Содержание

Лабораторная работа №1. Логические и условные операторы в Java (if, if-else и switch)

1 Цель работы

Изучить операторы, используемые для организации ветвления в программе.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Для того, чтобы изменить последовательность выполнения фрагментов программы, в языке Java применяются конструкции `if`, `if-else` и `switch`.

В лабораторной работе будут рассмотрены следующие вопросы:

— Как применяются условные операторы `if`, `if-else` и `switch`.

— Какие существуют логические операторы в Java.

— В чем отличие между оператором == и методом `equals()` при сравнении.

— Конструкция `if-else-if`.

— Что такое тернарный оператор.

В конце предложены упражнения для закрепления материала.

Управляющие операторы применяются для реализации переходов и ветвлений в потоке исполнения команд программы, исходя из ее состояния. Управляющие операторы в Java бывают следующих типов: операторы выбора, операторы цикла, операторы перехода. В данной лабораторной работе рассматриваются операторы выбора или условные операторы – `if` и `switch`.

Операторы выбора позволяют управлять порядком выполнения команд программы в соответствии с условиями, которые известны только во время выполнения.

Конструкция `if-else` имеет следующий вид:

```
if (условие) {
    // блок кода 1
}
else{
    // блок кода 2
}
```

Блок `else` не является обязательным. Если «условие» истинно, то выполняется «блок кода 1», иначе – «блок кода 2». Никогда не выполняется оба блока кода. Условие представлено логическим выражением.

Логическое выражение – это выражение, возвращающее значение типа boolean. Условие может быть представлено одной переменной типа boolean.

Пример:

```
boolean b = true;
if (b) {
    System.out.println("b - истина");
}
```

Логические операции выполняются только с операндами типа boolean. Практически все они бинарные (операция выполняется с двумя операндами), одна унарная – отрицание, один тернарный. На рисунке 1 перечислены все логические операции языка Java.

Рассмотрим на примере:

```
int a = 4;
int b = 5;
```

```
boolean result;

result = a == b // a равно b - false
result = a != b // a не равно b - true
result = a < b; // a меньше b - true
result = a > b; // a больше b - false
result = a <= 4 // a меньше или равно 4 - true
result = b >= 6 // b больше или равно 6 - false
result = a > b || a < b // (a больше b) логическое или (a меньше
b) - true
result = 3 < a && a < 6 // (3 меньше a) логическое и(a меньше 6) -
true
result = !result // логическое нет – false
```

Операция	Описание
&	Логическая операция И
|	Логическая операция ИЛИ
^	Логическая операция исключающее ИЛИ
||	Укороченная логическая операция ИЛИ
&&	Укороченная логическая операция И
!	Логическая унарная операция НЕ
&=	Логическая операция И с присваиванием
|=	Логическая операция ИЛИ с присваиванием
^=	Логическая операция исключающее ИЛИ с присваиванием
==	Равенство
!=	Неравенство
?:	Тернарная условная операция типа *если..., то..., иначе...*

Рисунок 1 – Логические операции в Java

Отличие == и `equals()`. Оператор == выполняет сравнение по ссылке, то есть проверяет, указывают ли обе переменные на один и тот же объект в памяти, а метод `equals()` сравнивает объекты по значению. Данное свойство проявляется для объектов сложных типов данных (не примитивных, а представляющих собой экземпляры классов). Например:

```
String str1 = new String("Привет");
String str2 = new String("Привет");
String sameStr = str1;

boolean b1 = str1 == str2; // b1 - false, потому что str1 и str2
это 2 разных объекта
```

```
boolean b2 = str1.equals(str2); // b2 - true, потому что str1 и
str2 логически эквивалентны

boolean b3 = str1 == sameStr; // b3 - true, потому что str1 и
sameStr в действительности один и тот же объект
```

Конструкция `if-else-if` состоит из последовательности вложенных условных операторов `if`. Данная конструкция достаточно распространена. Синтаксис:

```
if (условие) {
    // блок кода 1
}
else if (условие) {
    // блок кода 2
...
else if (условие) {
    // блок кода N
else {
    // блок кода N + 1
}
```

Тернарный оператор – означает тройной, имеющий три операнда. Тернарный оператор имеет следующую структуру:

```
условие ? блок кода 1 : блок кода 2
```

Если «условие» истинно, то выполняется «блок кода 1», иначе «блок кода 2».

В языке Java оператора `switch` является оператором ветвления. Он предоставляет простой способ направить поток исполнения команд по разным ветвям кода в зависимости от значения управляющего выражения. Условный оператор `switch – case` удобен в тех случаях, когда количество вариантов очень много и писать для каждого `if-else` очень долго. Конструкция имеет следующий вид:

```
    switch (выражение) {
      case значение1:
        // блок кода 1;
        break;
      case значение2:
        // блок кода 2;
```

```
        break;
        ...
      case значениеN:
        // блок кода N;
        break;
      default:
        // блок кода N+1;
  }
```

`switch` работает с такими примитивными типами данных как: `byte`, `short`, `char`, и `int`. Также с типом `Enum`, классом `String` и несколькими специальными классами-оболочками примитивных типов: `Character`, `Byte`, `Short`, `Integer`.

Выражение в круглых скобках после `switch` сравнивается со значениями, указанными после слова `case`, и, в случае совпадения, управление передается соответствующему блоку кода. Если выражение не совпадает ни с одним вариантом `case`, то управление передается блоку `default`, который **не является обязательным**. После выполнения соответствующего блока, оператор `break` вызывает завершение выполнения оператора `switch`. Если `break` отсутствует, то управление передается следующему блоку за только что выполненным.

В следующем примере, для целочисленного значения, представляющего день недели, определяется его название в виде строки. Для значения 3, переменная dayString примет значение «Среда».

```
int day = 3;
String dayString;
switch (day) {
    case 1:  dayString = "Понедельник";
             break;
    case 2:  dayString = "Вторник";
             break;
    case 3:  dayString = "Среда";
             break;
    case 4:  dayString = "Четверг";
             break;
    case 5:  dayString = "Пятница";
             break;
    case 6:  dayString = "Суббота";
             break;
    case 7:  dayString = "Воскресенье";
```

```
                break;
        default: dayString = "Ошибка";
                break;
    }
    System.out.print(dayString);
```

5 Упражнения на тему условных операторов в Java:

1. Даны 4 числа типа `int`. Сравнить их и вывести наименьшее на консоль.

2. Вывести на консоль количество максимальных чисел седи чисел из задания 1.

3. Даны 5 чисел (тип `int`). Вывести вначале наименьшее, а затем наибольшее из данных чисел.

4. Даны имена 2-х человек (тип `String`). Если имена равны, то вывести сообщение о том, что люди являются тезками.

5. Дано число месяца (тип `int`). Необходимо определить время года (зима, весна, лето, осень) и вывести на консоль.

Лабораторная работа № 2. Java операторы цикла (for, while, do-while), оператор break

1 Цель работы

Изучить операторы циклов, используемые для организации повторяющихся процессов в программах.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Цикл в программировании используется для многократного повторения определенного фрагмента кода, в Java существует 3 оператора цикла: `for`, `while`, `do-while`.

В лабораторной работе будут рассмотрены следующие вопросы:

— Как применяются операторы цикла `while`, `do-while`, `for`.

— Как досрочно выйти из цикла или пропустить итерацию – ключевые слова `break` и `continue`.

В конце предложены упражнения для закрепления материала.

Циклы многократно выполняют один и тот же набор инструкций до тех пор, пока не будет удовлетворено условие завершения цикла.

Оператор цикла `while` является самым простым для организации циклов в Java. Он повторяет оператор или блок операторов до тех пор, пока значение его управляющего оператора истинно. Этот оператор цикла имеет следующую общую форму:

```
while(условие) {
     // тело цикла
}
```

Как было рассмотрено в 1 лабораторной работе, условием может быть любое логические выражение. Пока оно истинно, тело цикла будет выполняться. Пример отсчета от 10 до 1 с помощью `while`:

```
int n = 10;
while(n > 0) {
     System.out.println("такт " + n);
     n--;
}
```

Возможно реализовать бесконечный цикл `while`:

```
while(true) {
     // тело цикла
}
```

Если в начальный момент условное выражение, управляющее циклом `while`, ложно, то код в теле цикла не будет выполняться. Когда нужна гарантия того, что тело цикла выполнится хотя бы один раз, то используется цикл `do-while`, в котором проверка условия выполняется в конце цикла после тела цикла.

```
do {
     // тело цикла
} while (условие);
```

Ниже представлен пример, который в случае использования цикла `while` не выполнится ни разу, а цикл `do-while` выполнит код из тела один раз. После запуска кода, в консоль будет выведено «i = 10».

```
int i = 10;
```

```
do {
    System.out.println("I = " + i);
    i++;
} while(i < 5);
```

Цикл `for` в общем случае выглядит так:

```
for(инициализация; условие; итерация) {
     // тело цикла
}
```

Конкретный пример, который выводит в консоль числа от 1 до 10:

```
for (int i = 1; i <= 10; i++) {
     System.out.print(i + " ");
}
```

Инициализация — первый параметр, который содержит переменную управления циклом и ее начальное значение. С помощью этой переменной будет подсчитываться количество повторений цикла. В нашем примере это переменная int i = 1.

Условие — второй параметр, содержит некоторое логическое выражение — условие при котором будет выполняться цикл. В нашем примере это условие i <= 10.

Итерация — третий параметр, выражение, изменяющее переменную (заданную в инициализации) после каждого шага цикла. Изменение происходит только в случае выполнения условия. В нашем примере итерация i++ — увеличение переменной на единицу. Также для итерации часто используется i-- уменьшение переменной на единицу.

Когда в теле цикла встречается оператор `break`, выполнение цикла прекращается и управление передается оператору, следующему за циклом. Например, в коде ниже происходит символа в строке. Символы строки просматриваются в цикле, а когда нужное значение будет найдено, происходит выход из цикла с помощью `break`.

```
public class BreakDemo {
```

```
    public static void main(String[] args) {
        // строка для поиска
        String searchMe = "Мама мыла раму";
        //количество символов в строке
        int max = searchMe.length();
        // символ, который нужно найти в строке
        char symb = 'ы';
        // переключатель найдено/не найдено
        boolean find = false;
        for (int i = 0; i < max; i++) {
            // если символ на позиции i в строке равен искомому
символу
            if (searchMe.charAt(i) == symb) {
                //переключаемся в режим "найдено" и выходим из
цикла
                find = true;
                break;
            }
        }
      // выводим соответствующее сообщение: найден или не найден
символ
        if (find) {
            System.out.println("Символ '" + symb + "' найден в
строке");
        } else {
            System.out.println("Символ '" + symb + "' не найден в
строке");
        }
    }
}
```

Оператор `continue` предназначен для передачи управления циклу из тела цикла. Тогда как `break` позволяет полностью выйти из цикла, `continue` позволяет пропустить весь оставшийся после него код тела цикла и перейти к новой итерации цикла (конечно, если управляющее условие истинно). Например, можно вывести нечетные числа таким образом:

```
for(int i = 0; i < 10; i++) {
     System.out.print(i + " ");
     // Если остаток от деления на 2 равен нулю
     if (i % 2 == 0) continue;
     System.out.println("");
}
```

5 Упражнения по теме операторов цикла в Java:

1. При помощи цикла `for` вывести на экран нечетные числа от 1 до 99.
2. Дано число n при помощи цикла `for` посчитать факториал n!.

3. Выполните задания 1 и 2 с использованием цикла `while`.

4. Даны переменные x и n вычислить x в степени n.

5. Вывести 10 первых чисел последовательности {0, -5, -10, -15, …}.

6. Переделайте пример для оператора `break` (в котором выполнялся поиск символа в строке). Необходимо, чтобы заданный символ встречался в строке хотя бы 2 раза. Программа должна подсчитывать число вхождений символа в строке и выводить это число в консоль.

Лабораторная работа № 3. Разработка алгоритмов и программ обработки одномерных массивов

1 Цель работы

Изучить методы алгоритмизации и программирования с использованием ссылочных типов данных (массивов).

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Цикл в программировании используется для многократного повторения определенного фрагмента кода, в Java существует 3 оператора цикла: `for`, `while`, `do-while`.

В лабораторной работе будут рассмотрены следующие вопросы:

— Что такое массив в Java.

— Объявление и инициализация массива в Java.

— Что такое инициализатор массива.

— Как получать и изменять значения ячеек массива.

— Итерация по массиву с помощью цикла.

В конце предложены упражнения для закрепления материала.

Массив – группа однотипных переменных, для обращения к которым используется общее имя. В языке Java допускается создание массивов любого типа и разной размерности. Доступ к конкретному элементу массива осуществляется по его индексу. Массивы предоставляют удобный способ группирования связанной вместе информации.

Одномерные массивы представляют собой список однотипных переменных. Чтобы создать массив, необходимо сначала **объявить** переменную массива требуемого типа. В общем виде одномерный массив определяется так:

```
тип имя_переменной[];
```

Параметр тип указывает на тип каждого элемента массива. Конструкция выше, на самом деле, пока не создает никакого массива, его на данный момент еще не существует. Чтобы данная переменная указывала на массив, необходимо зарезервировать область памяти с помощью операции `new`, а затем связать ее с переменной. Пример **объявления** и **инициализации** массива:

```
тип имя_переменной[] = new тип[размер];
```

Тип показывает тип данных, для которых резервируется память, параметр размер – количество элементов в списке, параметр имя_переменной является ссылкой на область памяти, в которой находятся элементы массива.

Элементы массива, для которых память была выделена операцией new, инициализируются нулевыми значениями (для числовых примитивных типов), логическими значениями `false` (для логического типа `boolean`) или пустыми значениями null (для сложных ссылочных типов данных).

Таким образом, процесс создания массива происходит в два этапа. Сначала объявляется переменная нужного типа, а затем резервируется память для хранения элементов массива и присвоить ее переменной массива.

Данную концепцию полезно знать, хотя на практике всю работу с памятью выполняет Java, так это язык с автоматическим управлением памятью.

Для получения или изменения значения ячейки массива используется следующая конструкция:

```
// получение значения
имя_переменной[индекс_элемента];
// изменение элемента
имя_переменной[индекс элемента] = новое_значения;
```

Индексация элементов массива в Java начинается с 0.

Создать массив можно с помощью так называемого **инициализатора массива**. Инициализатор массива – список выражений, разделяемый запятыми и заключаемый в фигурные скобки. Массив создается настолько большим, чтобы вместить все элементы.

Для примера создадим массив с помощью инициализатора, а затем выведем в консоль все его элементы с помощью цикла `for`. Заметьте, что получение длины массива выполняется с помощью свойства `length`.

```
int monthDays[] = {31, 28, 31, 30, 31, 30,
                   31, 31, 30, 31, 30, 31};
for(int i = 0; i < monthDays.length; i++) {
     System.out.println(monthDays[i]);
}
```

5 Упражнения на тему одномерных массивов в Java:

1. Создайте массив, содержащий 10 первых нечетных чисел. Выведете элементы массива на консоль в одну строку, разделяя запятой.
2. Дан массив размерности N, найти наименьший элемент массива и вывести на консоль (если наименьших элементов несколько — вывести их все).
3. В массиве из задания 2 найти наибольший элемент.
4. Поменять наибольший и наименьший элементы массива местами. Пример: дан массив {4, -5, 0, 6, 8}. После замены будет выглядеть {4, 8, 0, 6, -5}.
5. Найти среднее арифметическое всех элементов массива.

Лабораторная работа № 4. Сортировка массивов

1 Цель работы

Изучить алгоритмы сортировки одномерных массивов.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Темой работы является сортировка массивов.

В лабораторной работе будут рассмотрены следующие вопросы:

— Что такое сортировка.

— Алгоритм сортировки выбором на Java.

— Алгоритм сортировки пузырьком на Java.

— Метод `sort()` класса `java.util.Arrays`.

— Сортировка целочисленного массива по возрастанию и убыванию.

— Сортировка массива строк.

В конце предложены упражнения для закрепления материала.

Одной из частых задач при работе с массивами является сортировка массива. Сортировкой массива называется процесс упорядочивания элементов массива по возрастанию или по убыванию. В данной лабораторной работе будут рассмотрены некоторые способы сортировки.

Тривиальным решением задачи «Отсортировать массив по возрастанию» является перебор, то есть: найти минимальный элемент и поменять его местами с начальным, потом, в оставшейся части массива (кроме первого элемента), найти снова минимальный элемент и поменять его со вторым элементом и т.д. Такой алгоритм называется «Сортировка выбором».

Ниже приведен код, реализующий сортировку массива по алгоритму «Сортировки выбором».

```
public static void selectionSort(int[] arr){
    /* По очереди будем просматривать все подмножества
      элементов массива (0 - последний, 1-последний,
      2-последний,...) */
    for (int i = 0; i < arr.length; i++) {
        /* Предполагаем, что первый элемент (в каждом
           подмножестве элементов) является минимальным */
        int min = arr[i];
        int min_i = i;
        /* В оставшейся части подмножества ищем элемент,
           который меньше предположенного минимума  */
        for (int j = i+1; j < arr.length; j++) {
            // Если находим, запоминаем его индекс
            if (arr[j] < min) {
                min = arr[j];
                min_i = j;
            }
        }
        /* Если нашелся элемент, меньший, чем на текущей позиции,
          меняем их местами */
        if (i != min_i) {
            int tmp = arr[i];
            arr[i] = arr[min_i];
            arr[min_i] = tmp;
        }
     }
}
```

Следующим достаточно известным способом сортировки массивов является алгоритм «Сортировки пузырьком».

Алгоритм проходит массив от начала и до конца, сравнивая попарно соседние элементы, если элементы стоят в неправильном порядке, то они меняются местами, таким образом, после первого прохода на конце массива оказывается максимальный элемент (для сортировки по возрастанию). Затем

проход массива повторяется, и на предпоследнем месте оказывается другой наибольший после максимального элемент и т.д. В итоге, наименьший элемент постепенно перемещается к началу массива («всплывает» до нужной позиции как пузырёк в воде).

Реализация алгоритма «Сортировка пузырьком» на Java (по возрастанию):

```
public static void bubbleSort(int[] arr){
    /* Внешний цикл каждый раз сокращает фрагмент массива,
      так как внутренний цикл каждый раз ставит в конец
      фрагмента максимальный элемент */
    for(int i = arr.length-1 ; i > 0 ; i--){
        for(int j = 0 ; j < i ; j++){
            /* Сравниваем элементы попарно,
              если они имеют неправильный порядок,
              то меняем местами */
            if(arr[j] > arr[j+1]){
                int tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}
```

Ниже расположен пример применения метода сортировки `bubbleSort()`. Аналогично применяется и метод `selectionSort()`.

```
int arr[] = {62, 84, 32, 5, 0, 14, 52, 82, 58, 71};
bubbleSort(arr);
for (int i = 0; i < arr.length; i++) {
    System.out.print(arr[i] + "  ");
}
System.out.print("\n");
```

На сегодняшний день данные алгоритмы сортировки используются разве что в учебных целях. Существуют гораздо более быстрые и эффективные алгоритмы.

В стандартной библиотеке Java реализован метод сортировки `sort()`, который сортирует массив по усовершенствованному алгоритму «Быстрой сортировки» (QuickSort).

Чтобы воспользоваться этим методом, нужно импортировать класс java.util.Arrays.

```
import java.util.*;
```

Пример использования библиотечного метода для сортировки массива:

```
// Создаем массив случайных чисел
int arr[] = new int[10];
for (int i = 0; i < arr.length; i++) {
    arr[i] = (int) (Math.random() * 100);
    System.out.print(arr[i] + "  ");
}
System.out.print("\nSorted: \n");
// Сортируем массив
Arrays.sort(arr);
// Выводим отсортированный массив на консоль
for (int i = 0; i < arr.length; i++) {
    System.out.print(arr[i] + "  ");
}
```

В примере выше числа массива сортируются по возрастанию. Для сортировки по убыванию выполните следующий код:

```
// Создаем массив случайных чисел
Integer arr[] = new Integer[10];
for (int i = 0; i < arr.length; i++) {
    arr[i] = (int) (Math.random() * 100);
    System.out.print(arr[i] + "  ");
}
System.out.print("\nSorted: \n");
// Сортируем массив
Arrays.sort(arr, Collections.reverseOrder());
// Выводим отсортированный массив на консоль.
for (int i = 0; i < arr.length; i++) {
    System.out.print(arr[i] + "  ");
}
```

Заметьте, что в данном случае вместо примитивного типа `int` используется класс-обертка (wrapper) `Integer`. Это связано с тем, что метод `sort()` класса `Arrays`, принимающий два параметра, требует в качестве первого параметра только массив объектов, а массив из примитивных типов данных не удовлетворяет этому условию. Вторым параметром передается объект типа `Comparator`. Он определяет логику сравнения двух объектов между собой.

Данный метод позволяет также отсортировать массив строк:

```
String[] names = new String[]{"Roman", "Anna", "Petr", "Maria"};

Arrays.sort(names);
for (int i = 0; i < names.length; i++) {
    System.out.print(names[i] + "  ");
}
```

5 Упражнения на тему сортировки массивов в Java:

1. Создайте массив из 20 случайных чисел (числа должны быть в диапазоне от 0 до 1000) и отсортируйте массив по убыванию при помощи сортировки пузырьком.

2. Создайте массив, содержащий марки автомобилей, отсортируйте его сначала по возрастанию, потом по убыванию.

Лабораторная работа № 5. Разработка алгоритмов и программ обработки многомерных массивов

1 Цель работы

Изучить методы алгоритмизации и программирования с использованием многомерных массивов.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Данная работа посвящена многомерным массивам в Java.

В лабораторной работе будут рассмотрены следующие вопросы:

— Что такое многомерный массив в Java.

— Объявление и инициализация двумерного массива в Java.

— Создание неоднородных двумерных массивов.

— Заполнение двумерных массивов значениями.

— Итерация по двумерному массиву с помощью цикла.

В конце предложены упражнения для закрепления материала.

В языке Java многомерные массивы представляют собой массивы массивов. При объявлении и инициализации, например, двумерного массива используется еще один ряд квадратных скобок. Например:

```
int twoD[][] = new int[4][5];
```

Данная строка инициализирует память для массива размерностью 4 на 5, где 4 – число строк, а 5 – число столбцов, а затем присваивает его переменной `twoD`.

В математике такая структура называется матрицей, но на языке Java она представляется двумерным массивом – массивом массивов.

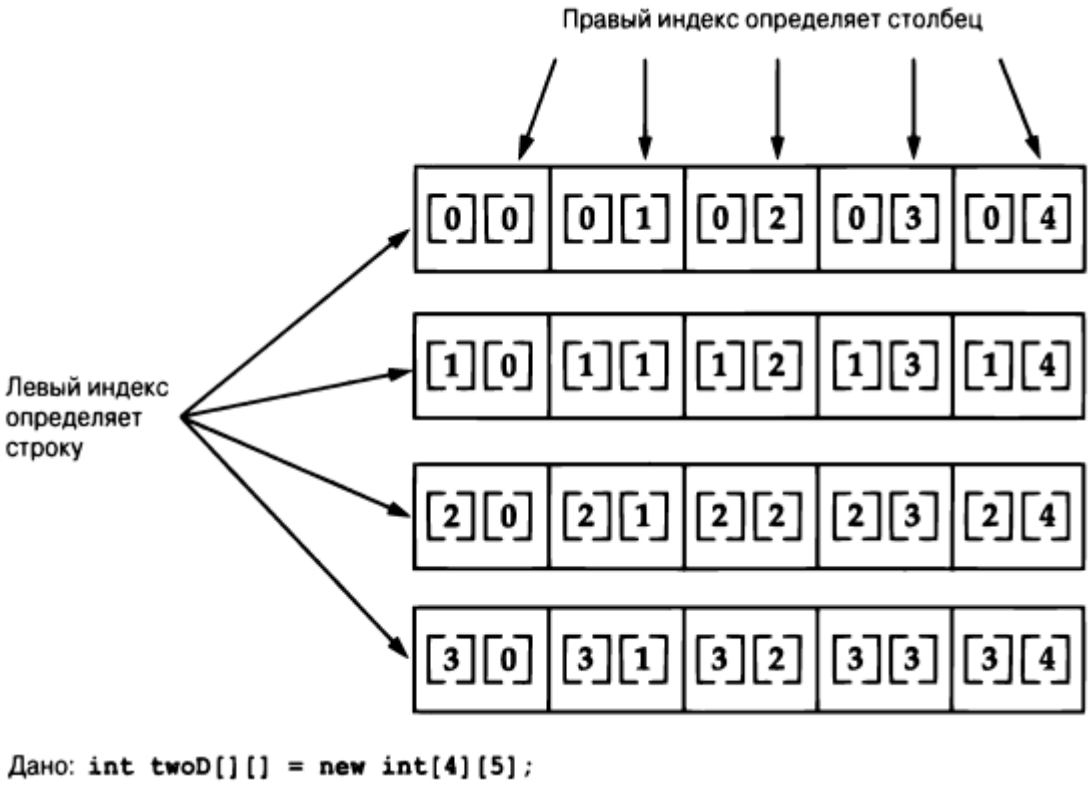


Рисунок 3 – Визуализация двумерного массива

В примере ниже приведен код для заполнения и получения значений элементов двумерного массива. Программа выводит нумерацию элементов массива слева направо и сверху вниз.

```
int twoD[][] = new int[4][5];
int i, j, k = 0;

for (i = 0; i < 4; i++) {
    for (j = 0; j < 5; j++) {
        twoD[i][j] = k;
        k++;
    }
}
```

```
for (i = 0; i < 4; i++) {
    for (j = 0; j < 5; j++) {
        System.out.print(twoD[i][j] + " ");
    }
    System.out.println();
}
```

При инициализации массива (когда указываются размерности), необходимо указать размерность только для первого измерения массива. Указания второго измерения является опциональным и используется, когда необходимо сделать прямоугольный двумерный массив. Но не всегда требуется именно прямоугольный массив. Существуют также зубчатые и треугольные (ступенчатые) массивы. Такие массивы называются неоднородными или нерегулярными. Они могут быть неприемлемы в большинстве приложений, так как их поведение не так предсказуемо, как у прямоугольных.

Ниже представлен код для создания ступенчатого двумерного массива.

```
int twoD[][] = new int[4][];
for (int n = 0; n < 4; n++) {
    twoD[n] = new int[n + 1];
}
int i, j, k = 0;

for (i = 0; i < 4; i++) {
    for (j = 0; j < i + 1; j++) {
        twoD[i][j] = k;
        k++;
    }
}

for (i = 0; i < 4; i++) {
    for (j = 0; j < i + 1; j++) {
        System.out.print(twoD[i][j] + " ");
    }
    System.out.println();
}
```

Данный массив графически изображен на рисунке 4.

Аналогично определяется двумерный массив любой другой формы. Просто указывается необходимая размерность второго измерения для каждого элемента первого измерения.

Двумерные массивы, также, как и одномерные, могут быть заполнены с помощью инициализатора массива.

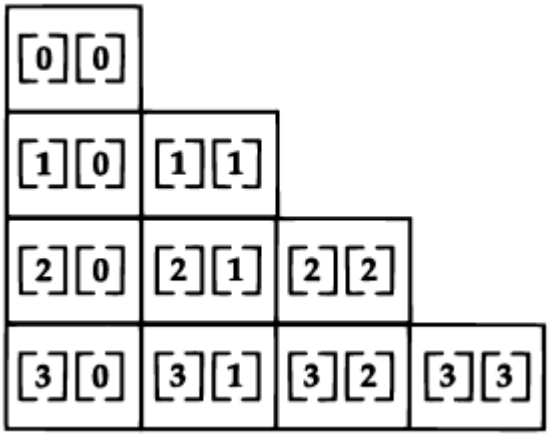


Рисунок 4 – Визуализация двумерного массива с разной размерностью второго измерения

В примере ниже показывается, как заполнить двумернай массив с помощью инициализатора.

```
double m[][] = {
        {0, 1, 2},
        {3, 4, 5,},
        {6, 7, 8}
};
for(int i = 0; i < m.length; i++) {
    for(int j = 0; j < m[i].length; j++) {
        System.out.print(m[i][j] + " ");
    }
    System.out.println();
}
```

5 Упражнения на тему многомерных массивов в Java:

1. Создайте массив размерностью 5 на 6 и заполните его случайными числами (в диапазоне от 0 до 99). Выведите на консоль третью строку.

2. Даны матрицы C и D размерностью 3 на 3 и заполненные случайными числами в диапазоне от 0 до 99. Выполните по отдельности сначала сложение, потом умножения матриц друг на друга. Выведете исходные матрицы и результат вычислений на консоль.

3. Просуммируйте все элементы двумерного массива.

4. Дан двумерный массив, содержащий отрицательные и положительные числа. Выведете на экран номера тех ячеек массива, которые содержат отрицательные числа.

5. Отсортируйте элементы в строках двумерного массива по возрастанию.

Лабораторная работа № 6. Арифметические операторы и математика в Java

1 Цель работы

Изучить арифметические операторы в Java.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Данная работа посвящена арифметическим операциям в Java.

В лабораторной работе будут рассмотрены следующие вопросы:

— Какие бывают арифметические операторы в Java.

— Применение арифметических операций в Java.

— Операция деления по модулю в Java.

— Префиксная и постфиксная форма инкремента и декремента.

— Класс `java.lang.Math`.

В конце предложены упражнения для закрепления материала.

Арифметические операции применяются в математических выражениях таким же образом, как и в алгебре. Все арифметические операции, доступные в Java, перечислены ниже.

Операнды арифметических операций должны иметь числовой тип. Арифметические операции нельзя выполнять над логическими типами данных, но допускается над типами данных `char`, поскольку в Java это тип является, по сути, разновидностью типа `int`.

Операция	Описание
+	Сложение (а также унарный плюс)
-	Вычитание (а также унарный минус)
*	Умножение
/	Деление
%	Деление по модулю
++	Инкремент (приращение на 1)
+=	Сложение с присваиванием
-=	Вычитание с присваиванием
*=	Умножение с присваиванием
/=	Деление с присваиванием
%=	Деление по модулю с присваиванием
--	Декремент (отрицательное приращение на 1)

Рисунок 5 – Арифметические операции в Java

Все основные арифметические операции (сложения, вычитания, умножения и деления) воздействуют на числовые типы данных так, как этого и следовало ожидать. Операция унарного вычитания изменяет знак своего единственного операнда. Операция унарного сложения просто возвращает значение своего операнда. Нужно иметь в виду, что операция деления над целочисленным типом данных, ее результат не будет содержать дробный компонент, так как будет иметь целочисленный тип данных.

Следующий код демонстрирует применение арифметических операций в зависимости от типа операндов.

```
System.out.println("Целочисленная арифметика");
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
```

```
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("e = " + e);

System.out.println("Арифметика с плавющей точкой");
double da = 1+ 1;
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
```

Также в Java определена операция деления по модулю. Операция деления по модулю % возвращает остаток от деления. В следующем примере программы демонстрируется применение операции %.

```
int x = 42;
double y = 42.25;

System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
```

Данный код напишет в консоль числа 2 и 2.25.

В языке Java определены специальные составные операции, которые объединяют арифметические операции с операцией присваивания. Например, для сложения:

```
a = a + 4;
// можно записать как
a += 4;
```

Аналогично работают и другие составные операторы. Кажется, что данные операции созданы только для сокращения объема кода. Это только отчасти так. Реализация составных операций в исполняющей системе Java оказывается эффективнее реализации эквивалентных длинных форм.

Операции «++» и «--» выполняют инкремент и декремент. Они уже использовались в предыдущих работах, поэтому пример кода для них приведен не будет. Но еще не были рассмотрены формы этих операций. Они могут быть

записаны в постфиксной форме, когда операция следует за операндом, а также в префиксной форме, когда операция предшествует операнду.

В префиксной форме значение операнда увеличивается или уменьшается до извлечения значения для применения в выражении. А в постфиксной форме предыдущее значение извлекается для применения в выражении, и только после этого изменяется значение операнда.

Для решения задач нередко требуется использование математических функций. В Java такие функции включены в класс `Math`. Для того, чтобы использовать методы класса `Math`, нужно подключить его в начале .java файла с вашим кодом.

```
import static java.lang.Math.*;
```

Часто используемые математические функции

— `sqrt(a)` — извлекает квадратный корень из числа a.

— `pow(a, n)` — a возводится в степень n.

— `sin(a)`, `cos(a)`, `tan(a)` — тригонометрические функции sin, cos и tg угла a, указанного в радианах.

— `asin(n)`, `acos(n)`, `atan(n)` — обратные тригонометрические функции, возвращают угол в радианах.

— `exp(a)` — возвращает значение экспоненты, возведенной в степень a.

— `log(a)` — возвращает значение натурального логарифма числа a.

— `log10(a)` — возвращает значение десятичного логарифма числа a.

— `abs(a)` — возвращает модуль числа a.

— `round(a)` — округляет вещественное число до ближайшего целого.

Константы

— `PI` — число «пи», с точностью в 15 десятичных знаков.

— `E` — число «e» (основание экспоненциальной функции), с точностью в 15 десятичных знаков.

5 Упражнения по теме арифметических операторов в Java:

1. Дан массив целых чисел, найти среди элементов массива числа, которые делятся на 3 и на 6.
2. Посчитать среднее арифметическое чисел в массиве.
3. Известны катеты прямоугольного треугольника, найти его площадь и периметр.
4. Даны два целых числа, найти их наибольший общий делитель и наименьшее общее кратное.
5. Даны радиус вращения и высота конуса, вычислить объем конуса.

Лабораторная работа № 7. Классы и экземпляры классов в Java

1 Цель работы

Изучить основные принципы разработки классов в Java и получить представление об экземплярах классов и методах.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Данная работа посвящена классам и их возможностям в Java.

В лабораторной работе будут рассмотрены следующие вопросы:

— Что такое класс в Java и как его создать.

— Что относится к членам класса, поля и методы класса.

— Модификаторы доступа и инкапсуляция.

— Как создается экземпляр класса, ключевое слово `new`.

— Методы класса, параметры и аргументы, перегрузка методов.

— Передача аргументов по значению и по ссылке.

— Конструкторы класса, перегрузка конструкторов.

— Ключевое слово `this`.

— Базовая информация о сборщике мусора.

— Ключевые слова `static` и `final`.

В конце предложены упражнения для закрепления материала.

Класс является конструкцией, на которой построен весь язык Java. Объектно-ориентированное в Java также невозможно без класса и объекта.

Главный смысл создания класса – определение **нового типа данных**. Класс тогда является шаблоном для создания объекта. Объект же является экземпляром класса. В связи с этим, понятия экземпляр и объект в данном случае могут употребляться в одном значении.

Класс объявляется с помощью ключевого слова `class`. Ниже приведен обобщенный вид простого класса.

```
class имя_класса {
     тип переменная_экземпляра1;
     тип переменная_экземпляра2;
     …
     тип переменная_экземпляраN;

     тип имя_метода1(список_параметров) {
         // тело метода
     }

     тип имя_метода2(список_параметров) {
         // тело метода
     }
     …
     тип имя_методаN(список_параметров) {
         // тело метода
     }
}
```

Класс содержит члены класса – переменные экземпляра и методы. Данные, определенные в классе, называются **переменными экземпляра**. Код, реализующий логику обработки данных, находится в **методах**.

Для объяснения классов в Java будет использоваться класс `IntegerStack`, который реализует простую структуру данных, называемую стеком. Для простоты, стек будет хранить только значения целочисленного типа. Хотя, конечно, Java поддерживает создание параметризованных классов, что делает возможным создать класс стека, который будет хранить любой тип данных, но эта тема выходит за рамки данной лабораторной работы.

```
public class IntegerStack {
    private int size;
    private int stack[];
    private int count;

    public IntegerStack(int size) {
        this.size = size;
        stack = new int[size];
        count = 0;
    }

    public void push(int item) {
        if(!isFull()) {
            stack[count++] = item;
        } else {
            System.out.println("Стек переполнен");
        }
    }

    public int pop() {
        if(!isEmpty()) {
            return stack[--count];
        } else {
            System.out.println("Стек пуст");
            return 0;
        }
    }

    public boolean isFull() {
        return count == size;
    }

    public boolean isEmpty() {
        return count == 0;
    }
}
```

Использование данного класса продемонстрировано в коде ниже.

```
import java.util.Arrays;
import java.util.Collections;

public class Main {
    public static void main(String[] args) {
        IntegerStack st1 = new IntegerStack(5);
        IntegerStack st2 = new IntegerStack(10);
        // Попытка извлечения из пустых стеков
        st1.pop();
        st2.pop();
        // Заполнение первого стека значениями от 1 до 4
        for (int i = 0; i < 5; i++) {
            st1.push(i);
```

```
        }
        // Заполнение второго стека значениями от 10 до 1
        for (int i = 10; i > 0; i--) {
            st2.push(i);
        }
        // Поытка добавить элементы в полные стеки
        st1.push(100);
        st2.push(100);
        // Извлечение элементов из первого стека
        System.out.println("Первый стек");
        for (int i = 0; i < 5; i++) {
            System.out.println(st1.pop());
        }
        // Извлечение элементов из второго стека
        System.out.println("Второй стек");
        for (int i = 0; i < 10; i++) {
            System.out.println(st2.pop());
        }
    }
}
```

При запуске данного кода, в консоль будет выведена следующая информация:

```
Стек пуст
Стек пуст
Стек переполнен
Стек переполнен
Первый стек
4
3
2
1
0
Второй стек
1
2
3
4
5
6
7
8
9
10
```

Начнем с того, что рассмотрим **модификаторы доступа**, которые были применены при определении класса. С помощью модификаторов доступа реализуется механизм **инкапсуляции** или сокрытия реализации. С помощью них возможно создавать классы, которые действуют практически как черный

ящик. Это позволяет защитить внутренние данные класса от изменения извне, предоставляя пользователю класса только определенный набор методов, которые безопасно работают с данными и не повреждают их, как и задумывается разработчиком класса.

В языке Java определены следующие модификаторы доступа: `public` (открытый), `private` (закрытый), `protected` (защищенный). Модификатором по умолчанию является модификатор `package`.

Когда член класса определяется с модификатором доступа `public`, он становится доступным из любого другого кода. Если член класса объявлен как `private`, то он доступен только в области видимости класса. Модификатор `protected` применяется в случае наследования, поля суперкласса с таким модификатором будут доступны членам суперкласса и классам-наследникам (о наследовании будет рассказано в следующей лабораторной работе). Модификатор `package` делает доступным член класса в рамках пакета.

В нашем классе `IntegerStack` (который сам является публичным, открытым), имеется три закрытых поля и четыре открытых метода. Поле `size` содержит размер стека, поле `stack[]` – переменная массива, который является внутренним представлением стека, поле `count` – текущий объем стека (сколько элементов в нем содержится на данный момент). Почему эти поля являются закрытыми? Именно для защиты данных класса от повреждения. Если бы они были публичными, любой мог бы изменить их. Например, можно было бы изменить значение размера стека на большее. Но тогда при операции добавления элемента в стек, рано или поздно наступил бы выход за границы массива, который представляет стек. Для недопущения такой ситуации данные поля объявлены закрытыми. Вся работа с классом возможна только через открытые методы класса, которые не приводят к неожиданным изменениям полей. Метод `push()` предназначен для добавления элемента в стек, а метод `pop()` – для извлечения элемента из стека. Методы `isFull()` и `isEmpty()` предназначены для проверки стека на пустоту или переполненность соответственно.

Теперь поговорим о создании объектов или экземпляров класса. Сначала объявляется переменная, которая имеет тип класса. Сама по себе переменная не определяет объект. Эта переменная может только ссылаться на объект. Создание объекта происходит с помощью оператора `new`. Эта операция резервирует память для объекта и возвращает ссылку на него. Эта ссылка сохраняется в объявленной ранее переменной. В нашем примере объекты класса создаются следующим образом:

```
IntegerStack st1 = new IntegerStack(5);
IntegerStack st2 = new IntegerStack(10);
```

Создание каждого из экземпляров происходит с помощью оператора `new`. Затем следует имя класса и передается параметр – размер стека. Оператор `new` вызывает специальный метод – конструктор класса. Прежде чем поговорить о конструкторе, рассмотрим методы.

Метод – еще один возможный член класса, наряду с переменными экземпляра. Общий вид метода следующий:

```
модификатор_доступа тип имя(список параметров) {
     // тело метода
}
```

«Тип» обозначает конкретный тип данных, который определяет тип значения, возвращаемого методом. Метод, который не возвращает значения, должен иметь модификатор `void`. Возврат значения из метода происходит с помощью оператора `return`. Имя метода служит для идентификации метода среди других членов класса, поэтому он должен быть уникален в области видимости класса (но есть исключение, о котором будет сказано дальше).

В отношении методов необходимо различать два понятия – параметр и аргумент. **Параметр** – определенная в методе переменная, которая принимает заданное значение при вызове метода. **Аргумент** – значение, передаваемое методу при вызове.

Есть два способа передачи аргументов – по значению и по ссылке. **Передача по значению** означает, что в параметр копируется значение

передаваемого аргумента. По значению передаются все параметры примитивного типа. Если в теле метода изменяется значение параметра, значение переменной, которая была передана как аргумент, не изменится. В этом смысл передачи по значению. Все объекты (то есть экземпляры классов) **передаются по ссылке**. В этом случае параметру присваивается ссылка на объект в памяти. В случае изменения такого параметра в теле метода, объект будет изменяться и там, откуда он был передан в метод, потому что объект всегда один, просто к нему получают доступ по ссылке из разных мест программы. Вообще, строго говоря, **в Java все аргументы передаются по значению**, потому что ссылка на объект хранится в переменной, и при передаче объекта класса ссылка на этот объект копируется в параметр метода. А «копируется» означает передачу по значению. Дальнейшее поведении зависит лишь от того, какого типа переменная – примитивного или ссылочного.

В Java разрешается определять в одном и том же классе несколько методов с одним именем, но с разными параметрами. Это называется **перегрузкой методов**. Перегружаемые методы различаются типом параметров и/или их количеством. Не каждый язык программирования допускает перегрузку методов.

Особый тип метода класса – **конструктор**. Конструктор инициализирует объект класса во время его создания. Именно конструктор вызывается оператором `new`. Имя конструктора должно совпадать с именем класса. Конструктор неявно возвращает объект, который имеет тип данных класса. В каждом классе должен быть конструктор. Даже если разработчик не определили ни одного конструктора, он создается неявно при компиляции. Такой конструктор не имеет параметров и не выполняет никакой логики, кроме создания экземпляра класса. Как и методы, конструктор допускает перегрузку. Перегрузка конструктора полезна, когда, например, необходимо в разных ситуациях по-разному инициализировать поля экземпляра, из-за чего конструктор должен принимать разное число параметров или параметры

разного типа. Для рассматриваемого класса `IntegerStack`, конструктор имеет вид:

```
public IntegerStack(int size) {
     this.size = size;
     stack = new int[size];
     count = 0;
}
```

Видно, что данный конструктор принимает на вход только один параметр, который определяет размер создаваемого стека. В теле конструктора это значения затем используется для инициализации массива, который является внутренним представлением стека в этом классе. Но мы также можем, к примеру, передавать этот массив как параметр. Логически данный функционал может понадобиться тогда, когда мы хотим уже заполненный значениями массив превратить в стек, чтобы потом с ним работать как со стеком. Такой конструктор будет выглядеть так:

```
public IntegerStack(int stack[], int count) {
    this.stack = stack;
    this.count = count;
    size = stack.length;
}
```

Также здесь в качестве параметра конструктора передается число элементов, находящихся в стеке. Теперь можно создавать объекты класса с помощью этого конструктора.

```
int stack3[] = {10, 8, 6, 4, 2};
IntegerStack st3 = new IntegerStack(stack3, 5);
```

Часто в методах класса используется ключевое слово `this`. Это ключевое слово возвращает ссылку на текущий объект. В теле метода класса мы можем использовать это ключевое слово для получения ссылки на объект, вызывавший этот метод.

Java является языком с автоматическим управлением памятью. Для высвобождения памяти используется так называемый **сборщик мусора**. Он подсчитывает для каждого существующего объекта в оперативной памяти

число ссылок на него. Когда на объект не ссылается ни одной переменной, то он попадает в кандидаты на удаление. В какой-то момент запускается процедура удаления и такие объекты удаляются. Это упрощенное объяснение работы сборщика мусора, на деле он устроен несколько сложнее.

Важным также является понятие статического члена класса. Для объявления статического члена класса в Java используется ключевое слово `static`. Статический член класса используется независимо от какого-либо объекта класса. В случае переменной это будет означать, что любой объект класса может получать доступ к этой статической переменной, при этом ее значение будет одинаковым для всех экземпляров. Если такая статическая переменная имеет публичный модификатор доступа, то она может быть получена через оператора точки после имени класса даже без создания экземпляра класса. Таким образом из класса `Math` получается константа «пи» (лабораторная работа №6). Статические методы могут вызывать только другие статические методы, им доступны только статические переменные, они не могут использовать в теле ключевое слово `this` (логично, ведь они не связаны ни с каким объектом, поэтому `this` не на что ссылаться).

Последним моментом, который будет рассмотрен в данной лабораторной работе, является ключевое слово `final`. Поле, объявленное как `final`, называется конечным и его значение не может быть изменено. Это означает, что такое поле должно быть инициализировано при объявлении, либо в конструкторе класса. Данное ключевое слово располагается перед типом объявляемой переменной. Например, `public final int DATABASE_NAME = "mydb"`. Класс также может быть конечным. Класс с модификатором `final` не может быть расширен, то есть не может быть унаследован другим классом. Таким образом можно явным образом запретить наследоваться от класса.

Наследование, абстрактные классы и интерфейсы будут рассмотрены в следующей лабораторной работе.

5 Упражнения по теме создания классов Java (по вариантам):

Каждый разрабатываемый класс должен, как правило, содержать следующие элементы: скрытые поля, конструкторы с параметрами и без параметров, методы, свойства. Методы и свойства должны обеспечивать непротиворечивый, полный, минимальный и удобный интерфейс класса. В программе должна выполняться проверка всех разработанных элементов класса.

1. Описать класс, реализующий десятичный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, — и метод, позволяющее получить его текущее состояние. Написать программу, демонстрирующую все разработанные элементы класса.

2. Описать класс, реализующий шестнадцатеричный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, — и метод, позволяющее получить его текущее состояние. При выходе за границы диапазона выбрасываются исключения.

Написать программу, демонстрирующую все разработанные элементы класса.

3. Описать класс, представляющий треугольник. Предусмотреть методы для создания объектов, перемещения на плоскости, изменения размеров и вращения на заданный угол. Описать метод для получения состояния объекта.

Написать программу, демонстрирующую все разработанные элементы класса.

4. Построить описание класса, содержащего информацию о почтовом адресе организации. Предусмотреть возможность раздельного изменения составных частей адреса и проверки допустимости вводимых значений.

Написать программу, демонстрирующую все разработанные элементы класса.

5. Составить описание класса для представления комплексных чисел. Обеспечить выполнение операций сложения, вычитания и умножения комплексных чисел.

Написать программу, демонстрирующую все разработанные элементы класса.

6. Составить описание класса для вектора, заданного координатами его концов в трехмерном пространстве. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.

Написать программу, демонстрирующую все разработанные элементы класса.

7. Составить описание класса прямоугольников со сторонами, параллельными осям координат. Предусмотреть возможность перемещения прямоугольников на плоскости, изменение размеров, построение наименьшего прямоугольника, содержащего два заданных прямоугольника, и прямоугольника, являющегося общей частью (пересечением) двух прямоугольников.

Написать программу, демонстрирующую все разработанные элементы класса.

8. Составить описание класса для представления даты. Предусмотреть возможности установки даты и изменения ее отдельных полей (год, месяц, день) с проверкой допустимости вводимых значений. В случае недопустимых значений полей выбрасываются исключения. Создать методы изменения даты на заданное количество дней, месяцев и лет.

Написать программу, демонстрирующую все разработанные элементы класса.

9. Составить описание класса для представления времени. Предусмотреть возможности установки времени и изменения его отдельных полей (час, минута, секунда) с проверкой допустимости вводимых значений. В

случае недопустимых значений полей выбрасываются исключения. Создать методы изменения времени на заданное количество часов, минут и секунд.

Написать программу, демонстрирующую все разработанные элементы класса.

10. Составить описание класса многочлена вида ax2 + bx + c. Предусмотреть методы, реализующие:

— вычисление значения многочлена для заданного аргумента;

— операцию сложения, вычитания и умножения многочленов с получением нового объекта-многочлена;

— вывод на экран описания многочлена.

Написать программу, демонстрирующую все разработанные элементы класса.

11. Описать класс, представляющий треугольник. Предусмотреть методы для создания объектов, вычисления площади, периметра и точки пересечения медиан. Описать метод для получения состояния объекта.

Написать программу, демонстрирующую все разработанные элементы класса.

12. Описать класс, представляющий круг. Предусмотреть методы для создания объектов, вычисления площади круга, длины окружности и проверки попадания заданной точки внутрь круга. Описать метод для получения состояния объекта.

Написать программу, демонстрирующую все разработанные элементы класса.

13. Описать класс для работы со строкой, позволяющей хранить только двоичное число и выполнять с ним арифметические операции. Предусмотреть инициализацию с проверкой допустимости значений.

Написать программу, демонстрирующую все разработанные элементы класса.

14. Описать класс дробей — рациональных чисел, являющихся отношением двух целых чисел. Предусмотреть методы сложения, вычитания, умножения и деления дробей.

Написать программу, демонстрирующую все разработанные элементы класса.

15. Описать класс «файл», содержащий сведения об имени, дате создания и длине файла. Предусмотреть инициализацию с проверкой допустимости значений полей. В случае недопустимых значений полей выбрасываются исключения. Описать метод добавления информации в конец файла и свойства для получения состояния файла. Написать программу, демонстрирующую все разработанные элементы класса.

Лабораторная работа № 8. Разработка интерфейсов

1 Цель работы

Изучить основные принципы разработки интерфейсов в Java.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Данная работа посвящена интерфейсам в Java.

В лабораторной работе будут рассмотрены следующие вопросы:

— Наследование в Java, понятия суперкласса и подкласса, ключевое слово `extends`.

— Ключевое слово `super`.

— Переопределение методов, сигнатура метода.

— Абстрактный класс в Java, ключевое слово `abstract`.

— Дополнительные функции ключевого слова `final`.

— Интерфейс в Java, ключевое слово `interface`.

— Реализация интерфейса, ключевое слово `implements`.

— Полиморфизм в Java, динамическая диспетчеризация методов.

— Методы с реализацией по умолчанию, ключевое слово `default`.

— Сравнительная таблица абстрактного класса и интерфейса в Java.

В конце предложены упражнения для закрепления материала.

Перед рассмотрением интерфейсов логично поговорить сначала о наследовании и абстрактных классах. Это наиболее правильный путь изучения. **Наследование** является одним из основополагающих принципов ООП. В общем случае создается класс с характеристиками, общими для группы классов. Затем этот общий класс наследуется более специализированными классами из этого набора. Каждый специализированный класс добавляет свои собственные возможности к унаследованным от родителя возможностям.

В Java наследуемый класс называется **суперклассом**, а наследующий класс – **подклассом**. Пример наследования классов приведен ниже.

Суперкласс `Box`.

```
public class Box {
    private double width;
    private double height;
    private double depth;

    public Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    public Box(double len) {
        width = height = depth = len;
    }

    public double volume() {
        return width * height * depth;
    }
}
```

Наследующий класс `BoxWeight`.

```
class BoxWeight extends Box {
    private double weight;


    public BoxWeight(double w, double h, double d, double m) {
        super(w, h, d);
        weight = m;
    }

    public BoxWeight(double len, double m) {
```

```
        super(len);
        weight = m;
    }
}
```

Использование класса `BoxWeight`.

```
public class Main {
    public static void main(String[] args) {
        BoxWeight box1 = new BoxWeight(10, 20, 30, 40);
        BoxWeight box2 = new BoxWeight(20, 50);
        System.out.println("Объем первой коробки: " +
box1.volume());
        System.out.println("Объем второй коробки: " +
box2.volume());
    }
}
```

Чтобы наследовать класс необходимо после определения одного класса (после его имени) написать ключевое слово `extends`, а потом написать название суперкласса.

Класс `BoxWeight` наследует все характеристики класса `Box` и добавляет к ним компонент `weight`. Класс `BoxWeight` включает все члены класса Box. Именно поэтому экземпляры класса `BoxWeight` могут вызывать метод `volume()`, хотя в классе `BoxWeight` этого метода явно не определено.

Несмотря на то, что класс Box является суперклассом, он в то же время остается полностью независимым классом. В то же время, подкласс сам может являться для другого класса суперклассом.

Нужно отметить, что в Java **нет множественного наследования**, и один класс может наследоваться только от одного класса.

В приведенном примере используется ключевое слово `super`, оно может использоваться для вызова конструктора суперкласса (но это не единственное его применение). Очевидно, использование конструктора суперкласса сокращает написание излишнего кода. Зачем в подклассе заново определять параметры длины, ширины и глубины коробки, если в суперклассе уже определен такой конструктор? Гораздо удобнее вызвать конструктор суперкласса, принимающий три эти параметра, а потом уже инициализировать

специфичное для класса `BoxWeight` поле `weight`. При этом нужно помнить, что вызов конструктора суперкласса всегда должно выполняться в конструкторе подкласса в первую очередь. Также, при иерархическом наследовании стоит понимать, что в подклассе вызывается конструктор ближайшего в иерархии суперкласса.

Интересным механизмом является **переопределение методов** (overriding), который не нужно путать с перегрузкой методов (overloading). Важным здесь является понятие сигнатуры метода. **Сигнатура метода** – имя метода и его параметры в упорядоченном виде. Параметры определяются в виде упорядоченного множества пар «тип параметра – имя параметра». Так вот, подкласс может определить в своем теле метод с такой же сигнатурой, которую имеет метод в суперклассе, и предложить собственную реализацию метода. Таким образом, у нас будет подкласс, который специализировал метод суперкласса с помощью переопределения. В примере с классами `Box` и `BoxWeight`, в подклассе `BoxWeight` можно было переопределить метод `volume()`, чтобы данный класс имел собственную реализацию данного метода.

Тема наследования в Java очень обширна, но в данной краткой теории ограничимся этими сведениями и перейдем к рассмотрению абстрактных классов.

Иногда суперкласс требуется определить таким образом, чтобы объявить в нем структуру заданной абстракции, не предоставляя полную реализацию каждого метода. Для этого нужно создать суперкласс, определяющий только обобщенную форму для совместного использования всеми его подклассами, в каждом из которых могут добавлены требующиеся подробности. В таком классе определяется характер методов, которые должны быть реализованы в подклассах. Подобная ситуация может, например, возникнуть тогда, когда в суперклассе не удается полностью реализовать метод. Пример объявления и использования **абстрактного класса** приведен ниже.

Абстрактный класс `Figure` (Фигура).

```
public abstract class Figure {
    protected double dim1;
    protected double dim2;

    public Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    abstract double area();
}
```

Класс `Rectangle` (Прямоугольник).

```
public class Rectangle extends Figure {
    public Rectangle(double a, double b) {
        super(a, b);
    }

    double area() {
        System.out.println("Площадь четырехугольника");
        return dim1 * dim2;
    }
}
```

Класс `Triangle` (Треугольник).

```
public class Triangle extends Figure {
    public Triangle(double a, double b) {
        super(a, b);
    }

    double area() {
        System.out.println("Площадь треугольника");
        return dim1 * dim2 / 2;
    }
}
```

Использование данных классов:

```
public class Main {
    public static void main(String[] args) {
        Rectangle r = new Rectangle(2, 2);
        Triangle t = new Triangle(2, 5);
        System.out.println(r.area());
        System.out.println(t.area());
```

```
    }
}
```

При выполнении данного кода в консоль будет выведено следующее:

```
Площадь четырехугольника
4.0
Площадь треугольника
5.0
```

Метод суперкласса, который должен быть переопределен в подклассе, помечается с помощью модификатора `abstract`. Любой класс, который содержит хотя бы один абстрактный метод, должен быть объявлен как абстрактный. Для этого при объявлении класса перед ключевым словом `class` указывается слово `abstract`. Абстрактный класс не может быть инстанцирован, то есть он не может иметь экземпляров. Любой класс, наследующийся от абстрактного класса, должен реализовывать все абстрактные методы, либо сам должен быть объявлен абстрактным.

В приведенном примере классы прямоугольника и треугольника наследуются от абстрактного класса фигуры. В классе `Figure` определен абстрактный метод `area()`, который должен вычислять площадь фигуры. В классе `Figure` этот метод объявлен абстрактным, потому что у фигуры в общем понимании нельзя вычислить площадь, нам необходимо знать, что это за фигура. Поэтому использование абстрактного метода в данном случае логически оправдано. Классы `Rectangle` и `Triangle` предоставляют собственную реализацию данного метода.

Ранее уже было сказано про ключевое слово `final`. Но было сказано только о создании констант с помощью него. В области наследования это ключевое слово обретает два новых значения. Первое использование – запрет на переопределение метода. Для этого в объявлении метода перед указанием возвращаемого типа указывается слово `final`. Тогда при попытке переопределить данный метод в подклассе возникнет ошибка. Второе использование – запрет на наследование класса. Класс с модификатором `final` не может быть суперклассом для других классов, то есть не может быть унаследован каким-либо другим классом.

Когда наследование и абстрактные классы рассмотрены, можно перейти к рассмотрению интерфейсов.

С помощью ключевого слова `interface` можно полностью абстрагировать интерфейс класса от его реализации, то есть можно указать, что именно должен делать класс, при этом не определяя то, как он должен это делать. Интерфейсы похожи на классы, но не содержат переменные экземпляра, а у методов нет тела. Как уже было сказано ранее, один класс может непосредственно наследовать только один класс, но при этом класс может реализовывать любое количество интерфейсов. Общая форма объявления интерфейса указана ниже.

```
модификатор_доступа interface имя {
     тип имя_конечной_переменной1 = значение;
     возвращаемый_тип имя_метода1(список_параметров);
     ...
}
```

Если интерфейс объявлен без модификатора доступа, интерфейс доступен только другим членам пакета, в котором он объявлен. Публичный интерфейс доступен в любом другом коде.

Члены интерфейса (поля и методы) имеют свои особенности. Поля интерфейса неявно объявляются с модификаторами `final` и `static`, то есть они должны быть проинициализированы при объявлении, а затем их значение не может быть изменено. Объявленные методы не содержат тел, по сути, они являются абстрактными методами, поэтому они должны быть определены в классе, который реализует интерфейс. Если класс реализует не все методы интерфейса, он должен быть абстрактным.

Чтобы реализовать интерфейс, нужно в определение класса добавить ключевое слово `implements`, а после него указать необходимые интерфейсы через запятую. При этом сначала перечисляются, если необходимо, наследуемые классы через ключевое слово `extends`, а уже потом через `implements` интерфейсы.

```
модификатор_доступа class имя_класса extends суперкласс implements
```

```
                    интерфейс1, интерфейс2 ... {
    // тело класса
}
```

В лабораторной работе №7, которая была посвящена рассмотрению классов в Java, в качестве примера использовался класс `IntegerStack`. Теперь, зная о интерфейсах, можно определить интерфейс, который будет объявлять общий функционал целочисленных стеков. Можно, например, создать два класса, которые будут реализовывать интерфейс, один класс будет представлять стек фиксированного размера для хранения целых чисел, а второй будет представлять динамически расширяющийся стек для хранения целых чисел. Код интерфейса, код классов и пример их использования приведен ниже.

Интерфейс `IntegerStack`.

```
public interface IntegerStack {
    void push(int item);
    int pop();
}
```

Класс `FixedIntegerStack`.

```
public class FixedIntegerStack implements IntegerStack {
    private int size;
    private int stack[];
    private int count;

    public FixedIntegerStack(int size) {
        this.size = size;
        stack = new int[size];
        count = 0;
    }

    public void push(int item) {
        if (!isFull()) {
            stack[count++] = item;
        } else {
            System.out.println("Стек переполнен");
        }
    }

    public int pop() {
        if (!isEmpty()) {
            return stack[--count];
```

```
        } else {
            System.out.println("Стек пуст");
            return 0;
        }
    }

    public boolean isFull() {
        return count == size;
    }

    public boolean isEmpty() {
        return count == 0;
    }
}
```

Класс `DynamicIntegerStack`.

```
public class DynamicIntegerStack implements IntegerStack {
    private int size;
    private int stack[];
    private int count;

    public DynamicIntegerStack(int size) {
        this.size = size;
        stack = new int[size];
        count = 0;
    }

    public void push(int item) {
        if (!isFull()) {
            int temp[] = new int[size * 2];
            System.arraycopy(stack, 0, temp, 0, size);
            size = size * 2;
            stack = temp;
            stack[count++] = item;
        } else {
            System.out.println("Стек переполнен");
        }
    }

    public int pop() {
        if (!isEmpty()) {
            return stack[--count];
        } else {
            System.out.println("Стек пуст");
            return 0;
        }
    }

    public boolean isFull() {
```

```
        return count == size;
    }

    public boolean isEmpty() {
        return count == 0;
    }
}
```

Использование классов.

```
public class Main {
    public static void main(String[] args) {
        FixedIntegerStack st1 = new FixedIntegerStack(5);
        DynamicIntegerStack st2 = new DynamicIntegerStack(5);
        System.out.println("Заполнение стека 1");
        for (int i = 0; i < 10; i++) {
            st1.push(i);
        }
        System.out.println("Заполнение стека 2");
        for (int i = 0; i < 10; i++) {
            st2.push(i);
        }
    }
}
```

Результат выполнения кода.

```
Заполнение стека 1
Стек переполнен
Стек переполнен
Стек переполнен
Стек переполнен
Стек переполнен
Заполнение стека 2
```

Таким образом, определен один интерфейс с двумя основными методами стека – `push()` и `pop()`. Классы предоставляют свою реализацию каждого из методов. В классе `DynamicIntegerStack` в методе `push()` реализовано автоматическое увеличение размера массива, который хранит элементы стека. Как видно из примера использования экземпляров класса, было создано два стека объемом 5 элементов. Затем они намеренно заполнялись 10 значениями. Очевидно, что первый стек переполнился после 5 элемента и больше не принял ни одного элемента. Внутренний массив второго стека также переполнился и был расширен путем создания массива большего размера и копирования в него элементов из старого массива.

Заметьте, что для копирования содержимого одного массива в другой используется статический метод `arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`. В качестве параметров он ожидает массив-источник `src`, индекс `srcPos`, с которого нужно копировать данные, третьим параметром нужно передать массив `dest`, в который копируются данные, четвертый параметр – индекс `destPos` во втором массиве, начиная с которого нужно записывать данные, пятым параметром передается число элементов `length`, которые будут скопированы.

Вообще, можно было построить иерархию классов по-другому. Можно определить «промежуточный» абстрактный класс, который бы содержал в себе всю общую логику для классов стеков фиксированного и динамического размера. Реализация данного подхода приведена ниже.

Интерфейс `IntegerStack`.

```
public interface IntegerStack {
    void push(int item);
    int pop();
}
```

Абстрактный класс `AbstractIntegerStack`.

```
public abstract class AbstractIntegerStack implements IntegerStack
{
    protected int size;
    protected int stack[];
    protected int count;

    public abstract void push(int item);

    public int pop() {
        if (!isEmpty()) {
            return stack[--count];
        } else {
            System.out.println("Стек пуст");
            return 0;
        }
    }

    public boolean isFull() {
        return count == size;
    }
```

```
    public boolean isEmpty() {
        return count == 0;
    }
}
```

Класс `FixedIntegerStack`.

```
public class FixedIntegerStack extends AbstractIntegerStack {

    public FixedIntegerStack(int size) {
        this.size = size;
        stack = new int[size];
        count = 0;
    }

    public void push(int item) {
        if (!isFull()) {
            stack[count++] = item;
        } else {
            System.out.println("Стек переполнен");
        }
    }
}
```

Класс `DynamicIntegerStack`.

```
public class DynamicIntegerStack extends AbstractIntegerStack {

    public DynamicIntegerStack(int size) {
        this.size = size;
        stack = new int[size];
        count = 0;
    }

    public void push(int item) {
        if (!isFull()) {
            int temp[] = new int[size * 2];
            System.arraycopy(stack, 0, temp, 0, size);
            size = size * 2;
            stack = temp;
            stack[count++] = item;
        } else {
            System.out.println("Стек переполнен");
        }
    }
}
```

Использование классов не изменилось, как не изменился и вывод в консоль, поэтому он здесь не приводится. Теперь каждый из классов стека реализуют только конструктор и метод `push()`, всю остальную логику реализует абстрактный класс. В том числе он объявляет и поля для этих классов.

Может возникнуть закономерный вопрос – для чего это нужно? Зачем строить такую иерархию классов, если можно просто создать два файла и два класса `FixedIntegerStack` и `DynamicIntegerStack`, и они буду выполнять нужные функции? Неужели просто сократить число строк кода? Да, число строк кода уменьшается, но это далеко не главное. С помощью абстрактных конструкций (интерфейсов и абстрактных классов) реализуется важнейший механизм ООП – полиморфизм.

Полиморфизм – способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе объекта. В Java можно работать со многими типами, как с одним. Например, ранее в лабораторной работе были приведены классы `Figure`, `Rectangle` и `Triangle`. Для создания объекта классов прямоугольника и треугольника использовалась обычная конструкция, например `Rectangle r = new Rectangle(5, 5)`. В данном случае и переменная ссылочного типа `r`, и сам объект имеют один и тот же тип - `Rectangle`. Но мы знаем, что классы прямоугольника и треугольника наследуются от абстрактного класса `Figure`. Создать объекты класса `Figure` нельзя (он абстрактный), но вот ссылочную переменную такого типа объявить можно. Теперь у нас есть ссылка типа `Figure`, но чем же ее инициализировать, если объект этого класса создать нельзя? И тут начинается самое интересное – ссылочная переменная суперкласса может ссылаться на объект подкласса – `Figure f = new Rectangle(5, 5)`. Так же можно создать и треугольник.

Далее нужно сказать, что в Java полиморфизм происходит во время выполнения, а не компиляции. В определении полиморфизма есть слово «использовать», а используются объекты путем вызова их методов.

Соответственно, в суперклассе может быть объявлен метод, но не реализован. В нашем примере это метод для вычисления площади фигуры `area()`. Это метод реализуется в подклассах. Когда мы вызываем у объекта типа `Rectangle` или `Triangle` метод `area()`, то все просто – вызывается конкретная реализация, описанная в этих классах. Но какой метод вызывается, когда ссылка типа `Figure`, а объект типа `Rectangle` или `Triangle`? Здесь в дело вступает один из полезнейших механизмов в Java – **динамическая диспетчеризация методов**. Когда переопределенный метод вызывается по ссылке на суперкласс `Figure`, нужный вариант этого метода выбирается в Java в зависимости от типа объекта, на который делается ссылка в момент вызова - `Rectangle` или `Triangle`. Этот выбор делается во время выполнения программы. По ссылке на разные типы объектов будут вызываться разные варианты переопределенного метода.

Иначе говоря, вариант переопределенного метода выбирается для выполнения в зависимости от типа объекта, на который делается ссылка, а не типа ссылочной переменной. Так, если суперкласс содержит метод, переопределяемый в подклассе, то по ссылке на разные типы объектов через ссылочную переменную суперкласса будут выполняться разные варианты этого метода. На практике часто возникает ситуация, когда метод должен принимать объекты и работать с ними, не задумываясь об их конкретной внутренней реализации. Для примера создадим метод, который будет распечатывать площадь любой фигуры в консоль. Пример кода приведен ниже.

```
public class Main {
    public static void main(String[] args) {
        Figure f1 = new Rectangle(2, 2);
        Figure f2 = new Triangle(5, 4);
        System.out.println("Вычисляем площадь прямоугольника: ");
        printArea(f1);
        System.out.println("Вычисляем площадь треугольника: ");
        printArea(f2);
    }

    public static void printArea(Figure f) {
        System.out.println("Площадь фигуры: " + f.area());
    }
}
```

Как видно из кода выше, метод `printArea()` принимает в качестве параметра ссылку типа `Figure`. Метод внутри тела вызывает метод `area()` для вычисления площади фигуры. В результате запуска этого кода, в консоль будет выведено следующее:

```
Вычисляем площадь прямоугольника:
Площадь четырехугольника
Площадь фигуры: 4.0
Вычисляем площадь треугольника:
Площадь треугольника
Площадь фигуры: 10.0
```

Методу `printArea()` абсолютна неважна внутренняя реализация метода `area()`, ему важно лишь то, что в типе `Figure` определен этот метод (это класс предоставляет «интерфейс» из одного этого метода). Во время выполнения конкретная реализация будет вызвана у объекта конкретного типа `Rectangle` или `Triangle`, на которую указывает ссылка типа `Figure`. Теперь, даже если будут определены новые классы фигур, метод `printArea()` не нужно будет переписывать. В этом проявляется мощность полиморфизма.

В примере со стеками все точно также. Можно создавать ссылки типа `IntegerStack` или `AbstractIntegerStack` и инициализировать их объектами классов `FixedIntegerStack` или `DynamicIntegerStack`. Тогда, метод, которому требуется стек для выполнения своих функций, может в качестве одного из параметров принимать ссылку типа `IntegerStack`. В этом заключается сила полиморфизма – мы можем работать с объектами разными типов, как будто все они одного типа.

В JDK (Java Development Kit) 8 появилась возможность создавать в интерфейсах методы с реализацией по умолчанию. Это позволяет объявлять в интерфейсе метод, который имеет тело. Методы с реализацией по умолчанию создаются с помощью ключевого слова `default`. Важно отметить, что внедрение методов с реализацией по умолчанию не изменяет главную особенность интерфейсов: неспособность сохранять данные состояния. В частности, в интерфейсе по-прежнему недопустимы переменные экземпляра. Следовательно, интерфейс отличается от класса тем, что он не допускает сохранение состояния. Более того, создавать экземпляр самого интерфейса

нельзя. Поэтом интерфейс должен быть по-прежнему реализован в классе, если требуется получить его экземпляр, несмотря на возможность определять в интерфейсе методы с реализацией по умолчанию.

В конце теоретических сведений для выполнения данной лабораторной работы приведем таблицу, в которой сравниваются абстрактный класс и интерфейс.

Т а б л и ц а 1 — Отличия абстрактного класса от интерфейса в Java

	AbstractClass	Interface
Ключевое слово, используемое для описания	Для определения абстрактного класса используется ключевое слово `abstract`	Для определения интерфейса используется ключевое слово `interface`
Ключевое слово, используемое в реализации	Для наследования от абстрактного класса используется ключевое слово `extends`	Для определения интерфейса используется ключевое слово `implements`
Конструктор	Определение конструктора разрешено	Объявление конструктора запрещено
Реализация методов	Допустима	Вплоть до Java 8 разрешено только объявлять метода, но реализация их запрещена Начиная с Java 8 стала допустима реализация `static` методов, а реализация `non-static` (нестатических) методов стала доступна посредством ключевого слова `default`
Поля	Может иметь как `static`, так и `non-static` поля (также и с `final` полями)	Любое поле интерфейса по умолчанию является `public static final` (и иных иметь не может)
Модификаторы доступа	Члены абстрактного класса могут иметь любой модификатор доступа: `public`, `protected`, `private`, либо модификатор доступа по умолчанию (`package`)	Члена интерфейса по умолчанию являются `public`
Наследование	Может расширить (`extends`) любой другой класс и реализовать (`implements`) сколько угодно интерфейсов	Может расширить (`extends`) другой интерфейс

5 Упражнения по теме интерфейсов в Java (по вариантам):

Общая часть задания: из предыдущей работы по классам, взять задание соответствующее варианту и разработать собственный интерфейс на основе этого задания. Можно реализовать иерархию из интерфейса, абстрактного класса и его подклассов по теме варианта.

Лабораторная работа № 9. Программирование сетевых сокетов. Разработка клиент-серверного приложения UDP

1 Цель работы

Изучить классы и методы, которые позволяют устанавливать соединение с удалённым компьютером, используя протокол UDP.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Данная работа посвящена программированию сетевых UDP сокетов в Java.

В лабораторной работе будут рассмотрены следующие вопросы:

— Понятие сокета.

— Реализация сокета в пакете `java.net`.

— UDP и TCP, их особенности и отличия.

— Что такое дейтаграмма.

— Классы `DatagramPacket` и `DatagramSocket`.

— Создание простейшего клиент-серверного приложения с использованием UDP протокола.

В конце предложены упражнения для закрепления материала.

Java был разработан как язык сетевого программирования для обеспечения возможности создания клиент/серверных приложений, которые взаимодействуют друг с другом в сети. Java обеспечивает обширную библиотеку сетевых классов, которые позволяют быстро получать доступ к сетевым ресурсам.

Основные сетевые возможности Java реализуются в классах и интерфейсах пакета `java.net`, посредством которых Java обеспечивает потоковое взаимодействие, позволяющее приложениям рассматривать соединение, как поток данных. Классы и интерфейсы пакета `java.net` также предлагают коммуникации на основе передачи отдельных пакетов информации, которые обычно используется для передачи аудио и видео через Интернет.

Взаимодействие в сети рассматривается на основе понятия сокетов, которые позволяют приложениям рассматривать сетевые подключения как файлы, и программа может читать из сокета или писать в сокет, как она делает это с файлом. Слово «сокет» в переводе на русский язык означает «гнездо». Это название образовалось по аналогии с гнёздами (разъемами) на аппаратуре, с которыми стыкуются разъёмы. Сокет представляет собой программную конструкцию (объект), которая определяет конечную точку соединения.

Существуют два механизма, предназначенных для сетевого взаимодействия программ, – это сокеты датаграмм, которые используют пользовательский датаграммный протокол (User Datagram Protocol) (UDP) без установления соединения, и сокеты, использующие Протокол управления передачей / Межсетевой протокол (Transmission Control Protocol/Internet Protocol) (TCP/IP), устанавливающий соединение.

Дейтаграмма - пакет данных, отправленный по сети, прибытие которого, время прибытия и содержание не гарантировано. Не гарантируется также и порядок доставки пакетов. При передаче пакета UDP по какому-либо адресу нет никакой гарантии того, что он будет принят, а также, что по этому адресу вообще существует потребитель пакетов. Аналогично, когда вы получаете датаграмму, у вас нет никаких гарантий, что она не была повреждена в пути

следования или что отправитель ожидает подтверждения получения датаграммы. Использование UDP может привести к потере или к дублированию пакетов, что приводит к дополнительным проблемам, связанным с проверкой ошибок и обеспечением надёжности передачи данных. Если вам необходимо добиться оптимальной производительности, и вы готовы сократить затраты на проверку целостности информации, пакеты UDP могут оказаться весьма полезными.

При использовании потоковых сокетов, программа устанавливает соединение с другим сокетом и, пока соединение установлено, поток данных протекает между программами, и говорят, что потоковые сокеты обеспечивают обслуживание на основе установления соединения. TCP/IP является потоковым протоколом на основе установления двунаправленных соединений точка-точка между узлами Интернет, и взаимодействие между компьютерами по этому протоколу предназначено для реализации надежной передачи данных. Все данные, отправленные по каналу передачи, получаются в том же порядке, в котором они передавались. В отличие от дейтаграммных сокетов, сокеты TCP/IP реализуют высоконадежные устойчивые соединения между клиентом и сервером.

Java позволяет разрабатывать сетевые приложения с использованием дейтаграммных сокетов UDP и сокетов TCP/IP. Сокеты UDP используют протокол UDP для взаимодействия приложений через сеть. UDP является быстрым, без установления соединения и ненадежным протоколом. Пакет `java.net` содержит следующие два класса, позволяющие применять сокеты UDP в приложении Java: Класс `DatagramPacket`, Класс `DatagramSocket`.

Объект типа `DatagramPacket` является контейнером данных, состоящим из дейтаграммных пакетов, которые посылаются или принимаются через сеть. Следующие конструкторы используются для инициализации объектов `DatagramPacket`.

`public DatagramPacket(byte[] buffer, int buffer_length)`: Создает объект `DatagramPacket`, который принимает и сохраняет данные в

массиве `byte`. Длина буфера массива `byte` задается вторым параметром `buffer_length`.

`public DatagramPacket(byte[] buffer, int buffer_length, InetAddress address, int port)`: Создает объект `DatagramPacket`, который посылает пакеты данных заданной длины. Пакеты данных посылаются на компьютер с заданным IP адресом и номером порта, передаваемыми как параметры.

Методы, определенные в классе `DatagramPacket`, могут быть использованы после инициализации объекта класса `DatagramPacket`. Таблица 2 представляет методы класса `DatagramPacket`.

Т а б л и ц а 2 — Методы класса `DatagramPacket`

Метод	Описание
`InetAddress getAddress()`	Возвращает адрес источника (для принимаемых дейтаграмм) или места назначения (для отправляемых дейтаграмм)
`byte[] getData()`	Возвращает массив байтов данных, содержащихся в дейтаграмме. Используется в основном для извлечения данных из дейтаграммы после ее приема
`int getLength()`	Возвращает длину достоверных данных, содержащихся в массиве байтов, который должен быть возвращен из метода `getData()`. Эта длина может не полностью совпадать с длиной массива байтов
`int getOffset()`	Возвращает начальный индекс данных
`int getPort()`	Возвращает номер порта
`void setAddress(InetAddress ip_address)`	Устанавливает адрес, по которому отправляется пакет. Адрес передается в параметр `ip_address`
`void setData(byte[] data)`	Устанавливает буфер в виде заданного массива

	`data`, нулевое смещение и длину, равную количеству байтов в указанном массиве `data`
`void setData(byte[] data, int index, int length)`	Устанавливает буфер в виде заданного массива `data`, смещение – по указанному индексу `index`, а длину – по заданному параметру `length`
`void setLength(int length)`	Устанавливает длину пакета равной `length`
`void setPort(int port)`	Устанавливает заданный порт

Класс `DatagramSocket` содержит функциональность для управления объектами `DatagramPacket`. Объекты `DatagramPacket` отправляют и принимают сохраненные данные, используя объект `DatagramSocket`. Следующие конструкторы используются для инициализации объекта `DatagramSocket`:

```
DatagramSocket() throws SocketException

DatagramSocket(int port) throws SocketException

DatagramSocket(int port, InetAddress ip_address) throws
SockerException

DatagramSocket(SocketAddress address) throws SocketException
```

Первый конструктор создает объект класса `DatagramSocket`, связанный с любым незанятым портом локального компьютера. Второй конструктор – объект класса `DatagramSocket`, связанный с указанным портом. Третий конструктор – объект класса `DatagramSocket`, связанный с указанным портом и адресом. Четвертый конструктор – объект класса `DatagramSocket`, связанный с заданным объектом класса `SocketAddress`. Класс `SocketAddress` является абстрактным и реализуется конкретным классом `InetSocketAddress`, который инкапсулирует IP-адрес и номер порта. Все конструкторы класса могут генерировать исключение типа `SocketException`, если при создании сокета возникают ошибки.

В классе `DatagramSocket` определяется большое количество методов. Наиболее важными из них являются методы `send()` и `receive()`, общие формы которых представлены ниже:

```
void send(DatagramPacket packet) throws IOException
void receive(DatagramPacket packet) throws IOException
```

Метод `send()` отправляет указанный пакет по указанному адресу и порту. Метод `receive()` ожидает прихода пакета от отправителя.

В классе `DatagramSocket` определяется также метод `close()`, закрывающий сокет данного типа. Начиная с версии JDK 7, класс `DatagramSocket` реализует интерфейс `AutoCloseable`, что позволяет управлять сокетом типа `DatagramSocket` в блоке оператора `try-with-resources` (`try` с ресурсами). Этот оператор автоматически закрывает сокет в конце работы. Другие методы из данного класса предоставляют доступ к различным атрибутам сокета, они перечислены в таблице 3.

Т а б л и ц а 3 — Методы класса `DatagramSocket`

Метод	Описание
`InetAddres getInetAddress()`	Возвращает адрес, если сокет подключен, а иначе – `null`
`int getLocalPort()`	Возвращает номер локального порта
`int getPort()`	Возвращает номер порта, к которому подключен сокет. Если сокет не подключен ни к одному из портов, то возвращается значение -1
`boolean isBound()`	Возвращает логическое значение `true`, если сокет привязан к адресу, а иначе – логические значение `false`
`boolean isConnected()`	Возвращает логическое значение `true`, если сокет подключен к серверу, а иначе – логические значение `false`

`void setSoTimeout(int timeout) throws SocketException`	Устанавливает период ожидания, равный заданному количеству миллисекунд

Необходимо разработать клиент/серверное приложение, в котором сервер может распространять сообщения всем клиентам, зарегистрированным в группе 233.0.0.1, порт 1502. Пользователь сервера должен иметь возможность ввода и отправки текстовых сообщений, а пользователь-клиент просматривает полученные сообщения. Ниже будет приведен исходный код простого UDP-сервера и клиента, решающий поставленную задачу.

Класс `Server`.

```
import java.io.*;
import java.net.*;

public class Server {

    public Server() throws IOException {
        System.out.println("Sending messages");

        // Вызов метода transmit() для передачи сообщение всем
        // клиентам, зарегистрированным в группе
        transmit("233.0.0.1");
    }

    public void transmit(String ipGroup) {
        // Создается объект DatagramSocket для
        // приема запросов клиента
        try (DatagramSocket socket = new DatagramSocket()) {
            // создается входной поток для приема
            // данных с консоли
            try (BufferedReader in = new BufferedReader(new
InputStreamReader(System.in))) {
                while (true) {
                    System.out.println("Введите строку для
передачи клиентам: ");
                    // Чтение из консоли
                    String str = in.readLine();
                    // Преобразование строки в байты
                    byte[] buffer = str.getBytes();
                    // Получение ip-адреса
                    InetAddress address =
InetAddress.getByName(ipGroup);
                    // Посылка пакета датаграмм на порт номер 1502
                    DatagramPacket packet = new
DatagramPacket(buffer,
                            buffer.length, address,
                            1502);
```

```
                    // Посылка сообщений всем клиентам в группе
                    socket.send(packet);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] arg) throws Exception {
        // Запуск сервера
        new Server();
    }
}
```

Класс `Client`.

```
import java.net.*;

public class Client {
    public static void main(String[] arg) throws Exception {
        InetAddress address = InetAddress.getByName("233.0.0.1");
        System.out.println("Ожидание сообщения от сервера");
        // Создание объекта MulticastSocket для получения
        // данных от группы, используя номер порта 1502
        try (MulticastSocket socket = new MulticastSocket(1502)) {
            // Регистрация клиента в группе
            socket.joinGroup(address);
            while (true) {
                // Создание буфера для хранения данных
                byte[] buffer = new byte[256];
                // Создание дейтаграммы, связывание с буфером
                DatagramPacket packet = new DatagramPacket(buffer,
buffer.length);
                // Получение данных от сервера
                socket.receive(packet);
                // Получение данных из дейтаграммы
                String str = new String(packet.getData());
                // Вывод сообщения на экран
                System.out.println("Получено сообщение: " +
str.trim());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Проблем с пониманием кода возникнуть не должно, так как каждая строчка снабжена комментарием.

Теперь будет описана процедура запуска данной программы в среде Intellij IDEA.

Каждый класс содержит метод `main()`, поэтому он может быть запущен как отдельная программа. Действительно, программы сервера и клиента являются отдельными, самостоятельными программами. Удобство Intellij IDEA заключается в том, что в данной среде разработки возможен одновременный запуск нескольких файлов с методом `main()`. Но при этом нужно понимать, что это не единственный способ запуска клиента и сервера. Можно сделать это в консоли, сначала скомпилировав исходный код, а затем передать виртуальной машине Java (JVM) этот скомпилированный файл, содержащий байт-код для JVM. Для запуска клиента и сервера понадобится, при использовании такого способа, две консоли. Для удобства будем запускать файлы в Intellij IDEA.

Для начала необходимо открыть файлы клиента и сервера. Там, где показывается нумерация строк кода, будет зеленый треугольник. Нажав на него, откроется меню запуска, для наших целей нужно выбрать первую опцию. Данный процесс проиллюстрирован на рисунках 6 и 7.

Рисунок 6 – Кнопка запуска файла

Рисунок 7 – Выбор нужной опции запуска

Запустить файл в Intellij IDEA можно разными способами, но этот самый удобный.

Итак, сервер запущен. Откроется консоль, которая является стандартным вводом/выводом для программы сервера. Зеленая точка слева от имени класса означает, что данная программа в данный момент выполняется. На рисунке 8 показана консоль сервера.

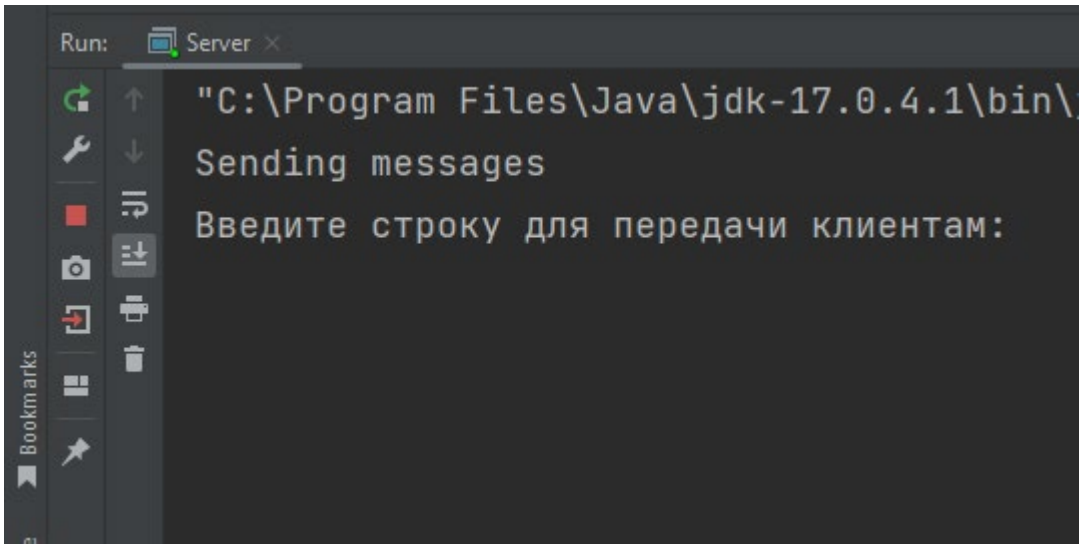


Рисунок 8 – Консоль сервера

Аналогичные действия выполняются для запуска клиентской программы. На рисунке 9 показана консоль клиента.

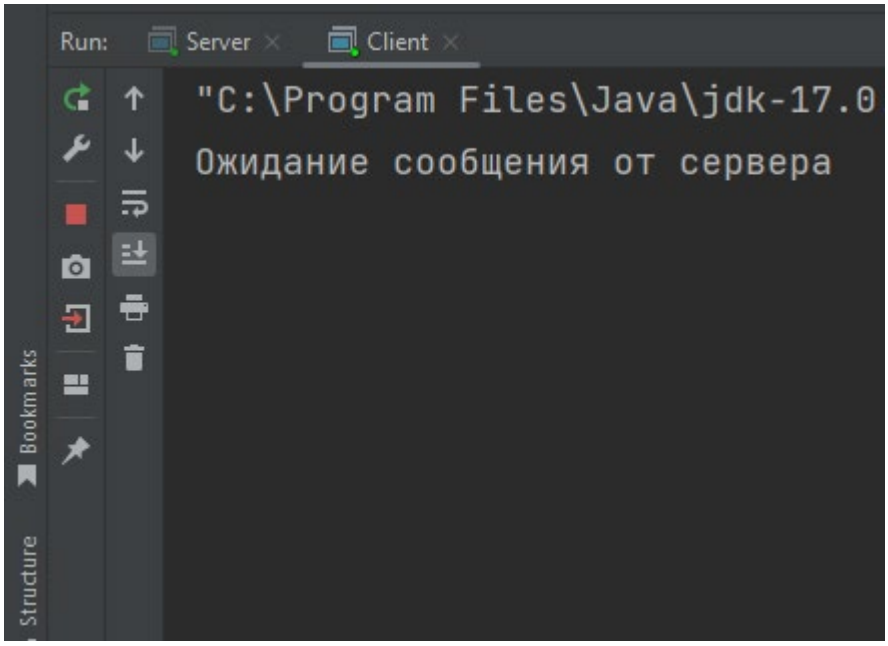


Рисунок 9 – Консоль клиента

Демонстрация работы не приводится, тестирование программы необходимо провести самостоятельно и отразить в отчете.

5 Упражнения по теме UDP сокетов в Java:

Разработать клиент-серверное приложение с использованием протокола UDP, в соответствии с рассмотренным примером. Изучить все основные классы для разработки данного вида сетевых приложений, алгоритмы работы клиента и сервера. В отчете привести результат работы программы (отправку сообщения сервером и прием этого сообщения клиентом).

Лабораторная работа № 10. Разработка сетевых клиент-серверных приложений с использованием TCP

1 Цель работы

Изучить классы и методы, которые позволяют устанавливать соединение с удалённым компьютером, используя протокол TCP. Разработать сетевое приложение с архитектурой клиент-сервер, использующее протокол TCP.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Данная работа посвящена программированию сетевых UDP сокетов в Java.

В лабораторной работе будут рассмотрены следующие вопросы:

— Особенности TCP сокета.

— Классы `Socket` и `ServerSocket`, их конструкторы и основные методы.

— Знакомство с классом `Thread` и его методами.

— Создание класса, реализующего механизм многопоточности, переопределение метода `run()`, запуск потока через метод `start()`.

— Создание клиент-серверного приложения с использованием TCP сокетов и механизма многопоточности.

В конце предложены упражнения для закрепления материала.

Java поддерживает классы и методы, которые позволяют устанавливать соединение с удаленным компьютером, используя протокол TCP. В отличие от UDP, TCP является протоколом, ориентированным на установление соединения, которое гарантирует надежную связь между приложениями клиента и сервера. Взаимодействие с использованием протокола TCP, начинается после установления соединения между сокетами клиента и сервера. Сокет сервера "слушает" запросы на установление соединения, отправленные сокетами клиентов, и устанавливает соединение. После установления соединения между приложениями клиента и сервера, они могут взаимодействовать друг с другом.

Java упрощает сетевое программирование, путем инкапсуляции функциональности соединения сокета TCP в классы сокета, в которых класс `Socket` предназначен для создания сокета клиента, а класс `ServerSocket` для создания сокета сервера.

`Socket` является базовым классом, поддерживающим протокол TCP. Класс `Socket` обеспечивает методы для потокового ввода/вывода, облегчает выполнение операций чтения и записи в сокет и является обязательным для программ, выполняющих сетевое взаимодействие.

Для создания объектов класса `Socket` используются следующие конструкторы.

`public Socket (InetAddress IP_address, int port)`: создает объект `Socket`, который соединяется хостом, заданным в параметрами `IP_address` и `port`.

`public Socket (String hostname, int port)`: создает объект `Socket`, который соединяется с хостом, заданным параметрами имя хоста (IP-адрес) `hostname` и `port`, который сервер «слушает».

Основные методы класса `Socket` приведены в таблице 4.

Т а б л и ц а 4 — Методы класса `Socket`

Метод	Описание
`public InetAddress getInetAddress()`	Возвращает объект `InetAddress`, который содержит IP-адрес, с которым соединяется объект `Socket`
`public InputStream getInputStream()`	Возвращает входной поток для объекта `Socket`
`public InetAddress getLocalAddress()`	Возвращает объект `InetAddress`, содержащий локальный адрес, с которым соединяется объект `Socket`
`public int getPort()`	Возвращает удаленный порт, с которым соединяется объект `Socket`
`public int getLocalPort()`	Возвращает локальный порт, с которым соединяется объект `Socket`
`public OutputStream getOutputStream()`	Возвращает выходной поток объекта `Socket`
`void close()`	Закрывает объект `Socket`
`public String toString()`	Возвращает IP-адрес и номер порта сокета клиента как `String`

`ServerSocket` представляет собой класс, используемый программами сервера для прослушивания запросов клиентов. `ServerSocket` реально не выполняет сервис, но создаёт объект `Socket` от имени клиента, через который выполняется взаимодействие с сокетом клиента.

Для создания и инициализации объектов `ServerSocket` используются следующие конструкторы, определённые в классе `ServerSocket`.

`public ServerSocket(int port_number)`: создает сокет сервера на заданном порту на локальной машине. Клиентам следует использовать этот порт, чтобы общаться с сервером. Если номер порта 0, то сокет сервера создаётся на любом свободном порту локальной машины.

`public ServerSocket(int port, int backlog)`: создает сокет сервера на заданном порту на локальной машине. Второй параметр задаёт

максимальное количество соединений клиентов, которые сокет сервера поддерживает на заданном порту.

`public ServerSocket(int port, int backlog, InetAddress bindAddr)`: создает сокет сервера на заданном порту. Третий параметр используется для создания сокета сервера хоста, подключенного к нескольким физическим линиям (multi-homed host). Сокет сервера принимает запросы клиента только с заданных IP адресов.

Методы класса `ServerSocket` представлены в таблице 5.

Т а б л и ц а 5 — Методы класса `ServerSocket`

Метод	Описание
`public InetAddress getInetAddress()`	Возвращает объект `InetAddress`, который содержит адрес объекта `ServerSocket`
`public int getLocalPort()`	Возвращает номер порта, с которого объект `ServerSocket` слушает запросы клиента
`public Socket accept() throws IOException`	Заставляет сокет сервера слушать соединение клиента и принимать его. После установления соединения клиента с сервером метод возвращает сокет клиента
`public void bind( SocketAddress address) throws IOException`	Связывает объект `ServerSocket` с заданным адресом (IP адрес и порт). Этот метод вызывает исключительную ситуацию `IOException`, когда происходит ошибка
`public void close() throws IOException`	Закрывает объект `ServerSocket`. Этот метод вызывает исключительную ситуацию `IOException`, когда происходит ошибка
`public String toString()`	Возвращает IP адрес и номер порта сокета сервера как `String`

Процедура разработки сервера состоит в создании объекта класса `ServerSocket`, который «слушает» клиентские запросы на установление соединения с определённого порта. Когда сервер распознает допустимый

запрос, объект `ServerSocket` получает объект `Socket`, созданный клиентом. Взаимодействие между сервером и клиентом происходит с использованием этого сокета.

Класс `ServerSocket` пакета `java.net` используется для создания объекта, с помощью которого сервер слушает запросы удалённого входа. Класс `BufferedInputStream` управляет передачей данных от клиента к серверу, а класс `PrintStream` управляет передачей данных от сервера к клиенту.

Метод `accept()` ожидает соединения клиента, прослушивая порт, с которым он связан. Когда клиент пытается соединиться с сокетом сервера, метод принимает соединение и возвращает сокет клиента, после чего этот сокет используется клиентом для общения с сервером. Выходной поток этого сокета является входным потоком для связанного клиента и наоборот, входной поток сокета является выходным для сервера. Исключение `IOException` генерируется, если происходит какая-либо ошибка во время установления соединения.

Разрабатываемый в данной лабораторной работе сервер будет многопоточным. Таким образом, он в один момент сможет обслуживать сразу нескольких клиентов. Тема многопоточности является очень обширной, и в данной работе она не будет рассматриваться. Будет только показано, как на практике реализовать самую простую многопоточную программу.

Создатели Java предоставили две возможности создания потоков: реализация (`implements`) интерфейса `Runnable` и расширение (`extends`) класса `Thread`. Расширение класса - это путь наследования методов и переменных класса родителя. В этом случае можно наследоваться только от одного родительского класса `Thread`. Данное ограничение внутри Java можно преодолеть реализацией интерфейса `Runnable`.

Каждое Java приложение имеет хотя бы один выполняющийся поток. Поток, с которого начинается выполнение программы, называется главным. После создания процесса, как правило, JVM начинает выполнение главного потока с метода `main()`. Затем, по мере необходимости, могут быть запущены

дополнительные потоки. Многопоточность — это два и более потоков, выполняющихся одновременно в одной программе.

В классе `Thread` определены семь перегруженных конструкторов, большое количество методов, предназначенных для работы с потоками, и три константы (приоритеты выполнения потока).

```
Thread();
Thread(Runnable target);
Thread(Runnable target, String name);
Thread(String name);
Thread(ThreadGroup group, Runnable target);
Thread(ThreadGroup group, Runnable target, String name);
Thread(ThreadGroup group, String name);
```

`target` – экземпляр класса, реализующего интерфейс `Runnable`, name – имя создаваемого потока, `group` – группа, которой относится поток.

Наиболее часто используемые методы класса `Thread` для управления потоками перечислены в таблице 6.

Т а б л и ц а 6 — Методы класса `Thread`

Метод	Описание
`long getId()`	получение идентификатора потока
`String getName()`	получение имени потока
`int getPriority()`	получение приоритета потока
`State getState()`	определение состояния потока
`void interrupt(`	прерывание выполнения потока
`boolean isAlive()`	проверка, выполняется ли поток
`boolean isDaemon()`	проверка, является ли поток «демоном»
`void join()`	ожидание завершения потока
`void join(millis)`	ожидание `millis` миллисекунд завершения потока
`void notify()`	«пробуждение» отдельного потока, ожидающего «сигнала»

`void notifyAll()`	«пробуждение» всех потоков, ожидающих «сигнала»
`void run()`	запуск потока, если поток был создан с использованием интерфейса `Runnable`
`void setDaemon(bool)`	определение потока-демона
`void setPriority(int)`	определение приоритета потока
`void sleep(int)`	приостановка потока на заданное время
`void start()`	запуск потока
`void wait()`	приостановка потока, пока другой поток не вызовет метод `notify()`
`void wait(millis)`	приостановка потока на `millis` миллисекунд или пока другой поток не вызовет метод `notify()`

При выполнении программы объект `Thread` может находиться в одном из четырех основных состояний: «новый», «работоспособный», «неработоспособный» и «пассивный». При создании потока он получает состояние «новый» (NEW) и не выполняется. Для перевода потока из состояния «новый» в «работоспособный» (RUNNABLE) следует выполнить метод `start()`, вызывающий метод `run()`.

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного перечисления Thread.State :

– NEW — поток создан, но еще не запущен;

– RUNNABLE — поток выполняется;

– BLOCKED — поток блокирован;

– WAITING — поток ждет окончания работы другого потока;

– TIMED_WAITING — поток некоторое время ждет окончания другого потока;

– TERMINATED — поток завершен.

Как понятно из приведенной выше теории, для создания класса, который может запускаться в отдельном потоке, необходимо наследовать класс `Thread`, переопределить в классе метод `run()`, затем вызвать у класса, который теперь является подклассом класса `Thread`, метод `start()`, который затем вызовет выполнение метода `run()` в отдельном потоке.

Поставновка задачи: необходимо разработать клиент-серверное приложение, в котором сервер слушает запросы клиентов на порт 1500 и отправляет объект-сообщение, содержащий текущую дату/время сервера и строку сообщения. Пользователь-клиент должен иметь возможность просмотра полученного сообщения.

Класс сервера, который выполняет данную задачу с применением механизма многопоточности, приведен ниже.

Класс `ServerTCP`.

```
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Calendar;

public class ServerTCP extends Thread {
    // Объявляется ссылка на объект - сокет сервера
    ServerSocket serverSocket = null;

    public ServerTCP() {
        try {
            // Создается объект ServerSocket, который получает
            // запросы клиента на порт 1500
            serverSocket = new ServerSocket(1500);
            System.out.println("Запуск сервера");
            // Запускаем процесс, неявно вызывается метод run()
            start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Метод класс Thread, который необходимо переопределить
    @Override
    public void run() {
        try {
            while (true) {
                // Ожидание запросов на соединение от клиентов
```

```
                Socket clientSocket = serverSocket.accept();
                System.out.println("Принято соединение " +
clientSocket.getInetAddress().getHostAddress());
                // Получение выходного потока, связанного с
объектом Socket
                try (ObjectOutputStream out = new
ObjectOutputStream(clientSocket.getOutputStream())) {
                    // Создание объекта для передачи клиентам
                    DateMessage dateMessage = new
DateMessage(Calendar.getInstance().getTime(),
                            "Текущая дата/время на сервере");
                    // Запись объекта в выходной поток
                    out.writeObject(dateMessage);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                // Закрытие сокета
                serverSocket.close();
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }

    public static void main(String[] args) {
        // Запуск сервера
        new ServerTCP();
    }
}
```

Класс клиента `ClientTCP`.

```
import java.io.ObjectInputStream;
import java.net.Socket;
import java.util.Scanner;

public class ClientTCP {
    public static void main(String[] args) {
        // Создается объект Socket
        // для соединения с сервером
        try (Socket clientSocket = new Socket("localhost", 1500))
{
            // Получаем ссылку на поток, связанный с сокетом
            try (ObjectInputStream in = new
ObjectInputStream(clientSocket.getInputStream())) {
                // Извлекаем объект из входного потока
                DateMessage dateMessage = (DateMessage)
```

```
in.readObject();
                // Выводим полученные данные на консоль
                System.out.println(dateMessage.getMessage());
                System.out.println(dateMessage.getDate());
                // Реализация ожидания, чтобы программа не
выключалась сразу
                System.out.println("Нажмите Enter для выхода...");
                Scanner scanner = new Scanner(System.in);
                scanner.nextLine();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Также понадобится вспомогательный класс, который будет просто хранить данные даты/времени и сообщения от сервера. Он содержит несколько полей и сеттеры/геттеры для них.

Класс `DateMessage`.

```
import java.io.Serializable;
import java.util.Date;

public class DateMessage implements Serializable {
    // Поле для хранения даты и времени
    private Date date;
    // Поле для хранения текстового сообщения
    private String message;

    //  Конструктор класс
    public DateMessage(Date date, String message) {
        this.date = date;
        this.message = message;
    }

    // Геттер date
    public Date getDate() {
        return date;
    }

    // Сеттер date
    public void setDate(Date date) {
        this.date = date;
    }

    // Геттер message
    public String getMessage() {
        return message;
```

```
    }

    // Сеттер message
    public void setMessage(String message) {
        this.message = message;
    }
}
```

Данная программа запускается аналогично программе, которая демонстрировала клиент-сервер на UDP-сокетах. Только теперь сервер поддерживает много подключений одновременно. Для того, чтобы запускать несколько экземпляров одной программы, необходимо выполнить некоторые настройки.

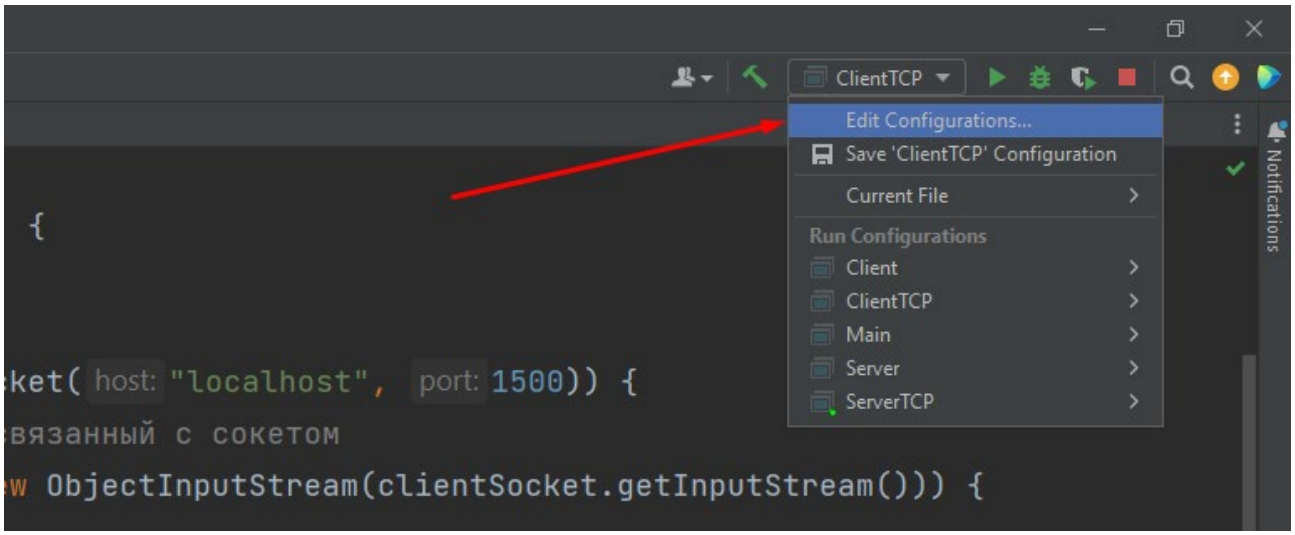


Рисунок 10 – Необходимая вкладка

Для перехода в необходимые настройки, необходимо нажать кнопку «Edit Configurations…», как на рисунке 10. В открывшемся окне необходимо выбрать класс клиента и нажать «Modify options», выбрать опцию «Allow multiple instances» (рисунок 11). Сохраните настройки.

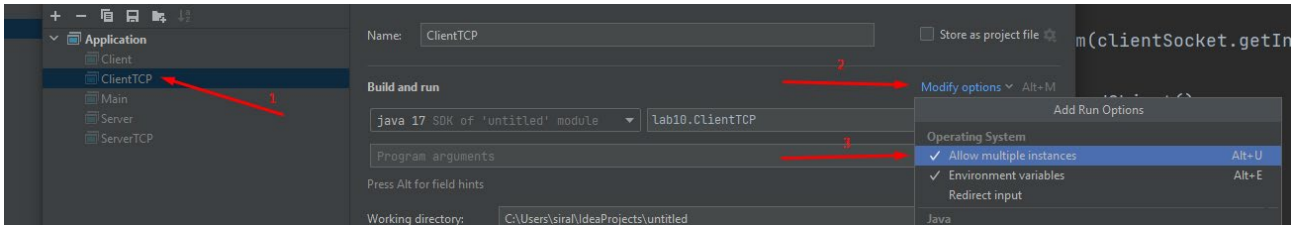


Рисунок 11 – Разрешение на одновременный запуск нескольких экземпляров

Теперь можно запускать множество клиентов. Демонстрация работы данной программы должна быть приведена в отчете по лабораторной работе.

5 Упражнения по теме TCP сокетов в Java:

Разработать клиент-серверное приложение с использованием протокола TCP/IP, в соответствии с рассмотренным примером. Изучить все основные классы для разработки данного вида сетевых приложений, алгоритмы работы клиента и сервера. В отчете привести результат работы программы.

Лабораторная работа № 11. Разработка распределённых приложений

1 Цель работы

Изучить классы и методы, которые позволяют разрабатывать приложения использующие распределённую архитектуру. Изучить технологию разработки распределённых приложений.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Данная работа посвящена разработке распределенных приложений.

В лабораторной работе будут рассмотрены следующие вопросы:

— Спецификация Java RMI.

— Архитектура Java RMI.

— Как работает приложение, использующее RMI.

— Пакет `java.rmi`.

— Знакомство с СУБД SQLite.

— Импорт библиотеки в проект в среде разработки Intellij IDEA.

— Реализация простого приложения с использованием RMI для регистрации участников конференции.

В конце предложены упражнения для закрепления материала.

В настоящее время существуют три подхода к разработке приложений: традиционный подход, клиент/серверный подход и компонентный подход.

При использовании традиционного подхода единственное приложение управляет логикой представления, логикой обработки и взаимодействием с базой данных (на локальном компьютере). Монолитное приложение при незначительной модификации, расширении и развитии приложения, приложение должно перекомпилироваться и собираться вновь. В Клиент/Серверной архитектуре (также называемая двухслойная (two-tier) архитектура) данные от клиентской части, хранятся централизованно на сервере и к ним предоставляется доступ по технологии клиент/сервер. При этом логика обработки объединяется со слоем представления либо на стороне клиента, либо на стороне сервера, которая содержит код для связи с базой данных. Если логика обработки объединяется со слоем представления, то клиент называется «толстым», а если логика обработки объединяется с сервером базы данных, то сервер называется «толстым».

Однако и клиент/серверная архитектура также имеет недостатки.

Любое изменение бизнес-логики требует изменений в алгоритмах обработки. При изменении алгоритмов обработки либо слой представления, либо код доступа к базе данных нуждается в изменении, в зависимости от места расположения бизнес-логики.

Реализованные приложения, использующие двухслойную архитектуру, могут трудно масштабироваться из-за ограниченного количества доступных для клиента соединителей с базой данных. Запросы соединения, превышающие определенное количество, просто отбрасываются сервером.

В трехслойной архитектуре, логика представления располагается на стороне клиента, доступ к базе данных контролируется на стороне сервера, а логика обработки находится между этими двумя слоями. Слой логики обработки относится к серверу приложений (также называемый средним слоем трехслойной компонентной архитектуры). Этот тип архитектуры называется серверо-центрическим, поскольку он даёт возможность компонентам

приложения выполняться на среднем слое сервера приложений, применяющего правила обработки независимо от интерфейса представления и реализации базы данных.

Приложения, в которых логика представления, логика обработки и база данных размещаются на нескольких компьютерах, называется распределенными (distributed) приложениями.

Вызов удаленных методов является реализацией идей (Remote Procedure Call - RPC) для языка программирования Java. Расширении механизма передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных через сеть. То есть, клиентское приложение обращается к процедурам, хранящимся на сервере. Средства удаленного вызова процедур предназначены для облегчения организации распределенных вычислений. Наибольшая эффективность использования RPC достигается в тех приложениях, в которых существует интерактивная связь между удаленными компонентами с небольшим временем ответов и относительно малым количеством передаваемых данных. Такие приложения называются RPC-ориентированными.

RMI представляет собой спецификацию, которая дает возможность одной виртуальной Java-машине - Java Virtual Machine (JVM), вызывать методы в объектах, расположенных в другой JVM. Эти две JVM могут быть запущены на различных компьютерах или выполняться в отдельных процессах одного компьютера. RMI применяется на среднем слое трехслойной архитектуры, облегчая возможность программистам вызывать распределенные компоненты в сетевой среде. Для использования RMI, программисту нет необходимости знать программирование сокета или организовывать многопоточное выполнение, что позволяет ему концентрироваться на реализации логики обработки.

Одной из центральных достоинств RMI является возможность загрузки определения класса объекта в ситуации, когда загружаемый класс не представлен в виртуальной машине клиента. Вся функциональность объекта, доступная только на одной JVM, может быть передана в другую JVM,

расположенную удалённо. RMI доставляет объекты в соответствии с шаблоном, определенным классом, следовательно, не меняется их поведение при пересылке. Эта особенность позволяет динамически определять и расширять функциональность клиентского приложения путем использования поведенческих возможностей серверного приложения.

Архитектура RMI состоит из трех уровней (рисунок 12):

– уровень стаб/скелетон (Stub/Skeleton);
– уровень удаленной ссылки (Remote Reference Layer);
– транспортный уровень (Transport Layer).

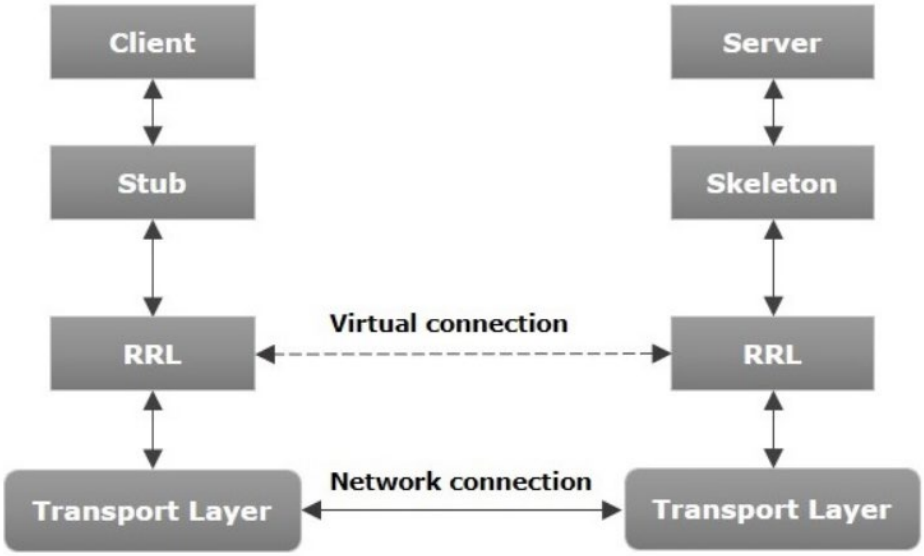


Рисунок 12 – Архитектура RMI

Уровень стаб/скелетон прослушивает вызовы удаленных методов от клиента, направляет их на удаленные сервисы на сервере. Стаб представляет собой удаленный объект, связанный с объектом на сервере через скелетон. Скелетон читает параметры для вызова метода, выполняет вызов метода объекта на сервере, получает возвращаемое значение и записывает его на стаб.

Уровень удаленной ссылки занимается преобразованием потоков данных между стабом и скелетоном. Клиент отправляет параметры для удаленного метода на RRL, и он преобразует их формат, в котором возможна их отправка по сети. Этот процесс называется упаковкой (marshalling). На стороне сервера

RRL преобразует данные обратно и передает их скелетону. На данном уровне клиент и сервер связаны виртуально.

На транспортном уровне непосредственно происходит передача данных между клиентом и сервером путем создания сокетных соединений. На этом уровне клиент и сервер связаны физически.

Интересным механизмом является распределенный сборщик мусора (Distributed Garbage Collector). Java является языком с автоматическим управлением памятью, реализованным с помощью сборщика мусора (Garbage Collector). По сути, он выгружает из памяти объекты, на которые нет ни одной активной ссылки. Распределенный сборщик мусора является реализацией сборщика мусора, который действует в рамках не одной JVM, а в рамках всех JVM, участвующих в сетевом взаимодействии.

Процесс работы приложения с применением RMI может быть описан последовательностью следующих этапов:

– клиент вызывает метод удаленного объекта, которые определен на стабе (уровень клиента);

– переданные параметры для метода упаковываются и отправляются на скелетон (уровень сети);

– данные поступают на скелетон, распаковываются и передаются в метод на серверном объекте, который был запрошен клиентом (промежуточный уровень);

– метод выполняется и ответ возвращается стабу (уровень сети);

– клиент получает данные от стаба (уровень клиента).

Пакет `java.rmi` обеспечивает удаленный интерфейс, класс для доступа к удаленным именам, зарегистрированным на сервере и менеджер безопасности для RMI.

Пакет `java.rmi.registry` обеспечивает классы и интерфейсы, которые используются удаленным регистром.

Пакет `java.rmi.server` обеспечивает классы и интерфейсы, которые используются для реализации удаленных объектов, стабы и скелетоны, а также поддержку RMI связи.

Пакет `java.rmi.dgc` обеспечивает классы и интерфейсы, которые используются распределенным сборщиком мусора RMI.

Пакет `java.rmi` объявляет интерфейс `Remote` и классы `Naming`. Он также содержит ряд классов исключительных ситуаций, которые используются с RMI.

Все удаленные объекты должны реализовать интерфейс `Remote`. Этот интерфейс не содержит методов и используется для идентификации удаленных объектов. Пакет `java.rmi` состоит из следующих классов:

Класс `Naming`: Содержит статические методы для доступа удаленных объектов через URL. Этот класс поддерживает следующие методы:

`rebind()`: Связывает имя удаленного объекта с заданным URL и обычно используются объектом сервера.

`unbind()`: Удаляет связь между именем объекта и URL.

`lookup()`: Возвращает удаленный объект, заданный URL и обычно используется объектом клиента.

`list()`: Возвращает список URL, которые известны регистру RMI.

Пакет `java.rmi` определяет набор исключительных ситуаций. Класс `RemoteException` является родительским для всех исключительных ситуаций, которые генерируются во время использования RMI.

Удаленным объектам, которые доступны локально, нет необходимости вызывать исключительную ситуацию `RemoteException`.

Пакет `java.rmi.registry` содержит интерфейсы `Registry` и `RegistryHandler` и используется регистром. Эти интерфейсы используются для регистрации и доступа к удаленным объектам по имени. Объекты `Remote` регистрируются, когда они связываются с процессом регистрации хоста.

Пакет `java.rmi.server` реализует интерфейсы и классы, которые поддерживают клиентскую и серверную части RMI. Пакет `java.rmi.server` состоит из следующих классов и интерфейсов:

Класс `RemoteObject`: реализует интерфейс `Remote` и обеспечивает удаленную реализацию класса `Object`. Все объекты, которые реализуют удаленные объекты, расширяют класс `RemoteObject`.

Класс `RemoteServer`: расширяет класс `RemoteObject` и является общим классом, который является подклассом конкретных типов реализации удаленного объекта.

Класс `UnicastRemoteObject`: расширяет класс `RemoteServer` и обеспечивает реализацию `RemoteObject` по умолчанию. Классы, которые реализуют `RemoteObject` обычно подклассы объекта `UnicastRemoteObject`. Путем расширения `UnicastRemoteObject`, подкласс может использовать по умолчанию RMI для общения на основе транспортного сокета.

Класс `RemoteStub`: обеспечивает абстрактную реализацию стаба на стороне клиента. Статический метод `setRef()` используется, чтобы связать стаб на стороне клиента с соответствующими удаленными объектами.

Интерфейс `RemoteCall`: обеспечивает методы, которые используются всеми стабами и скелетонами в RMI.

Интерфейс `Skeleton`: обеспечивает метод для получения доступа к методам удаленного объекта и этот интерфейс реализуется удаленными скелетонами.

Интерфейс `Unreferenced`: дает возможность определить, когда клиент больше не ссылается на удаленный объект.

Распределенное приложение, использующее RMI, содержит несколько компонентов, в том числе, файл интерфейса, файл сервера и файл клиента. В соответствии со спецификацией RMI, необходимо выполнять определенную последовательность шагов для создания распределенного приложения RMI, а именно:

— создать удаленный интерфейс;

— реализовать удаленный интерфейс;

— создать сервер RMI;

— создать клиент RMI;

— выполнить приложение RMI.

Необходимо разработать клиент-серверное приложение для удаленной регистрации участников научной конференции. Сервер организаторов конференции должен содержать базу данных (БД) и RMI-сервис для приема и записи регистрационных сведений в БД. Участникам конференции необходимо предоставить приложение с графическим интерфейсом для ввода и отправки данных на сервер с помощью вызова удаленного метода RMI.

Для решения поставленной задачи необходимо выполнить следующие шаги:

1. Создать новый проект.
2. Создать сериализуемый Java-класс `RegistrationInfo` для представления и передачи данных регистрации.
3. Создать класс, реализующий логику работы с БД – добавление нового участника и получение числа зарегистрированных участников.
4. Реализовать интерфейс `ConfServer` в классе `ConfServerImpl`.
5. Создать клиентское приложение – разработать графический интерфейс (с использованием библиотеки Swing) и обеспечить вызов удаленного метода сервера.
6. Выполнить приложение.

Класс RegistrationInfo будет представлять сущность участника конференции. Подобный класс, который только хранит данные, уже был создан в лабораторной работе 10, он назывался DateMessage.

Ниже представлен код класса RegistrationInfo.

```
import java.io.Serializable;

public class RegistrationInfo implements Serializable {
    private String firstName;
    private String lastName;
    private String organization;
    private String reportTheme;
    private String email;

    public RegistrationInfo(String firstName, String lastName,
                            String organization, String
```

```
reportTheme, String email) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
        this.organization = organization;
        this.reportTheme = reportTheme;
        this.email = email;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getOrganization() {
        return organization;
    }

    public void setOrganization(String organization) {
        this.organization = organization;
    }

    public String getReportTheme() {
        return reportTheme;
    }

    public void setReportTheme(String reportTheme) {
        this.reportTheme = reportTheme;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

Объекты данного класса будут передаваться по сети, поэтому необходимо унаследоваться от класса `Serializable`. Данный класс позволяет преобразовывать объекты в поток байт. Объекты по сети могут передаваться только в виде потока байт.

Класс содержит несколько полей и геттеры/сеттеры для них.

Теперь необходимо создать класс, который будет работать с БД. В качестве БД будет использоваться SQLite. SQLite – компактная встраиваемая СУБД. Слово «встраиваемый» (англ. embedded) означает, что SQLite не использует парадигмы клиент-сервер, то есть движок SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а представляет собой библиотеку, с которой программа компонуется, и движок становится составной частью программы. Таким образом, в качестве протокола обмена используются вызовы функций (API) библиотеки SQLite. Такой подход уменьшает накладные расходы, время отклика и упрощает программу. SQLite хранит всю базу данных (включая определения, таблицы, индексы и данные) в единственном стандартном файле на том компьютере, на котором исполняется программа. Простота реализации достигается за счёт того, что перед началом исполнения транзакции записи весь файл, хранящий базу данных, блокируется; ACID-функции достигаются в том числе за счёт создания файла журнала.

Для использования SQLite в Java, как и любой другой БД, требуется библиотека, которая позволяет работать с ней как с объектом Java. Процесс импортирования/подключения библиотеки будет описан ниже. Подключение к базе данных Java DataBase Connectivity (JDBC) – это интерфейс прикладного программирования (API) для языка программирования Java, который определяет, как клиент может получить доступ к базе данных. Это технология доступа к данным на основе Java, используемая для подключения к базе данных Java.

Для начала необходимо скачать библиотеку, она официально находится на github. Необходимо зайти в репозиторий «xerial/sqlite-jdbc» и перейти на страницу релизов «Releases» – [страница релизов]. На этой странице выберите

последнюю версию библиотеки и перейдите в область «Assets». Далее нужно скачать jar-файл.

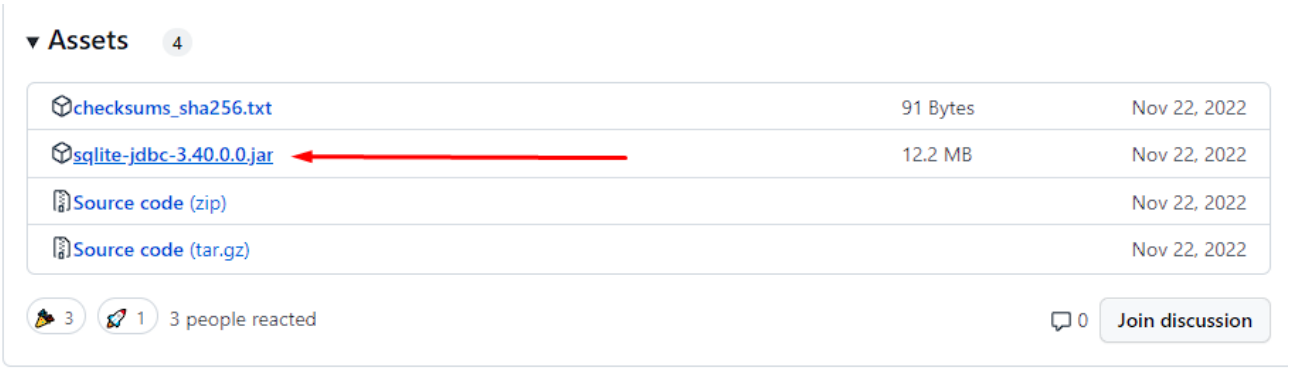


Рисунок 13 – Нужный файл библиотеки

Теперь нужно добавить библиотеку в проект. В открытом проекте, в который нужно добавить библиотеку, нажимаем File и выбираем Project Structure... или же используем горячие клавиши: Ctrl + Alt + Shift + S.

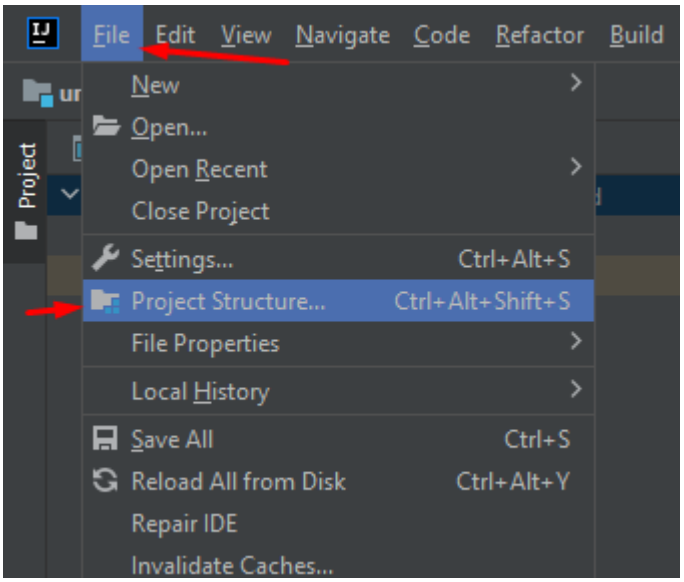


Рисунок 14 – Открываем настройки файловой структуры проекта

В открывшемся окне Project Settings > Modules > Dependencies > знак ‘+’ > JARs or directories… (рисунок 15).

После этого нужно выбрать файл на диске и нажать «Ок» (рисунок 16).

Применяем изменения.

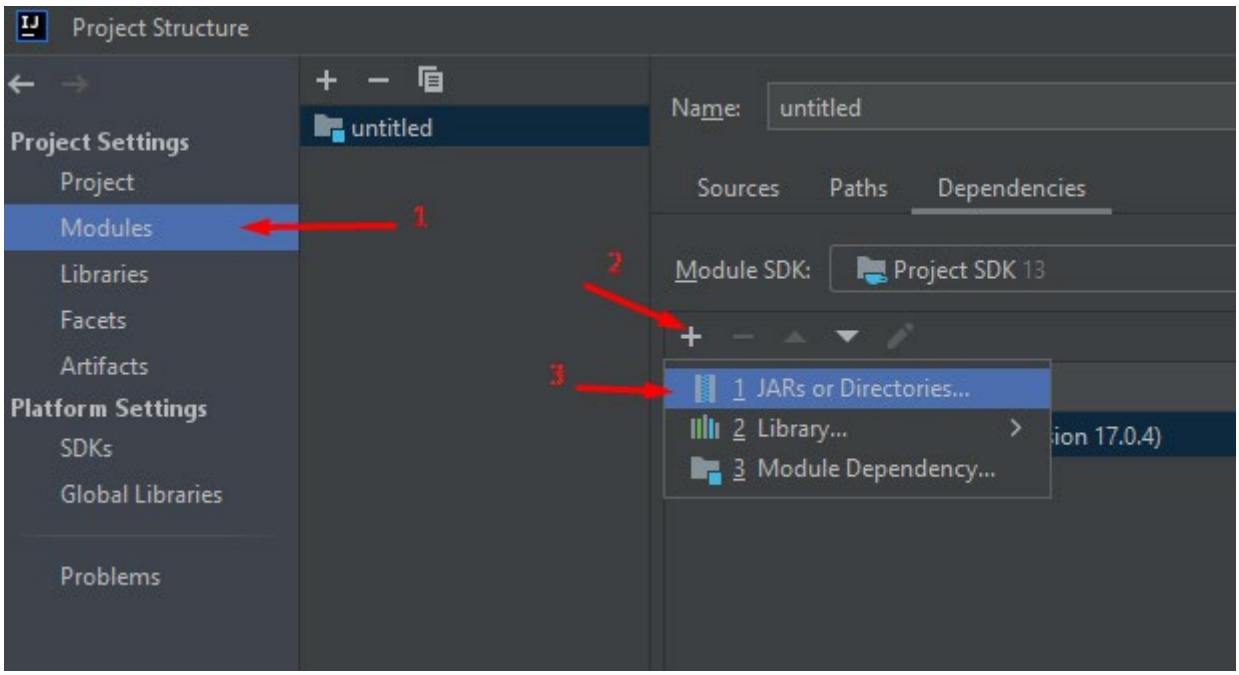


Рисунок 15 – Порядок добавления библиотеки

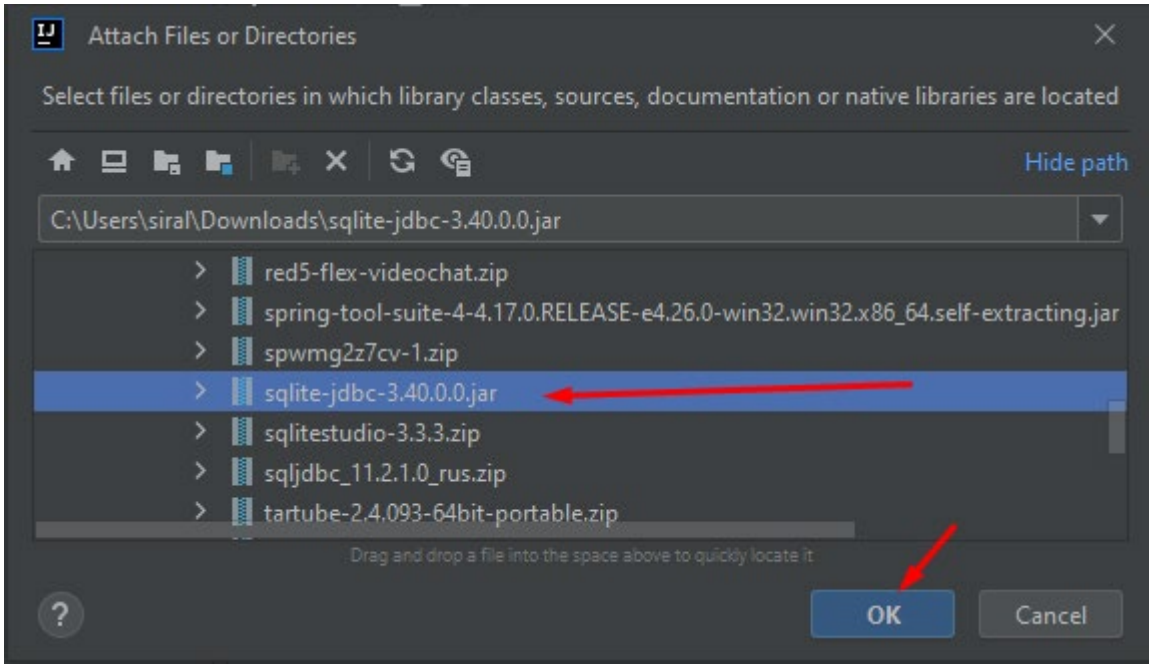


Рисунок 16 – Выбор файла библиотеки

Библиотека импортирована. Теперь можно перейти к написанию класса `DbHandler`, который будет работать с БД.

Класс `DbHandler`.

```
import java.sql.*;
```

```
public class DbHandler {
    private static final String CONNECTION_STRING =
            String.format("jdbc:sqlite:%s\\src\\lab11\\bd.db",
System.getProperty("user.dir"));

    private static DbHandler instance = null;

    private DbHandler() {
    }

    public static synchronized DbHandler getInstance() throws
SQLException {
        if (instance == null) {
            instance = new DbHandler();
            instance.createTable();
        }
        return instance;
    }

    public void createTable() throws SQLException {
        try (Connection connection =
DriverManager.getConnection(CONNECTION_STRING)) {
            String sql = "create table if not exists
registration_info(first_name varchar(20), " +
                    "last_name varchar(20), " +
                    "organization varchar(100), " +
                    "report_theme varchar(300), " +
                    "email varchar(20));";
            connection.createStatement().execute(sql);
        }
    }

    public void addParticipant(RegistrationInfo participant)
throws SQLException {
        try (Connection connection =
DriverManager.getConnection(CONNECTION_STRING)) {
            String statement = "insert into registration_info "
                    + "(first_name, last_name, organization, "
                    + "report_theme, email) "
                    + "values (?, ?, ?, ?, ?)";
            try (PreparedStatement st =
connection.prepareStatement(statement)) {
                st.setString(1, participant.getFirstName());
                st.setString(2, participant.getLastName());
                st.setString(3, participant.getOrganization());
                st.setString(4, participant.getReportTheme());
                st.setString(5, participant.getEmail());
                st.executeUpdate();
            }
        }
        System.out.println("Добавлен новый участник");
    }
```

```
    public int getParticipantCount() throws SQLException {
        try (Connection connection =
DriverManager.getConnection(CONNECTION_STRING)) {
            try (Statement statement =
connection.createStatement()) {
                int count = 0;
                ResultSet rs = statement.executeQuery("Select
count(*) from registration_info");
                if (rs.next()) {
                    count = rs.getInt(1);
                }
                return count;
            }
        }
    }
}
```

Данный класс также реализует шаблон Одиночка (Singleton). Для этого в Java необходимо конструктор класса объявить с модификатором доступа `private`. Далее объявляется метод `getInstance()`, который будет возвращать экземпляр данного класса. Если экземпляр ранее не был создан, он создается и записывается в поле класса, затем возвращается методом. Если экземпляр уже существует (поле `instance` не `null`), то метод просто возвращает экземпляр `DbHandler`.

```
// реализация Singleton
...
private static DbHandler instance = null;

private DbHandler() {
}

public static synchronized DbHandler getInstance() throws
SQLException {
    if (instance == null)
        instance = new DbHandler();
    return instance;
}
...
```

Опишем процесс подключения к БД. В классе определена константа `CONNECTION_STRING`, которая содержит в себе строку подключения к БД, по сути, адрес, по которому находится БД.

```
private static final String CONNECTION_STRING =
        String.format("jdbc:sqlite:%s\\src\\lab11\\bd.db",
System.getProperty("user.dir"));
```

Данная строка строится следующим образом: получаем текущую пользовательскую директорию с помощью `System.getProperty("user.dir")`. Затем эта строка подставляется вместо так называемого спецификатора формата «%s», который как раз и ожидает строку. Подстановка выполняется статическим методом `format()` из класса `String`. В итоге строка подключения будет содержать в себе абсолютный путь к базе данных. Заметьте, что предполагается, что файлы лабораторной работы находятся в пакете «lab11». Укажите тот путь, который нужен вам. Можно также использовать и относительный путь.

Само подключение к БД выполняется следующим образом:

```
Connection connection = DriverManager.getConnection(
                                                CONNECTION_STRING)
```

Если файла БД не существует, он будет создана при первом подключении. Таблица для хранения данных участников будет создана методом `createTable()`, если ее не существует в БД.

Методы класса действуют по одному принципу – создается подключение к БД, создается строка, содержащая SQL выражение, выражение передается соединению, вызывается выполнение SQL инструкций, далее происходит обработка ответа базы данных.

Теперь можно перейти непосредственно к реализации классов, которые будут использовать RMI. Для начала необходимо определить удаленный интерфейс сервера. Для этого необходимо создать интерфейс и выполнить наследование от интерфейса `Remote`. Далее необходимо объявить методы, которые будут предоставляться клиентам.

Интерфейс `ConfServer`.

```
import java.rmi.*;

public interface ConfServer extends Remote {
    int registerConfParticipant(RegistrationInfo registrationInfo)
```

```
            throws RemoteException;
}
```

Как видно из кода интерфейса, сервер будет предоставлять удаленный метод, который позволит клиентам регистрироваться как участникам конференции. Класс сервера, реализующий данный интерфейс приведен ниже.

```
import java.net.InetAddress;
import java.rmi.*;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

public class ConfServerImpl extends UnicastRemoteObject implements
ConfServer {
    // Определяется конструктор по умолчанию
    public ConfServerImpl() throws RemoteException {
        super();
    }

    // Определение удаленного метода
    @Override
    public int registerConfParticipant(RegistrationInfo
participant)
            throws RemoteException {
        try {
            DbHandler dbh = DbHandler.getInstance();
            dbh.addParticipant(participant);
            return dbh.getParticipantCount();
        } catch (Exception e) {
            e.printStackTrace();
            throw new RemoteException(e.getMessage(), e);
        }
    }

    // Метод main()
    public static void main(String[] args) {
        try {
            // Установка параметра адреса сервера (чтобы избежать
               ошибки создания сервера на локальной машине)
            System.setProperty("java.rmi.server.hostname",
                    InetAddress.getLocalHost().getHostAddress());
            // Создание RMI регистра
            LocateRegistry.createRegistry(1099);
            // Создание экземпляра класса ConfServerImpl
            ConfServerImpl instance = new ConfServerImpl();
            // Регистрация объекта RMI под именем ConfServer
            Naming.rebind("ConfServer", instance);
            System.out.println("Сервис зарегистрирован");
        } catch (Exception e) {
            e.printStackTrace();
```

```
        }
    }
}
```

Класс `ConfServerImpl`, помимо реализации интерфейса `ConfServer`, также должен наследоваться от класса `UnicastRemoteObject`. Класс содержит метод `main()`, в котором сначала создается регистр RMI, а затем экземпляр класса сервера регистрируется как удаленный сервер, доступный по имени «ConfServer».

Осталось создать клиентскую часть программы.

```
import javax.naming.NamingException;
import javax.swing.*;
import java.rmi.*;
import java.awt.event.*;
import java.awt.*;

public class ConfClient {
    private static JFrame frame;
    private final JTextField txtLastName;
    private final JTextField txtFirstName;
    private final JTextField txtOrganization;
    private final JTextField txtReportTheme;
    private final JTextField txtEmail;

    // Определяется конструктор по умолчанию
    public ConfClient() {
        // Создается JFrame
        frame = new JFrame("Регистрация участника конференции");
        JPanel panel = new JPanel();
        // Набор менеджеров разметки
        panel.setLayout(new GridLayout(5, 2));
        frame.setBounds(100, 100, 400, 200);
        frame.getContentPane().setLayout(new BorderLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Создание текстовых компонентов
        JLabel lbLastName = new JLabel("Фамилия");
        JLabel lbFirstName = new JLabel("Имя");
        JLabel lbReportTheme = new JLabel("Тема доклада");
        JLabel lbOrganization = new JLabel("Организация");
        JLabel lbEmail = new JLabel("Емайл");
        // Создание полей ввода
        txtLastName = new JTextField(15);
        txtFirstName = new JTextField(15);
        txtOrganization = new JTextField(70);
        txtReportTheme = new JTextField(100);
```

```
        txtEmail = new JTextField(15);
        // Создание кнопки отправки
        JButton submit = new JButton("Отправить");
        // Добавление в панель компонентов swing
        panel.add(lbLastName);
        panel.add(txtLastName);
        panel.add(lbFirstName);
        panel.add(txtFirstName);
        panel.add(lbOrganization);
        panel.add(txtOrganization);
        panel.add(lbReportTheme);
        panel.add(txtReportTheme);
        panel.add(lbEmail);
        panel.add(txtEmail);
        frame.getContentPane().add(panel, BorderLayout.CENTER);
        frame.getContentPane().add(submit, BorderLayout.SOUTH);
        frame.setVisible(true);
        // Добавляем слушателя события нажатия кнопки
        submit.addActionListener(new ButtonListener());
    }

    // Создание класса ButtonListener
    class ButtonListener implements ActionListener {
        // Определение метода actionPerformed()
        public void actionPerformed(ActionEvent evt) {
            try {
                // Получение удаленного объекта
                // Если сервер размещен на удаленном компьютере,
                // то вместо localhost указывается имя
                // хоста сервера
                ConfServer server = (ConfServer)
Naming.lookup("rmi://localhost/ConfServer");
                // Формирование сведений о регистрации для
                // отправки на сервер
                RegistrationInfo registrationInfo = new
RegistrationInfo(
                        txtFirstName.getText(),
txtLastName.getText(),
                        txtOrganization.getText(),
txtReportTheme.getText(),
                        txtEmail.getText());
                // Вызов удаленного метода
                int count =
server.registerConfParticipant(registrationInfo);
                JOptionPane.showMessageDialog(
                        frame,
                        "Регистрация выполнена успешно"
                                + "\nКоличество зарегистрированных
участников - "
                                + count + "\nСпасибо за участие");
            } catch (Exception e) {
                JOptionPane.showMessageDialog(frame, "Ошибка");
```

```
                e.printStackTrace();
            }
        }
    }

    // Определение метода main()
    public static void main(String[] args) throws NamingException
{
        new ConfClient();
    }
}
```

Конструктор класса инициализирует пользовательский графический интерфейс. Создаются компоненты окна, затем они добавляются на общий фрейм (типа `JFrame`). Для обработки события нажатия кнопки, необходимо создать «слушателя» `ActionListener`, который реализует выполнение логики после нажатия на кнопку. В данном случае создается внутренний (вложенный) класс `ButtonListener`, который наследуется от класса `ActionListener`. В данном классе реализован один метод `actionPerformed()`, в котором находится основная логика взаимодействия с удаленным сервером. Когда клиент заполнил текстовые поля и нажал на кнопку «Отправить», вызывается метод `actionPerformed()`. Сначала выполняется подключение к серверу. Затем создается объект `RegistarionInfo`, а потом он отправляется серверу в качестве аргумента удаленной функции `registerConfParticipant()`. Сервер выполняет логику по регистрации нового участника и возвращает клиенту в качестве ответа число зарегистрированных участников.

Таким образом создается клиент-серверное приложения с использованием Java RMI.

Нужно понимать, что на клиентской машине и на серверной должны быть разные файлы/классы. В клиентском подкаталоге должны быть следующие файлы:

— ConfClient.java;

— ConfServer.java;

— RegistrationInfo.java.

В то же время, в подкаталоге сервера должны быть следующие файлы:

— ConfServer.java;

— ConfServerImpl.java;

— DbHandler.java;

— RegistrationInfo.java;

— bd.db – файл БД.

В рамках локального запуска это не так важно, поэтому можно хранить все файлы проекта в одном каталоге.

Запуск нескольких «исполняемых» файлов уже был ранее рассмотрен в лабораторных работах. Поэтому ниже просто будет продемонстрирована работа программы. Сначала запускается сервер, а уже потом клиент.

Если в консоль сервера было выведено «Сервис зарегистрирован», то сервер успешно запущен, БД доступна.

В форму клиентской части приложения вписываются данные, нажимается кнопка «Отправить». В случае успешной регистрации появляется диалоговое окно с сообщением об успешной регистрации и текущем количестве участников. В консоль сервера в этот момент выводится сообщение «Участник зарегистрирован». Работа программы проиллюстрирована на рисунках 17-19.

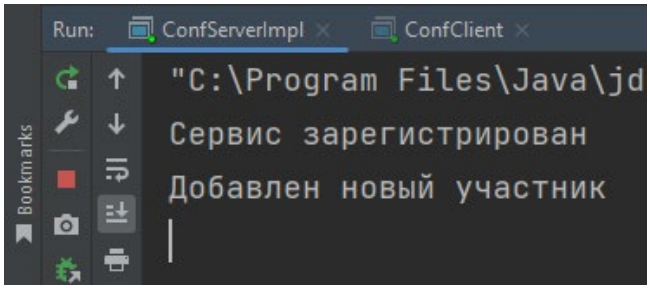


Рисунок 17 – Консоль сервера

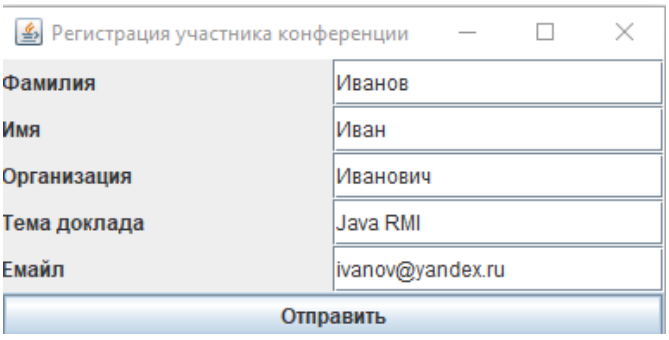


Рисунок 18 – Графическое окно пользователя

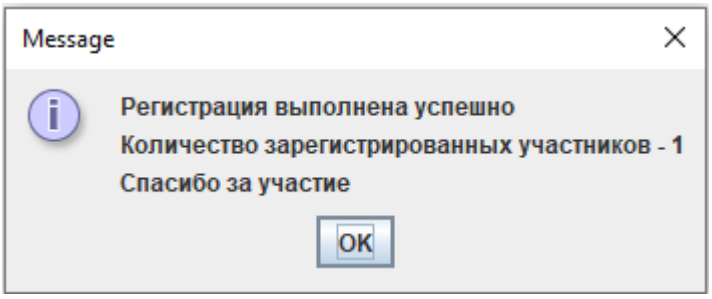


Рисунок 19 – Сообщение об успешной регистрации

Просмотреть данные добавленного пользователя, можно, например, с помощью программы SQLiteStudio, которую не нужно устанавливать.

Подключение БД в данной программе показано на рисунке 20.

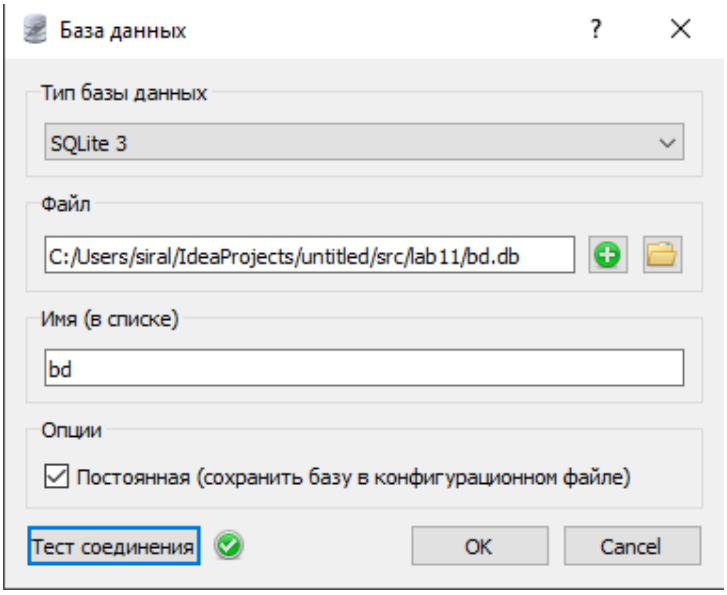


Рисунок 20 – Добавление БД в SQLiteStudio

Теперь можно просматривать данные, которые были добавлены программой.

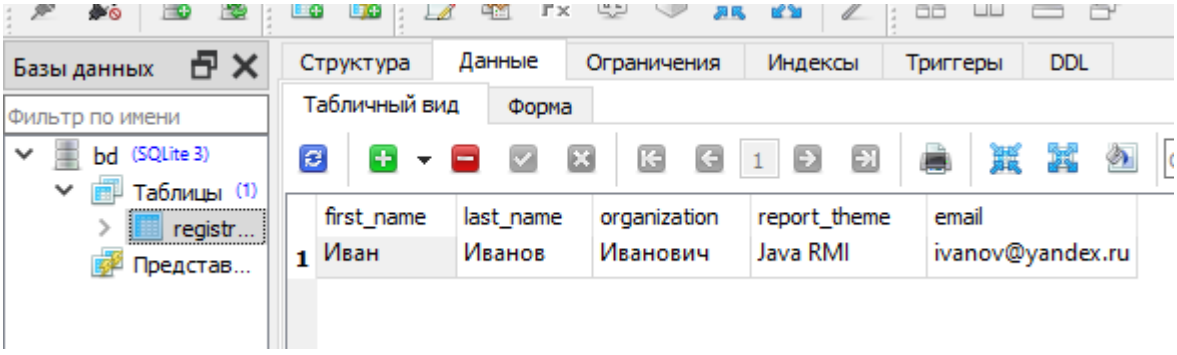


Рисунок 21 – Просмотр данных в БД

5 Упражнения по теме Java RMI:

Разработать распределённое приложения в соответствии с рассмотренным примером. Изучить все основные классы для разработки данного вида сетевых приложений, алгоритмы работы клиента и сервера.

Лабораторная работа № 12. Разработка сетевых программ с использованием технологии Java Servlet

1 Цель работы

Изучить классы и методы, которые позволяют разрабатывать приложения, использующие технологию Java Servlet и динамические веб-страницы.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Данная работа посвящена разработке веб-приложений с использованием сервлетов.

В лабораторной работе будут рассмотрены следующие вопросы:

— Спецификация Java Servlet API.

— Что такое сервлет.

— Основные классы, реализующие сервлеты.

— Установка Apache NetBeans.

— Создание проекта веб-приложения в NetBeans.

— Создание сервера GlassFish Server.

— Подключение к базе данных Derby DataBase.

— Создание таблицы в БД и заполнение ее данными с помощью средств NetBeans.

— Создание сервлета.

— Запуск веб-приложения.

В конце предложены упражнения для закрепления материала.

Очень часто веб-сайт принимает введенные данные и проверяет их, используя серверные программы. Эти серверные программы могут быть написаны с использованием различных серверных технологий, таких как Common Gateway Interface (CGI – Общий шлюзовой интерфейс), Active Server Pages (ASP – Активные серверные страницы) и сервлеты.

CGI-скрипты разрабатывются на языках программирования C, C++ или Perl. Если серверное приложение использует CGI-скрипты для обработки клиентских запросов, то сервер создает отдельную виртуальную машину для выполнения каждого экземпляра CGI-скрипта, в результате чего производительность сервера уменьшается из-за большого количества одновременных запросов.

ASP – это серверная технология, которая позволяет разработчику совмещать HTML и скриптовый язык на одной странице. Для объектов ASP поддерживается многопоточность, и, следовательно, они могут одновременно обслуживать несколько запросов. Ограничение ASP состоит в том, что эта технология не совместима со многими коммерческими веб-серверами.

Сервлеты – это серверные программы, написанные на Java. В отличие от CGI-скриптов, код инициализации сервлета выполняется только один раз. Кроме того, обработка каждого клиентского запроса выполняется в отдельном потоке на сервере, что предотвращает создание лишних процессов, увеличивая, таким образом, производительность сервера. При этом сервлеты наследуют все свойства языка программирования Java. Например, подобно всем стандартным классам Java, сервлет не зависит от платформы и может использоваться в различных операционных системах. Также сервлет может пользоваться

богатыми возможностями Java, такими как работа в сети, многопоточность, JDBC, локализация, RMI и многими другими.

Сервлеты позволяют расширять функциональность серверного приложения для формирования динамического содержимого в веб-приложении. Кроме наследуемых достоинств языка Java, сервлеты обладают перечисленными ниже характеристиками.

Безопасность. Веб-контейнер обеспечивает среду выполнения для сервлета. Сервлеты наследуют параметры безопасности, предоставляемые веб-контейнером, что позволяет разработчикам сосредоточиться на функциональности сервлета и оставить обеспечение проблем безопасности веб-контейнеру.

Управление сессией. Это механизм отслеживания состояния пользователя при выполнении нескольких запросов. Сессия сохраняет идентичность и состояние клиента при выполнении нескольких запросов.

Сохранение данных экземпляра объекта. Сервлеты позволяют увеличить производительность сервера, сокращая количество обращений к диску. Например, если клиент заходит на сайт банка для проверки баланса или получения ссуды, номер его счета должен проверяться в базе данных при выполнении каждого действия. Вместо многократной проверки номера счета в базе данных, сервлеты сохраняют номер счета в памяти объекта до тех пор, пока пользователь находится на веб-сайте.

Чтобы получить доступ к сервлету, сначала необходимо откомпилировать сервлет, а затем развернуть файл класса сервлета на Java-совместимом сервере приложений, таком, например, GlassFish Application Server. Сервер приложений JavaEE предоставляет веб-контейнер для управления различными компонентами веб-приложения, в том числе, HTML-страницами и сервлетами. Веб-контейнер обеспечивает среду исполнения для сервлета.

Веб-контейнер обеспечивает управление различными стадиями жизненного цикла сервлета, в том числе, инициализация экземпляра сервлета, обработка клиентского запроса и удаление экземпляра сервлета. Также он

определяет ограничения безопасности, которые разрешают доступ к развернутым сервлетам только авторизованным пользователям. Он управленяет транзакциями во время записи и чтения данных сервлетом из базы данных. Веб-контейнер отвечает за создание и удаление экземпляров сервлетов из пула экземпляров в процессе обслуживания многократных запросов.

Когда клиент отправляет запрос конкретного сервлета серверу приложений JavaEE, последний передает запрос веб-контейнеру, который проверяет, существует ли экземпляр требуемого сервлета и, если экземпляр сервлета существует, то веб-контейнер передает запрос сервлету, который обрабатывает запрос клиента и отправляет ответ обратно клиенту. Взаимодействие между сервером приложений и веб-контейнером при обработке клиентского запроса показано на рис. 22.

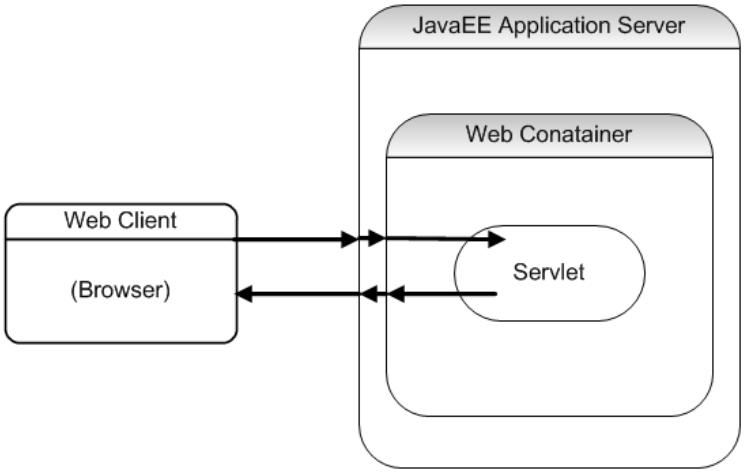


Рисунок 22 – Взаимодействие между сервером приложений и веб-контейнером при обработке клиентского запроса

Если экземпляр сервлета не существует, то веб-контейнер находит и загружает класс сервлета. Затем веб-контейнер создает экземпляр сервлета и инициализирует его. После этого экземпляр сервлета начинает обработку запроса и Веб-контейнер передает ответ клиенту, сгенерированный сервлетом. Рис. 23 показывает этапы, выполняемые веб-контейнером при первом получении запроса от клиента.

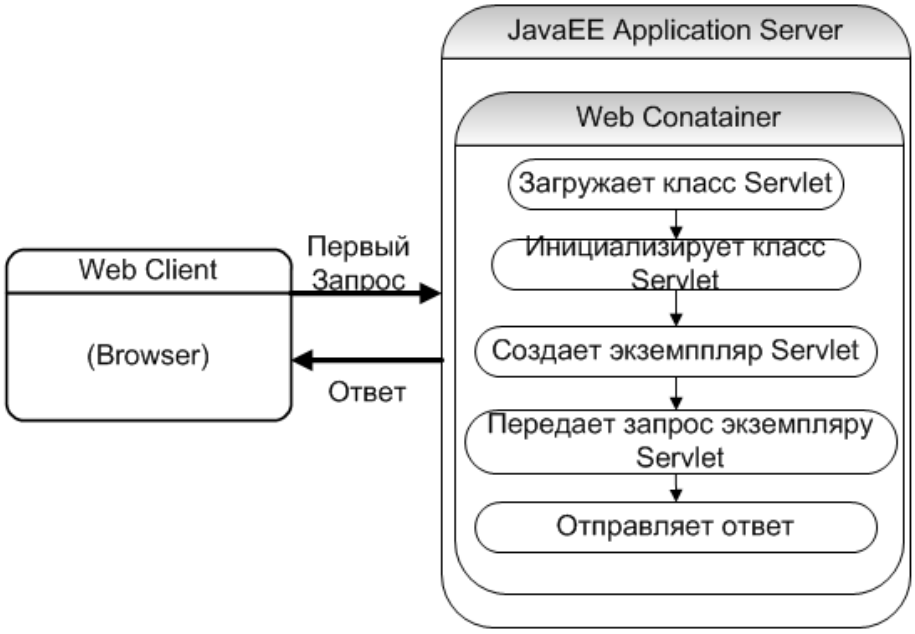


Рисунок 23 – Действия, выполняемые веб-контейнером

Веб-контейнер управляет сервлетом, вызывая различные методы жизненного цикла. Эти методы определены в Java Servlet API и сосредоточены в наборе классов и интерфейсов, которые можно использовать для разработки сервлета. Эти классы и интерфейсы находятся в пакетах `javax.servlet` и `javax.servlet.http`.

Интерфейс `Servlet` является корневым интерфейсом иерархии сервлетов и всем сервлетам необходимо явно или неявно реализовывать этот интерфейс. Класс `GenericServlet` из Servlet API реализует интерфейс `Servlet` и кроме того, он реализует интерфейс `ServletConfig` из Servlet API и интерфейс `Serializable` из стандартного пакета `java.io`. Объект интерфейса `ServletConfig` используется веб-контейнером для передачи конфигурационной информации сервлету при его инициализации.

Для разработки сервлета, который взаимодействует по протоколу HTTP, необходимо расширить класс `HttpServlet` в сервлете. Класс `HttpServlet` расширяет класс `GenericServlet` и таким образом предоставляет встроенную функциональность HTTP. Например, `HttpServlet` предоставляет методы, которые позволяют сервлету обрабатывать запрос клиента, приходящий с

помощью определенного метода HTTP. Рис. 24 показывает общую схему иерархии интерфейсов и классов в пакетах `javax.servlet` и `javax.servlet.http`.

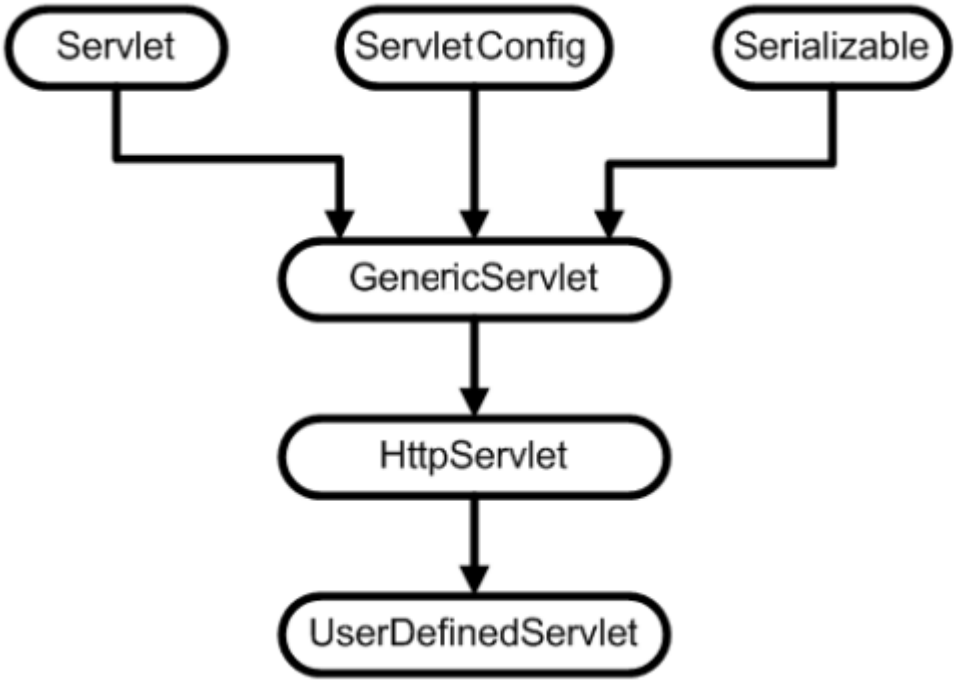


Рисунок 24 – Иерархия классов сервлетов

Интерфейс `Servlet` из пакета `javax.servlet` определяет методы, которые веб-контейнер вызывает в процессе управления жизненным циклом сервлета. В таблице 7 представлены методы интерфейса `javax.servlet.Servlet`.

Т а б л и ц а 7 — Методы интерфейса `Servlet`

Метод	Описание
`public void destroy()`	Веб-контейнер вызывает метод `destroy()` перед уничтожением экземпляра сервлета.
`public ServletConfig getServletConfig()`	Возвращает объект `ServletConfig`, который содержит данные о параметрах инициализации сервлета.
`public String getServletInfo()`	Возвращает строку, которая содержит

	информацию о сервлете, такую как автор, версия и авторское право.
`public void init(ServletConfig config) throws ServletException`	Веб-контейнер вызывает этот метод после создания экземпляра сервлета.

Интерфейс `javax.servlet.ServletConfig` реализуется веб-контейнером, чтобы иметь возможность передавать данные о конфигурации сервлету во время инициализации сервлета. Сервлет инициализируется передачей веб-контейнером объекта `ServletConfig` методу `init()`. Объект `ServletConfig` содержит данные для инициализации, а также предоставляет доступ к объекту `ServletContext`.

Параметры инициализации – это пары имен и их значений, которые можно использовать для передачи информации сервлету. Например, можно указать URL JDBC как параметр инициализации сервлета. При инициализации сервлет может использовать значение URL, чтобы получить соединение с базой данных. Контекст, известный как «servlet context» или «web context» создается веб-контейнером как объект интерфейса `ServletContex` и позволяет сервлету взаимодействовать с веб-контейнером, который содержит этот сервлет. Для каждого веб-приложения существует единственный `ServletContext`, и он доступен всем ресурсам приложения. В таблице 8 представлены несколько методов интерфейса `javax.servlet.ServletConfig`.

Т а б л и ц а 8 — Методы интерфейса `ServletConfig`

Метод	Описание
`public String getInitParameter(String param)`	Возвращает строку, содержащую значение указанных параметров инициализации или `null`, если параметр не существует.
`public Enumeration getInitParameterNames()`	Возвращает имена всех параметров инициализации как объект `Enumeration`, содержащий объекты `String`. Если параметры инициализации не

	определены, то возвращается пустой `Enumeration`.
`public ServletContext getServletContext()`	Возвращает объект `ServletContext` для сервлета, который позволяет взаимодействие с веб-контейнером.

Интерфейс `javax.servlet.Servlet` определяет методы жизненного цикла сервлета: `init()`, `service()` и `destroy()`. Веб-контейнер вызывает методы сервлета `init()`, `service()` и `destroy()` во время его жизни.

1. Веб-контейнер загружает класс сервлета и создает один или более экземпляров класса сервлета;
2. Веб-контейнер вызывает метод `init()` экземпляра сервлета во время инициализации сервлета. Метод `init()` вызывается только один раз в жизненном цикле сервлета;
3. Веб-контейнер вызывает метод `service()`, разрешая сервлету выполнить обработку запроса клиента;
4. Метод `service()` обрабатывает запрос и возвращает ответ веб-контейнеру;
5. После этого сервлет ожидает получения и обработки последующих запросов, как это происходило на этапах 3 и 4;
6. Веб-контейнер вызывает метод `destroy()` перед удалением экземпляра сервлета. Метод `destroy()` также вызывается только один раз во время жизни сервлета.

Метод `init()` вызывается только однажды во время инициализации жизненного цикла сервлета, что дает возможность сервлету выполнить любую работу, например, открытие файлов или установку соединений с серверами БД. Если сервлет установлен на сервере постоянно, он загружается при запуске сервера. В противном случае сервер активизирует сервлет при получении первого запроса от клиента на выполнение услуги данным сервлетом. Веб-контейнер определяет нужный сервлет для обработки запроса клиента. После

этого веб-контейнер создает объект интерфейса `ServletConfig`, который содержит данные конфигурации при запуске - параметры инициализации сервлета. Далее веб-контейнер вызывает метод `init()` сервлета и передает ему объект `ServletConfig`.

Метод `init()` генерирует исключение `ServletException`, если веб-контейнер не может инициализировать ресурсы сервлета. Гарантируется, что метод `init()` закончится перед любым другим обращением к сервлету, например, вызовом метода `service()`.

Метод `service()` является ядром сервлета и выполняет обработку запросов клиента. Всякий раз, когда веб-контейнер получает клиентский запрос, он вызывает метод `service()`. Метод `service()` вызывается только после завершения инициализации сервлета. При вызове метода `service()` веб-контейнер передает объект интерфейса `ServletRequest` и объект интерфейса `ServletResponse`. Объект `ServletRequest` содержит данные о запросе клиента, а объект `ServletResponse` содержит информацию, возвращаемую сервлетом клиенту. Следующий фрагмент кода показывает сигнатуру метода `service()`:

```
public void service(ServletRequest request, ServletResponse
                    response) throws ServletException,IOException
```

Метод `service()` генерирует исключение `ServletException`, когда возникает исключительная ситуация, которая прерывает нормальную работу сервлета. Также метод `service()` генерирует исключение `IOException`, когда возникает исключительная ситуация ввода или вывода.

Метод `service()` отправляет клиентский запрос одному из методов обработки запроса интерфейса `HttpServlet`: `doGet()`, `doPost()`, `doHead()` или `doPut()`. Методы обработки запросов принимают объекты `HttpServletRequest` и `HttpServletResponse` как параметры от метода `service()`.

Метод `service()` не переопределяется в `HttpServlet`, так как веб-контейнер автоматически вызывает метод `service()`. Функциональность HTTP сервлетов содержится в методах `doGet()` или `doPost()`.

Метод `doGet()` обрабатывает клиентский запрос, использующий GET-метод протокола HTTP. GET является одним из методов запроса HTTP, который, в основном, используется для извлечения статических ресурсов. Когда в строке браузера вводите адрес URL для просмотра статической веб-страницы, браузер использует метод GET для выполнения запроса. Аналогично, когда выбирается гиперссылка на веб-странице для получения доступа к некоторому ресурсу, то запрос отправляется с применением метода GET. Также, используя метод GET, можно отправлять данные, введенные пользователем в теге `<form>` HTML. Данные, отправленные с применением GET-метода, добавляются как строка запроса к URL. Тип метода HTTP указывается в форме HTML, используя атрибут `method` тега `<form>`.

Для обработки клиентских запросов, которые получены с использованием метода GET, необходимо переопределить метод `doGet()` в классе сервлета. В методе `doGet()` можно извлекать данные от клиента из объекта `HttpServletRequest`, а также использовать объект `HttpServletResponse` для формирования ответа клиенту. Следующий фрагмент программы демонстрирует сигнатуру метода `doGet()`:

```
protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
```

Метод `doPost()` обрабатывает запросы в сервлете, которые отправлены клиентом, используя метод POST протокола HTTP. Например, если клиент вводит регистрационные данные в форму HTML, данные могут быть отправлены с использованием метода POST. В отличие от метода GET, запрос POST отправляет данные как часть тела запроса HTTP и в результате отправленные данные не видны как часть URL.

Для обработки запросов в сервлете, которые отправляются с использованием метода POST, необходимо переопределить `doPost()`. В методе `doPost()` можно обработать запрос и отправить ответ клиенту. Следующий фрагмент кода демонстрирует сигнатуру метода `doPost()`:

```
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
```

Метод `doPut()` обрабатывает запросы, отправленные с использованием метода PUT протокола HTTP. Метод PUT позволяет клиенту сохранить информацию на сервере. Например, можно использовать метод PUT для отправки файла с изображением на сервер. Следующий фрагмент кода демонстрирует сигнатуру метода `doPut()`:

```
protected void doPut(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
```

Метод `destroy()` вызывается для освобождения всех ресурсов, занимаемых сервлетом, например, открытых файлов и соединений с базой данных, перед выгрузкой сервлета. Этот метод может быть пустым, если нет необходимости выполнения каких-либо завершающих операций, или в нем можно выполнить действия по освобождению удерживаемых ресурсов. Веб-контейнер вызывает метод `destroy()` перед удалением экземпляра сервлета в следующих случаях

— период времени, указанный для жизни сервлета, истек (период времени сервлета – это период, в течение которого сервлет сохраняется в активном состоянии веб-контейнером для обслуживания клиентских запросов);

— веб-контейнеру нужно выгрузить экземпляры сервлетов для экономии памяти;

— веб-контейнер заканчивает работу.

В методе `destroy()` можно написать код для освобождения ресурсов, занимаемых сервлетом. Метод `destroy()` также используется для сохранения любой предназначенной для длительного использования ин-формации перед тем, как экземпляр сервлета будет удален. Необходимо создавать метода `destroy()` таким образом, чтобы избежать закрытия необходимых ресурсов до тех пор, пока все вызовы `service()` не завершатся.

Для создания сервлета, необходимо выполнить следующие шаги:

1. написать код сервлета;
2. скомпилировать и упаковать сервлет;
3. развернуть сервлет как приложение JavaEE;
4. вызвать сервлет из браузера.

Пакеты сервлетов определяют два абстрактных класса, которые реализуют интерфейс `Servlet`: класс `GenericServlet` (из пакета `javax.servlet`) и класс `HttpServlet` (из пакета `javax.servlet.http`). Эти классы предоставляют реализацию по умолчанию для всех методов интерфейса `Servlet`. Большинство разработчиков используют либо класс `GenericServlet`, либо класс `HttpServlet`, и замещают некоторые или все методы. Чтобы написать класс сервлета мы будем использовать расширение интерфейса `HttpServlet`. Двумя наиболее распространенными типами запросов HTTP (их также называют методами запросов) являются GET и POST, которые соответственно получают (или извлекают) информацию от клиента и помещают (или отправляют) данные клиенту. В классе `HttpServlet` определены методы `doGet()` и `doPost()` для реакции на запросы типа GET и POST клиента. Эти методы вызываются методом `service()` класса `HttpServlet`, который, в свою очередь, вызывается при поступлении запроса на сервер. Метод `service()` сначала определяет тип запроса, а затем вызывает соответствующий метод.

После объявления и инициализации различных переменных сервлета, необходимо описать функциональность сервлета. Для этого переопределяются методы `doGet()` и `doPost()` класса `HttpServlet`. Методы `doGet()` и `doPost()`

получают запросы HTTP, используя интерфейс `HttpServletRequest`. Объект `HttpServletRequest` содержит информацию о запросе, отправленном клиентом с помощью запроса HTTP. Можно получить значения параметра запроса, вызывая метод `getParameter()` интерфейса `HttpServletRequest`. Метод `getParameter()` интерфейса `HttpServletRequest` принимает имя параметра запроса как строковое значение и возвращает строку, которая содержит соответствующее значение параметра.

Для отправки ответа клиенту HTTP-сервлеты используют объект интерфейса `HttpServletResponse`. Чтобы отправить символьные данные клиенту, нужно получить объект `java.io.PrintWriter`. Можно использовать метод `getWriter()` интерфейса `HttpServletResponse`, чтобы получить объект `PrintWriter`.

Отправить данные клиенту можно, используя метод `println()` класса `PrintWriter`.

Теперь перейдем непосредственно к созданию веб-приложения, которые будет использовать сервлеты.

Для выполнения работы потребуется установка другой интегрированной среды разработки – Apache NetBeans. Ее нужно скачать с официального сайта и выполнить стандартную установку. Далее необходимо запустить программу. В области обозревателя проектов правой кнопкой мыши надо вызывать контекстное меню и выбрать «New Project…». См. рисунок 25.

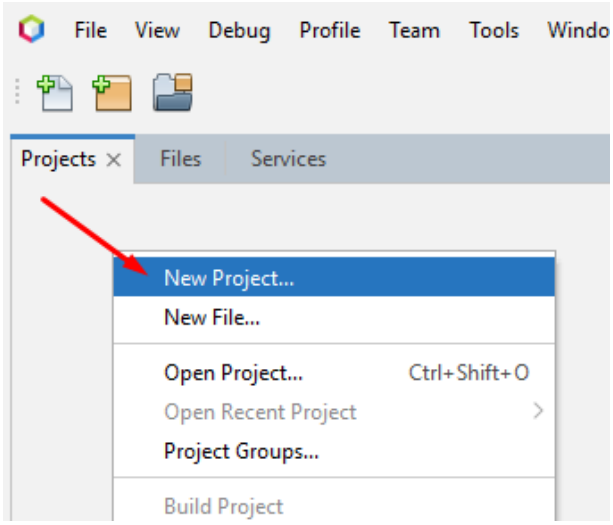


Рисунок 25 – Выбор опции для создания нового проекта

Откроется окно создания нового проекта. Нужно выбрать «Java with Maven» и «Web Application». Затем нужно нажать «Next». Как написано в информационном поле, создание веб-приложений в данный момент неактивно и нужно нажать «Далее», чтобы активировать эту возможность (рисунок 26).

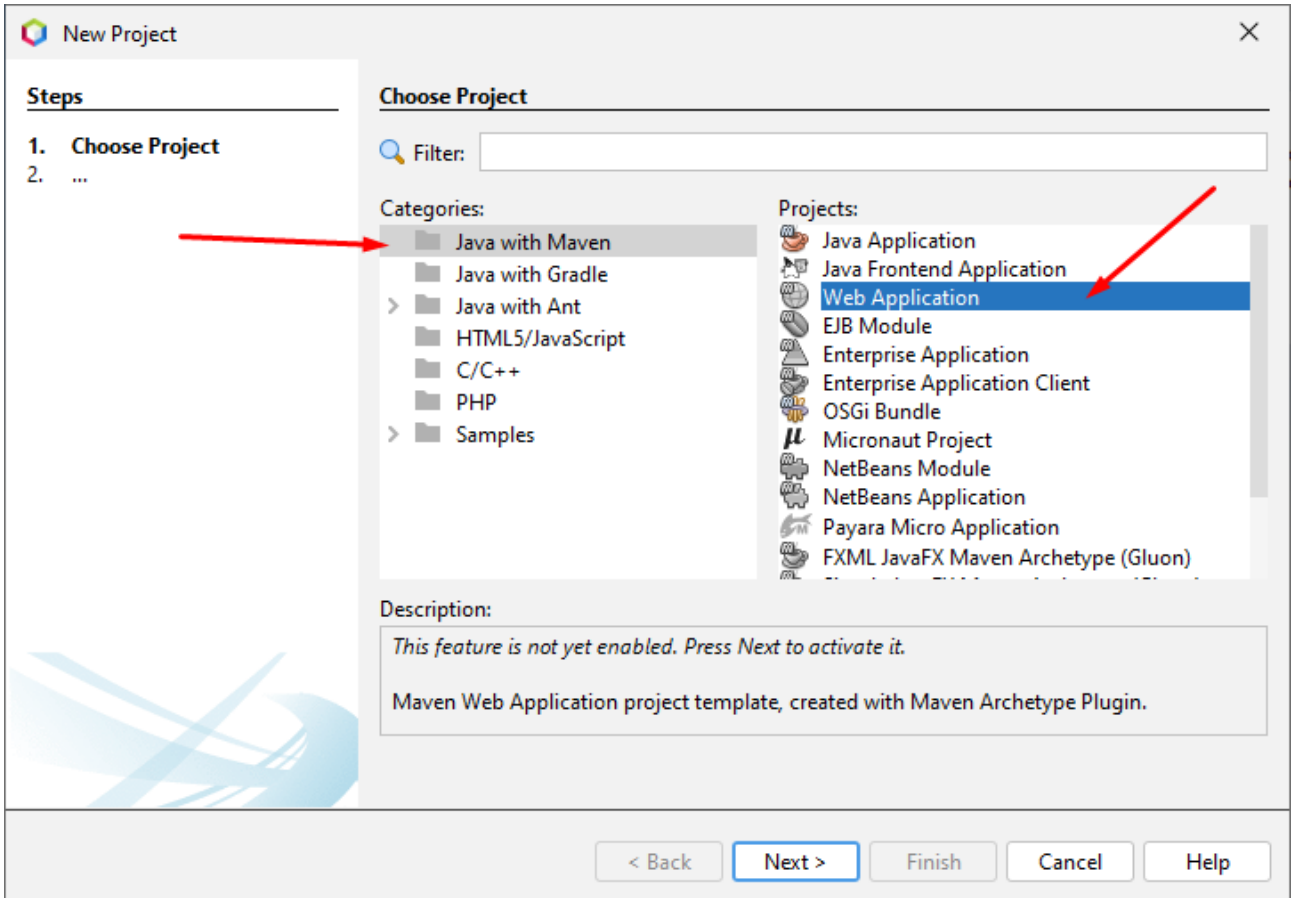


Рисунок 26 – Выбор типа нового проекта

Среда разработки выполнит активацию необходимых компонентов для веб-разработки. Далее необходимо указать имя создаваемого проекта и некоторую дополнительную информацию о проекте (рисунок 27).

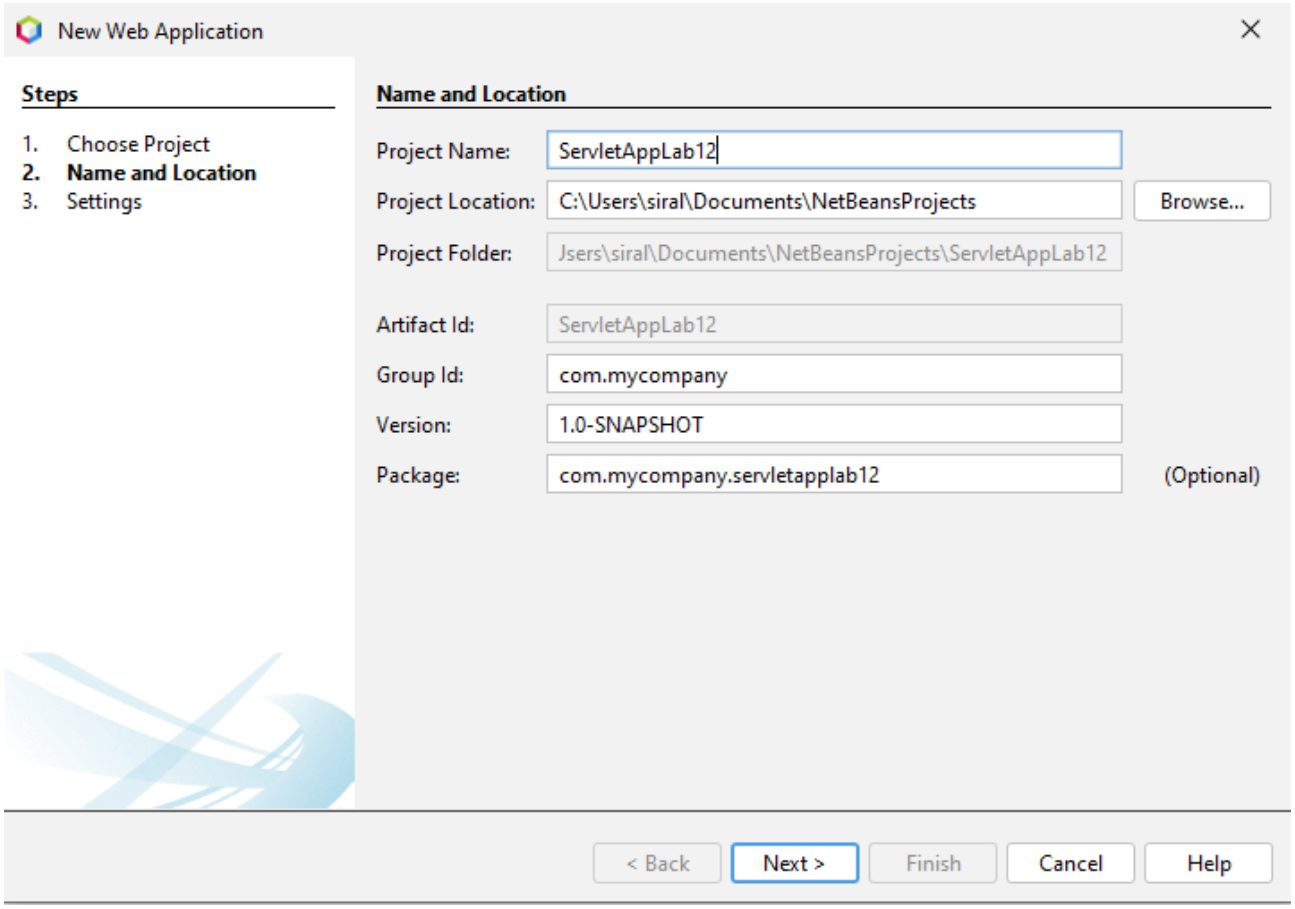


Рисунок 27 – Настройки проекта

Нажимаем «Next». Открывается меню выбора сервера приложений и версии JavaEE. На рисунке 28 показано это меню. Сейчас нет ни одного доступного сервера для выбора. Нажимаем кнопку «Add…» для создания нового сервера.

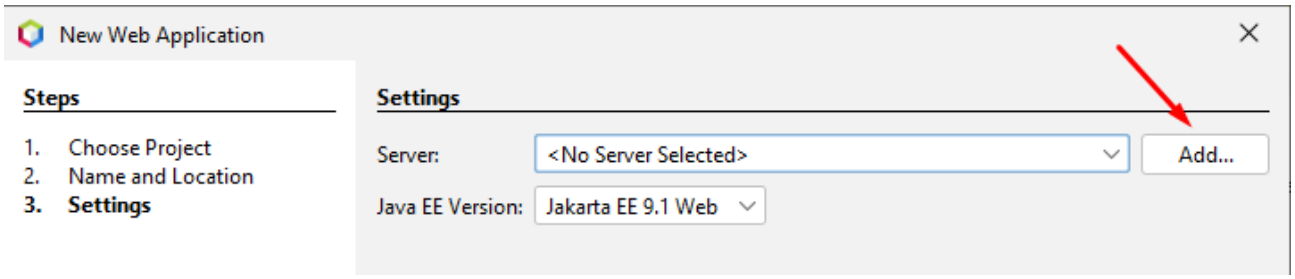


Рисунок 28 – Выбор сервера и версии JavaEE

Выбираем GlassFish Server (рисунок 29).

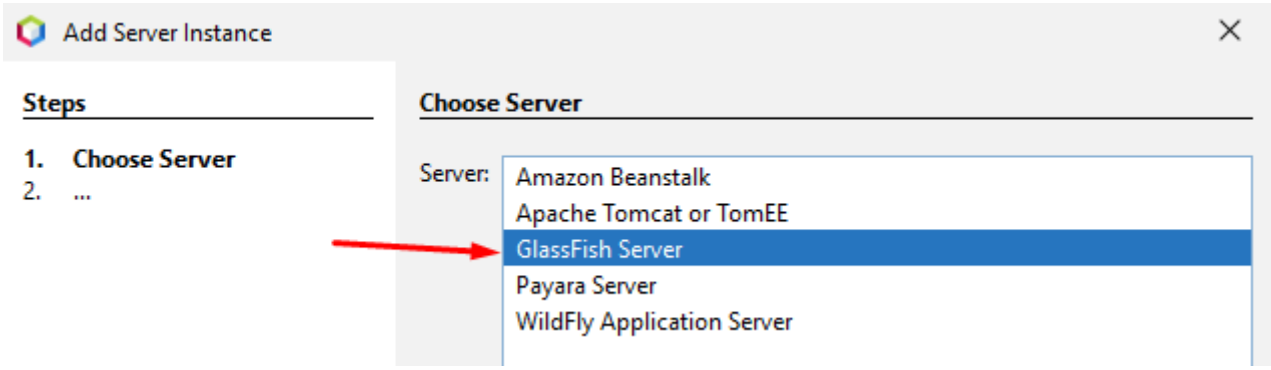


Рисунок 29 – Выбор сервера

Далее выполняется первичная настройка сервера и скачивание его файлов. На рисунке 30 показан порядок действий.

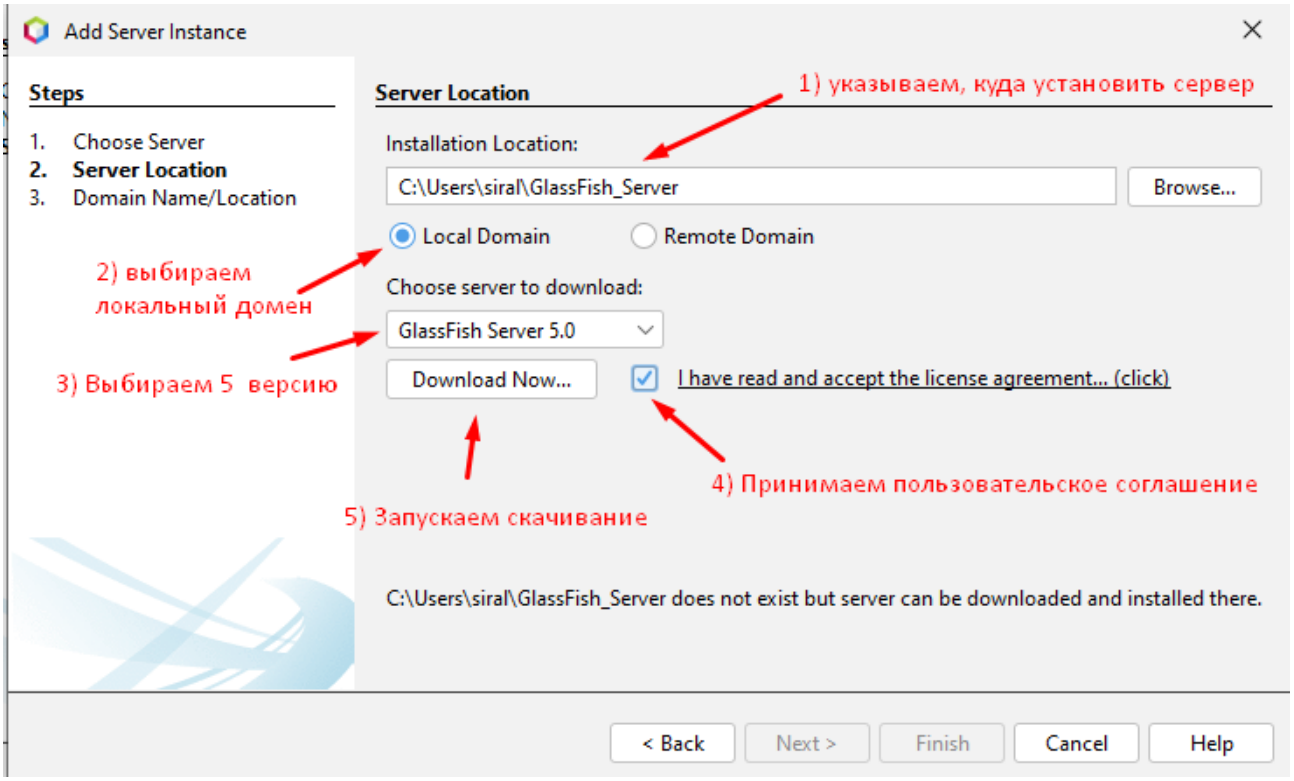


Рисунок 30 – Первичная настройка и загрузка файлов сервера

После скачивания переходим далее. Последним этапом является настройка домена. Так как проект учебный, оставим все по умолчанию и нажимаем «Finish» (рисунок 31).

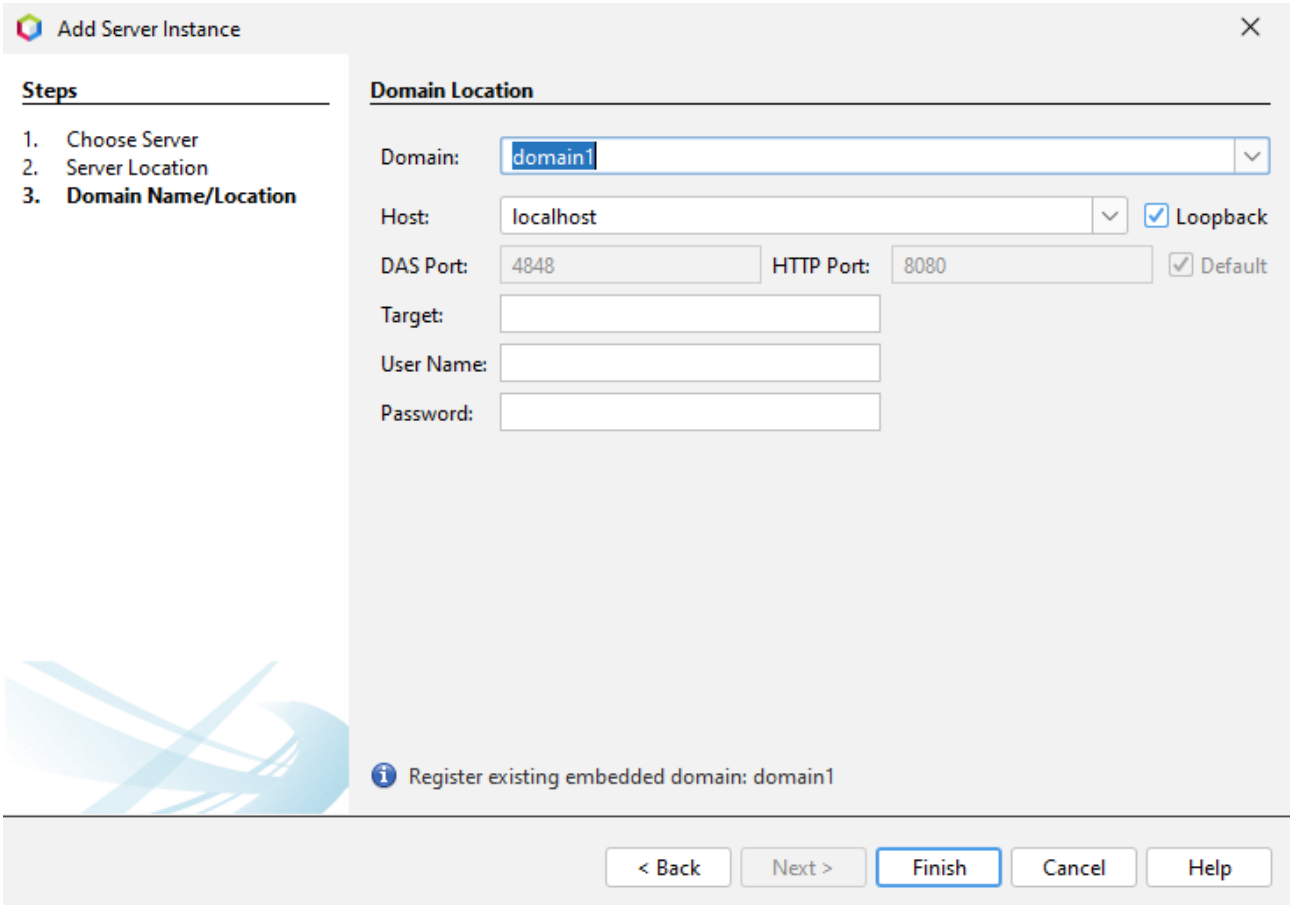


Рисунок 31 – Настройки домена

Далее возвращаемся обратно в окно выбора сервера и версии JavaEE. Выбираем только что созданный сервер и версию JavaEE 8 Web (рисунок 32).

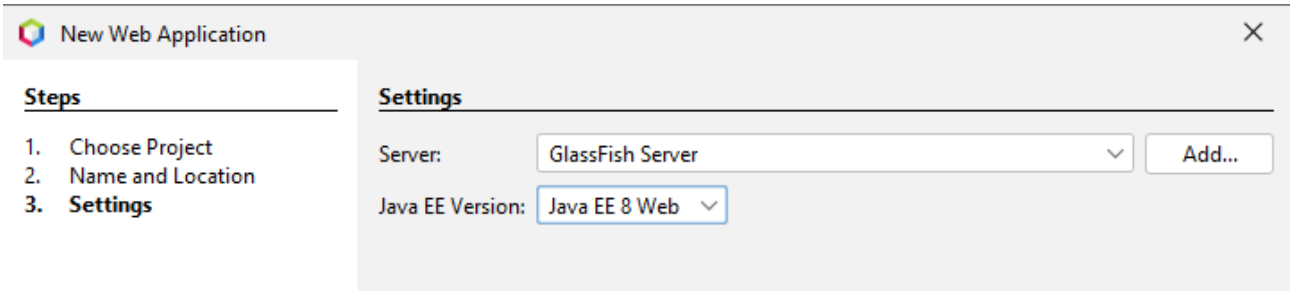


Рисунок 32 – Сервер и версия выбраны

Нажимаем «Finish», проект создан.

Теперь выполним настройки базы данных. Подключимся к ней, создадим таблицу и наполним ее данными. На сервере GlassFish по умолчанию создается база данных Apache Derby, она и будет использоваться в работе.

Для начала необходимо запустить созданный сервер (рисунок 33). Для переходим во вкладку «Services», раскрываем список «Servers», в контекстном меню сервера выбираем «Start».

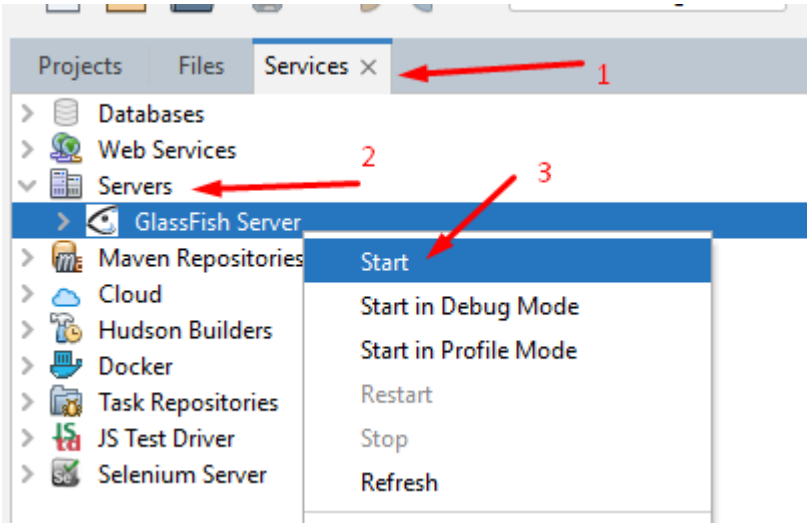


Рисунок 33 – Запуск сервера

Может возникнуть ошибка, как на рисунке 34. Для решения этой проблемы нужно выбрать другую версию Java SE. Этот процесс будет описан ниже. Нажимаем «Manage Platforms».

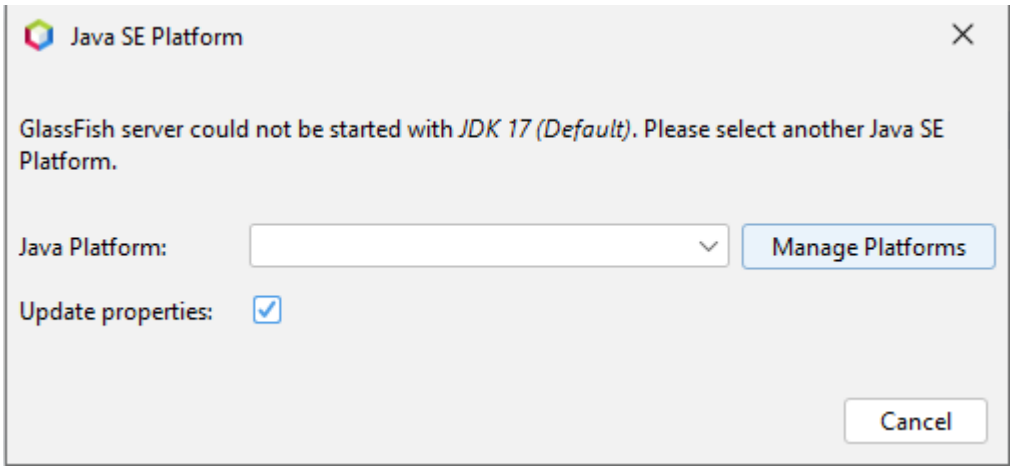


Рисунок 34 – Возможная ошибка при запуске сервера

В открывшемся окне нажимаем «Add Platform…» (рисунок 35).

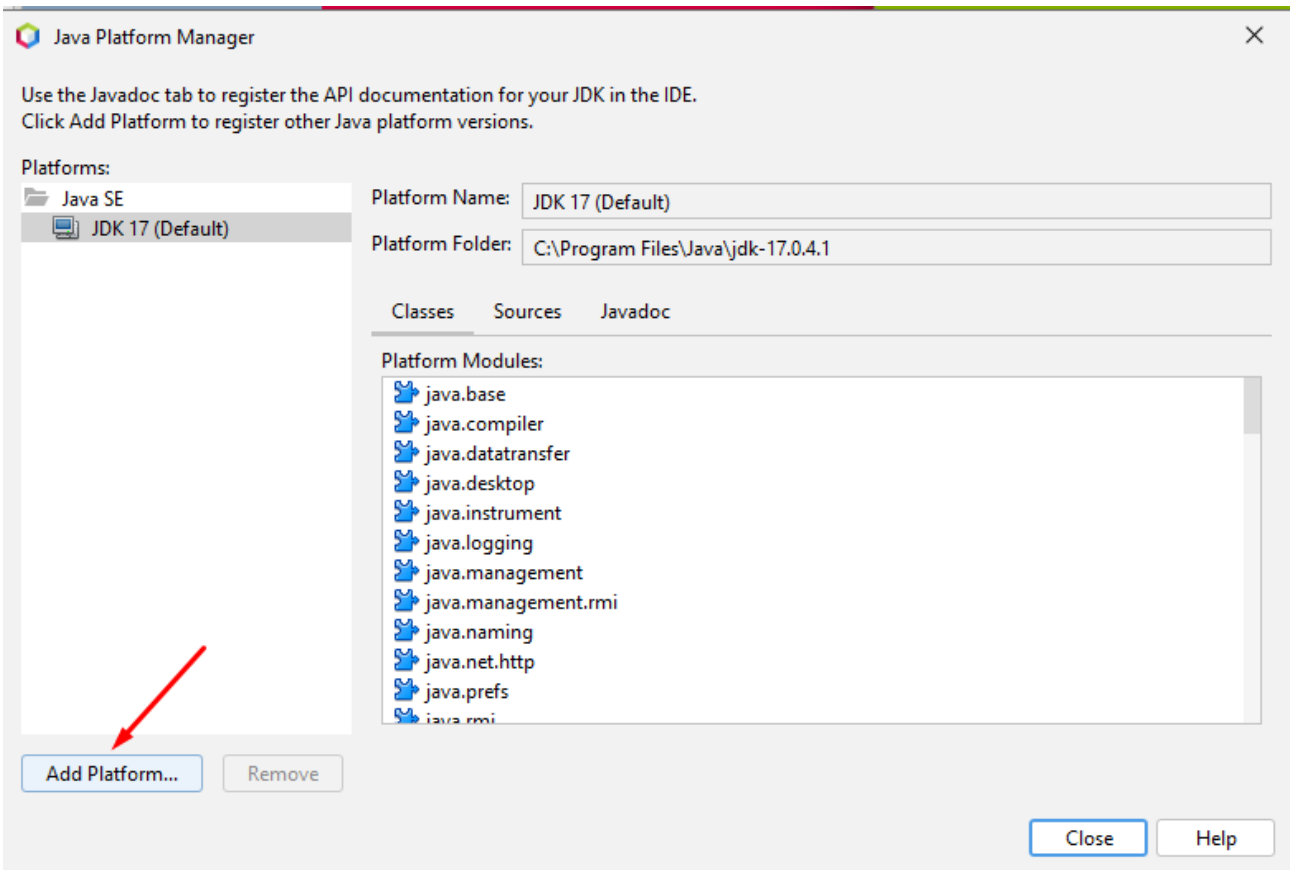


Рисунок 35 – Добавление платформы

В открывшемся окне выбираем опцию «Download OpenJDK» (рисунок 36).

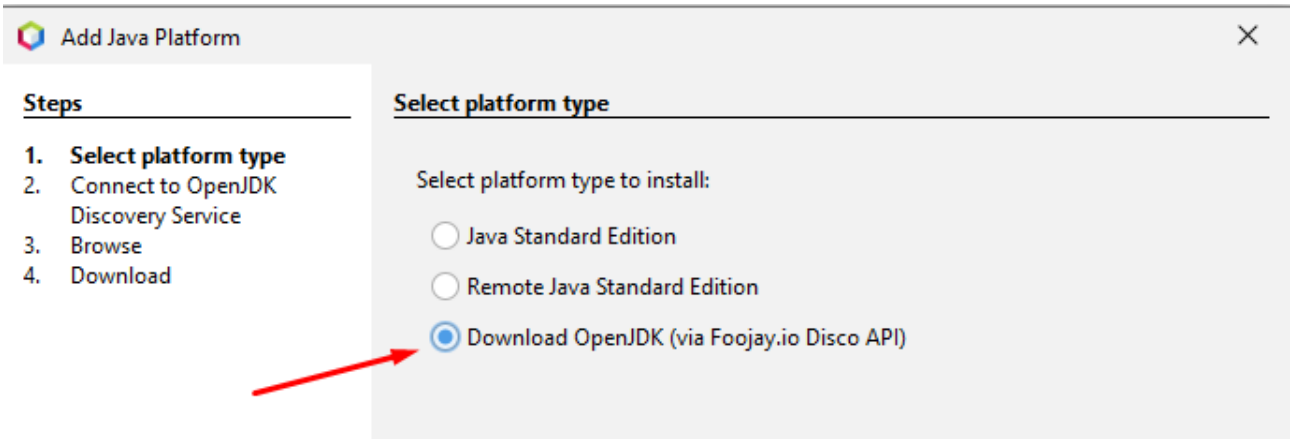


Рисунок 36 – Выбор типа загружаемой платформы Java SE

Далее выбираем 8 версию и distribution «OJDKBuild» (рисунок 37).

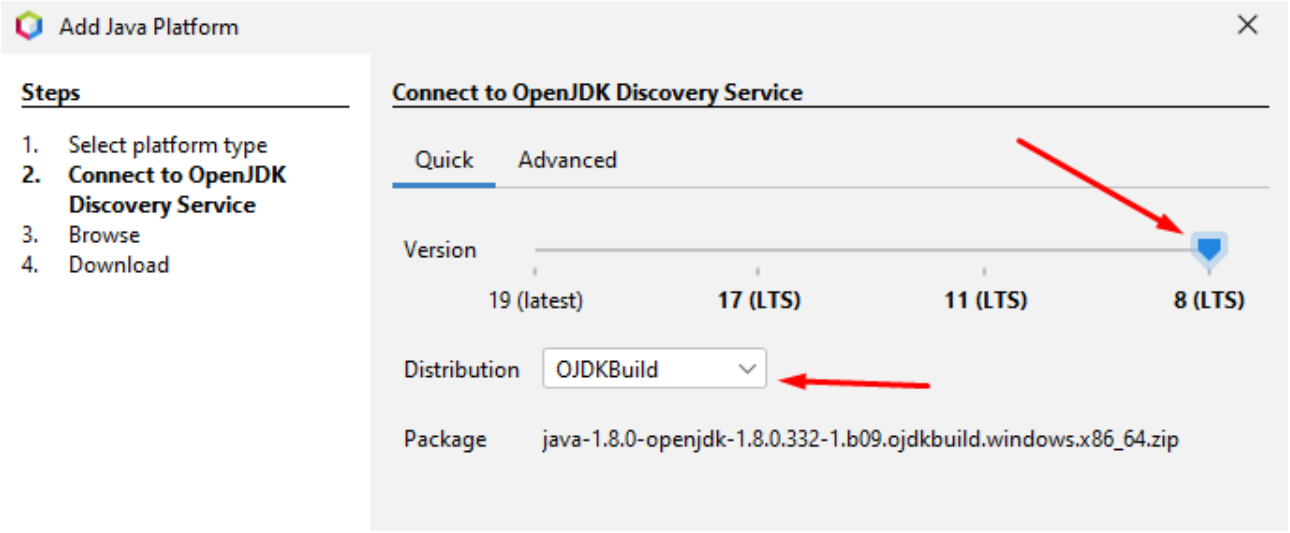


Рисунок 37 – Выбор версии загружаемой платформы Java SE

Жмем «Next» и указываем директорию, в которую будет установлена платформа (рисунок 38).

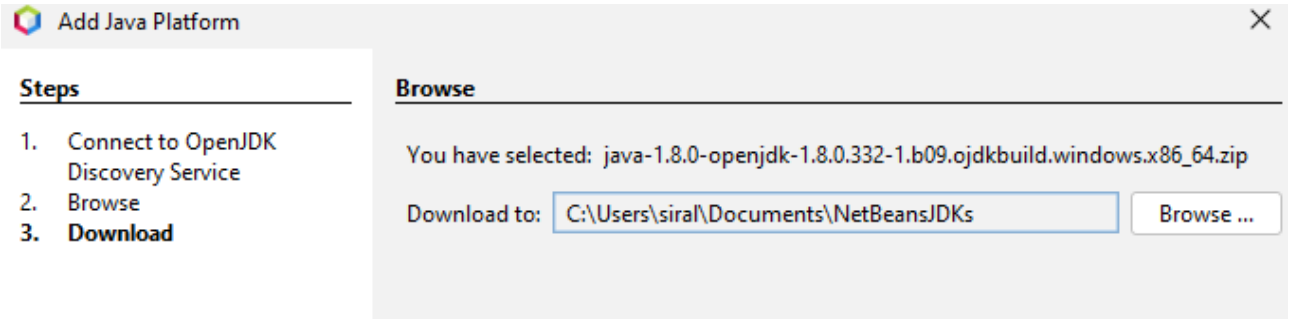


Рисунок 38 – Выбор директории для установки

Жмем «Next» и запускается процедура скачивания. После этого можно выбрать установленную платформу для запуска сервера (рисунок 39).

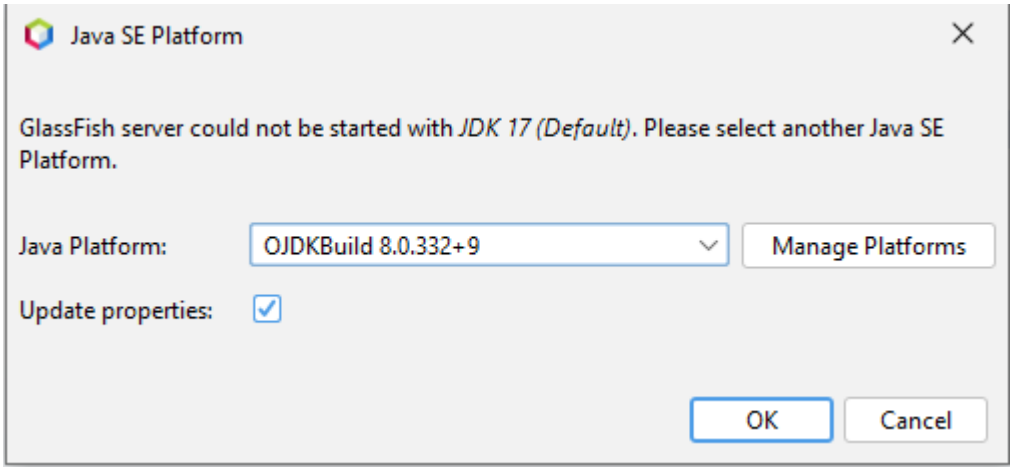


Рисунок 39 – Выбор платформы для запуска сервера

После этого сервер запустится автоматически.

Продолжим настройку базы данных. Во вкладке «Services» нужно вызвать контекстное меню списка «Databases». Выбираем создание нового подключения.

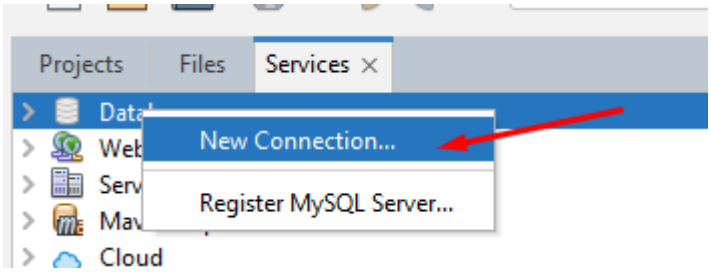


Рисунок 40 – Выбор создания нового подключения

В открывшемся окне нужно выбрать «драйвер» БД. Выбираем «Java DB (Network)», так как будет подключаться к БД на сервере. Далее настраиваем подключение, процесс показан на рисунке 41.

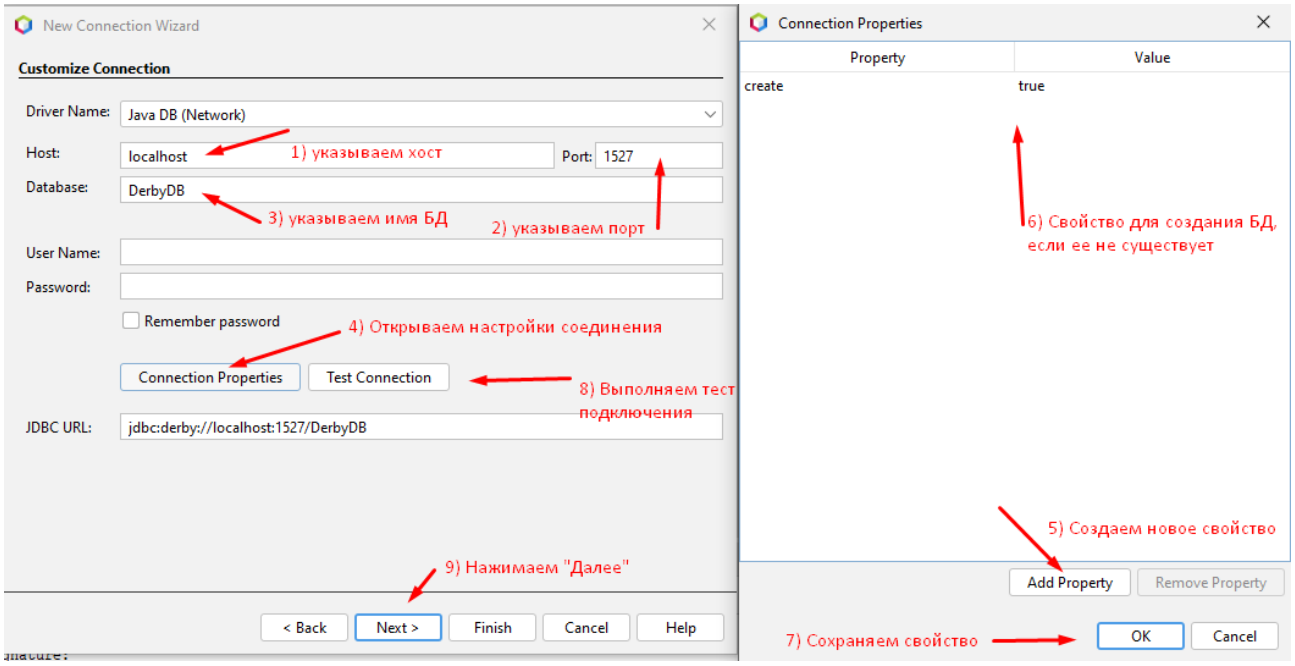


Рисунок 41 – Настройки нового подключения

Нужно выбрать хост и порт, имя БД. Указать логин и пароль для подключения к БД. В данном случае логин и пароль не устанавливаются. Затем нужно добавить свойство подключения «create=true», чтобы создать базу данных, так как ее не существует. Для проверки созданного подключения

нужно нажать «Test Connection», должно отобразиться «Connection Succeeded». Далее выбираем схему, оставляем по умолчанию «APP». Нажимаем «Finish».

В обозревателе баз данных отобразится созданное подключение. Также IDE автоматически подключится к БД. Перейдем к заданию на данную лабораторную работу.

Постановка задачи: необходимо разработать веб-приложение, использующее сервлет, для поиска информации о сотрудниках организации. Данные о сотрудниках хранятся в таблице Employee в базе данных. Для выполнения поиска пользователь указывает фамилию сотрудника и просматривает информацию о найденных сотрудниках (допускается существование нескольких сотрудников с одинаковыми фамилиями).

Таким образом, в БД нужно создать таблицу Employee и добавить в нее данные сотрудников. Для этого на созданном подключении вызываем контекстное меню и выбираем «Execute Command…».

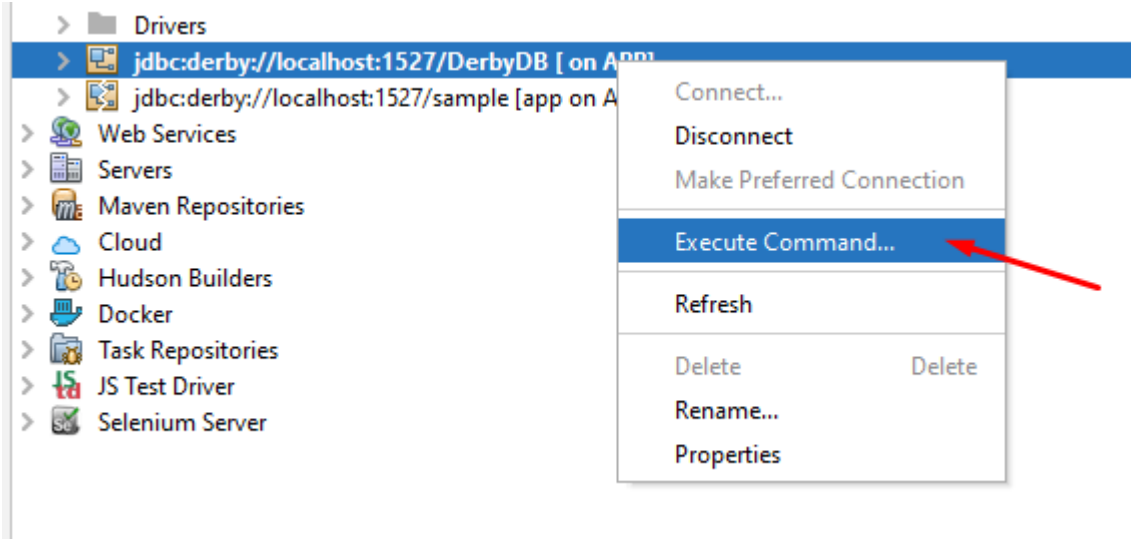


Рисунок 42 – Вызов редактора SQL-инструкций

Далее нужно вставить SQL-инструкции, приведенные ниже, и выполнить выражения SQL (рисунок 43).

```
drop table employee;
-- создание таблицы
create table employee(id integer, first_name varchar(20),
last_name varchar(20), designation varchar(20), phone
varchar(20));
```

```
--вставка тестовых данных
insert into employee values (1, 'Ivan', 'Ivanov', 'Manager', '11-
22-33');
insert into employee values (2, 'Nikolay', 'Ivanov', 'Programmer',
'33-44-55');
insert into employee values (3, 'Sergey', 'Petrov', 'System
administrator', '12-34-56');
insert into employee values (4, 'Alexey', 'Petrov', 'Manager',
'56-78-90');
insert into employee values (5, 'Vitaliy', 'Kuznetsov',
'Technician', '55-66-77');
-- выбрать все из таблицы для проверки
select * from employee;
```

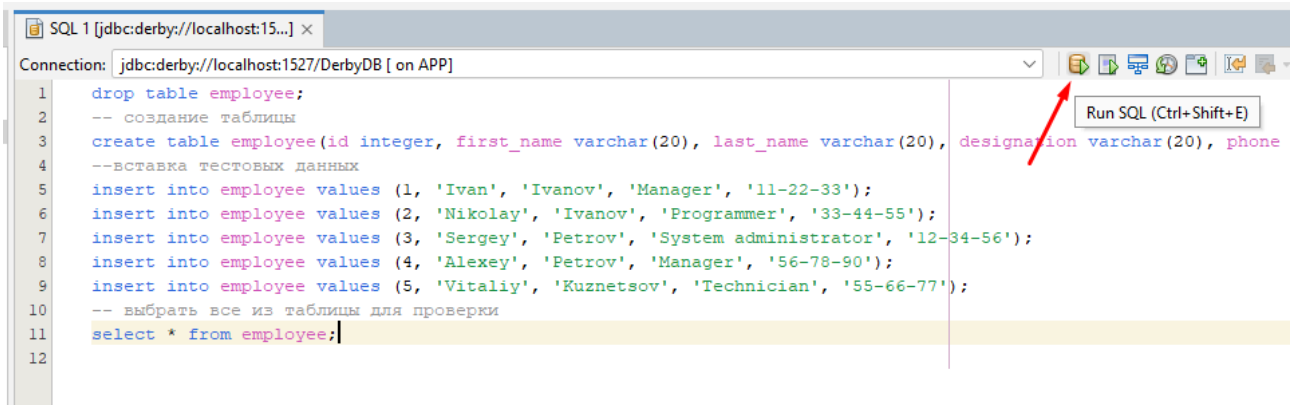
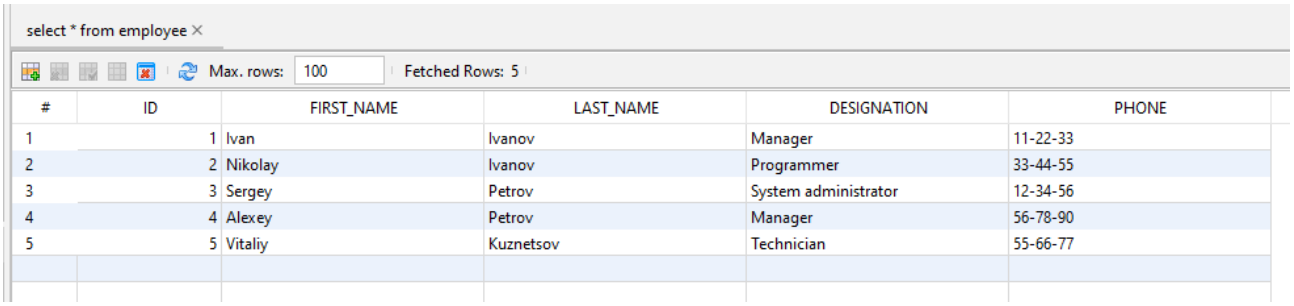


Рисунок 43 – Выполнение SQL-инструкций

В случае успешного выполнения инструкций, отобразится таблица Employee с добавленными данными (рисунок 44).



#	ID	FIRST_NAME	LAST_NAME	DESIGNATION	PHONE
1	1	Ivan	Ivanov	Manager	11-22-33
2	2	Nikolay	Ivanov	Programmer	33-44-55
3	3	Sergey	Petrov	System administrator	12-34-56
4	4	Alexey	Petrov	Manager	56-78-90
5	5	Vitaliy	Kuznetsov	Technician	55-66-77

Рисунок 44 – Таблица данных из БД

Переходим в директорию проекта в обозревателе проектов. В директории «Source Packages», создаем новый сервлет в корневом пакете проекта com.mycompany.servletapplab12 (рисунок 45).

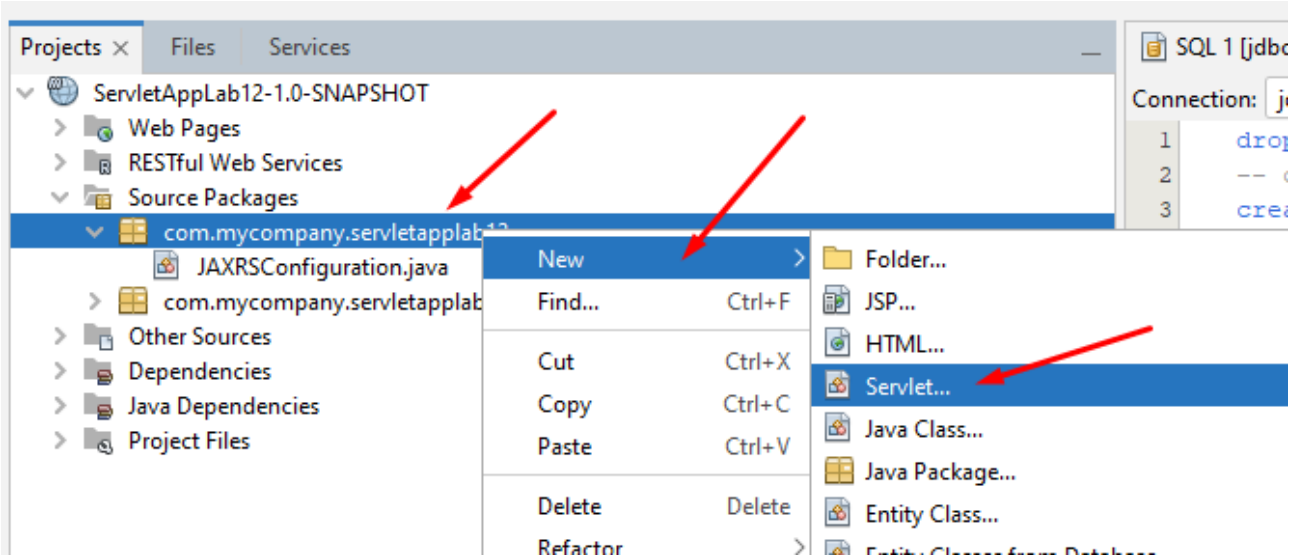


Рисунок 45 – Создание сервлета

Указываем имя сервлета как на рисунке 46 и жмем «Finish».

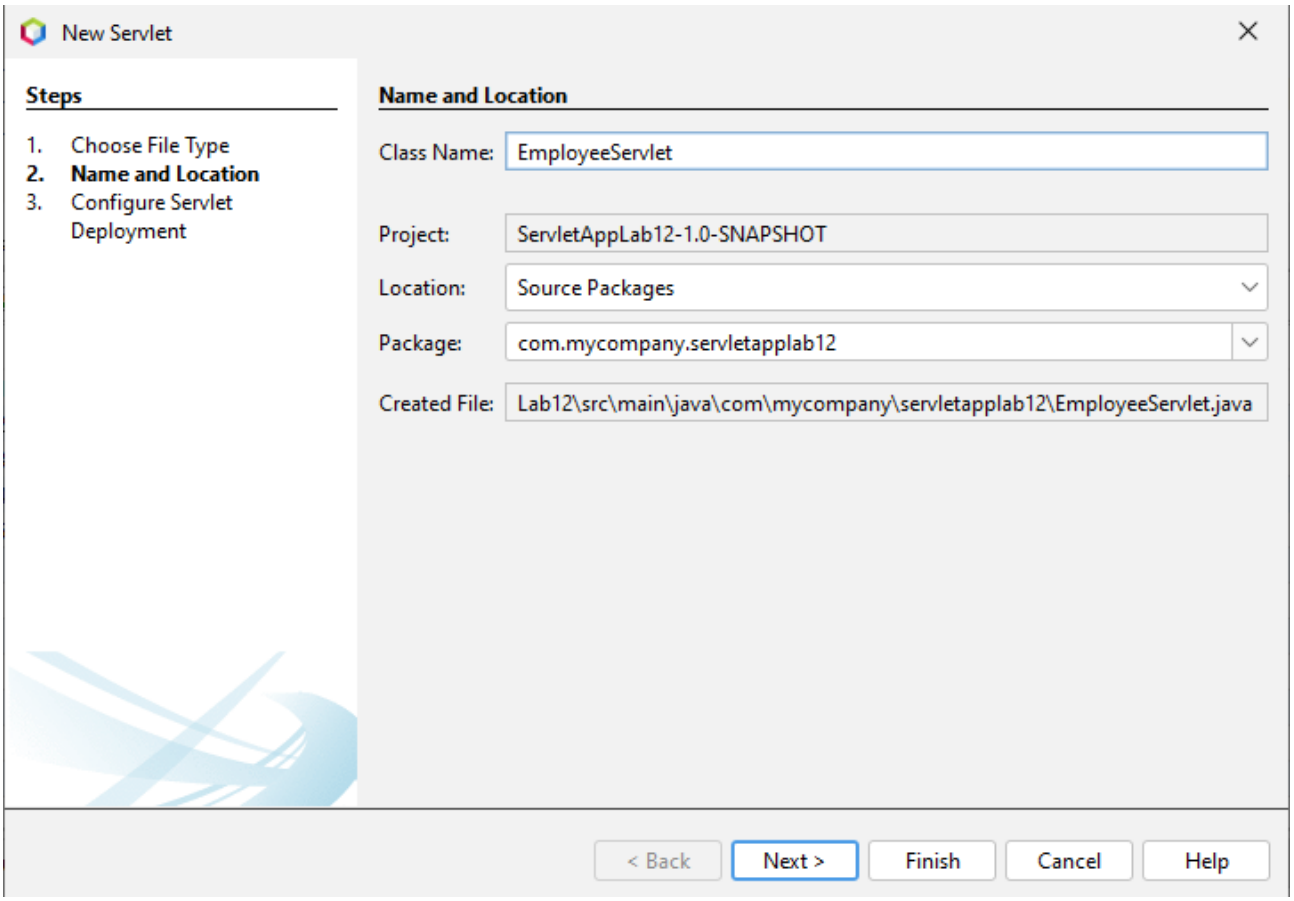


Рисунок 46 – Указание имени сервлета

Будет создан сервлет со стандартным наполнением. Теперь нужно создать класс, который будет представлять сущность сотрудника – `Employee`.

Как добавить новый класс, показано на рисунке 47. Укажите имя «Employee».

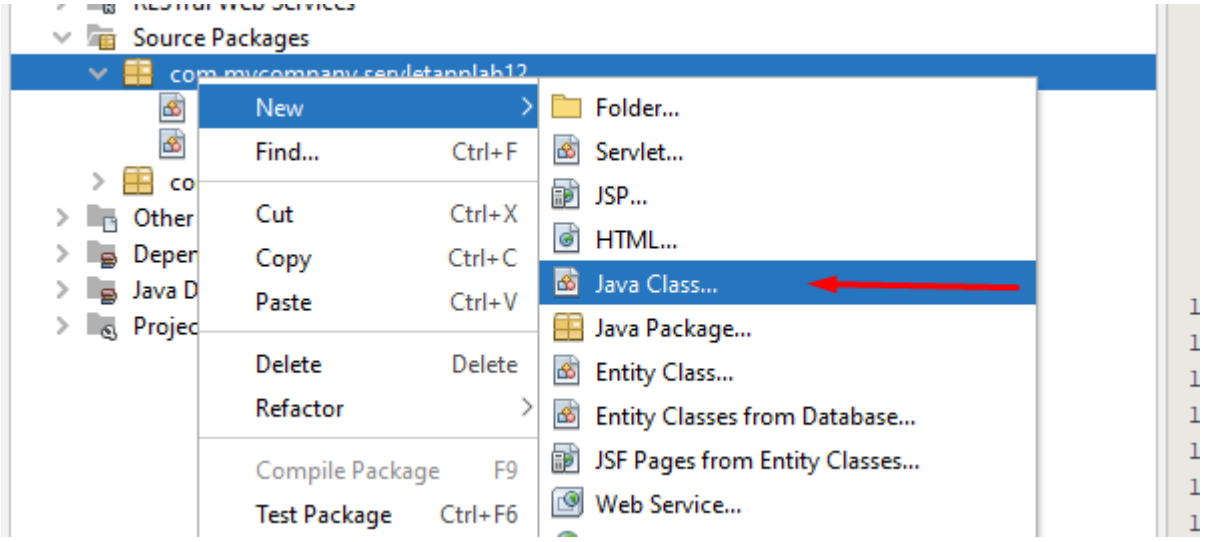


Рисунок 47 – Добавление нового класса

Код класса `Employee` приведен ниже.

```
package com.mycompany.servletapplab12;

import java.io.Serializable;

public class Employee implements Serializable {

    private Long id;
    private String firstName;
    private String lastName;
    private String designation;
    private String phone;

    public Employee() {
    }

    public Employee(Long id, String firstName, String lastName,
            String designation, String phone) {
        super();
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.designation = designation;
        this.phone = phone;
    }

    public Long getId() {
        return id;
    }
```

```
    public void setId(Long id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getDesignation() {
        return designation;
    }

    public void setDesignation(String designation) {
        this.designation = designation;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }
}
```

Теперь вернемся к коду сервлета. Замените код в файле сервлета на следующий:

```
package com.mycompany.servletapplab12;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import javax.servlet.ServletException;
```

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/EmployeeServlet")
public class EmployeeServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public EmployeeServlet() {
        super();
    }

    @Override
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        try {
            response.setContentType("text/html;charset=UTF-8");
            // Получение из http-запроса значения параметра
lasname
            String lastname = request.getParameter("lastname");
            // Коллекция для хранения найденных сотрудников
            ArrayList<Employee> employees = new ArrayList<>();
            // Выполнение SQL-запроса
            // Получение соединения с БД
            try ( Connection con = DriverManager.getConnection(
                    "jdbc:derby://localhost:1527/DerbyDB")) {
                // Выполнение SQL-запроса
                ResultSet rs = con.createStatement().executeQuery(
                        "Select id, first_name, last_name,
designation, phone "
                        + "From employee " + "Where last_name like
'"
                        + lastname + "'");
                // Перечисление результатов запроса
                while (rs.next()) {
                    // По каждой записи выборки формируется
                    // объект класса Employee.
                    // Значения свойств заполяются из полей записи
                    Employee emp = new Employee(
                            rs.getLong(1),
                            rs.getString(2),
                            rs.getString(3),
                            rs.getString(4),
                            rs.getString(5));
                    // Добавление созданного объекта в коллекцию
                    employees.add(emp);
                }
            }
            // Выводим информацию о найденных сотрудниках
```

```
            PrintWriter out = response.getWriter();
            out.println("Найденные сотрудники<br>");
            for (Employee emp : employees) {
                out.print(emp.getFirstName() + " "
                        + emp.getLastName() + " "
                        + emp.getDesignation() + " "
                        + emp.getPhone() + "<br>");
            }
        } catch (IOException | SQLException ex) {
            throw new ServletException(ex);
        }
    }

    @Override
    protected void doPost(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
    }
}
```

Это простейший сервлет, который переопределяет метод `doGet()`. Клиент запрашивает у сервлета записи из БД, для которых параметр «lastname» равен переданному в URL. Сервлет подключается к БД и извлекает нужные записи. В качестве ответа сервлет отправляет простую статическую HTML страницу, которая и содержит найденные записи.

Последнее, что нужно сделать, это создать страницу index.html, которая будет открываться при запуске веб-приложения. Ее расположение показано на рисунке 48.

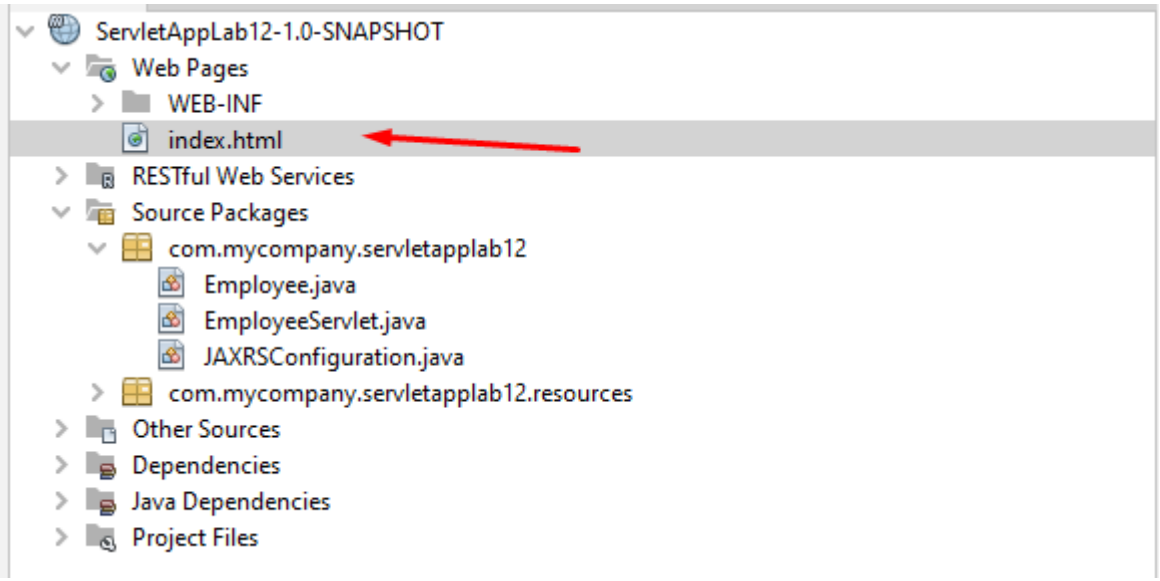


Рисунок 48 – Расположение index.html в проекте

Замените содержимое данного файла на следующее:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Start Page</title>
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
    </head>
    <body>
        <h1>Hello World!</h1>
        <p><a
href="http://localhost:8080/ServletAppLab12/EmployeeServlet?lastna
me=Ivanov">Servlet Employee Test</a></p>
    </body>
</html>
```

Веб-приложение готово. Перед запуском перейдем во вкладку «Tools» > «Options» (рисунок 49).

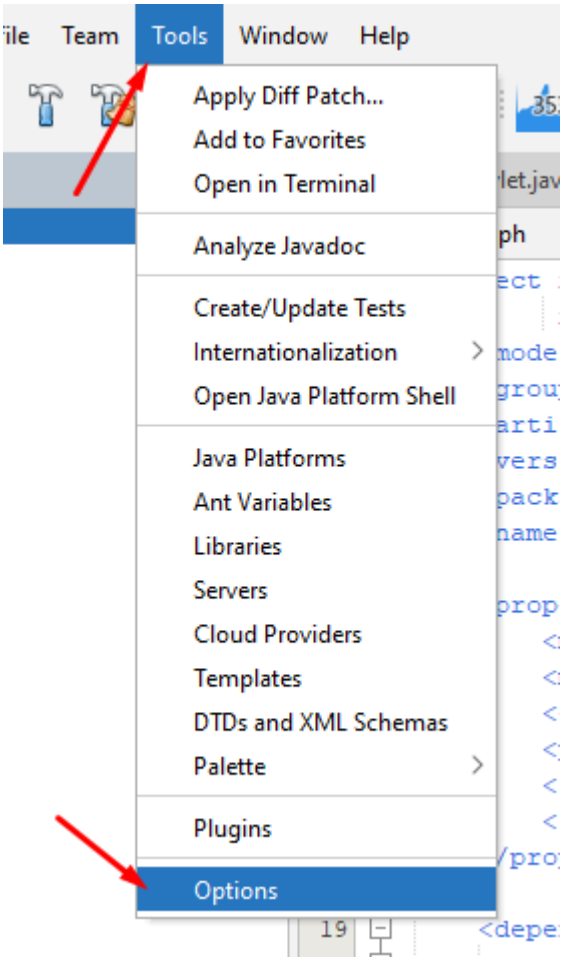


Рисунок 49 – Открываем настройки

В открывшемся окне переходим во вкладку «Java» > «Maven». Там выбираем в качестве «Default JDK» установленную ранее платформу (рисунок 50). Это сделано для того, чтобы не возникло ошибок при сборке проекта сборщиком проектов Maven.

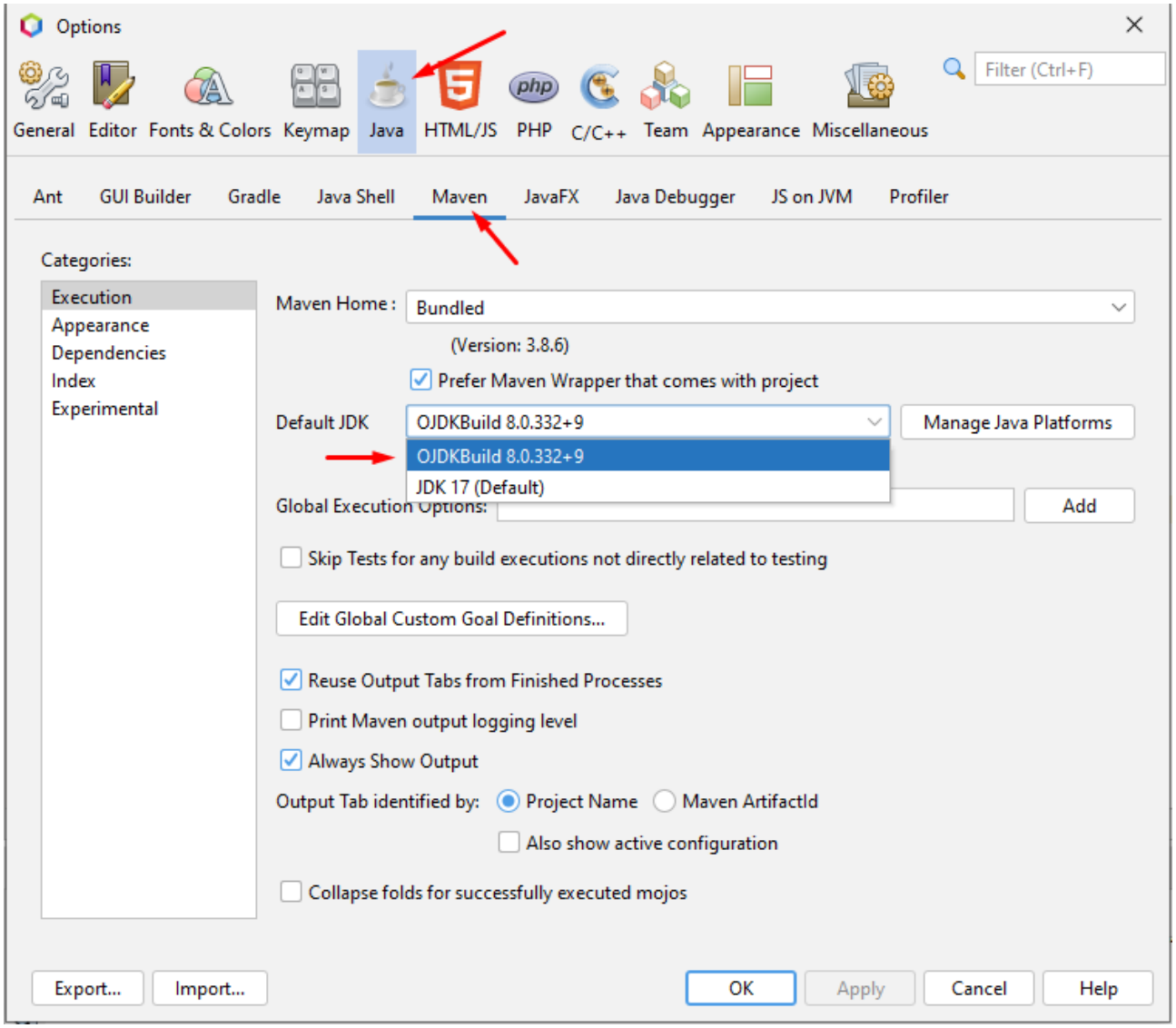


Рисунок 50 – Настройки Maven

Необходимо запустить приложенин с помощью опции «Run» (рисунок 51). Первый запуск приложения может быть долгим, так как Maven будет обновлять свои внутренние индексы. Индексацию можно отключить, но не рекомендуется (если все же требуется, то в окне на рисунке 50 слева нужно выбрать Index, затем установить там настройку в значение «Never»).

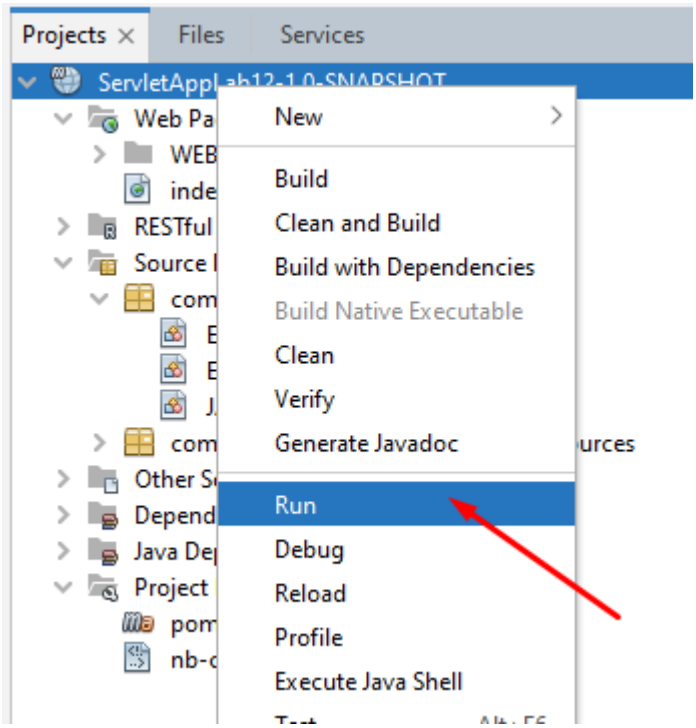


Рисунок 51 – Запуск веб-приложения

Откроется главная страница. На ней будет ссылка, которая вызовет выполнение разработанного сервлета (рисунок 52).

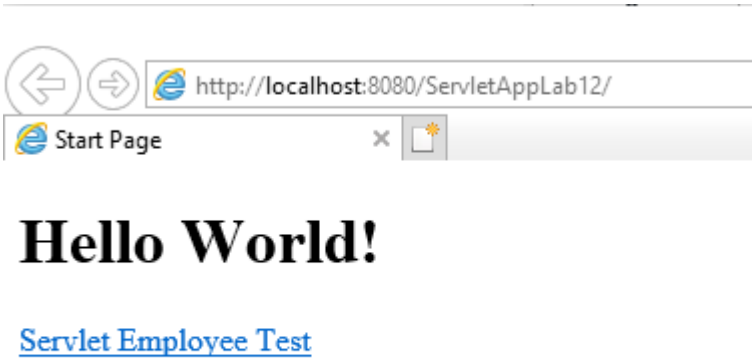


Рисунок 52 – Главная страница приложения

Нажимаем на гиперссылку «Servlet Employee Test». Происходит переход на адрес:

[http://localhost:8080/ServletAppLab12/EmployeeServlet?lastname=Ivanov]

Страница по данному адресу будет содержать записи, у которых в БД параметр «lastname» равен «Ivanov». Можно поменять в адресной строке

параметр, например, на «Petrov» и нажать Enter. Сервлет отправит в ответ соответствующую страницу. Данные страницы показаны на рисунках 53 и 54.

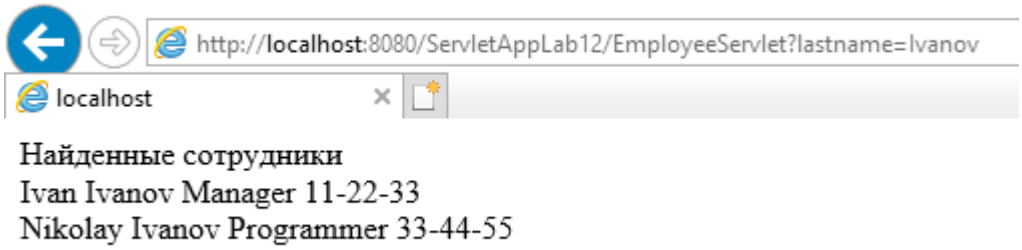


Рисунок 53 – Страница, сгенерированная сервлетом, параметр lastname=Ivanov

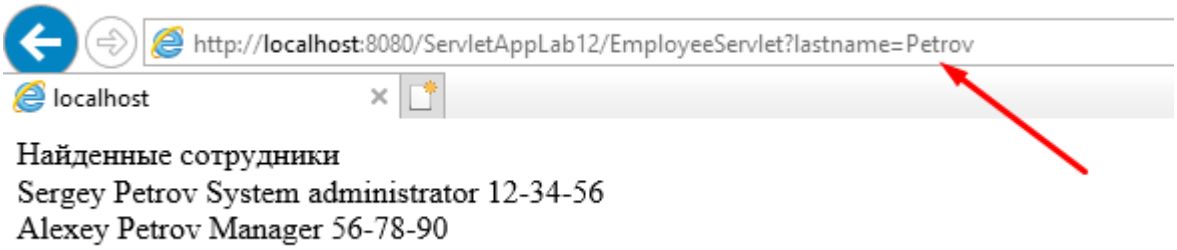


Рисунок 54 – Страница, сгенерированная сервлетом, параметр lastname=Petrov

5 Упражнения по теме сервлетов в Java:

Разработать сервлет в соответствии с рассмотренным примером. Изучить все основные классы для разработки данного вида сетевых приложений, алгоритмы сервлета.

Лабораторная работа № 13. Разработка сетевых программ с использованием Servlet API и событий жизненного цикла

1 Цель работы

Изучить классы и методы, которые позволяют разрабатывать приложения, использующие Servlet API и события жизненного цикла.

2 Порядок выполнения работы

Выполнить задание, указанное в конце методических указаний по данной лабораторной работе, составить отчет.

3 Содержание отчета

— наименование и цель работы;

— задание на лабораторную работу согласно варианту;

— схема алгоритма, текст программы на алгоритмическом языке;

— результаты работы программы.

4 Теоретическая часть

Данная работа посвящена разработке приложений с использованием Servlet API и событий жизненного цикла.

В лабораторной работе будут рассмотрены следующие вопросы:

— Интерфейсы в Java Servlet API, которые описывают запрос, ответ и контекст сервлета.

— Слушатели запросов к сервлету.

— Слушатели контекста сервлета.

— Создание класса-слушателя.

— Просмотр лог-файла веб-сервера GlassFish.

В конце предложены упражнения для закрепления материала.

Классы и интерфейсы пакета `javax.servlet` обеспечивают взаимодействие между сервлетом и клиентом. При создании сервлета необходимо реализовать интерфейс `Servlet` непосредственно или расширением класса, который реализует этот интерфейс. Интерфейс `Servlet` определяет различные методы жизненного цикла. Другими широко используемыми интерфейсами пакета `javax.servlet` являются:

— интерфейс `ServletRequest`;

— интерфейс `ServletResponse`;

— интерфейс `ServletContext`.

Интерфейс `ServletRequest` определяет несколько важных методов в дополнение к методам, представленным ниже:

`public String getProtocol()`, который возвращает название протокола и версии протокола, использованного для отправки запроса.

`public BufferedReader getReader()`, который возвращает тело объекта запроса в символьной форме.

`public String getScheme()`, который возвращает тип протокола, используемого для выполнения запроса: HTTP, FTP или HTTPS.

Интерфейс `ServletRequest` содержит методы для обработки клиентских запросов к сервлету. При вызове сервлета, веб-контейнер передает объекты, которые реализуют интерфейсы `ServletRequest` и `ServletResponse`, методу `service()` сервлета. Таблица 9 показывает различные методы интерфейса ServletRequest.

Т а б л и ц а 9 — Методы интерфейса `ServletRequest`

Метод	Описание
`public String getParameter(String paramName)`	Возвращает объект `String`, содержащий значение заданного параметра запроса.
`public String[] getParameterValues(String paramName)`	Возвращает массив объектов `String`, содержащий все значения параметра запроса.

`public Enumeration getParameterNames()`	Возвращает `Enumeration`, содержащий все имена параметров в виде объектов `String`, которые содержит запрос к сервлету.
`public String getRemoteHost()`	Возвращает `String`, содержащий точное имя компьютера, с которого был отправлен запрос.
`public String getRemoteAddr()`	Возвращает `String`, содержащий IP-адрес компьютера, с которого был отправлен запрос.

Интерфейс `ServletResponse` содержит методы, которые позволяют сервлету реагировать на клиентские запросы. Сервлет может отправить ответ в виде символьных или двоичных данных. Поток `PrintWriter` может быть использован для передачи символьных данных в ответе сервлета, а поток `ServletOutputStream` – для передачи двоичных данных. Таблица 10 показывает различные методы интерфейса `ServletResponse`.

Т а б л и ц а 10 — Методы интерфейса `ServletResponse`

Метод	Описание
`public ServletOutputStream getOutputStream() throws IOException`	Возвращает объект класса `ServletOutputStream`, который представляет выходной поток для передачи двоичных данных в ответе.
`public PrintWriter getWriter() throws IOException`	Возвращает объект класса `PrintWriter`, который сервлет использует для передачи символьных данных в ответе.
`public void setContentType(String type)`	Устанавливает тип Multipurpose Internet Mail Extensions (MIME) для ответа сервлета. Не-которые из MIME-типов: text/plain, im-age/jpeg и text/html .

Интерфейс `ServletContext` предоставляет информацию сервлетам о среде, в которой они выполняются. Контекст также называют контекстом

сервлета или веб-контекстом, и он создается веб-контейнером как объект интерфейса `ServletContext`. Этот объект представляет контекст, внутри которого выполняется веб-приложение. Веб-контейнер создает объект `ServletContext` для каждого развернутого веб-приложения. Объект `ServletContext` можно использовать для определения пути к другим файлам веб-приложения, для доступа к другим сервлетам веб-приложения и записи сообщений в журнал сервера приложений. Также объект `ServletContext` можно использовать для установки атрибутов, к которым другие сервлеты приложения могут получать доступ. Таблица 11 показывает методы интерфейса `ServletContext`.

Т а б л и ц а 11 — Методы интерфейса `ServletContext`

Метод	Описание
`public void setAttribute(String attname, Object value)`	Связывает объект с именем и сохраняет пару имя/значение как атрибут объекта `ServletContext`. Если атрибут уже существует, то этот метод замещает существующий атрибут.
`public Object getAttribute(String attrname)`	Возвращает объект, хранимый в объекте `ServletContext` с именем, переданным как параметр.
`public Enumeration getAttributeNames()`	Возвращает `Enumeration` объектов `String`, который содержит имена всех атрибутов контекста.
`public String getInitParameter(String pname)`	Возвращает значение параметра инициализации с именем, переданным как параметр.
`public Enumeration getInitParameterNames()`	Возвращает `Enumeration` объектов, который содержит имена всех атрибутов, доступных в этом запросе.

`public int getMajorVersion()`	Возвращает целое значение, определяющее номер версии Servlet API, которую поддерживает веб-контейнер. Если ваш веб-контейнер поддерживает Servlet API версии 2.4, этот метод вернет 2.
`public int getMinorVersion()`	Возвращает целое значение, определяющее номер подверсии Servlet API, которую поддерживает веб-контейнер. Если веб-контейнер поддерживает Servlet API версии 2.4, этот метод возвращает 4.

Для использования объекта `ServletContext` необходимо получить объект `ServletContext` в методе `init()` сервлета. Метод `getServletContext()` интерфейса `ServletConfig` позволяет получить объект `ServletContext`.

Пакет `javax.servlet.http` является расширением пакета `javax.servlet` и классы и интерфейсы этого пакета обрабатывают сервлеты, которые работают, используя протокол HTTP. Эти сервлеты также называются HTTP-сервлетами и для их создания необходимо расширить класс `HttpServlet`. Ниже представлены наиболее часто используемые интерфейсы пакета `javax.servlet.http`:

— интерфейс `HttpServletRequest`;

— интерфейс `HttpServletResponse`;

— интерфейс `HttpSession`.

Интерфейс `HttpServletRequest` расширяет интерфейс `ServletRequest` для представления информации из запроса, отправленного клиентом HTTP. Он включает поддержку получения параметров запроса и доступ к информации из заголовка HTTP-запроса.

HTTP-запросы имеют несколько сопутствующих заголовков, предоставляющих дополнительную информацию о клиенте, например,

название и версия браузера, отправляющего запрос и другие. Ниже представлены некоторые полезные заголовки HTTP-запроса:

Accept: Определяет MIME-тип, который клиент предпочитает использовать.

Accept-Language: Определяет язык, на котором клиент предпочитает получать запрос.

User-Agent: Определяет название и версию браузера, отправившего запрос.

В таблице 12 описаны некоторые методы интерфейса `HttpServletRequest`.

Т а б л и ц а 12 — Методы интерфейса `HttpServletRequest`

Метод	Описание
`public String getHeader(String fieldname)`	Возвращает значение поля заголовка запроса, такие как Cache-Control и Accept-Language, указанные в параметре.
`public Enumeration getHeaders(String sname)`	Возвращает все значения, связанные с конкретным заголовком запроса в виде объекта Enumeration
`public Enumeration getHeaderNames()`	Возвращает имена всех заголовков запроса, к которым сервлет может получить доступ, в виде объекта Enumeration

Интерфейс `HttpServletResponse` расширяет интерфейс `ServletResponse` и предоставляет методы для обработки ответов, кодов статуса и заголовков ответов для сервлетов, которые взаимодействуют с помощью HTTP. Таблица 13 описывает некоторые методы интерфейса `HttpServletResponse`.

Т а б л и ц а 13 — Методы интерфейса `HttpServletResponse`

Метод	Описание
`void addHeader(String hname, String hvalue)`	Добавляет заголовок `hname` со значением `hvalue`. Этот метод добавляет новый

	заголовок, если заголовок уже существует.
`void addIntHeader(String hname, int hvalue)`	Добавляет заголовок `hname` со значением `hvalue`.
`void addDateHeader(String hname, long datev)`	Добавляет заголовок hname со значением, равным `datev`. Значение `datev` должно быть в миллисекундах, которые прошли с полуночи 1 января 1970.
`boolean containsHeader(String hname)`	Возвращает `true`, если заголовок hname уже установлен с конкретным значением, и `false`, в противном случае.
`void sendRedirect(String url)`	Перенаправляет запрос указанному URL.

Протокол HTTP - это протокол без сохранения состояния, поскольку в протоколе нет ничего, что требует от браузера идентифицировать себя при каждом запросе, а также отсутствует постоянно установленное соединение между браузером и веб-сервером, которое сохранялось бы от страницы к странице. Когда пользователь посещает веб-страницу, браузер посылает HTTP-запрос серверу, который в свою очередь возвращает HTTP-ответ. Этим и ограничивается взаимодействие, и это представляет собой завершённую HTTP-транзакцию.

Интерфейс `HttpSession` обеспечивает методы для сохранения состояния конечного пользователя в веб-приложении на время сессии. Объект интерфейса `HttpSession` предоставляет средства для отслеживания и управления сессией конечного пользователя. Таблица 14 представляет несколько методов интерфейса `HttpSession`.

Т а б л и ц а 14 — Методы интерфейса `HttpSession`

Метод	Описание
`public void setAttribute(String name, Object value)`	Связывает объект с именем и сохраняет пару name/value как атрибут объекта `HttpSession`. Если атрибут уже существует, то замещается

	существующий атрибут.
`public Object getAttribute(String name)`	Извлекает объект `String`, указанный в параметре, из объекта сеанса. Если не найден объект для указанного атрибута, то метод `getAttribute()` возвращает `null`.
`public Enumeration getAttributeNames()`	Возвращает `Enumeration`, содержащий имена всех объектов, которые связаны с атрибутами объекта сеанса.

Во время жизненного цикла сервлета возникают различные события, связанные с контекстом сервлета, созданием сеанса и добавлением атрибутов в контекст сервлета. Веб-контейнер уведомляет класс-слушатель, когда возникает событие в течение жизненного цикла сервлета, и для того, чтобы получать уведомление о событии, классу-слушателю необходимо расширить интерфейс-слушатель Servlet API.

Ниже представлены различные события, которые генерируются во время жизненного цикла сервлета:

— события запроса к сервлету;

— события контекста сервлета;

— события сессии HTTP.

Следующие два интерфейса представляют события запроса сервлета, которые имеют отношение к изменениям в объектах запроса, связанных с веб-приложением:

— `javax.servlet.ServletRequestEvent`

— `javax.servlet.ServletRequestAttributeEvent`

Веб-контейнер создает объект `ServletRequestEvent` при:

— инициализации объекта запроса, когда запрос появляется;

— удалении объекта запроса, когда запрос не требуется.

Веб-контейнер создает объект `ServletRequestAttributeEvent`, когда происходит какое-либо изменение в атрибутах запроса к сервлету. Веб-контейнер создает объект `ServletRequestAttributeEvent` во время:

— добавления атрибута к объекту запроса к сервлету;

— удаления атрибута из объекта запроса к сервлету;

— замены атрибута в объекте запроса к сервлету другим атрибутом с тем же самым именем.

События, которые связаны с изменениями в контексте веб-приложения, известны как события контекста сервлета. Следующие два интерфейса представляют события контекста сервлета:

— `javax.servlet.ServletContextEvent`

— `javax.servlet.ServletContextAttributeEvent`

Веб-контейнер создает объект `ServletContextEvent` во время:

— создания объекта `ServletContext` при фазе инициализации жизненного цикла сервлета;

— удаления объекта `ServletContext`.

Веб-контейнер генерирует объект `ServletContextAttributeEvent`, когда происходит какое-либо изменение в атрибуте контекста сервлета веб-приложения, в частности, при:

— добавлении атрибута к объекту контекста сервлета;

— удалении атрибута в объекте контекста сервлета;

— замене атрибута в объекте контекста сервлета другим атрибутом с таким же именем.

События сессии HTTP связаны с изменениями в объекте сессии сервлета. Следующие интерфейсы представляют события сессии сервлета:

— `javax.servlet.http.HttpSessionEvent`;

— `javax.servlet.http.HttpSessionAttributeEvent`;

— `javax.servlet.http.HttpSessionActivationEvent`;

— `javax.servlet.http.HttpSessionBindingEvent`.

Веб-контейнер создает объект `HttpSessionEvent` при:

— создании новой сессии;

— недействительности сессии;

— завершении срока сессии.

Веб-контейнер генерирует объект `HttpSessionAttributeEvent`, когда происходит какое-либо изменение в объекте атрибута сессии, в частности, при:

— добавлении атрибута к объекту сессии;

— удалении атрибута из объекта сессии;

— замене атрибута в объекте сессии.

Веб-контейнер генерирует объект `HttpSessionBindingEvent`, когда объект сервлета связывается с сессией или разрывает связь сессии. Связывание объекта с сессией означает, что объект соотносится с сессией, а разрыв связи объекта сервлета означает, что объект не соотносится с сессией. Веб-контейнер генерирует объект `HttpSessionActivationEvent`, когда сеанс активизируется или деактивируется.

Для успешной работы управления событиями жизненного цикла сервлета необходимо реализовать интерфейс `ServletContextListener` в классе, и после этого сервер приложений JavaEE идентифицирует класс как класс-слушатель. Классы, которые получают уведомления о событиях жизненного цикла сервлета, известны как слушатели событий. Эти классы-слушатели реализуют один или более интерфейсов слушателей событий сервлета, которые определены в Servlet API. Классы-слушатели могут быть логически разделены на следующие категории:

— слушатели запросов к сервлету;

— слушатели контекста сервлета;

— слушатели сеанса http.

Слушатели запросов к сервлету – это классы, которые слушают и обрабатывают события запросов к сервлету. Слушатели запросов к сервлету могут реализовывать следующие интерфейсы для получения уведомления о событиях запроса:

— `javax.servlet.ServletRequestListener`;

— `javax.servlet.ServletRequestAttributeListener`.

Интерфейс `ServletRequestListener` позволяет классу-слушателю получать уведомления от веб-контейнера об изменениях в объекте запроса сервлета. Необходимо реализовать интерфейс `ServletRequestListener` в классе-слушателе для получения уведомлений о событиях запроса. Ниже представлены методы интерфейса `ServletRequestListener`:

`void requestInitialized(ServletRequestEvent e)`: уведомляет класс-слушатель об инициализации запроса к сервлету, связанного с веб-приложением.

`void requestDestroyed(ServletRequestEvent e)`: уведомляет сервлет об удалении запроса к сервлету, связанному с веб-приложением.

Интерфейс `ServletRequestAttributeListener` позволяет классу-слушателю получать уведомления от веб-контейнера об изменениях в атрибутах запроса. Ниже представлены методы интерфейса `ServletRequestAttributeListener`, которые можно использовать в классе-слушателе:

`void attributeAdded (ServletRequestAttributeEvent srae)`: информирует класс-слушатель о добавлении атрибута запроса.

`void attributeRemoved (ServletRequestAttributeEvent srae)`: информирует класс-слушатель об удалении атрибута запроса.

`void attributeReplaced (ServletRequestAttributeEvent srae)`: информирует класс-слушатель о замене существующего атрибута запроса новым атрибутом запроса.

Слушатели контекста сервлета – это классы-слушатели, которые обрабатывают события контекста сервлета. Слушатели контекста сервлета могут реализовывать следующие интерфейсы для получения уведомлений об изменениях в контексте сервлета:

— `javax.servlet.ServletContextListener`;

— `javax.servlet.ServletContextAttributeListener`.

Интерфейс `ServletContextListener` позволяет классу-слушателю получать уведомления от веб-контейнера об изменениях в контексте сервлета веб-приложения. Ниже представлены методы интерфейса `ServletContextListener`:

`void contextInitialized(ServletContextEvent ce)`: уведомляет класс-слушатель об инициализации контекста сервлета веб-приложения.

`void contextDestroyed(ServletContextEvent ce)`: уведомляет класс-слушатель об удалении контекста сервлета веб-приложения.

Интерфейс `ServletContextAttributeListener` позволяет классу-слушателю получать уведомления от веб-контейнера об изменениях, происходящих в атрибутах контекста сервлета. Ниже представлены методы интерфейса `ServletContextAttributeListener`:

`void attributeAdded (ServletContextAttributeEvent cae)`: уведомляет класс-слушатель о добавлении атрибута в контекст сервлета.

`public void attributeRemoved (ServletContextAttributeEvent cae)`: уведомляет класс-слушатель об удалении атрибута из контекста сервлета.

`public void attributeReplaced (ServletContextAttributeEvent cae)`: уведомляет класс-слушатель о замене существующего атрибута контекста сервлета новым атрибутом.

Перейдем к разработке приложения-примера. Постановка задачи: разработать приложение, которое будет записывать в журнал время, когда объекты запроса и контекста инициализируются, и когда добавляется атрибут к объекту контекста. Кроме того, приложение должно также записывать время, когда атрибут удаляется из объекта контекста и объекты запроса и контекста уничтожаются.

Создайте проект, аналогично тому, как это было сделано в лабораторной работе 12, за тем исключением, что базу данных создавать и подключать не нужно – в этой работе она не используется.

Создайте в проекте два файла – класс сервлета и класс слушателя сервлета. Создание сервлетов, опять же, было продемонстрировано в

предыдущей работе. Назовите сервлет ServletEvents. Вставьте в файл сервлета следующий код:

```
package com.mycompany.servletapplab13;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/ServletEvents")
public class ServletEvents extends HttpServlet {

    private static final long serialVersionUID = 1L;
    ServletContext ctx;
    PrintWriter pw;

    @Override
    public void init(ServletConfig cfig) {
        /*Получение объекта ServletContext*/
        ctx = cfig.getServletContext();
    }

    public ServletEvents() {
        super();
    }

    @Override
    public void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        /*Установка атрибута контекста*/
        ctx.setAttribute("URL", "jdbc:odbc:EmployeesDB");
        /*Получение объекта PrintWriter*/
        pw = response.getWriter();
        /*Отправка ответа, что атрибут URL установлен*/
        response.setContentType("text/html");
        /*Печать времени добавления атрибута к объекту контекста.
*/
        pw.println("<B>The JDBC URL has been set as a context
attribute at "
                + new Date() + "</B></BR>");
        /*Удаление атрибута из объекта контекста. */
        ctx.removeAttribute("URL");
```

```
        /*Печать времени удаления атрибута из объекта контекста.
*/
        pw.println("<B>The JDBC URL has been removed from context
at " + new Date() + "</B>");
    }
}
```

Для создания класса-слушателя ServletEventListener создайте обычный класс и вставьте в него следующий код:

```
package com.mycompany.servletapplab13;

import java.util.Date;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
import javax.servlet.annotation.WebListener;

@WebListener
public class ServletEventListener implements
ServletContextListener,
        ServletContextAttributeListener, ServletRequestListener {

    ServletContext contx;
    String name;
    String value;

    public ServletEventListener() {
    }

    @Override
    public void requestDestroyed(ServletRequestEvent sre) {
        /* Запись в журнал времени удаления запроса. */
        contx.log("A request has been destroyed at: " + new
Date());
    }

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        contx = sce.getServletContext();
        /* Запись в журнал времени инициализации контекста. */
        contx.log("Context has been initialized at " + new
Date());
    }

    @Override
```

```
    public void attributeAdded(ServletContextAttributeEvent event)
{
        /* Получение имени и значения атрибута. */
        name = event.getName();
        value = (String) event.getValue();
        /* Запись в журнал времени добавления атрибута в объект
контекста. */
        contx.log("An attribute with name: " + name + " and value:
" + value
                + " has been added to the context at: " + new
Date());
    }

    @Override
    public void attributeReplaced(ServletContextAttributeEvent
event) {
        /* Запись в журнал времени замены атрибута. */
        contx.log("Attribute with name: " + name + " and value: "
+ value
                + " has been replaced context at: " + new Date());
    }

    @Override
    public void attributeRemoved(ServletContextAttributeEvent
event) {
        /* Запись в журнал времени удаления атрибута из объекта
контекста. */
        contx.log("Attribute with name: " + name + " and value: "
+ value
                + " has been removed from the context at: " + new
Date());
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        /* Запись в журнал времени удаления контекста. */
        contx.log("Context has been destroyed at " + new Date());
    }

    @Override
    public void requestInitialized(ServletRequestEvent sre) {
        /* Запись в журнал времени инициализации запроса. */
        contx.log("A request has been initialized at: " + new
Date());
    }
}
```

В предложенном коде определяются все методы трех интерфейсов `ServletContextListener`, `ServletContextAttributeListener` и `ServletRequestListener`. Внутри каждого метода записывается время

появления события в файл server.log. О том, что данное приложение представляет собой слушатель свидетельствует аннотация `@WebListener` перед заголовком класса, которая значительно упрощает развертывание класса.

Каждая строчка кода прокомментирована, поэтому проблем с пониманием кода возникнуть не должно.

Файла index.html оставьте с содержимым по умолчанию.

Запустите приложение. Откроется главная страница приложения, нам нужно перейти по адресу

[http://localhost:8080/ServletAppLab13/ServletEvents]

«ServletAppLab13» - название проекта. Если ваш проект называется по-другому, не забудьте об этом.

Рисунок 55 – Страница, сгенерированная сервлетом

На рисунке 55 показана страница, сгенерированная сервлетом. Как видно из содержимого страницы, атрибут был сначала добавлен в контекст, а затем удален.

Чтобы увидеть результаты работы слушателя событий сервлета, необходимо открыть лог файл сервера. Для этого нужно перейти в директорию, в которую в 12 лабораторной работе был установлен GlassFish Server. А далее – Директория установки\glassfish\domains\domain1\logs. В папке logs будет файл server.log. Его можно открыть любым текстовым редактором.

В логе можно найти следующие строки, которые как раз и были записаны классом-слушателем. Записи приведены на рисунке 56.

```
6484 [2023-01-23T15:44:33.655+0300] [glassfish 5.0] [INFO] [] [javax.enterprise.web] [tid:
     _ThreadID=116 _ThreadName=http-listener-1(4)] [timeMillis: 1674477873655]
     [levelValue: 800] [[
6485   WebModule[/ServletAppLab13] ServletContext.log():A request has been initialized at:
       Mon Jan 23 15:44:33 MSK 2023]]
6486
6487 [2023-01-23T15:44:33.657+0300] [glassfish 5.0] [INFO] [] [javax.enterprise.web] [tid:
     _ThreadID=116 _ThreadName=http-listener-1(4)] [timeMillis: 1674477873657]
     [levelValue: 800] [[
6488   WebModule[/ServletAppLab13] ServletContext.log():A request has been destroyed at:
       Mon Jan 23 15:44:33 MSK 2023]]
6489
6490 [2023-01-23T15:44:44.382+0300] [glassfish 5.0] [INFO] [] [javax.enterprise.web] [tid:
     _ThreadID=113 _ThreadName=http-listener-1(1)] [timeMillis: 1674477884382]
     [levelValue: 800] [[
6491   WebModule[/ServletAppLab13] ServletContext.log():A request has been initialized at:
       Mon Jan 23 15:44:44 MSK 2023]]
6492
6493 [2023-01-23T15:44:44.385+0300] [glassfish 5.0] [INFO] [] [javax.enterprise.web] [tid:
     _ThreadID=113 _ThreadName=http-listener-1(1)] [timeMillis: 1674477884385]
     [levelValue: 800] [[
6494   WebModule[/ServletAppLab13] ServletContext.log():An attribute with name: URL and
       value: jdbc:odbc:EmployeesDB has been added to the context at: Mon Jan 23 15:44:44
       MSK 2023]]
6495
6496 [2023-01-23T15:44:44.386+0300] [glassfish 5.0] [INFO] [] [javax.enterprise.web] [tid:
     _ThreadID=113 _ThreadName=http-listener-1(1)] [timeMillis: 1674477884386]
     [levelValue: 800] [[
6497   WebModule[/ServletAppLab13] ServletContext.log():Attribute with name: URL and
       value: jdbc:odbc:EmployeesDB has been removed from the context at: Mon Jan 23
       15:44:44 MSK 2023]]
6498
6499 [2023-01-23T15:44:44.386+0300] [glassfish 5.0] [INFO] [] [javax.enterprise.web] [tid:
     _ThreadID=113 _ThreadName=http-listener-1(1)] [timeMillis: 1674477884386]
     [levelValue: 800] [[
6500   WebModule[/ServletAppLab13] ServletContext.log():A request has been destroyed at:
       Mon Jan 23 15:44:44 MSK 2023]]
```

Рисунок 56 – Записи в логе, созданные классом-слушателем

5 Упражнения по теме сервлетов и событий жизненного цикла в Java:

Разработать сервлет и класс-слушатель в соответствии с рассмотренным примером. Изучить все основные классы для разработки данного вида сетевых приложений, алгоритмы сервлета. В отчете привести записи из лога сервера, сгенерированные классом-слушателем.