // Выводим количество обработанных строк
        }
    }
}
```

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Потоки (многопоточность). Практика. Жизненный цикл потока - заключение

Таким образом, **многопоточность** является гибким и эффективным инструментом для ускорения выполнения задач, которые могут быть разделены на независимые части. Например, в контексте работы с базами данных многопоточность может значительно улучшить производительность программы путем повышения скорости обработки и внесения информации, особенно при обработке больших объемов данных. Количество потоков выставляйте в зависимости от ресурсов ПК/сервера.

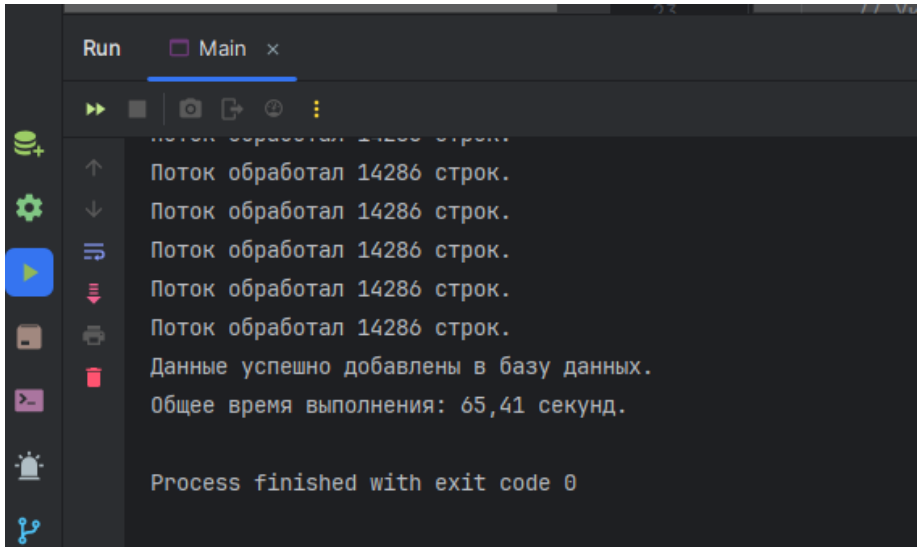


Рисунок 1 – Пример добавления данных с использованием 14-ти потоков (200 тысяч строк). Если использовать 94 потока, то время добавления снижается до 12.5 секунд.

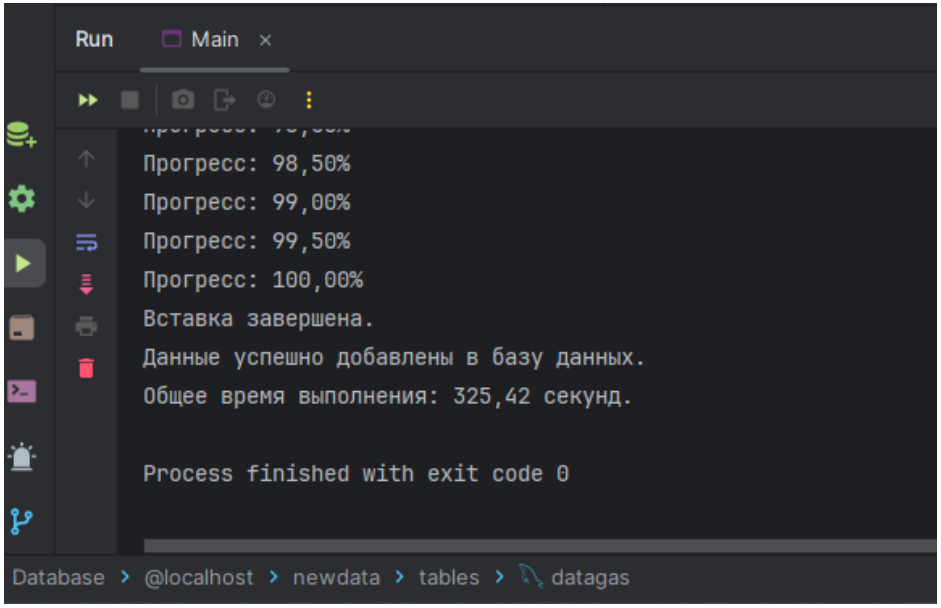


Рисунок 2 – Пример добавления данных (200 тысяч строк) через обычный цикл без использования многопоточности

Основы Java: основные конструкции. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №1

Выполняем только сложный вариант.

Сложный вариант.* **Задача №1**

Базовый вариант. Реализовать программу для выполнения следующих математических операций с целочисленным, байтовым и вещественным типами данных: сложение, вычитание, умножение, деление, деление по модулю (остаток), модуль числа, возведение в степень. Все данные вводятся с клавиатуры (класс Scanner, System.in, nextint).

Сложный вариант. Целочисленные, байтовые и вещественные типы данных. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задания полностью идентичны базовому варианту. При этом в программе данные пункты должны называться следующим образом:

1. Вывести все таблицы из MySQL.
2. Создать таблицу в MySQL.
3. Сложение чисел, результат сохранить в MySQL с последующим выводом в консоль.
4. Вычитание чисел, результат сохранить в MySQL с последующим выводом в консоль.
5. Умножение чисел, результат сохранить в MySQL с последующим выводом в консоль.
6. Деление чисел, результат сохранить в MySQL с последующим выводом в консоль.
7. Деление чисел по модулю (остаток), результат сохранить в MySQL с последующим выводом в консоль.
8. Возведение числа в модуль, результат сохранить в MySQL с последующим выводом в консоль.
9. Возведение числа в степень, результат сохранить в MySQL с последующим выводом в консоль.
10. Сохранить все данные (вышеполученные результаты) из MySQL в Excel и вывести на экран.

Основы Java: основные конструкции. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №2

Выполняем только сложный вариант.

Задача №2

Базовый вариант. Ввести две строки (не менее 50 символов каждая) с клавиатуры. Необходимо вывести на экран две введенных ранее строки, подсчитать и вывести размер длины каждой строки, объединить данные строки в одну, сравнить данные строки и результат сравнения вывести на экран.

Сложный вариант. Строковый тип данных. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задания полностью идентичны базовому варианту. При этом в программе данные пункты должны называться следующим образом:

1. Вывести все таблицы из MySQL.
2. Создать таблицу в MySQL.
3. Ввести две строки с клавиатуры, результат сохранить в MySQL с последующим выводом в консоль.
4. Подсчитать размер ранее введенных строк, результат сохранить в MySQL с последующим выводом в консоль.
5. Объединить две строки в единое целое, результат сохранить в MySQL с последующим выводом в консоль.
6. Сравнить две ранее введенные строки, результат сохранить в MySQL с последующим выводом в консоль.
7. Сохранить все данные (вышеполученные результаты) из MySQL в Excel и вывести на экран.

Основы Java: основные конструкции. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №3

Выполняем только сложный вариант.

Задача №3

Сложный вариант.*

Базовый вариант. Необходимо с клавиатуры задать несколько чисел. Реализовать программу, проверяющую каждое число на целостность (т.е. целое число или нет) и четность. При этом необходимо реализовать условие на проверку целостности числа (т.е. если пользователь вводит не целое число или вообще не число, то сообщать ему об ошибке).

Целочисленный тип данных. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задания полностью идентичны базовому варианту. При этом в программе данные пункты должны называться следующим образом:

1. Вывести все таблицы из MySQL.
2. Создать таблицу в MySQL.
3. Выполнение задачи базового варианта, результат сохранить в MySQL с последующим выводом в консоль.
4. Сохранить все данные (вышеполученные результаты) из MySQL в Excel и вывести на экран.

Основы Java: основные конструкции. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №4

Выполняем только сложный вариант.

Задача №4 (Контрольная работа №1)

Сложный вариант.*

Базовый вариант. Ввести две строки (не менее 50 символов каждая) с клавиатуры. Необходимо вывести на экран две введенных ранее строки, вернуть подстроку по индексам (substring()), перевести все строки в верхний и нижний регистры, найти подстроку (тоже вводить с клавиатуры) и определить: заканчивается ли строка данной подстрокой (endsWith()).

Сложный вариант. Строковый тип данных. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задания полностью идентичны базовому варианту. При этом в программе данные пункты должны называться следующим образом:

1. Вывести все таблицы из MySQL.
2. Создать таблицу в MySQL.
3. Возвращение подстроки по индексам, результат сохранить в MySQL с последующим выводом в консоль.
4. Перевод строк в верхний и нижний регистры, результат сохранить в MySQL с последующим выводом в консоль.
5. Поиск подстроки и определение окончания подстроки, результат сохранить в MySQL с последующим выводом в консоль.
6. Сохранить все данные (вышеполученные результаты) из MySQL в Excel и вывести на экран.

Основы Java: основные конструкции. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №5

Выполняем только сложный вариант.

Задача №5

Сложный вариант. *

Базовый вариант. Строковый тип данных (класс StringBuffer). Ввести две строки (не менее 50 символов каждая) с клавиатуры. Необходимо вывести на экран две введенных ранее строки, изменить порядок символов на обратный и добавить одну строку в другую.

Сложный вариант. Строковый тип данных. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задачи полностью идентичны базовому варианту.

1. Вывести все таблицы из MySQL.
2. Создать таблицу в MySQL.
3. Изменить порядок символов строки на обратный, результат сохранить в MySQL с последующим выводом в консоль.
4. Добавить одну строку в другую, результат сохранить в MySQL с последующим выводом в консоль.
5. Сохранить все данные (вышеполученные результаты) из MySQL в Excel и вывести на экран.

Базовые принципы в ООП Java. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №6

Выполняем только сложный вариант.

Задача №6

Сложный вариант. *

Базовый вариант. Объектно-ориентированное программирование. Необходимо создать класс ArrayPI (модификатор доступа – public), в котором необходимо создать два двухмерных массива (ввод с клавиатуры, 7 столбцов и 7 строк). Далее необходимо создать класс-наследник Matrix (модификатор доступа – public final), в котором будут наследоваться данные матрицы и перемножаться. Необходимо перемножить две данных матрицы и итоговую матрицу (произведение двух матриц) вывести на экран.

Сложный вариант. Объектно-ориентированное программирование. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задачи полностью идентичны базовому варианту. Каждый пункт меню должен быть отдельным классом-наследником (подклассом).

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL.
3. Ввести две матрицы с клавиатуры и каждую из них сохранить в MySQL с последующим выводом в консоль.
4. Перемножить матрицу, сохранить перемноженную матрицу в MySQL и вывести в консоль.
5. Сохранить результаты из MySQL в Excel и вывести их в консоль.

Семинарское занятие №7

Выполняем только сложный вариант.

Задача №7

Сложный вариант. *

Базовый вариант. Объектно-ориентированное программирование. Необходимо создать класс ArrayPI (модификатор доступа – public), в котором необходимо создать одномерный массив (ввод с клавиатуры, 35 элементов). Далее необходимо создать класс-наследник Sort (модификатор доступа – public final), в котором будет наследоваться и сортироваться введенный ранее массив. Необходимо отсортировать введенный ранее массив по возрастанию и убыванию методом пузырьковой сортировки.

Сложный вариант. Объектно-ориентированное программирование. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задачи полностью идентичны базовому варианту. Каждый пункт меню должен быть отдельным классом-наследником (подклассом).

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL.
3. Ввести одномерный массив и сохранить в MySQL с последующим выводом в консоль.
4. Отсортировать массив и сохранить в MySQL с последующим выводом в консоль.
5. Сохранить результаты из MySQL в Excel и вывести их в консоль.

Базовые принципы в ООП Java. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №8

Выполняем только сложный вариант.

Задача №8 (Контрольная работа №2)

Сложный вариант. *

Базовый вариант. Объектно-ориентированное программирование. Сделайте абстрактный класс Student, в котором будут следующие поля (private): name (имя), age (возраст). Также создайте public-методы setName, getName, setAge, getAge для передачи и присвоения значений в классе-наследнике Worker. В данном классе реализуйте метод, который вносит дополнительное private-поле salary (зарплата), а также методы public getSalary и setSalary для передачи и присвоения значений зарплаты. На выходе мы должны получить имя и возраст студента, а также его предполагаемую зарплату.

Сложный вариант. Объектно-ориентированное программирование. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задачи полностью идентичны базовому варианту. Каждый пункт меню должен быть отдельным классом-наследником (подклассом).

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL.
3. Ввод с клавиатуры значений ВСЕХ полей, сохранить их в MySQL с последующим выводом в консоль.
4. Сохранение всех результатов в MySQL с последующим выводом в консоль.
5. Сохранить результаты из MySQL в Excel и вывести их в консоль.

Базовые принципы в ООП Java. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №8

За данную задачу дополнительно начисляется 4.5 балла к уже существующим баллам

Полухардкорный (полуспециальный) вариант. Задача без порядкового номера.

Объектно-ориентированное программирование и большие данные. Дан файл Excel, в котором хранится больше миллиона значений с данными по гидрологическим постам и автоматическим станциям Республики Башкортостан. Необходимо реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Каждый пункт меню должен быть отдельным классом-наследником (подклассом). Пункты консольного меню:

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL (одна для всего файла).
3. Экспортировать все данные из файла в Excel в MySQL (с учетом структуры данных: столбцы должны называться точно также, либо перевести названия на английский язык).
4. Дополнительно реализовать экспорт из эксель-файла по столбцам (название столбца вводится с клавиатуры). Т.е. мы вводим название столбца и происходит экспорт данных в таблицу MySQL.
5. Реализовать вывод из MySQL всех данных по коду гидрологического поста (задается с клавиатуры). Т.е. вводим код гидрологического поста и выводятся данные только по этому гидропосту.
6. Реализовать вывод из MySQL данных по коду гидропоста (вводится с клавиатуры) и дате (вводится с клавиатуры). Т.е. вводим дату и код гидропоста, и в консоль должны выводится все данные по гидропосту на задаваемую дату.

Ссылка на файл Excel: https://palchevsky.ru/uploads/Urovni2_1_1_new_1_2.xlsx

Для отображения всех данных необходимо проделать показанные на следующем слайде действия.

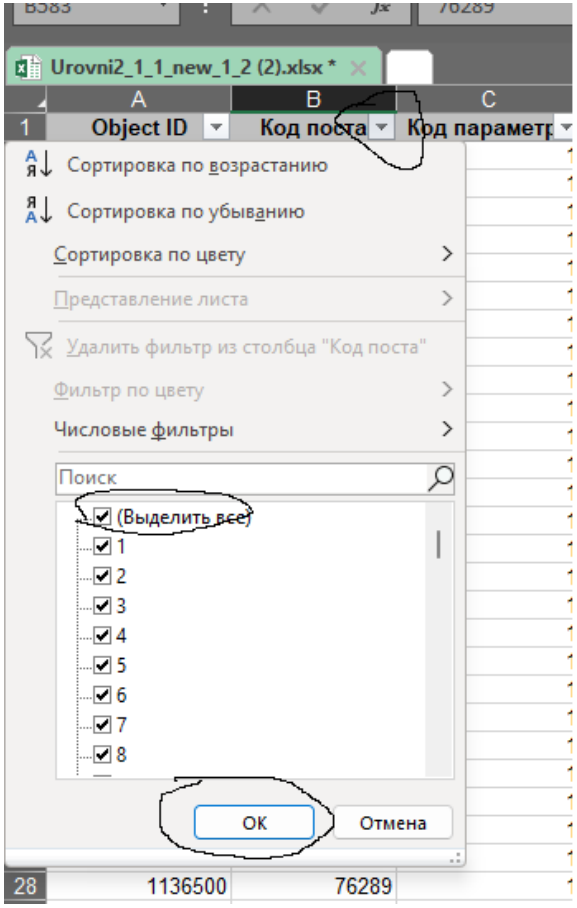
Базовые принципы в ООП Java. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №8

За данную задачу дополнительно начисляется 4.5 балла к уже существующим баллам

Полухардкорный (полуспециальный) вариант. Пояснение к задаче.



Классы, методы и объекты в Java. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №9

Выполняем только сложный вариант.

Задача №9

Сложный вариант. *

Базовый вариант. Объектно-ориентированное программирование. Необходимо создать класс ArrayPI (модификатор доступа – public), в котором необходимо создать два двухмерных массива (ввод с клавиатуры, 7 столбцов и 7 строк). Далее необходимо создать классы-наследники (у всех – модификатор доступа – public final, каждая операция с матрицами – отдельный класс-наследник), в которых будут наследоваться данные матрицы и: перемножаться, складываться, вычитаться, возводиться в степень. После выполнения каждого класса необходимо выводить итоговый результат.

Сложный вариант. Объектно-ориентированное программирование. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задачи полностью идентичны базовому варианту. Каждый пункт меню должен быть отдельным классом-наследником (подклассом).

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL.
3. Ввести две матрицы с клавиатуры и каждую из них сохранить в MySQL с последующим форматированным выводом в консоль.
4. Перемножить, сложить, вычесть, возвести в степень матрицы, а также сохранить результаты в MySQL и вывести в консоль.
5. Сохранить результаты из MySQL в Excel и вывести их в консоль.

Классы, методы и объекты в Java. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №10

Выполняем только сложный вариант.

Задача №10

Сложный вариант. *

Базовый вариант. Объектно-ориентированное программирование. Создать класс Student с модификатором доступа public. В данном классе создать метод, в котором добавляем поля с модификатором доступа private, с их последующим вводом с клавиатуры: Количество студентов, данные о которых вводим с клавиатуры; ID студента; Направление подготовки студента; ФИО студента; Группа студента. Далее создать два класса-наследника, в которых мы переопределяем данный метод супер-класса. Необходимо реализовать полиморфный вывод данного метода. Минимальное количество студентов 5. На выходе мы должны получить список студентов в табличном виде (с заголовками столбцов).

Сложный вариант. Объектно-ориентированное программирование. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задачи полностью идентичны заданию №1. Каждый пункт меню должен быть отдельным классом-наследником (подклассом).

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL.
3. Ввести данные о всех студентах и сохранить их в MySQL с последующим табличным (форматированным) выводом в консоль.
4. Вывести данные о студенте по ID из MySQL.
5. Удалить данные о студенте из MySQL по ID.
6. Сохранить итоговые результаты из MySQL в Excel и вывести их в консоль.

Классы, методы и объекты в Java. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №11

Выполняем только сложный вариант.

Задача №11 (Контрольная работа №3)

Сложный вариант. *

Базовый вариант. Объектно-ориентированное программирование.

1. ***Подзадача 1.*** Разработать программу, в которой требуется создать класс, описывающий геометрическую фигуру – треугольник. Методами класса должны быть – вычисление площади, периметра. Создать класс-наследник, определяющий прямоугольный треугольник. Все данные для треугольника вводим с клавиатуры.

2. ***Подзадача 2.*** Разработать программу, в которой требуется создать класс с модификатором доступа Public. В данном классе необходимо создать метод, реализующий вычисление четных и нечетных факториалов. Все данные вводим с клавиатуры.

Сложный вариант. Объектно-ориентированное программирование. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задачи полностью идентичны заданию №1. Каждый пункт меню должен быть отдельным классом-наследником (подклассом).

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL.
3. Решение базового варианта, сохранение результатов в MySQL.
4. Вывод данных с условием: вывести данные по ID строки. Каждая строка – результаты, сохраненные в MySQL в ходе решения подзадач №1 и №2 базового варианта.
5. Сохранить результаты из MySQL в Excel (заголовки столбцов должны присутствовать при экспорте данных из MySQL в Excel) и вывести их в консоль.

Классы, методы и объекты в Java. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №11

За данную задачу дополнительно начисляется 6 баллов к уже существующим баллам

Полухардкорный (полуспециальный) вариант. Задача без порядкового номера.

4. Объектно-ориентированное программирование и большие данные. Дан файл Excel, в котором хранится больше миллиона значений с данными по гидрологическим постам и автоматическим станциям Республики Башкортостан. Необходимо реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Каждый пункт меню должен быть отдельным классом-наследником (подклассом). Пункты консольного меню:

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL (одна для всего файла).
3. Экспортировать все данные из файла в Excel в MySQL (с учетом структуры данных: столбцы должны называться точно также, либо перевести названия на английский язык).
4. Дополнительно реализовать экспорт из эксель-файла по столбцам (название столбца вводится с клавиатуры). Т.е. мы вводим название столбца и происходит экспорт данных в таблицу MySQL.
5. Реализовать вывод из MySQL всех данных по коду гидрологического поста (задается с клавиатуры). Т.е. вводим код гидрологического поста и выводятся данные только по этому гидропосту.
6. Реализовать вывод из MySQL данных по коду гидропоста (вводится с клавиатуры) и дате (вводится с клавиатуры). Т.е. вводим дату и код гидропоста, и в консоль должны выводится все данные по гидропосту на задаваемую дату.
7. Восстановить все пропущенные данные (за исключением столбца «Описание») в MySQL методом k-ближайших соседей (метрика: Евклидово расстояние).

Ссылка на файл Excel: https://palchevsky.ru/uploads/Urovni2_1_1_new_1_2.xlsx

Коллекции. Интерфейсы. Исключения. Отладка. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №12

Выполняем только сложный вариант.

Задача №12

Сложный вариант. *

Базовый вариант. Объектно-ориентированное программирование и коллекции. Создать класс Students с модификатором доступа public. В данном классе создать метод, в котором добавляем поля с модификатором доступа private, с их последующим вводом с клавиатуры: Количество студентов, данные о которых вводим с клавиатуры; ID студента; Направление подготовки студента; ФИО студента; Группа студента. Далее создать класс-наследник, в котором мы переопределяем данный метод супер-класса и создадим в нем отсортированный по алфавиту (по фамилии) список. Необходимо сделать вывод в виде таблицы с помощью форматированных строк. Минимальное количество студентов 7.

Сложный вариант. Объектно-ориентированное программирование. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задачи полностью идентичны заданию №1. Каждый пункт меню должен быть отдельным классом-наследником (подклассом).

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL.
3. Ввести данные о всех студентах и сохранить список в MySQL с последующим табличным (форматированным) выводом в консоль.
4. Вывести данные о студенте по ID из MySQL.
5. Удалить данные о студенте из MySQL по ID.
6. Сохранить итоговые результаты из MySQL в Excel и вывести их в консоль.

Коллекции. Интерфейсы. Исключения. Отладка. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №13

Выполняем только сложный вариант.

Задача №13

Сложный вариант. *

Базовый вариант. Объектно-ориентированное программирование и коллекции. Создать класс Listik с модификатором доступа public. В данном классе создать два метода: random и input с модификатором доступа protected. Внутри метода random создать список длиной из 1000 случайных значений int (генератор). В методе input необходимо создать список из 10 значений String, вводимых с клавиатуры. На выходе должны получить два списка.

Объектно-ориентированное программирование и коллекции. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задачи полностью идентичны заданию №1. Каждый пункт меню должен быть отдельным классом-наследником (подклассом).

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL.
3. Ввести список и сохранить в MySQL.
4. Удалить элемент из списка в MySQL по ID.
5. Сохранить итоговые результаты из MySQL в Excel и вывести их в консоль.

Коллекции. Интерфейсы. Исключения. Отладка. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №14

Выполняем только сложный вариант.

Задача №14

Сложный вариант. *

Базовый вариант. Объектно-ориентированное программирование и коллекции. Создать класс Listik с модификатором доступа public. В данном классе создать два метода: random и input с модификатором доступа protected. Внутри метода random создать список длиной из 1000 случайных значений int (генератор). В методе input необходимо создать список из 10000 значений Integer. При этом количество элементов и числовой диапазон элементов вводятся пользователем. Выполнить ввод числа с клавиатуры с последующей проверкой на принадлежность данного числа данному списку.

Объектно-ориентированное программирование и коллекции. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задачи полностью идентичны заданию №1. Каждый пункт меню должен быть отдельным классом-наследником (подклассом).

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL.
3. Сохранить вводимое число и списки в MySQL.
4. Удалить элемент из списка в MySQL по ID.
5. Сохранить итоговые результаты из MySQL в Excel и вывести их в консоль.

Коллекции. Интерфейсы. Исключения. Отладка. Практические задачи

ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ

Семинарское занятие №15

Выполняем только сложный вариант.

Задача №15 (контрольная работа №4)

Сложный вариант. *

Базовый вариант. Объектно-ориентированное программирование и коллекции. Создать класс Listik с модификатором доступа public. В данном классе создать метод, в котором с клавиатуры вводим 50 элементов (создать условие на ввод не менее и не более 50 элементов). Создать класс-наследник Listik1, в котором необходимо переопределить метод супер-класса и сконвертировать список в строку, а также множество.

Сложный вариант. Объектно-ориентированное программирование и коллекции. Реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). Задачи полностью идентичны заданию №1. Каждый пункт меню должен быть отдельным классом-наследником (подклассом).

1. Вывести все таблицы из базы данных MySQL.
2. Создать таблицу в базе данных MySQL.
3. Сохранить вводимый с клавиатуры список, а также строку и множество в MySQL.
4. Удалить элемент из списка, строки и множества в MySQL по ID.
5. Сохранить итоговые результаты из MySQL в Excel и вывести их в консоль.