Содержание

[Критерии оценки 2](#_Toc440985517)

[Лабораторная работа №1 3](#_Toc440985518)

[Тема: Установка и настройка среды разработки NetBeans. Проект: создание, тестирование, отладка. Структура проекта. Пример. Компиляция и запуск готового приложения. Использование системы контроля версий Git. 3](#_Toc440985519)

[Лабораторная работа №2 6](#_Toc440985520)

[Тема: Программирование типовых алгоритмов для обработки числовых типов данных. Класс Math. Преобразование типов. 6](#_Toc440985521)

[Лабораторная работа №3 12](#_Toc440985522)

[Тема: Программирование типовых алгоритмов для обработки массивов. 12](#_Toc440985523)

[Лабораторная работа №4 22](#_Toc440985524)

[Тема: Классы и объекты в Java. Иерархия классов. Создание классов и объектов. Интерфейсы. Использование пакетов и jar-файлов. 22](#_Toc440985525)

[Лабораторная работа №5 31](#_Toc440985526)

[Тема: Исключительные ситуации. Обработка исключений. 31](#_Toc440985527)

[Лабораторная работа №6 34](#_Toc440985528)

[Тема: Библиотеки классов. Классы для обработки строк. Программирование типовых алгоритмов для обработки строк. Объекты класса String. Методы для работы со строками. 34](#_Toc440985529)

[Лабораторная работа №7 38](#_Toc440985530)

[Тема: Система ввода-вывода. Байтовые и символьные потоки. Консольный ввод с использованием объекта System.in. Консольный ввод с помощью класса Scanner. Использование диалогового окна. Работа с файлами. 38](#_Toc440985531)

[Лабораторная работа №8 43](#_Toc440985532)

[Тема: Декомпозиция простых алгоритмов. Коллективная разработка проекта. 43](#_Toc440985533)

[Контрольная работа №1 44](#_Toc440985534)

[Лабораторная работа №9 45](#_Toc440985535)

[Тема: Создание программ с графическим интерфейсом. Создание оконных приложений. События. Обработка событий фреймового окна, мыши и клавиатуры. 45](#_Toc440985536)

[Лабораторная работа №10 48](#_Toc440985537)

[Тема: Создание программ с графическим интерфейсом. Классы основных компонентов. 48](#_Toc440985538)

[Лабораторная работа №11 50](#_Toc440985539)

[Тема: Работа с сетью, многопоточные приложения. Классы для организации работы с сетями. 50](#_Toc440985540)

[Лабораторная работа №12 57](#_Toc440985541)

[Тема: Работа с базами данных. Доступ к базам данных из программ на языке Java. JDBC-драйверы. Регистрация драйверов. Основные приемы программирования JDBC. Выполнение запросов. Построение модели данных для JTable на основе результирующего набора данных. 57](#_Toc440985542)

[Лабораторная работа №13 62](#_Toc440985543)

[Тема: Функциональная и компонентная декомпозиция проекта. Коллективная разработка проекта. 62](#_Toc440985544)

[Контрольная работа №2 63](#_Toc440985545)

# Критерии оценки

|  |  |  |  |
| --- | --- | --- | --- |
| **Параметр** | **Оценка** | | |
| **«удовлетворительно»** | **«хорошо»** | **«отлично»** |
| **1. Сроки** | Нет требований | В текущем семестре | В текущем месяце |
| **2. Плагиат** | <40% | <30% | <20% |
| **3. Посещение** | ≥25% | ≥75% | 100% |
| **4. Тестирование** | Нет требований (типовой тест) | Все возможные сочетания входных параметров, согласно условию задачи | Все возможные сочетания входных параметров, согласно условию задачи. Описание исключительных ситуаций. |
| **5. Оформление программы** | Нет требований | Обязательная разметка кода и именование элементов программы согласно нотации (например, венгерской). | Обязательная разметка кода и именование элементов программы согласно нотации (например, венгерской). Наличие комментариев к коду программы (основным блокам и функциям). |
| **6. Требования к алгоритму** | Нет требований | Нет требований | Алгоритм проанализирован с точки зрения быстродействия. |
| **7. Требования к памяти** | Нет требований | Нет требований | Корректный выбор типов данных, исходя из условий задачи. Отсутствие заведомо избыточных переменных и структур. |
| **8. Требования к интерфейсу программы** | Нет требований | Интерфейс позволяет пользователю самостоятельно использовать программу. | Дружелюбный (общепринятый) интерфейс, с рациональным размещением элементов и использованием цветовой гаммы (например, калькулятор Windows, MSWord и т.д.) |
| **9. Ответы на вопросы по теме лабораторной** | Владеет материалом описанным в отчете по лабораторной и ориентируется в исходном коде | Свободно владеет теоретическим материалом по теме лабораторной работы и способен модифицировать код при небольших изменениях условий задачи | |

# Лабораторная работа №1

## Тема: Установка и настройка среды разработки NetBeans. Проект: создание, тестирование, отладка. Структура проекта. Пример. Компиляция и запуск готового приложения. Использование системы контроля версий Git.

**Цели работы:** научиться настраивать среду разработки NetBeans для создания, тестирования и отладки программ на языке Java.

**Постановка задачи**

Создать и настроить проект в среде разработки NetBeans для написания программ на языке Java. Написать первую программу на Java («Привет, мир!»). Установить программное обеспечение системы контроля версий (рекомендуется использование SourceTree), создать профиль на хостинге проектов GitHub или Bitbucket. Разместить первую лабораторную работу на хостинге.

**Используемый метод**

Рекомендации по настройке ПО системы контроля версий:

GIT and GitHub Source Tree

<http://www.bogotobogo.com/cplusplus/Git/Git_GitHub_Source_Tree_1_Commit_Push.php>

<https://github.com/GSoft-SharePoint/Dynamite/wiki/Getting-started-with-SourceTree,-Git-and-git-flow>

Система контроля версий — это система, записывающая изменения в файл или набор файлов в течение большого периода времени, так что вы сможете позже вернуться к определенной версии. Для контроля версий файлов в этой книге, в качестве примера, будет использоваться исходный код программного обеспечения, хотя на самом деле вы можете использовать контроль версий практически для любых типов файлов.

Основное отличие Git’а от любой другой СКВ (Subversion и друзья включительно), это подход Git’а к работе со своими данными. Концептуально, большинство других систем хранят информацию в виде списка изменений в файлах. Эти системы (CVS, Subversion, Perforce, Bazaar и т.д.) представляют информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени.

Git не хранит и не обрабатывает данные таким способом. Вместо этого, подход Git’а к хранению данных больше похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта в Git’е, система запоминает как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увелечения эффективности, если файлы не были изменены, Git не запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как, скажем, **поток снимков.**

Первая программа на языке JAVA:

*class HelloWorld {*

*public static void main(String args []) {*

*System.out.print("Hello World!");*

*}*

*}*

Для того, чтобы программировать на языке программирования Java, Вам необходимо установить Java Developers Kit - пакет для разработки Java-приложений (<http://java.sun.com/products/jdk>).

Язык Java требует, чтобы весь программный код был заключен внутри поименованных классов. Приведенную выше программу надо записать в файл HelloWorld.java. Для того, чтобы оттранслировать этот пример необходимо запустить транслятор Java - javac, указав в качестве параметра имя файла с исходным текстом:

*С: \> javac HelloWorld.Java*

Транслятор создаст файл HelloWorld.class с независимым от процессора байт-кодом нашего примера. Для того, чтобы исполнить полученный код, необходимо иметь среду выполнения языка Java, в которую надо загрузить новый класс для исполнения. Подчеркнем, что указывается имя класса, а не имя файла, в котором этот класс содержится.

*С: > java HelloWorld*

*Hello World!*

***Или, если Вы работаете в среде разработки NetBeans, то среда все сделает за вас.***

Теперь подробно разберем строки кода.

*class HelloWorld {*

В этой строке использовано зарезервированное слово class. Оно говорит транслятору, что мы собираемся описать новый класс. Полное описание класса располагается между открывающей фигурной скобкой в первой строке и парной ей закрывающей фигурной скобкой в строке 5. Фигурные скобки в Java используются точно так же, как в языках С и С++.

*public static void main (String args []) {*

**public**

Это - *модификатор доступа,* который позволяет программисту управлять видимостью любого метода и любой переменной. В данном случае модификатор доступа public означает, что метод main виден и доступен любому классу. Существуют еще 2 указателя уровня доступа - private и protected, с которыми мы более детально познакомимся позже.

**static**

С помощью этого слова объявляются методы и переменные класса, используемые для работы с классом в целом. Методы, в объявлении которых использовано ключевое слово static, могут непосредственно работать только с локальными и статическими переменными.

**void**

У вас нередко будет возникать потребность в методах, которые возвращают значение того или иного типа: например, int для целых значений, float - для вещественных или имя класса для типов данных, определенных программистом. В нашем случае нужно просто вывести на экран строку, а возвращать значение из метода main не требуется, т.е. тип void – это пустой тип данных.

**main**

Все существующие реализации Java-интерпретаторов, получив команду интерпретировать класс, начинают свою работу с вызова метода main. Java-транслятор может оттранслировать класс, в котором нет метода main. А вот Java-интерпретатор запускать классы без метода main не умеет.

Все параметры, которые нужно передать методу, указываются внутри пары круглых скобок в виде списка элементов, разделенных символами ";" (точка с запятой). Каждый элемент списка параметров состоит из разделенных пробелом типа и идентификатора. Даже если у метода нет параметров, после его имени все равно нужно поставить пару круглых скобок. В примере, который мы сейчас обсуждаем, у метода main только один параметр, правда довольно сложного типа.

Элемент **String args[]** объявляет параметр с именем args, который является массивом объектов - представителей класса String. Обратите внимание на квадратные скобки, стоящие после идентификатора args. Они говорят о том, что мы имеем дело с массивом, а не с одиночным элементом указанного типа. Мы вернемся к обсуждению массивов в будущем, а пока отметим, что тип String - это класс.

*System.out.print("Hello World!");*

В этой строке выполняется метод print объекта out. Объект out объявлен в классе OutputStream и статически инициализируется в классе System.

Закрывающей фигурной скобкой в строке 4 заканчивается объявление метода main, а такая же скобка в строке 5 завершает объявление класса HelloWorld.

# Лабораторная работа №2

## Тема: Программирование типовых алгоритмов для обработки числовых типов данных. Класс Math. Преобразование типов.

**Цели работы:** научиться создавать переменные и константы базовых типов данных; научиться преобразовывать переменные и результаты операций в другой тип данных; познакомиться с классом Math и его методами.

**Постановка задачи**

Создать переменные базовых типов, выполнить операции над ними (математические, логические и преобразование типов).

**Используемый метод**

**Идентификаторы**

Идентификаторы используются для именования классов, методов и переменных. В качестве идентификатора может использоваться любая последовательность строчных и прописных букв, цифр и символов \_ (подчеркивание) и $ (доллар). Идентификаторы не должны начинаться с цифры, чтобы транслятор не перепутал их с числовыми литеральными константами. Java - язык, чувствительный к регистру букв. Это означает, что, к примеру, Value и VALUE - различные идентификаторы.

**Константы**

Константы в Java задаются их литеральным представлением. Целые числа, числа с плавающей точкой, логические значения, символы и строки можно располагать в любом месте исходного кода.

**Операторы**

Оператор - это команда, выполняющая некоторое действие над одним или двумя аргументами и выдающая результат. Синтаксически операторы чаще всего размещаются между идентификаторами и литералами.

Примеры операторов языка Java:

|  |  |  |  |
| --- | --- | --- | --- |
| + | += | - | -= |
| \* | \*= | / | /= |
| | | |= | ^ | ^= |
| & | &= | % | %= |
| > | >= | < | <= |
| ! | != | ++ | -- |
| >> | >>= | << | <<= |
| >>> | >>>= | && | | | |
| == | = | ~ | ?: |

**Разделители**

Разделители влияют на внешний вид и функциональность программного кода.

|  |  |  |
| --- | --- | --- |
| **Символы** | **Название** | **Для чего применяются** |
| ( ) | круглые скобки | Выделяют списки параметров в объявлении и вызове метода, также используются для задания приоритета операций в выражениях, выделения выражений в операторах управления выполнением программы, и в операторах приведения типов. |
| { } | фигурные скобки | Содержат значения автоматически инициализируемых массивов, также используются для ограничения блока кода в классах, методах и локальных областях видимости. |
| [ ] | квадратные скобки | Используются в объявлениях массивов и при доступе к отдельным элементам массива. |
| ; | точка с запятой | Разделяет операторы. |
| , | запятая | Разделяет идентификаторы в объявлениях переменных, также используется для связи операторов в заголовке цикла for. |
| . | точка | Отделяет имена пакетов от имен подпакетов и классов, также используется для отделения имени переменной или метода от имени переменной. |

**Переменные**

Переменная - это основной элемент хранения информации в Java-программе. Переменная характеризуется комбинацией идентификатора, типа и области действия. В зависимости от того, где вы объявили переменную, она может быть локальной, например, для кода внутри цикла for, либо это может быть переменная экземпляра класса, доступная всем методам данного класса. Локальные области действия объявляются с помощью фигурных скобок.

**Объявление переменной**

Основная форма объявления переменной такова:

*тип идентификатор [ = значение] [, идентификатор [ = значение* *]*...*];*

*Тип -* это либо один из встроенных типов, то есть, byte, short, int, long, char, float, double, boolean, либо имя класса или интерфейса. Ниже приведено несколько примеров объявления переменных различных типов. Обратите внимание на то, что некоторые примеры включают в себя инициализацию начального значения. Переменные, для которых начальные значения не указаны, автоматически инициализируются нулем.

*int a, b, с; //Объявляет три целых переменных а, b, с.*

*int d = 3, e, f = 5; //Объявляет еще три целых переменных, инициализирует d и f.*

*byte z = 22; //Инициализирует z.*

*double pi = 3. 14159; //Объявляет число пи.*

*char x = 'x'; //Переменная х получает значение 'х'.*

В приведенном ниже примере создаются три переменные, соответствующие сторонам прямоугольного треугольника, а затем cпомощью теоремы Пифагора вычисляется длина гипотенузы, в данном случае числа 5, величина гипотенузы классического прямоугольного треугольника со сторонами 3, 4, 5.

*class Variables {*

*public static void main (String args []) {*

*double a = 3;*

*double b = 4;*

*double c;*

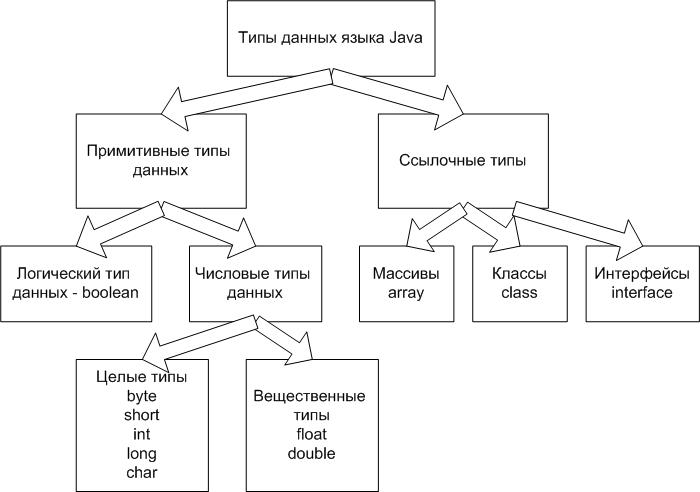
*с = Math.sqrt (a\* a + b\* b);*

*System.out.println ("c = "+ c);*

*}*

*}*

**Типы данных**



|  |  |  |
| --- | --- | --- |
| Имя | Разрядность | Диапазон |
| long | 64 | -9,223,372,036,854,775,808.. 9,223,372,036,854,775,807 |
| Int | 32 | -2,147,483,648.. 2,147,483,647 |
| Short | 16 | -32,768.. 32,767 |
| byte | 8 | -128.. 127 |
| char | 16 | 0.. 65,535 |

|  |  |  |
| --- | --- | --- |
| Имя | Разрядность | Диапазон |
| double | 64 | 1. 7е-308.. 1. 7е+ 308 |
| float | 32 | 3. 4е-038.. 3. 4е+ 038 |

**Логический тип данных**

*boolean b = true, bb = false, bool2; // логический тип данных*

Логические операции:

- логическое отрицание (!);

- логическое И (&);

- логическое ИЛИ (|);

- исключающее ИЛИ (^).

**Целые типы данных**

*byte b1 = 50, b2 = -99, bЗ;*

*short det = 0, ind = 1;*

*int i = -100, j = 100, k = 9999;*

*long big = 50, veryBig = 2147483648L;*

*char c1 = 'A', c2 = '?', newLine = '\n';*

Арифметические операции на целыми числами

1. Сложение (+)
2. Вычитание (-)
3. Умножение (\*)
4. Деление (/)
5. Взятие остатка от деления (%)
6. Инкремент (++)
7. Декремент (--)

**Приведение типов**

Ручное преобразование

*Тип\_1 a;*

*Тип\_2 b = (Тип\_2) a;*

Пример:

*int a = 100;*

*byte b = (byte) a;*

Автоматическое преобразование типов

*byte a = 40;*

*byte b = 50;*

*byte с = 100;*

*int d = a\* b / с;*

**Комментарий:**

// комментарий

или

/\* комментарий 1

\* комментарий 2

\* комментарий 3\*/

**Класс Math**

Класс Math содержит набор математических функций, часто оказывающихся не­обходимыми при решении разных задач.

Math.abs(n) - возвращает модуль числа n.

Math.round(n) - возвращает целое число, ближайшее к вещественному числу n (округляет n).

Math.ceil(n) - возвращает ближайшее к числу n справа число с нулевой дробной частью (например, Math.ceil(3.4) в результате вернёт 4.0).

Math.cos(n), Math.sin(n), Math.tan(n) - тригонометрические функции sin, cos и tg от аргумента n, указанного в радианах.

Math.acos(n), Math.asin(n), Math.atan(n) - обратные тригонометрические функции, возвращают угол в радианах.

Math.toDegrees(n) - возвращает градусную меру угла в n радианов.

Math.toRadians(n) - возвращает радианную меру угла в n градусов.

Math.sqrt(n) - возвращает квадратный корень из n.

Math.pow(n, b) - возвращает значение степенной функции, n в степени b, основание и показатель степени могут быть вещественными.

Math.log(n) - возвращает значение натурального логарифма числа n.

Math.log10(n) - возвращает значение десятичного логарифма числа n.

Все перечисленные функции принимают вещественные аргументы, а тип возвращаемого значения зависит от типа аргумента и от самой функции.

Кроме функций в рассматриваемом классе имеются две часто используемых константы:

Math.PI - число «пи», с точностью в 15 десятичных знаков.

Math.E - число Неппера (основание экспоненциальной функции), с точностью в 15 десятичных знаков.

Примеры использования:

*System.out.println(Math.abs(-2.33)); // выведет 2.33*

*System.out.println(Math.round(Math.PI)); // выведет 3*

*System.out.println(Math.round(9.5)); // выведет 10*

*System.out.println(Math.round(9.5-0.001)); // выведет 9*

*System.out.println(Math.ceil(9.4)); // выведет 10.0*

*double c = Math.sqrt(3\*3 + 4\*4);*

*System.out.println(c); // выведет гипотенузу треугольника с катетами 3 и 4*

*double s1 = Math.cos(Math.toRadians(60));*

*System.out.println(s1); // выведет косинус угла в 60 градусов*

**Задачи**

1. Дано: два целых числа A и B. Требуется: найти разность, сумму, отношение и произведение чисел A и B.
2. Дано: три целых числа A, B и C. Требуется: найти периметр и площадь треугольника с длинами сторон A, B и C.
3. Дано: дано вещественное число A и целое число B. Требуется: найти значения

А) А в степени В;

Б) целая часть А в степени В;

В) корень квадратный из целой части А.

1. Дано: три вещественных числа А, В и С. Требуется: рассчитать выражение
2. Дано: два целых числа А и В. Требуется: вывести целую часть и остаток от деления А на В.

# Лабораторная работа №3

## Тема: Программирование типовых алгоритмов для обработки массивов.

**Цели работы:** научиться использовать операторы выбора if и switch,циклы for, while и do-while, операторы перехода break, continue и метки, а также научиться объявлять, инициализировать и работать с массивами.

**Постановка задачи**

Используя условные операторы и циклы выполнить операции над массивами данных.

**Используемый метод**

**Условный оператор if()**

В Java существует две условных конструкции, которые позволяют выполнять разные операции в зависимости от некоторого условия. В первую очередь рассмотрим условный оператор if(). Синтаксис его вызова имеет в общем случае вид:

*if(условие){инструкции\_1}*

*else{инструкции\_2}*

Условный оператор if() выполняется в следующей последовательности. Сначала проверяется условие, указанное в круглых скобках после ключевого слова if. Если условие верное (значение соответствующего выражения равно true), выполняется блок инструкций, указанный сразу после оператора if(условие) (в данном случае это инструкции\_1). В противном случае, то есть если значение выражения в круглых скобках после ключевого слова if равно false, выполняется блок инструкций, указанных после ключевого слова else (в данном случае это инструкции\_2). После выполнения условного оператора управление передается следующей после него инструкции. Обращаем внимание читателя на несколько обстоятельств. Во-первых, если любой из двух блоков инструкций состоит всего из одной команды, фигурные скобки для соответствующего блока можно не использовать. Тем не менее, лишними фигурные скобки никогда не бывают, поскольку на быстродействии они не сказываются, а читабельность программы значительно улучшают. Во-вторых, ветвь else условного оператора не является обязательной. Синтаксис вызова такой упрощенной формы условного оператора if() имеет следующий вид:

*if(условие){инструкции}*

В этом случае сначала проверяется на истинность условие, указанное в скобках после ключевого слова if. Если условие истинно, выполняется следующий после условия if блок инструкций. Если условие ложно, указанные инструкции не выполняются. На практике нередко используются вложенные операторы if(). С точки зрения синтаксиса языка Java такая ситуация проста: в ветви else условного оператора указывается другой условный оператор и т.д. Синтаксическая конструкция имеет вид:

*if(условие\_1){инструкции\_1}*

*else if(условие\_2){инструкции\_2}*

*else if(условие\_3){инструкции\_3}*

*...*

*else if(условие\_N){инструкции\_N}*

*elase{инструкции}*

Пример использования оператора if:

*class UsingIf{*

*public static void main(String[] args){*

*int x=3,y=6,z;*

*// Условная инструкция:*

*if(x!=0){*

*z=y/x;*

*System.out.println("Значение z="+z);*

*}*

*else System.out.println("Деление на нуль!");*

*}*

*}*

**Условный оператор switch()**

Обычно к услугам условной инструкции switch() прибегают в случае, когда при проверке условия альтернатив больше, чем две. Эту инструкцию еще называют инструкцией выбора. Синтаксис вызова инструкции switch() следующий:

*switch(условие){*

*case значение\_1:*

*//команды\_1*

*break;*

*case значение\_2:*

*//команды\_2*

*break;*

*...*

*case значение\_N:*

*//команды\_N*

*break;*

*default:*

*//команды*

*}*

После ключевого слова switch в круглых скобках указывается переменная или выражение, значение которого проверяются (условие). Возможные значения, которые может принимать условие, перечисляются после ключевых слов case. Каждому значению соответствует свой блок case. Каждый блок case заканчивается инструкцией break. Последним блоком в инструкции switch() является блок команд, выполняемых по умолчанию. Блок выделяется инструкцией default(). Блок не является обязательным, и инструкцию break в конце этого блока размещать не нужно. Наконец, все блоки case и блок default, если он есть, заключаются в фигурные скобки. Именно эти фигурные скобки определяют тело инструкции switch().

Алгоритм выполнения инструкции switch() следующий. Сначала вычисляется выражение или значение переменной, указанной в качестве условия. Затем вычисленное значение последовательно сравнивается со значениями, указанными после инструкций case, пока не будет найдено совпадение или не встретится блок default (если блок default отсутствует, то пока не будет достигнут конец тела инструкции switch()). Если совпадение найдено, начинает выполняться программный код соответствующего блока case. Код выполняется до конца тела инструкции switch() или break(). Собственно, инструкции break в блоках case и нужны для того, чтобы остановить выполнение программного кода инструкции switch(). В противном случае продолжали бы выполняться следующие блоки case.

Выражение, которое указывается в качестве проверяемого условия, может возвращать в качестве значения целое число или символ. Значения, указываемые после инструкций case(), должны быть литералами или константами. Пример использования инструкции switch приведен ниже.

*class UsingSwitch{*

*public static void main(String[] args){*

*char s='П';*

*System.out.print("Фамилия пользователя: ");*

*// Инструкция выбора:*

*switch(s){*

*case 'И':*

*System.out.println("Иванов");*

*break;*

*case 'П':*

*System.out.println("Петров");*

*break;*

*case 'С':*

*System.out.println("Сидоров");*

*break;*

*default:*

*System.out.println("Не определена");*

*}*

*System.out.println("Программа завершила работу!");*

*}*

*}*

Пример использования оператора switch():

*class UsingSwitch2{*

*public static void main(String[] args){*

*char s='п';*

*System.out.print("Фамилия пользователя: ");*

*// Инструкция вывода:*

*switch(s){*

*case 'И':*

*case 'и':*

*System.out.println("Иванов");*

*break;*

*case 'П':*

*case 'п':*

*System.out.println("Петров");*

*break;*

*case 'С':*

*case 'с':*

*System.out.println("Сидоров");*

*break;*

*default:*

*System.out.println("Не определена");*

*}*

*System.out.println("Программа завершила работу!");*

*}*

*}*

**Цикл с параметром for()**

Для выполнения однотипных многократно повторяющихся действий используют инструкции цикла. В Java существует несколько инструкций цикла. Рассмотрим инструкцию цикла for(). Синтаксис вызова инструкции цикла for() следующий:

*for (инициализация; условие; итерация){*

*// тело цикла*

*}*

В круглых скобках после ключевого слова for указываются три группы, или блока, выражений. Блоки разделяются точкой с запятой. Первый блок обычно называется блоком инициализации. Как правило, в этом блоке размещается команда (или команды), которая перед выполнением цикла присваивает индексным переменным начальные значения. Второй блок - условие, при выполнении которого продолжается работа инструкции цикла. Третий блок содержит команды, которыми изменяются значения индексных переменных. Первый и третий блоки могут состоять из нескольких команд. Команды одного блока разделяются запятыми.

Далее, после ключевого слова for и круглых скобок в фигурных скобках идет блок команд, которые выполняются в рамках каждого цикла и фактически формируют тело инструкции цикла. Если тело инструкции цикла состоит из одной команды, фигурные скобки можно не ставить. Выполняется инструкция цикла по следующему алгоритму.

Начинается выполнение инструкции цикла с блока инициализации: последовательно выполняются все команды этого блока. Далее проверяется условие во втором блоке. Если оно истинно (значение true), выполняются команды тела инструкции цикла (команды в фигурных скобках). Далее выполняются команды третьего блока в круглых скобках и проверяется условие во втором блоке. Если условие истинно, выполняются команды основного тела инструкции цикла, команды блока изменения индексных переменных (третий блок), затем проверяется условие и т. д. Вся эта процедура продолжается до тех пор, пока при проверке условия его значение не становится равным false.

По завершении инструкции цикла выполняется следующая после него инструкция. Обращаем внимание читателя на то, что команды первого блока инициализации инструкции цикла выполняются только один раз на начальном этапе выполнения инструкции.

Пример использования цикла с параметром for():

*class UsingFor{*

*public static void main(String[] args){*

*// Индексная переменная:*

*int i;*

*// Переменная для вычисления суммы:*

*int sum=0;*

*// Инструкция цикла:*

*for(i=1;i<=100;i++){*

*sum+=i;}*

*System.out.println("Сумма чисел от 1 до 100: "+sum);}*

*}*

**Цикл с предусловием while()**

Для организации циклов кроме инструкции for() часто используется инструкция while() (или ее модификация - инструкция do-while(), которая рассматривается в следующем разделе). Далее приведен синтаксис вызова инструкции while():

*while(условие){*

*// команды цикла*

*}*

После ключевого слова while в круглых скобках указывается условие. В начале выполнения инструкции проверяется это условие. Если условие истинно, выполняются команды цикла - они заключаются в фигурные скобки. После этого снова проверяется условие и т.д.

От инструкции for() инструкция while() принципиально отличается тем, что инициализация индексной переменной, если такая имеется, выполняется до вызова инструкции, а команда изменения этой переменной размещается в теле цикла. Поэтому инструкция while() более гибкая в плане возможных вариантов ее использования. Все, что запрограммировано с помощью инструкции for(), может быть запрограммировано и с помощью инструкции while(). Ниже приведен пример программы для вычисления суммы нечетных натуральных чисел с использованием цикла while().

*class UsingWhile{*

*public static void main(String[] args){*

*int sum=0,i=1;*

*// Инструкция цикла:*

*while(i<=100){*

*sum+=i;*

*i+=2;*

*}*

*System.out.println("Сумма нечетных чисел от 1 до 100: "+sum);*

*}*

*}*

**Цикл с постусловием do-while()**

Инструкция do-while() является модификацией инструкции while(). Синтаксис ее вызова такой:

*do{*

*// команды цикла*

*}while(условие);*

Выполнение инструкции начинается с блока команд цикла, размещенных в фигурных скобках после ключевого слова do. Затем проверяется условие, указанное в круглых скобках после ключевого слова while. Если условие истинно, выполняются команды цикла и снова проверяется условие и т. д. Таким образом, хотя бы один раз команды цикла будут выполнены - в этом отличие инструкции do-while() от инструкции while(). Ниже приведен пример использования цикла do-while() в программе для вычисления суммы нечетных натуральных чисел.

*class UsingDoWhile{*

*public static void main(String[] args){*

*int sum=0,i=1;*

*// Инструкция цикла:*

*do{*

*sum+=i;*

*i+=2;*

*}while(i<=100);*

*System.out.println("Сумма нечетных чисел от 1 до 100: "+sum);}*

*}*

Результат выполнения этой программы точно такой же, как и рассмотренных ранее аналогичных программ по вычислению суммы нечетных натуральных чисел в диапазоне от 1 до 100.

**Метки и инструкции break() и continue()**

В Java, в отличие от языка С++, нет инструкции перехода goto(). Тем не менее в Java могут использоваться метки. Обычно для этого применяют инструкции break() и continue().

С инструкцией break() мы уже сталкивались. В общем случае она завершает работу инструкции цикла или инструкции выбора. Если после инструкции break() указать метку, то управление передается команде, размещенной за помеченной этой меткой инструкцией цикла, выбора или блоком команд (напомним, блок команд заключается в фигурные скобки).

Инструкция continue() используется для завершения текущего цикла в инструкции цикла и переходу к выполнению следующего цикла. Если после инструкции continue() указать метку, то выполняется переход для выполнения итерации помеченного меткой внешнего цикла. Таким образом, инструкция continue() с меткой применяется только тогда, когда имеются вложенные циклы. Что касается непосредственно меток, то в качестве метки применяется идентификатор, который не может начинаться с цифры, заканчивается двоеточием и предназначен для выделения места в программном коде. Никакого дополнительного описания метка не требует.

Пример использования меток приведен ниже.

*class LabelsDemo{*

*public static void main(String[] args){*

*MyLabel:*

*for(int i=1;i<=100;i++){*

*for(int j=1;j<=100;j++){*

*if(i!=j) continue;*

*if((j%3==0)||(i%2==0)) break;*

*if(i+j>20) break MyLabel;*

*System.out.println(i+":"+j);*

*}*

*}}}*

**Массивы**

Массив - это конечная последовательность упорядоченных элементов одного типа, доступ к каждому элементу в которой осуществляется по его индексу. Размер или длина массива - это общее количество элементов в массиве.

Переменные, относящиеся к одному массиву, называются элементами этого массива. Чтобы однозначно идентифицировать элемент массива, необходимо знать имя массива и позицию (размещение) элемента в массиве. Позиция элементов в массиве определяется с помощью целочисленных индексов. Количество индексов, необходимых для идентификации элемента массива, называется размерностью массива. Одномерный массив - это такой массив, в котором идентификация элементов осуществляется с помощью одного индекса.

Для объявления одномерного массива необходимо задать тип, к которому относятся элементы массива, название массива, а также количество элементов, входящих в массив. Синтаксис объявления одномерного массива имеет вид:

*тип[] имя=new тип[размер];*

Вначале указывается тип элементов массива. Обращаем внимание, что для массива после идентификатора типа ставятся пустые квадратные скобки. Далее следует имя массива, оператор присваивания, инструкция динамического выделения памяти new, снова тип элементов массива и в квадратных скобках размер массива (количество элементов в массиве). Например, командой int nums=new int[20] объявляется целочисленный массив nums из 20 элементов.

Строго говоря, представленная здесь команда объявления массива является симбиозом двух команд: команды int[] nums объявления переменной nums типа «целочисленный массив» и команды nums=new int[20] выделения памяти под массив и присваивания ссылки на этот массив переменной nums. Другими словами, процесс объявления массива можно выполнить двумя командами:

*int[] nums;*

*nums=new int[20];*

Допускается указывать квадратные скобки либо после имени типа массива, либо после имени массива. Например, вместо команды int[] nums можно использовать команду int nums[].

Обращение к элементу одномерного массива осуществляется через имя массива с указанием в квадратных скобках индекса элемента. Индексация элементов массива начинается с нуля. Таким образом, ссылка на первый элемент массива nums будет иметь вид nums[0]. Если в массиве 20 элементов, то последний элемент массива имеет индекс 19, то есть nums[19].

Длину массива можно узнать с помощью свойства **length**.

При инициализации массива списком значений при объявлении переменной массива после нее указывается (через оператор присваивания) заключенный в фигурные скобки список значений. Например:

*int[] data={3,8,1,7};*

Пример объявления, инициализации и использования массивов приведен ниже.

*class MyArray{*

*public static void main(String[] args){*

*// Индексная переменная и размер массива:*

*int i,n;*

*// Объявление переменной массива:*

*int[] data;*

*// Инициализация массива:*

*data=new int[]{3,8,1,7};*

*// Длина второго массива:*

*n=data.length;*

*// Объявление второго массива:*

*int[] nums=new int[n];*

*// Заполнение второго массива:*

*for(i=0;i<n;i++){*

*nums[i]=2\*data[i]-3;*

*System.out.println("nums["+i+"]="+nums[i]);*

*}}}*

**Двумерные и многомерные массивы**

Объявляются двухмерные массивы практически так же, как и одномерные, с той лишь разницей, что при этом используются две пары квадратных скобок (как при объявлении переменной массива, так и при выделении для массива области памяти). При этом размер массива указывается по каждому из индексов. Синтаксис объявления двухмерного массива может быть следующим:

*тип[][] имя=new тип[размер\_1][размер\_2];*

Как и в случае одномерного массива, данная команда представляет собой объединение двух отдельных команд:

*тип[][] имя;*

*имя=new тип[размер\_1][размер\_2];*

Первой из них объявляется переменная двухмерного массива имя. Второй командой создается двухмерный массив с размерами размер\_1 и размер\_2, а ссылка на этот массив присваивается в качестве значения переменной массива имя. Например, командой double[][] data=new double[3][4] создается массив с именем data. Элементами массива являются значения типа double. Размер массива по первому индексу равен 3, а по второму - 4. К тому же результату приведет выполнение команд

*double[][] data;*

*data=new double[3][4];*

Пример объявления, инициализации и использования двумерного массива приведен ниже.

*class MyDArray{*

*public static void main(String[] args){*

*// Индексные переменные и размерность массива:*

*int i,j,n=3;*

*// Создание двухмерного массива:*

*int[][] nums=new int[n-1][n];*

*// Вложенные инструкции цикла:*

*for(i=0;i<n-1;i++){*

*for(j=0;j<n;j++){*

*// Заполнение элементов массива:*

*nums[i][j]=10\*(i+1)+j+1;*

*// Вывод значений в одну строку:*

*System.out.print(nums[i][j]+" ");*

*}*

*// Переход на новую строку*

*System.out.println();*

*}*

*}}*

**Присваивание и сравнение массивов**

В принципе, один массив может быть присвоен в качестве значения другому массиву, если их типы совпадают. Массивы также можно сравнивать.

Переменная, обозначающая массив (переменная массива), объявляется независимо от фактического выделения памяти под массив. Другими словами, непосредственно массив и переменная массива - это далеко не одно и то же. В этом смысле показательным является двухэтапный (двумя командами) процесс создания массива. Например:

*int[] nums;*

*nums=new int[]{1,2,3,4};*

В данном случае команда int[] nums есть не что иное, как объявление переменной nums. Тип этой переменной - «массив целых чисел». Значением переменной может быть ссылка (адрес) на какой-нибудь массив, состоящий из целых чисел.

*int[] nums,data;*

*nums=new int[]{1,2,3,4};*

*data=nums;*

Все достаточно просто. Первой командой int[] nums,data объявляются две переменные массива nums и data. Второй командой nums=new int[]{1,2,3,4} создается массив, а ссылка на него присваивается в качестве значения переменной nums. Далее командой data=nums значение переменной nums присваивается переменной data. Однако значение переменой nums - это ссылка на массив. Поэтому после присваивания переменная data ссылается на тот же массив! Например, элемент data[1] имеет такое же значение, что и nums[1] (значение 2). Точнее, это один и тот же элемент. Более того, если теперь изменить какой-нибудь элемент массива data (например, data[3]=-1), автоматически изменится и соответствующий элемент массива nums. Причина та же - массив на самом деле один, просто на него ссылаются две переменные. При сравнении массивов с помощью операторов равно == и не равно != (например, nums==data или nums!=data) сравниваются значения переменных массива, а не элементы в этих массивах. Поэтому результатом выражения nums==data является true, если обе переменные массива nums и data ссылаются на один и тот же массив.

Особенности сравнения массивов на предмет равенства (неравенства) иллюстрируются программным кодом ниже.

*class MyArrayDemo2{*

*public static void main(String[] args){*

*// Объявление массивов:*

*int[] nums=new int[]{1,2,3,4,5};*

*int[] data=new int[]{1,2,3,4,5};*

*// Комментирование следующей команды можно отменить:*

*// data=nums;*

*// Проверка совпадения ссылок:*

*if(data==nums){*

*System.out.println("Совпадающие массивы!");*

*return;*

*}*

*// Проверка размеров массивов:*

*if(data.length!=nums.length){*

*System.out.println("Разные массивы!");*

*return;*

*}*

*// Поэлементная проверка массивов:*

*for(int i=0;i<data.length;i++){*

*if(data[i]!=nums[i]){*

*System.out.println("Несовпадающие элементы!");*

*return;*

*}*

*}*

*System.out.println("Одинаковые массивы!");*

*}}*

Программа предназначена для сравнения двух целочисленных массивов. В программе объявляются два целочисленных массива nums и data и инициализируются одинаковыми наборами значений. Далее непосредственно выполняется проверка. Состоит она из трех этапов. Сначала выполняется проверка равенства переменных массивов nums и data. Если ссылки равны, то, очевидно, массивы одинаковы (совпадают). Проверка равенства ссылок на массивы выполняется с помощью условной инструкции if() с условием data==nums. При выполненном условии выводится сообщение «Совпадающие массивы!». При этом работа программы завершается, для чего используется команда return.

Если ссылки различны, выполняется поэлементная проверка массивов. Массивы считаются одинаковыми, если у них совпадают соответствующие элементы. Но прежде необходимо проверить, совпадают ли размеры массивов. Проверка равенства размеров массивов также выполняется с помощью условной инструкции if(), при этом проверяется условие data.length!=nums.length. Условие является истинным, если массивы имеют разные размеры. В этом случае выводится сообщение «Несовпадающие элементы!», и работа программы завершается.

При совпадающих размерах массивов запускается цикл, в рамках которого сравниваются элементы двух массивов. Для этого использована условная инструкция if() с проверяемым условием data[i]!=nums[i] (i — индексная переменная). Если встречаются несовпадающие элементы, выводится сообщение «Несовпадающие элементы!», и работа программы завершается.

В случае если два массива имеют только совпадающие элементы, цикл заканчивается без последствий, и, самое главное, работа программы продолжается, поэтому в конце выводится сообщение «Одинаковые массивы!».

**Задачи**

**Задачи повышенной трудности**

# Лабораторная работа №4

## Тема: Классы и объекты в Java. Иерархия классов. Создание классов и объектов. Интерфейсы. Использование пакетов и jar-файлов.

**Цели работы:** научиться использовать классы и создавать объекты классов.

**Постановка задачи**

**Используемый метод**

Обычно выделяют три фундаментальных принципа, составляющих основу ООП: *инкапсуляция*, *полиморфизм* и *наследование*. Кроме этих принципов, вводятся также два важных понятия - это понятия *класса* и *объекта*.

**Классы и объекты**

Чтобы решить проблему упорядочивания программного кода, было принято решение ввести четкое разграничение данных и методов обработки этих данных. Более того, данные и соответствующие им методы объединили в одну структуру, которая в ООП называется объектом. Такой на первый взгляд искусственный прием позволяет четко разграничить область применимости методов. Вся программа при этом имеет блочную структуру, что существенно упрощает анализ программного кода. Но даже в таком подходе было бы мало пользы, если бы каждый объект был абсолютно уникальным. Практика же такова, что каждый объект определяется некоторым общим шаблоном, который называется классом. В рамках класса задается общий шаблон, то есть структура, на основе которой затем создаются объекты. Данные, относящиеся к классу, называются полями класса, а программный код для их обработки - методами класса. В классе описывается, какого типа данные относятся к классу (данные называются полями класса), а также то, какие методы применяются к этим данным. Затем в программе на основе того или иного класса создается экземпляр класса (объект), в котором указываются конкретные значения полей и выполняются необходимые действия над ними.

Описание класса начинается с ключевого слова class. После этого следует имя класса и в фигурных скобках тело класса. Тело класса состоит из описания членов класса: полей и методов. По большему счету поля - это рассмотренные в предыдущих главах переменные. Таким образом, синтаксис объявления класса имеет следующий вид:

*class имя\_класса{*

*// тело класса*

*}*

В теле класса перечисляются с указанием типа переменные - поля класса (это в том числе могут быть массивы и объекты). Что касается методов, то это - обычные функции, только описанные в рамках класса.

Синтаксис объявления метода имеет вид:

*тип\_результат имя\_метода(аргументы){*

*// тело метода*

*}*

Объект создается в два этапа, которые обычно объединяют. На первом этапе объявляется объектная переменная или переменная объекта - формально, это имя объекта. С технической точки зрения объектная переменная содержит в качестве значения адрес объекта. Второй этап состоит в выделении в памяти места под объект (непосредственно создание объекта) и присваивание в качестве значения объявленной на предыдущем этапе объектной переменной ссылки на созданный объект. Синтаксис объявления объектной переменной мало отличается от объявления переменной базового типа с той лишь разницей, что в качестве типа переменной указывается имя класса, для которого создается объект. Создание объекта (выделение памяти под объект) выполняется с помощью оператора new, после которого указывается имя класса с пустыми круглыми скобками. На самом деле в этом месте указывается конструктор с аргументами, но поскольку конструкторы рассматриваются в следующей главе, пока что примем на веру, что объекты создаются именно так. Таким образом, синтаксис создания объекта имеет вид:

*имя\_класса имя\_объекта; // объектная переменная*

*имя\_объекта=new имя\_класса(); // выделение памяти*

Обычно эти две команды объединяют:

*имя\_класса имя\_объекта=new имя\_класса();*

Пример объявления класса, содержащего два поля и метод:

*class MyClass{*

*double x;*

*int m;*

*void set(double z, int n){*

*x=z;*

*m=n;}*

*}*

Класс имеет название MyClass и содержит два поля (поле x типа double и поле m типа int), а также метод с названием set(). Метод не возвращает результат, поэтому в сигнатуре метода в качестве типа возвращаемого результата указано ключевое слово void. У метода два аргумента: один типа double и второй типа int. Первый аргумент присваивается в качестве значения полю x, второй определяет значение поля m. Обращаем внимание, что описание класса к созданию объектов не приводит. Другими словами, описывающий класс код - это всего лишь шаблон, по которому впоследствии можно создавать объекты, а можно и не создавать. В данном случае команды по созданию объекта класса MyClass могут выглядеть так:

*MyClass obj; // Объектная переменная*

*obj=new MyClass(); // Создание объекта*

Или так:

*MyClass obj=new MyClass();*

Ниже приведен пример программы; в ней, кроме основного класса, описан еще один класс, в котором объявляются несколько полей и два метода, а также показано, как эти поля и метод используются в программе.

*class Coords{*

*// Координаты точки:*

*double x;*

*double y;*

*double z;*

*// Метод для присваивания значений полям:*

*void set(double a,double b,double c){*

*x=a;*

*y=b;*

*z=c;*

*}*

*// Методом вычисляется расстояние до точки:*

*double getDistance(){*

*return Math.sqrt(x\*x+y\*y+z\*z);*

*}*

*}*

*class CoordsDemo{*

*public static void main(String[] args){*

*// Создание объекта:*

*Coords obj=new Coords();*

*// Вызов метода:*

*obj.set(5.0,0,2.5);*

*// Обращение к полю объекта:*

*obj.y=-4.3;*

*// Обращение к методу объекта:*

*System.out.println("Расстояние до точки: "+obj.getDistance());*

*}}*

**Статические элементы**

Как уже отмечалось, помимо обычных полей и методов, в классе могут быть статические члены. От нестатических членов статические принципиально отличаются тем, что они общие для всех объектов данного класса. Например, если речь идет о нестатическом поле, то у каждого объекта класса это поле имеет свое уникальное для объекта значение. Если поле является статическим, то у всех объектов значение этого поля одно и то же. Для объявления статического члена класса, в том числе статического поля, используется идентификатор static. Синтаксис описания статического поля следующий:

*static тип\_поля имя\_поля;*

Синтаксис описания статического блока такой:

*static{ // начало статического блока*

*// статический блок*

*} // завершение статического блока*

**Доступ к членам класса**

В Java в зависимости от доступности все члены класса можно разделить на три группы: открытые, закрытые и защищенные. Во всех рассматривавшихся ранее примерах все члены классов были открытыми и, в силу этого обстоятельства, доступными за пределами класса. Таким образом, открытые члены класса – это члены класса, которые доступны вне этого класса. Если в каком-то месте программы создается объект класса, то к полям и методам этого объекта можно обращаться способом, описанным ранее (например, в формате объект.метод или объект.поле). Если поле или метод является закрытым, ничего из упомянутого сделать не удастся. К закрытым членам класса доступ осуществляется только в пределах класса. К закрытому полю можно обращаться в методах класса, но нельзя обратиться к полю вне класса. Закрытые методы класса могут вызываться только методами класса, но не могут вызываться извне класса. Разница между закрытыми и защищенными членами класса проявляется только при наследовании. Если о наследовании речь не идет, то можно полагать, что защищенный член класса - это аналог закрытого члена. Для определения уровня доступности используются три идентификатора: public, private и protected - идентификаторы доступа соответственно для открытых, закрытых и защищенных членов. Если идентификатор доступа не указан вовсе, соответствующий член считается открытым.

**Ключевое слово this**

Ключевое слово this является стандартной ссылкой на объект, из которого вызывается метод. При этом следует учесть, что в Java ссылка на объект фактически является именем этого объекта.

Пример использования ссылки this:

*class MyClass{*

*// Поля класса:*

*double Re,Im;*

*void set(double Re,double Im){*

*// Использование ссылки this:*

*this.Re=Re;*

*this.Im=Im;*

*}*

*void get(){*

*// Инструкция перехода на новую строку \n:*

*System.out.println("Значения полей:\nRe="+this.Re+" и Im="+this.Im);*

*}*

*}*

*class ThisDemo{*

*public static void main(String[] args){*

*MyClass obj=new MyClass();*

*obj.set(1,5);*

*obj.get();*

*}*

*}*

**Внутренние классы**

Внутренний класс – это класс, объявленный внутри другого класса. Эту ситуацию не следует путать с использованием в качестве поля класса объекта другого класса. Здесь речь идет о том, что в рамках кода тела класса содержится описание другого класса, который и называется внутренним. Класс, в котором объявлен внутренний класс, называется внешним. В принципе, внутренний класс может быть статическим, но такие классы используются на практике крайне редко, поэтому рассматривать мы их не будем, а ограничимся только нестатическими внутренними классами. Внутренний класс имеет несколько особенностей. Во-первых, члены внутреннего класса доступны только в пределах внутреннего класса и недоступны во внешнем классе (даже если они открытые). Во-вторых, во внутреннем классе можно обращаться к членам внешнего класса напрямую. Наконец, объявлять внутренние классы можно в любом блоке внешнего класса. Пример использования внутреннего класса приведен ниже.

*class MyOuter{*

*// Поле внешнего класса:*

*int number=123;*

*// Метод внешнего класса:*

*void show(){*

*// Создание объекта внутреннего класса:*

*MyInner InnerObj=new MyInner();*

*// Вызов метода объекта внутреннего класса:*

*InnerObj.display();*

*}*

*// Внутренний класс:*

*class MyInner{*

*// Метод внутреннего класса:*

*void display(){*

*System.out.println("Поле number="+number);*

*}*

*}*

*}*

*class InnerDemo{*

*public static void main(String args[]){*

*// Создание объекта внешнего класса:*

*MyOuter OuterObj=new MyOuter();*

*// Вызов метода объекта внешнего класса:*

*OuterObj.show();*

*}*

*}*

**Подкласс и суперкласс**

Для реализации наследования в описании подкласса после имени класса указывается ключевое слово extends и имя суперкласса. Во всем остальном описание подкласса не отличается от описания обычного класса (то есть класса, который создается «с нуля»). Синтаксис описания подкласса имеет вид:

*class A extends B{*

*// код*

*}*

В данном случае подкласс A создается на основе суперкласса B. В результате подкласс A получает (наследует) открытые и защищенные члены класса B.

Пример создания подкласса:

*class A{ // Суперкласс*

*int i,j;*

*void showij(){*

*System.out.println("Поля i и j: "+i+" и "+j);*

*}*

*}*

*class B extends A{ // Подкласс*

*int k;*

*void showk(){*

*System.out.println("Поле k: "+k);*

*}*

*void sum(){*

*// Обращение к наследуемым полям:*

*System.out.println("Сумма i+j+k="+(i+j+k));*

*}*

*}*

*class AB{*

*public static void main(String arg[]){*

*// Объект суперкласса:*

*A SuperObj=new A();*

*// Объект подкласса:*

*B SubObj=new B();*

*SuperObj.i=10;*

*SuperObj.j=20;*

*SuperObj.showij();*

*SubObj.i=7;*

*SubObj.j=8;*

*SubObj.k=9;*

*SubObj.showij();*

*SubObj.showk();*

*SubObj.sum();*

*}*

*}*

**Многоуровневое наследование**

Хотя множественное наследование (наследование сразу нескольких классов) в Java не допускается, с успехом может использоваться многоуровневое наследование. В этом случае подкласс становится суперклассом для другого подкласса.

Пример такой ситуации приведен ниже:

*class A{*

*int a;*

*A(int i){*

*a=i;*

*System.out.println("Поле a: "+a);*

*}*

*}*

*class B extends A{*

*int b;*

*B(int i,int j){*

*super(i);*

*b=j;*

*System.out.println("Поле b: "+b);*

*}*

*}*

*class C extends B{*

*int c;*

*C(int i,int j,int k){*

*super(i,j);*

*c=k;*

*System.out.println("Поле c: "+c);*

*}*

*}*

*class MultiCall{*

*public static void main(String args[]){*

*C obj=new C(1,2,3);*

*}*

*}*

**Пакеты в Java**

Для определения пакета необходимо в файле с описанием класса, включаемого в пакет, первой командой указать инструкцию package и имя пакета, например:

*package mypack;*

В данном случае mypack – это имя пакета. Если пакет с таким именем уже существует, соответствующий класс (или классы) из файла добавляется в этот пакет. Если такого пакета нет, он создается. Таким образом, одна и та же инструкция package может использоваться в нескольких файлах. Однако в файле может быть только одна инструкция package или не быть вовсе. В последнем случае классы попадают в так называемый *пакет по умолчанию*. Пакет, кроме классов, может содержать интерфейсы, а также подпакеты (то есть другие пакеты).

С практической точки зрения наиболее важный аспект в работе с пакетами – это, пожалуй, схема доступа к членам классов с учетом их размещения по пакетам. В таблице ниже перечислены уровни доступа к членам класса с учетом использованного при их объявлении спецификатора уровня доступа.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Идентификатор доступа | private | нет | protected | public |
| Тот же класс | + | + | + | + |
| Подкласс того же  пакета | - | + | + | + |
| Не подкласс того  же пакета | - | + | + | + |
| Подкласс в другом  пакете | - | - | + | + |
| Не подкласс  в другом пакете | - | - | - | + |

Символ плюс (+) означает, что член доступен, а символ минус (-) – что нет.

Если в программе выполняется обращение к классам, размещенным во внешних пакетах, необходимо указывать полное имя класса: через точку перечисляется вся иерархия пакетов, где размещен нужный класс. Например, если класс MyClass находится в подпакете subpack пакета mypack, то обращение к этому классу будет иметь вид:

*mypack.subpack.MyClass*

Чтобы можно было ссылаться на классы внешних пакетов в упрощенной форме, прибегают к импорту пакетов. При этом используется ключевое слово import (соответствующая команда размещается в начале файла после команды подключения пакета). Файл может содержать несколько инструкций импорта. Можно подключать (импортировать) отдельные классы пакета или весь пакет. В частности, для импорта класса после ключевого слова import указывают полное имя класса (то есть с учетом иерархии пакетов), например:

*import mypack.subpack.MyClass*

Для импорта всего пакета после имени пакета ставят звездочку (\*), например:

*import mypack.subpack.\**

Существуют некоторые ограничения, накладываемые на импорт пакетов.

1) Импортировать можно только открытые классы.

2) Пакет java.lang (базовая библиотека) можно не импортировать – он и так доступен.

3) Имя файла должно совпадать с именем открытого класса, если такой класс существует в файле.

4) Если в пакете несколько открытых классов, они должны размещаться в разных файлах.

**Интерфейсы**

Интерфейсы во многом напоминают классы. Принципиально от класса интерфейс отличается тем, что содержит только сигнатуры методов без описания, а также поля-константы (поля, значения которых постоянны и не могут изменяться). Описание интерфейса аналогично к описанию класса, только ключевое слово class необходимо заменить ключевым словом interface. Как отмечалось, для методов интерфейса указываются только сигнатуры. Описываемые в интерфейсе поля по умолчанию считаются неизменяемыми (как если бы они были описаны с ключевым словом final) и статическими (то есть static). Таким образом, поля интерфейса играют роль глобальных констант. Практическое использование интерфейса подразумевает его реализацию. Эта процедура напоминает наследование абстрактных классов. Реализуется интерфейс в классе. Класс, который реализует интерфейс, должен содержать описание всех методов интерфейса. Методы интерфейса при реализации описываются как открытые. Один и тот же класс может реализовать одновременно несколько интерфейсов, равно как один и тот же интерфейс может реализовываться несколькими классами. Для реализации интерфейса в классе в сигнатуре заголовка класса указывается инструкция implements(). С учетом того, что реализующий интерфейс класс может одновременно наследовать еще и суперкласс, общий синтаксис объявления класса, который наследует суперкласс и реализует несколько интерфейсов, имеет следующий вид:

*class имя [extends суперкласс] implements интерфейс1,интерфейс2,...{*

*// тело класса*

*}*

Если имеет место наследование классов, то после имени класса через ключевое слово extends указывается имя наследуемого суперкласса, затем идет ключевое слово implements. После ключевого слова implements через запятую перечисляются реализуемые в классе интерфейсы, а дальше все стандартно – указывается непосредственно тело класса. Напомним также, что перед ключевым словом class может размещаться ключевое слово public, определяющее уровень доступа класса.

Ниже приведен пример программы, в которой используется интерфейс.

*// Интерфейс:*

*interface MyMath{*

*// Сигнатура метода:*

*double Sinus(double x);*

*// Константа:*

*double PI=3.14159265358979;*

*}*

*// Класс реализует интерфейс:*

*class MyClass implements MyMath{*

*// Реализация метода (вычисление синуса):*

*public double Sinus(double x){*

*int i,n=1000;*

*double z=0,q=x;*

*for(i=1;i<=n;i++){*

*z+=q;*

*q\*=(-1)\*x\*x/(2\*i)/(2\*i+1);*

*}*

*return z;*

*}*

*}*

*class MyMathDemo{*

*public static void main(String args[]){*

*MyClass obj=new MyClass();*

*// Использование константы:*

*double z=MyClass.PI/6;*

*// Вызов метода:*

*System.out.println("sin("+z+")="+obj.Sinus(z));*

*}*

*}*

**Задачи**

1. Создать класс «Треугольник» с полями длин сторон А, В и С, а также методами вычисления площади и периметра. Программа должна выдавать результаты работы методов по введенным с клавиатуры длинам сторон.
2. Решить задачу 1 используя проверку существования такого треугольника, реализованную через методы внутреннего класса.
3. Решить задачу 1 используя подкласс, который будет определять тип треугольника (прямоугольный, тупоугольный или остроугольный).
4. С помощью интерфейса с полем значения числа Пи и методами определения длины и площади окружности найти длину и площадь по заданному радиусу.

# Лабораторная работа №5

## Тема: Исключительные ситуации. Обработка исключений.

**Цели работы:** научиться использовать инструмент автоматической сборки исключений.

**Постановка задачи**

Решить предложенные задачи, используя обработку исключений для проверки математических ошибок вычислений, ошибок ввода данных и пр.

**Используемый метод**

**Исключительные ситуации**

Исключительная ситуация – это ошибка, которая возникает в результате выполнения программы. Исключение в Java – это объект, который описывает исключительную ситуацию (ошибку). При возникновении исключительной ситуации в процессе выполнения программы автоматически создается объект, описывающий эту исключительную ситуацию. Этот объект передается для обработки методу, в котором возникла исключительная ситуация.

В Java для обработки исключительных ситуаций используется блок try-catch-finally. В блок try помещается программный код, который отслеживается на случай, если возникнет исключительная ситуация. Если исключительная ситуация возникает, то управление передается блоку catch. Программный код в этом блоке выполняется, только если возникает исключительная ситуация, причем не любая, а определенного типа. Аргумент, определяющий, какого типа исключительные ситуации обрабатываются в блоке catch, указывается после ключевого слова catch в круглых скобках, то есть в том же формате, что и аргумент метода. Поскольку в блоке try могут возникать исключения разных типов, для каждого из них можно предусмотреть свой блок catch. Если блоков catch несколько, при возникновении исключительной ситуации они перебираются последовательно до совпадения типа исключительной ситуации с аргументом блока catch. После блоков try и catch можно указать блок finally с кодом, который выполняется в любом случае вне зависимости от того, возникла исключительная ситуация или нет.

Общая схема использования блока try-catch-finally для обработки исключительных ситуаций выглядит так:

*try{*

*// код, который генерирует исключение*

*}*

*catch(Тип\_исключения\_1 объект){*

*// код для обработки исключения*

*}*

*catch(Тип\_исключения\_2 объект){*

*// код для обработки исключения*

*}*

*...*

*finally{*

*// код, который выполняется обязательно*

*}*

Если при исполнении программного кода в блоке try{} ошибок не возникает, после выполнения этого блока выполняется блок finally (если он имеется), затем управление передается следующей после конструкции try-catch-finally команде.

**Классы исключений**

В Java существует целая иерархия классов, предназначенных для обработки исключительных ситуаций. В вершине этой иерархии находится суперкласс Throwable. У этого суперкласса есть два подкласса: Exception и Error. К классу Error относятся «катастрофические» ошибки, которые невозможно обработать в программе, например переполнение стека памяти. У класса Exception есть подкласс RuntimeException. К классу RuntimeException относятся ошибки времени выполнения программы, которые перехватываются программами пользователя. Исключения для класса RuntimeException определяются автоматически. К ним относятся, например, деление на ноль, выход за пределы массива (недопустимая индексация массива).

Ниже приведен пример программы, в которой происходит обработка исключительной ситуации, заключающейся в делении на ноль.

*class ExceptionDemo{*

*public static void main(String args[]){*

*int a,b;*

*// Блок контроля исключительной ситуации:*

*try{*

*b=0;*

*// Деление на ноль:*

*a=100/b;*

*}*

*catch(ArithmeticException e){*

*// Обработка исключительной ситуации:*

*System.out.println("Деление на ноль!");*

*}*

*System.out.println("Выполнение программы продолжено!");*

*}*

*}*

**Описание исключительной ситуации**

В классе Throwable переопределяется метод toString(), который, как известно, определен в общем суперклассе Object, причем переопределяется он так, что в качестве результата возвращает строку, описывающую соответствующую ошибку. Напомним, что метод toString() вызывается автоматически, например, при передаче объекта исключения методу println() в качестве аргумента. Соответствующий пример приведен ниже.

*class MoreExceptionDemo{*

*public static void main(String args[]){*

*int a,b;*

*try{*

*b=0;*

*// Деление на ноль:*

*a=100/b;*

*}*

*catch(ArithmeticException e){*

*// При обработке ошибки использован объект исключения:*

*System.out.println("Ошибка: "+e);*

*}*

*System.out.println("Выполнение программы продолжено!");*

*}*

*}*

По сравнению с рассмотренным ранее особенность этого примера состоит в том, что в команде System.out.println("Ошибка: "+e) в качестве аргумента методу println() передается объект исключительной ситуации e. Результат выполнения программы в этом случае имеет такой вид:

*Ошибка: java.lang.ArithmeticException: / by zero*

*Выполнение программы продолжено!*

**Задачи**

# Лабораторная работа №6

## Тема: Библиотеки классов. Классы для обработки строк. Программирование типовых алгоритмов для обработки строк. Объекты класса String. Методы для работы со строками.

**Цели работы:** научиться работать со строками в языке программирования java.

**Постановка задачи**

Изучить класс String и его методы и решить предложенные задачи.

**Используемый метод**

В Java текст – это объект. Для работы с текстом служат два встроенных Java-класса: String и StringBuffer. Поэтому с формальной точки зрения создание текста сводится к созданию объекта одного из этих классов. Главное принципиальное различие состоит в том, что объекты класса String изменять нельзя, а объекты класса StringBuffer – можно. Классы String и StringBuffer определены в базовом пакете java.lang, который доступен по умолчанию, поэтому для создания объекта класса String или StringBuffer импорт пакетов выполнять не нужно. Оба класса определены как неизменяемые (final), то есть они не могут быть суперклассами для наследования.

**Объекты класса String**

Один из способов создания «текстовой переменной» подразумевает создание объекта класса String. Чтобы создать объект класса, необходимо, как минимум, знать, какие у этого класса есть конструкторы. Что касается класса String, то имеет смысл выделить следующие конструкторы.

*1) Конструктор создания пустой строки*. В этом случае конструктору аргументы не передаются. Пример команды создания объекта класса String со значением в виде пустой строки имеет вид:

*String s=new String();*

*2) Конструктор создания текстовой строки на основе символьного массива*. В этом случае аргументом конструктору передается имя массива символов. Результатом является текст, составленный из всех символов массива в порядке их размещения в массиве. Пример создания текстовой строки на основе символьного массива:

*char symbols[]={'a','b','c'};*

*String s=new String(symbols);*

В этом конструкторе, помимо имени массива, можно указать индекс элемента массива, начиная с которого будет извлекаться строка, а также длину строки в символах. Например, так:

*char symbols={'a','b','c','d','e','f'};*

*String s=new String(symbols,2,3); // s="cde"*

3) *Конструктор копирования объекта*. Аргументом конструктора указывается переменная текстового типа, ссылающаяся на уже существующий текстовый объект или текстовый литерал. В результате создается новый объект с таким же текстовым значением, как и исходный. Например:

*String obj=new String("Текстовая строка");*

*String s=new String(obj);*

**Длина строки length()**

Для определения длины строки, записанной в текстовом объекте, используют метод length().

**Класс StringBuffer**

Текстовые строки могут быть реализованы не только как объекты класса String, но и как объекты класса StringBuffer. Принципиальное отличие этих объектов состоит в том, что объекты класса StringBuffer можно изменять. Другими словами, если текст реализован в виде объекта класса StringBuffer, в этот текст можно вносить изменения, причем без создания нового объекта.

Работа с объектами класса StringBuffer:

*class StringBufferDemo{*

*public static void main(String args[]){*

*// Базовая строка - объект класса StringBuffer:*

*StringBuffer str=new StringBuffer("Мы программируем на С++");*

*// Длина строки:*

*System.out.println(str.length());*

*// Размер строки (максимальная длина в символах):*

*System.out.println(str.capacity());*

*// Вставка подстроки:*

*str.insert(20,"Java и ");*

*// Вывод строки на экран:*

*System.out.println(str);*

*// Замена подстроки:*

*str.replace(27,30,"Pascal");*

*// Вывод подстроки на экран:*

*System.out.println(str);*

*// Инверсия строки:*

*str.reverse();*

*// Вывод строки на экран:*

*System.out.println(str);*

*}*

*}*

**Операции со строками**

Что касается базовых операций, то по отношению к текстовым строкам можно применять только операцию сложения (с оператором +), которая интерпретируется как объединение (конкатенация) соответствующих строк. Пример конкатенации текстовых строк:

*String str="Евгению Петрову "+18+" лет.";*

В результате получаем текст "Евгению Петрову 18 лет.".

**Метод toString()**

Метод toString() определен в классе Object, находящемся на вершине иерархии классов Java (это общий суперкласс Java). Метод вызывается по умолчанию при преобразовании объекта в текстовый формат. Благодаря тому, что метод можно перегружать, открывается целый ряд достаточно интересных возможностей.

Для перегрузки метода toString(), как минимум, необходимо знать его сигнатуру. Метод в качестве значения возвращает объект класса String и не имеет аргументов. Как отмечалось, метод вызывается по умолчанию при преобразовании объекта в текстовый тип. Примеры таких ситуаций: сложение объекта с текстовой строкой или использование объекта в качестве аргумента метода println().

**Методы для работы со строками**

compareTo(String anotherString) - лексиграфическое сравнение строк;

compareToIgnoreCase(String str) - лексиграфическое сравнение строк без учета регистра символов;

regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) - тест на идентичность участков строк, можно указать учет регистра символов;

regionMatches(int toffset, String other, int ooffset, int len) - тест на идентичность участков строк;

concat(String str) - возвращает соединение двух строк;

contains(CharSequence s) - проверяет, входит ли указанная последовательность символов в строку;

endsWith(String suffix) - проверяет завершается ли строка указанным суффиксом;

startsWith(String prefix) - проверяет, начинается ли строка с указанного префикса;

startsWith(String prefix, int toffset) - проверяет, начинается ли строка в указанной позиции с указанного префикса;

equals(Object anObject) - проверяет идентична ли строка указанному объекту;

getBytes() - возвращает байтовое представление строки;

getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) - возвращает символьное представление участка строки;

hashCode() - хеш код строки;

indexOf(int ch) - поиск первого вхождения символа в строке;

indexOf(int ch, int fromIndex) - поиск первого вхождения символа в строке с указанной позиции;

indexOf(String str) - поиск первого вхождения указанной подстроки;

indexOf(String str, int fromIndex) - поиск первого вхождения указанной подстроки с указанной позиции;

lastIndexOf(int ch) - поиск последнего входения символа;

lastIndexOf(int ch, int fromIndex) - поиск последнего входения символа с указанной позиции;

lastIndexOf(String str) - поиск последнего вхождения строки;

lastIndexOf(String str, int fromIndex) - поиск последнего вхождения строки с указанной позиции;

replace(char oldChar, char newChar) - замена в строке одного символа на другой;

replace(CharSequence target, CharSequence replacement) - замена одной подстроки другой;

substring(int beginIndex, int endIndex) - возвратить подстроку как строку;

toLowerCase() - преобразовать строку в нижний регистр;

toLowerCase(Locale locale) - преобразовать строку в нижний регистр, используя указанную локализацию;

toUpperCase() - преобразовать строку в верхний регистр;

toUpperCase(Locale locale) - преобразовать строку в верхний регистр, используя указанную локализацию;

trim() - отсечь на концах строки пустые символы;

valueOf(a) - статические методы преобразования различных типов в строку.

Методы поиска возвращают индекс вхождения или -1 если искомое не найдено. Методы преобразования как replace не изменяют саму строку, а возвращают соответствующий новый объект строки.

Статические строки также имеют ряд методов использующие регулярные выражения:

matches(String regex) - удовлетворяет ли строка указанному регулярному выражению;

replaceAll(String regex, String rplс) - заменяет все вхождения строк, удовлетворяющих регулярному выражению, указанной строкой;

replaceFirst(String regex, String rplс) - заменяет первое вхождение строки, удовлетворяющей регулярному выражению, указанной строкой;

split(String regex) - разбивает строку на части, границами разбиения являются вхождения строк, удовлетворяющих регулярному выражению;

split(String regex, int limit) - аналогично предыдущему, но с ограничением применения регулярного выражения к строке значением limit. Если limit>0, то и размер возвращаемого массива строк не будет больше limit. Если limit<=0, то регулярное выражение применяется к строке неограниченное число раз.

Если необходимо сделать множество преобразований над строкой, то на это время эффективнее воспользоваться динамической строкой, реализуемой классом StringBuffer.

Некоторые методы объекта StringBuffer:

append(A) - преобразовать A в строку и добавить в конец;

insert(int offset, A) - преобразовать A в строку и вставить ее в указанную позицию;

delete(int start, int end) - удалить символы с указанной по указанную позицию;

deleteCharAt(int index) - удалить символ в указанной позиции;

getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) - сохранить последовательность символов в массив;

indexOf(String str) - поиск первого вхождения подстроки;

indexOf(String str, int fromIndex) - поиск первого вхождения подстроки с указанной позиции;

lastIndexOf(String str) - поиск последнего вхождения подстроки;

lastIndexOf(String str, int fromIndex) - поиск последнего вхождения подстроки с указанной позиции;

replace(int start, int end, String str) - замена участка строки указанной строкой;

reverse() - расположить символы в обратном порядке;

setCharAt(int index, char ch) - заменить символ в указанной позиции;

setLength(int newLength) - установить новый размер строки;

substring(int start) - вернуть подстроку с указанной позиции и до конца как строку;

substring(int start, int end) - вернуть подстроку как строку.

Оба строковых класса реализуют интерфейс CharSequence, т.е. включают в себя методы:

charAt(int index) - символ в указанной позиции;

length() - размер строки;

subSequence(int start, int end) - вернуть подстроку как последовательность символов;

toString() - вернуть строковое представление объекта.

**Задачи**

1. Дано: строка str. Требуется: определить самое короткое слово в этой строке и его длину.
2. Дано: строка str. Требуется: поменять местами первое и последнее слово.
3. Дано: две строки str1 и str2. Требуется: найти первое и последнее вхождение строки str2 в строку str1 и посчитать количество вхождений строки str2 в строку str1.
4. Дано: две строки str1 и str2. Требуется: удалить из строки str1 все вхождения строк str2.
5. Дано: три строки str1, str2 и str3. Требуется: заменить в строке str1 все вхождения строки str2 на строку str3.

**Задачи повышенной трудности**

1. Дано: строка str. Требуется: определить является ли строка str палиндромом.

# Лабораторная работа №7

## Тема: Система ввода-вывода. Байтовые и символьные потоки. Консольный ввод с использованием объекта System.in. Консольный ввод с помощью класса Scanner. Использование диалогового окна. Работа с файлами.

**Цели работы:** научиться использовать потоки ввода-вывода и работать с файлами.

**Постановка задачи**

Обработать потоки ввода-вывода используя объект System.in и класс Scanner. Также выполнить чтение и запись в файл.

**Используемый метод**

Ввод и вывод данных в Java реализуется через *потоки ввода-вывода* (stream). Выделяют два типа потоков: байтовые и символьные. На самом деле, в обоих случаях речь идет о байтах, но поскольку символ – это два байта, то при работе с символьными потоками байты обрабатываются, условно говоря, парами.

Для работы с символьными и байтовыми потоками используются специально разработанные для этого классы. Чтобы эти классы стали доступными в программе, необходимо подключить (импортировать) пакет java.io.

**Байтовые и символьные потоки**

На вершине иерархии байтовых потоков находятся два абстрактных класса: InputStream и OutputStream. В этих классах определены методы read() и write(), предназначенные для чтения данных из потока и записи данных в поток соответственно.

Некоторые другие классы байтовых потоков перечислены в таблице ниже:

|  |  |
| --- | --- |
| **Класс байтового потока** | **Описание** |
| InputStream | Абстрактный класс, который описывает поток ввода |
| OutputStream | Абстрактный класс, который описывает поток вывода |
| FilterInputStream | Класс, который реализует абстрактный класс InputStream |
| FilterOutputStream | Класс, который реализует абстрактный класс OutputStream |
| BufferedInputStream | Класс буферизованного потока ввода |
| BufferedOutputStream | Класс буферизованного потока вывода |
| ByteArrayInputStream | Класс потока ввода для считывания из массива |
| ByteArrayOutputStream | Класс потока вывода для записи в массив |
| FileInputStream | Класс потока ввода для считывания из файла |
| FileOutputStream | Класс потока вывода для записи в файл |
| DataInputStream | Класс потока ввода с методами для считывания данных стандартных типов Java |
| DataOutputStream | Класс потока вывода с методами для записи данных стандартных типов Java |
| PrintStream | Класс потока вывода, который поддерживает методы print() и println() |

Иерархия классов для символьных потоков ввода-вывода начинается с абстрактных классов Reader и Writer. В этих классах определены методы read() для считывания символьных данных из потока и write() для записи символьных данных в поток.

Некоторые из классов для символьных потоков представлены и кратко описаны в таблице ниже:

|  |  |
| --- | --- |
| **Класс символьного потока** | **Описание** |
| Reader | Абстрактный класс, который описывает поток ввода |
| Writer | Абстрактный класс, который описывает поток вывода |
| FilterReader | Класс, который описывает отфильтрованный поток ввода |
| FilterWriter | Класс, который описывает отфильтрованный поток вывода |
| InputStreamReader | Класс потока ввода, который переводит байты в символы |
| OutputStreamWriter | Класс потока вывода, который переводит символы в байты |
| StringReader | Класс потока ввода для считывания из текстовой строки |
| StringWriter | Класс потока вывода для записи в текстовую строку |
| FileReader | Класс потока ввода для считывания из файла |
| FileWriter | Класс потока вывода для записи в файл |
| BufferedReader | Класс буферизованного потока ввода |
| BufferedWriter | Класс буферизованного потока вывода |
| PrintWriter | Класс потока вывода, который поддерживает методы print() и println() |
| CharArrayReader | Класс потока ввода для считывания из массивам |
| CharArrarWriter | Класс потока вывода для записи в массив |
| LineNumberReader | Класс потока ввода для подсчета текстовых строк |

Часть возможностей ввода-вывода может быть реализована посредством класса System. Класс System содержит три переменных потока: in, out и err. Эти поля имеют атрибуты public и static. В частности:

1) Поле System.out – поток стандартного вывода. По умолчанию он связан с консолью. Поле System.out является объектом класса PrintStream.

2) Поле System.in – это поток стандартного ввода. По умолчанию он связан с клавиатурой. Поле является объектом класса InputStream.

3) Поле System.err – это стандартный поток ошибок. По умолчанию поток связан с консолью. Поле является объектом класса PrintStream.

**Консольный ввод с использованием объекта System.in**

Консольный ввод в Java реализуется посредством считывания объекта System.in. При этом используется класс BufferedReader – класс буферизованного входного потока. Применение этого класса требует подключения (импорта) пакета java.io (командой import java.io.\* в заголовке файла программы).

классический пример команды создания объекта буферизованного символьного потока ввода (объекта br):

*BufferedReader br=new BufferedReader(new InputStreamReader(System.in));*

Объект создается как относящийся к классу BufferedReader. При этом конструктору класса BufferedReader указан анонимный объект класса InputStreamReader. Этот анонимный объект создается конструктором с аргументом – объектом System.in. Такая нетривиальная цепочка инструкций связана с тем, что класс InputStreamReader переводит байты в символы, но позволяет считывать при этом только один символ. Поэтому объект класса InputStreamReader «вкладывается» в класс BufferedReader, который позволяет считывать несколько символов.

Для считывания символов служит метод read() класса BufferedReader. Метод имеет следующую сигнатуру:

*int read() throws IOException*

Таким образом, метод выбрасывает исключение IOException (ошибка ввода-вывода). Обращаем внимание на то, что в качестве результата метод возвращает число – код символа. Поэтому считанные с помощью этого метода значения необходимо через механизм явного приведения типов преобразовывать в тип char. Для считывания текстовых строк используют метод readLine() класса BufferedReader. У метода следующая сигнатура:

*String read() throws IOException*

Методом также выбрасывается исключение IOException, а результатом является считанная из буфера текстовая строка.

**Консольный ввод с помощью класса Scanner**

Для работы с этим классом необходимо включить в заголовок файла программы инструкцию import java.util.\*, то есть подключить (импортировать) пакет java.util.

Общая схема реализации процесса введения данных с консоли посредством класса Scanner такова: на основе стандартного потока ввода System.in создается объект класса Scanner, через который и осуществляется консольный ввод. При этом полезными могут оказаться методы класса Scanner, среди которых имеет смысл выделить следующие:

1) nextLine() – считывание текстовой строки;

2) next() – считывание одного слова;

3) nextInt() – считывание целого числа;

4) nextDouble() – считывание действительного числа.

Пример использования класса Scanner и его методов для реализации в программе консольного ввода приведен ниже.

*// Импорт пакета:*

*import java.util.\*;*

*class MyNewScanner{*

*public static void main(String args[]){*

*// Объект класса Scanner создается на основе объекта System.in:*

*Scanner inp=new Scanner(System.in);*

*// Текстовое поле (имя):*

*String name;*

*// Числовое поле (возраст):*

*int age;*

*// Задаем вопрос:*

*System.out.println("Как Вас зовут?");*

*// Считываем текст (имя):*

*name=inp.nextLine();*

*// Приветствие:*

*System.out.println("Добрый день, "+name+"!");*

*// Задаем вопрос:*

*System.out.println("Сколько Вам лет?");*

*// Считываем число (возраст):*

*age=inp.nextInt();*

*// Вывод сообщения:*

*System.out.println(name+", Вам "+age+" лет!");*

*}*

*}*

**Использование диалогового окна**

Введение данных во время выполнения программы можно реализовать не через консоль, а с помощью специального диалогового окна, которое является компонентом графической библиотеки Swing. Для подключения библиотеки необходимо в заголовок файла программы включить инструкцию import javax.swing.\*. Само диалоговое окно вызывается командой вида JOptionPane. showInputDialog(аргумент), где аргументом статического метода showInputDialog() класса JOptionPane указывается текст, отображаемый над полем ввода диалогового окна. В качестве результата методом showInputDialog() возвращается текст, который вводится пользователем в поле ввода диалогового окна.

Поскольку при вызове диалогового окна создается новый поток, который автоматически не завершается при закрытии окна, для завершения программы со всеми ее потоками требуется использовать инструкцию System.exit(0).

Для перевода текстового представления чисел в числовые форматы вызывают статический метод parseInt() класса-оболочки Integer и статический метод parseDouble() класса-оболочки Double. Аргументами методов указывают текстовое представление числа. Метод parseInt() предназначен для работы с целыми числами, а parseDouble() – с действительными.

Ввод с помощью диалогового окна:

*// Импорт пакета:*

*import javax.swing.\*;*

*class MyOptionPane{*

*public static void main(String args[]){*

*// Текстовое поле (имя):*

*String name;*

*// Числовое поле (возраст):*

*int age;*

*// Вопрос:*

*System.out.println("Как Вас зовут?");*

*// Открываем диалоговое окно:*

*name=JOptionPane.showInputDialog("Укажите Ваше имя");*

*// Выводим резльтат считывания:*

*System.out.println("Считано имя: "+name);*

*// Приветствие:*

*System.out.println("Добрый день, "+name+"!");*

*// Вопрос:*

*System.out.println("Сколько Вам лет?");*

*// Открываем диалоговое окно:*

*age=Integer.parseInt(JOptionPane.showInputDialog("Укажите Ваш возраст"));*

*// Выводим результат считывания:*

*System.out.println("Считан возраст: "+age+" лет.");*

*// Сообщение:*

*System.out.println(name+", Вам "+age+" лет!");*

*// Завершение всех потоков:*

*System.exit(0);*

*}*

*}*

**Работа с файлами**

Ввод и вывод информации в файлы и из файлов осуществляется достаточно просто. Для этого создается поток ввода или вывода и подключается к соответствующему файлу. При этом используют классы файловых потоков FileInputStream (файловый поток ввода) и FileOutputStream (файловый поток вывода). Конструкторы классов FileInputStream и FileOutputStream выбрасывают исключение FileNotFoundException (файл не найден). Имя (полное) файла, с которым связывается поток ввода или вывода, в виде текстовой строки передается аргументом конструктору.

После завершения работы с файлом поток необходимо закрыть, для чего используют метод close(). Пример программы, в которой реализована работа с файлами, приведен ниже.

*// Подключение пакета:*

*import java.io.\*;*

*class WriteToFile{*

*public static void main(String args[]) throws IOException{*

*// Целочисленное поле:*

*int a;*

*try{*

*// Поток файлового вывода:*

*FileOutputStream fout=new FileOutputStream("F:/Java\_2/Files/mydata.txt");*

*// Поток файлового ввода:*

*FileInputStream fin=new FileInputStream("F:/Java\_2/Files/base.txt");*

*// Считывание из файла:*

*a=fin.read();*

*while(a!=-1){*

*// Замена пробелов символами подчеркивания:*

*if(a==(int)' ') a=(int)'\_';*

*// Запись в файл:*

*fout.write(a);*

*// Считывание из файла:*

*a=fin.read();*

*}*

*// Закрыт поток вывода:*

*fout.close();*

*// Закрыт поток ввода:*

*fin.close();*

*}*

*catch(FileNotFoundException e){*

*// Обработка исключения "файл не найден":*

*System.out.println("Нет доступа к файлу: "+e);*

*}*

*// Сообщение программы:*

*System.out.println("Работа программы завершена!");*

*}*

*}*

# Лабораторная работа №8

## Тема: Декомпозиция простых алгоритмов. Коллективная разработка проекта.

**Цели работы:** научиться работать в команде, интегрируя два и более программных решения в один проект.

**Постановка задачи**

Разбиться на группы по 2-3 человека (в зависимости от исполнителей в задаче). Каждый исполнитель выполняет свою часть проекта.

**Пример задачи**

1. Дано: с клавиатуры вводится имя файла и две строки str1 и str2. Требуется: заменить в файле все вхождения строки str1 на строку str2.

Исполнитель\_1: осуществляет чтение данных (имя файла и строки) и чтение информации из файла, а также запись в файл измененной информации.

Исполнитель\_2: осуществляет операцию замены строки str1 на строку str2 и готовит отчет о проделанной работе.

# Контрольная работа №1

**Задачи**

1. Дано: координаты двух точек А=(x1;y1) и В=(x2;y2). Требуется: вычислить расстояние между двумя точками А и В.
2. Дано: действительное число. Требуется: определить, является ли это число натуральным.
3. Дано: целое число X и натуральное число N. Требуется: вычислить X в степени N.
4. Дано: целочисленный массив. Требуется: отсортировать массив по не убыванию.

**Задачи повышенной трудности**

1. Используя класс, описывающий работу с файлом, переписать информацию в файле в обратном порядке.
2. Используя класс, описывающий работу со строкой, зашифровать строку, сдвигом каждого символа на количество позиций, равное введенному целому числу.

# Лабораторная работа №9

## Тема: Создание программ с графическим интерфейсом. Создание оконных приложений. События. Обработка событий фреймового окна, мыши и клавиатуры.

**Цели работы:** научиться создавать и работать с графическим интерфейсом и обрабатывать события интерфейса.

**Постановка задачи**

Создать интерфейс и обработать различные события.

**Используемый метод**

В Java для создания графического интерфейса используются библиотеки AWT и Swing.

**Создание простого окна**

Рассмотрение методов создания и работы с графическим интерфейсом начнем с задачи создания фрейма. Фрейм – это такое окно, которое не содержит в себе других окон.

Для создания фрейма в Java необходимо расширить один из базовых классов: класс Frame из библиотеки AWT или класс JFrame из библиотеки Swing (причем класс JFrame является подклассом класса Frame). Далее необходимо предусмотреть возможность для этого окна реагировать на события, в частности на попытку это окно закрыть. Для этого создается специальный обработчик события закрытия окна. С помощью метода addWindowListener() ссылка на этот обработчик добавляется в класс, реализующий окно.

Для использования библиотек AWT и Swing, а также классов обработки событий подключают пакеты javax.swing (библиотека Swing), java.awt и java.awt.event (библиотека AWT).

**Создание графического окна средствами Swing**

*// Подключение библиотеки:*

*import javax.swing.\*;*

*// Расширение класса JFrame:*

*class JustAFrame extends JFrame{*

*// Конструктор класса:*

*public JustAFrame(int a,int b){*

*// Заголовок окна - аргумент конструктора суперкласса:*

*super("Простое графическое окно");*

*// Размеры окна:*

*setSize(a,b);*

*// Реакция на попытку закрыть окно:*

*setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);*

*// Отображение окна:*

*setVisible(true);*

*}*

*}*

*class MyFrame{*

*public static void main(String args[]){*

*// Создание окна:*

*JustAFrame frame=new JustAFrame(300,200);*

*}*

*}*

**Создание графического окна средствами AWT**

*// Подключение библиотеки AWT:*

*import java.awt.\*;*

*import java.awt.event.\*;*

*// Класс обработчика закрытия окна:*

*class MyAdapter extends WindowAdapter{*

*public void windowClosing(WindowEvent we){*

*System.exit(0);*

*}*

*}*

*// Класс окна:*

*class JustAFrame extends Frame{*

*// Конструктор:*

*JustAFrame(int a,int b){*

*// Аргумент конструктора суперкласса - имя окна:*

*super("Новое окно");*

*// Объект обработчика закрытия окна:*

*MyAdapter adapter=new MyAdapter();*

*// Размеры окна:*

*setSize(a,b);*

*// Отображение окна:*

*setVisible(true);*

*// Добавлен обработчик:*

*addWindowListener(adapter);*

*}*

*}*

*class MyAWTFrame{*

*public static void main(String args[]){*

*// Создание окна:*

*JustAFrame frame=new JustAFrame(400,300);*

*}*

*}*

Командой super("Новое окно") в конструкторе вызывается конструктор суперкласса с текстовым аргументом, который определяет название создаваемого окна – оно отображается в строке заголовка.

Командой setSize(a,b) с помощью унаследованного из класса Frame метода setSize() задаются (в пикселях) размеры окна по горизонтали и вертикали.

Поскольку создание окна не означает его отображения на экране, командой setVisible(true) окно выводится на экран (становится видимым).

Командой MyAdapter adapter=new MyAdapter() создается объект обработчика события закрытия окна, то есть объект adapter класса MyAdapter. Команда addWindowListener(adapter) означает, что объект adapter используется для обработки событий, происходящих с окном. В свою очередь, что именно будет происходить в рамках обработки событий, определяется кодом класса MyAdapter. В этом классе описан всего один метод windowClosing(), наследуемый от класса WindowAdapter. Аргументом методу передается объект события (генерируется автоматически при возникновении события). Метод windowClosing() определяет последовательность действий при попытке закрыть окно. Действие всего одно – командой System.exit(0) завершается работа всех процессов программы, в результате чего окно закрывается.

Аргументом методу setDefaultCloseOperation(), определяющему реакцию на попытку закрыть окно, передается статическая константа EXIT\_ON\_CLOSE класса JFrame, означающая, что работа программы будет завершена и окно закрыто.

**Обработка событий**

На вершине иерархии классов, описывающих события, находится класс EventObject, который описан в пакете java.util. В свою очередь, класс EventObject наследуется абстрактным классом AWTEvent. Этот класс описан в библиотеке AWT и, соответственно, находится в пакете java.awt. Все остальные классы обработки событий в библиотеке AWT являются подклассами класса AWTEvent и описаны в пакете java.awt.event.

**Классы событий библиотеки AWT**

|  |  |  |
| --- | --- | --- |
| **Класс события** | **Описание** | **Может возникать** |
| ActionEvent | Генерируется при щелчке мышью на кнопке | Возникает в компонентах классов Button, List и TextField |
| AdjustmentEvent | Возникает при изменении положения ползунка полосы прокрутки | Возникает в компонентах класса Scrollbar |
| ComponentEvent | Возникает при перемещении компонента, изменении его размеров, отображении и скрытии компонента | Возникает во всех компонентах |
| ItemEvent | Возникает при выборе или отказе от выбора элемента в соответствующих компонентах | Возникает в компонентах классов Checkbox, Choice и List |
| TextEvent | Происходит при изменении текста | Возникает в компонентах классов TextComponent, TextArea и TextField |
| ContainerEvent | Возникает, если в контейнер добавляется компонент или компонент из контейнера удаляется | Возникает в компонентах классов Container, Dialog, FileDialog, Frame, Panel, ScrollPane и Window |
| FocusEvent | Возникает, если соответствующий компонент получает или теряет фокус | Возникает во всех компонентах |
| InputEvent | Абстрактный класс, являющийся суперклассом для классов KeyEvent и MouseEvent. В классе определяются восемь целочисленных констант для получения информации о событии | Возникает при операциях ввода для компонентов |
| PaintEvent | Происходит при перерисовке компонента | Возникает в основных компонентах |
| WindowEvent | Происходит при открытии, закрытии, сворачивании, разворачивании окна, получении и передаче окном фокуса | Возникает в компонентах классов Dialog, FileDialog, Frame и Window |
| KeyEvent | Возникает при нажатии клавиши, отпускании клавиши, ввода символа | Возникает во всех компонентах |
| MouseEvent | Возникает при манипуляциях мышью с компонентом, таких как щелчок, перемещение указателя, появление указателя мыши на компоненте и т. д. | Возникает во всех компонентах |

# Лабораторная работа №10

## Тема: Создание программ с графическим интерфейсом. Классы основных компонентов.

**Цели работы:** научиться использовать компоненты графического интерфейса и обрабатывать события, связанные с их работой.

**Постановка задачи**

Создать графический интерфейс с различными компонентами ввода и вывода информации и обработать события, происходящие с компонентами и окном.

**Используемый метод**

Классы компонентов библиотеки AWT

|  |  |
| --- | --- |
| Класс | Компонент |
| Button | Стандартная для данной операционной системы кнопка |
| Checkbox | Флажок с квадратной областью для установки (активное состояние) и сброса (неактивное состояние) |
| CheckboxGroup | Группа переключателей |
| Choice | Выпадающий список |
| List | Список с полосой прокрутки, в котором можно выбирать сразу несколько пунктов |
| TextComponent | Абстрактный класс |
| TextField | Текстовое поле (одна строка) |
| TextArea | Текстовая область (несколько строк) |
| Scrollbar | Полоса прокрутки (и ползунок) |
| Panel | Панель (контейнер) – невидимый компонент интерфейса |
| ScrollPane | Контейнер для больших компонентов. Содержит только один компонент |
| Window | Пустое окно |
| Frame | Окно со строкой заголовка |
| Dialog | Диалоговое окно (обычно окно фиксированного размера) |
| FileDialog | Стандартное окно выбора файла |
| Canvas | Пустой компонент |

Некоторые классы компонентов библиотеки Swing

|  |  |
| --- | --- |
| **Класс** | **Описание** |
| AbstractButton | Абстрактный суперкласс для классов-компонентов кнопок в библиотеке Swing |
| ButtonGroup | Класс для создания групп кнопок, например групп переключателей |
| ImageIcon | Класс для инкапсуляции изображения (значка). Конструктору класса при создании объекта передается текстовая строка с именем файла изображения или url-адрес соответствующего ресурса |
| JApplet | Класс для работы с апплетами. Расширяет класс Applet |
| JButton | Класс для работы с элементами-кнопками |
| JCheckBox | Класс для работы с элементами-флажками |
| JComboBox | Класс для работы с элементами-раскрывающимися текстовыми списками |
| JLabel | Класс для работы с текстовыми метками |
| JRadioButton | Класс для работы с переключателями |
| JScrollPane | Класс для работ с полосами прокрутки |
| JTabbedPane | Класс для работы с панелями, содержащими вкладки |
| JTable | Класс для работы с таблицами |
| JTextField | Класс для работы с текстовыми полями |
| JTree | Класс для работы с деревьями |

# Лабораторная работа №11

## Тема: Работа с сетью, многопоточные приложения. Классы для организации работы с сетями.

**Цели работы:** научиться использовать классы для работы в сети TCP/IP и работать с потоками информации в сети.

**Постановка задачи**

**Используемый метод**

**Поточная модель Java**

В Java, поточная модель реализуется посредством иерархии классов, описывающих потоки. Основу этой иерархии составляют класс Thread и интерфейс Runnable. Для создания потока необходимо либо расширить класс Thread, либо реализовать интерфейс Runnable. При этом класс Thread инкапсулирует поток исполнения.

При запуске Java-программы начинает выполняться главный поток. Особенность главного потока состоит в том, что в нем порождаются все дочерние потоки. Главный поток отождествляется с программой. Программа начинается с выполнения главного потока и должна завершаться с завершением главного потока.

Методы класса Thread

|  |  |
| --- | --- |
| Метод | Описание |
| currentThread() | Методом в качестве результата возвращается ссылка на поток, из которого вызывается метод |
| getName() | Метод в качестве результата возвращает имя потока (текстовую строку) |
| getPriority() | Метод в качестве результата возвращает приоритет потока (целое число) |
| isAlive() | Метод позволяет выяснить, используется поток или нет |
| join() | Методом дается команда ожидания завершения потока |
| run() | Метод определения точки входа в поток |
| sleep() | Метод для приостановки потока на определенный промежуток времени (аргумент метода, в миллисекундах) |
| start() | Метод для запуска потока путем вызова его метода run() |

Метод имеет сигнатуру public static Thread currenThread(). Пример использования метода currentThread() для получения доступа к главному потоку приведен ниже:

*class CurrentThreadDemo{*

*public static void main(String args[]){*

*// Объектная переменная t класса Thread:*

*Thread t;*

*// Объектная переменная t ссылается на главный поток программы:*

*t=Thread.currentThread();*

*// Информация о потоке:*

*System.out.println("Активный поток: "+t);*

*// Потоку присвоено (изменено) имя:*

*t.setName("Самый главный поток");*

*// Информация о потоке:*

*System.out.println("После изменения имени: "+t);*

*try{*

*for(int n=5;n>0;n--){*

*System.out.println(n);*

*// Приостановка потока:*

*Thread.sleep(1000);*

*}*

*}*

*catch(InterruptedException e){*

*// Обработка исключения "прерывание потока"*

*System.out.println("Поток завершен!");*

*}*

*}*

*}*

**Создание потока**

Общая последовательность действий при создании нового потока путем реализации интерфейса Runnable следующая.

1. Определяется класс, реализующий интерфейс Runnable. В этом классе определяется метод run().

2. В этом классе создается объект класса Thread. Конструктору класса передается два аргумента: объект класса, реализующего интерфейс Runnable, и текстовая строка – название потока.

3. Для запуска потока из объекта класса Thread вызывается метод start().

Часто процесс создания нового потока реализуется расширением интерфейса Runnable.

1. При расширении интерфейса Runnable в соответствующем классе (для удобства назовем его внешним) не только определяется метод run(), но и описывается поле – объект класса Thread.

2. Создание объекта класса Thread (объекта потока), ссылка на который присваивается полю Thread, выполняется в конструкторе внешнего класса. При создании этого объекта вызывается конструктор класса Thread, первым аргументом которому передается ссылка this. Таким образом, одновременно с созданием объекта внешнего класса создается и объект потока, причем объект потока создается на основе объекта внешнего класса.

3. В конструкторе внешнего класса после команды создания объекта потока (объекта класса Thread) из этого потока вызывается метод start(). Это приводит к запуску потока.

4. Для создания и запуска потока в главном методе программы создается объект описанного ранее класса, расширяющего интерфейс Runnable. Поскольку для запуска потока достаточно самого факта создания объекта, нередко этот создаваемый объект является анонимным, то есть ссылка на него ни в какие объектные переменные не записывается.

Создание потока расширением класса Thread:

*// Класс NewThread расширяет класс Thread:*

*class NewThread extends Thread{*

*// Конструктор класса:*

*NewThread(){*

*// Вызов конструктора класса Thread:*

*super("Новый поток");*

*// Вывод сведений о потоке:*

*System.out.println("Дочерний поток: "+this);*

*// Запуск потока на выполнение:*

*start();*

*}*

*// Переопределение метода run():*

*public void run(){*

*try{*

*for(int i=5;i>0;i--){*

*System.out.println("Дочерний поток: "+i);*

*// Приостановка потока:*

*Thread.sleep(500);*

*}*

*}*

*// Обработка исключения прерывания потока:*

*catch(InterruptedException e){*

*System.out.println("Прерывание дочернего потока!");*

*}*

*System.out.println("Завершение дочернего потока!");*

*}*

*}*

*class ExtendsThreadDemo{*

*public static void main(String args[]){*

*new NewThread();*

*try{*

*for(int i=5;i>0;i--){*

*System.out.println("Главный поток: "+i\*100);*

*Thread.sleep(1000);*

*}*

*}*

*catch(InterruptedException e){*

*System.out.println("Прерывание главного потока!");*

*}*

*System.out.println("Завершение главного потока!");*

*}*

*}*

В результате выполнения программы получаем следующее:

*Дочерний поток: Thread[Новый поток,5,main]*

*Главный поток: 500*

*Дочерний поток: 5*

*Дочерний поток: 4*

*Дочерний поток: 3*

*Главный поток: 400*

*Дочерний поток: 2*

*Дочерний поток: 1*

*Главный поток: 300*

*Завершение дочернего потока!*

*Главный поток: 200*

*Главный поток: 100*

*Завершение главного потока!*

**Создание нескольких потоков**

*// Импорт класса Date:*

*import java.util.Date;*

*// Класс NewThread наследует класс Thread:*

*class NewThread extends Thread{*

*// Параметры потока (название, время задержки, количество итераций):*

*private String name;*

*private int time;*

*private int count;*

*// Конструктор:*

*NewThread(String name,int time,int count){*

*super(name);*

*this.name=name;*

*System.out.print("Создан новый поток: "+name+". ");*

*// Отображение даты и времени:*

*System.out.println("Время: "+new Date()+".");*

*this.time=time;*

*this.count=count;*

*// Запуск потока:*

*start();*

*}*

*// Переопределение метода run():*

*public void run(){*

*try{*

*for(int i=1;i<=count;i++){*

*System.out.print("Поток: "+name+". Сообщение "+i+" из "+count+". ");*

*// Отображение даты и времени:*

*System.out.println("Время: "+new Date()+".");*

*// Приостановка потока:*

*Thread.sleep(time);*

*}*

*}*

*catch(InterruptedException e){*

*System.out.println("Прерывание потока"+name);*

*}*

*finally{*

*System.out.print("Поток \""+name+"\" работу завершил! ");*

*// Отображение даты и времени:*

*System.out.println("Время: "+new Date()+".");*

*}*

*}*

*}*

*class MultiThreadDemo{*

*// Исключение InterruptedException в методе main() не обрабатывается:*

*public static void main(String args[]) throws InterruptedException{*

*System.out.print("Начало работы! ");*

*// Отображение даты и времени:*

*System.out.println("Время: "+new Date()+".");*

*// Создание трех дочерних потоков:*

*new NewThread("Красный",5000,5);*

*new NewThread("Желтый",6000,4);*

*new NewThread("Зеленый",7000,3);*

*// Приостановка главного потока:*

*Thread.sleep(30000);*

*System.out.print("Работа программы завершена! ");*

*// Отображение даты и времени:*

*System.out.println("Время: "+new Date()+".");*

*}*

*}*

Результат выполнения программы может иметь следующий вид:

*Начало работы! Время: Sat Sep 19 23:39:36 EEST 2009.*

*Создан новый поток: Красный. Время: Sat Sep 19 23:39:36 EEST 2009.*

*Создан новый поток: Желтый. Время: Sat Sep 19 23:39:36 EEST 2009.*

*Создан новый поток: Зеленый. Время: Sat Sep 19 23:39:36 EEST 2009.*

*Поток: Красный. Сообщение 1 из 5. Время: Sat Sep 19 23:39:36 EEST 2009.*

*Поток: Желтый. Сообщение 1 из 4. Время: Sat Sep 19 23:39:36 EEST 2009.*

*Поток: Зеленый. Сообщение 1 из 3. Время: Sat Sep 19 23:39:36 EEST 2009.*

*Поток: Красный. Сообщение 2 из 5. Время: Sat Sep 19 23:39:41 EEST 2009.*

*Поток: Желтый. Сообщение 2 из 4. Время: Sat Sep 19 23:39:42 EEST 2009.*

*Поток: Зеленый. Сообщение 2 из 3. Время: Sat Sep 19 23:39:43 EEST 2009.*

*Поток: Красный. Сообщение 3 из 5. Время: Sat Sep 19 23:39:46 EEST 2009.*

*Поток: Желтый. Сообщение 3 из 4. Время: Sat Sep 19 23:39:48 EEST 2009.*

*Поток: Зеленый. Сообщение 3 из 3. Время: Sat Sep 19 23:39:50 EEST 2009.*

*Поток: Красный. Сообщение 4 из 5. Время: Sat Sep 19 23:39:51 EEST 2009.*

*Поток: Желтый. Сообщение 4 из 4. Время: Sat Sep 19 23:39:54 EEST 2009.*

*Поток: Красный. Сообщение 5 из 5. Время: Sat Sep 19 23:39:56 EEST 2009.*

*Поток "Зеленый" работу завершил! Время: Sat Sep 19 23:39:57 EEST 2009.*

*Поток "Желтый" работу завершил! Время: Sat Sep 19 23:40:00 EEST 2009.*

*Поток "Красный" работу завершил! Время: Sat Sep 19 23:40:01 EEST 2009.*

*Работа программы завершена! Время: Sat Sep 19 23:40:06 EEST 2009.*

**Синхронизация потоков**

Существует два способа создания синхронизированного кода:

- создание синхронизированных методов;

- создание синхронизированных блоков.

В обоих случаях используется ключевое слово synchronized. Если создается синхронизированный метод, ключевое слово synchronized указывается в его сигнатуре. При вызове синхронизированного метода потоком другие потоки на этом методе блокируются – они не смогут его вызвать, пока работу с методом не завершит первый вызвавший его поток. Можно синхронизировать объект в блоке команд. Для этого блок выделяется фигурными скобками, перед которыми указывается ключевое слово synchronized, а в скобках после этого слова – синхронизируемый объект. Пример программы с синхронизированным методом приведен ниже.

*class MySource{*

*// Синхронизированный метод:*

*synchronized void showName(String msg1,String msg2,int time){*

*try{*

*// Приостановка потока, из которого вызван метод:*

*Thread.sleep(time);*

*// Вывод значения поля msg1:*

*System.out.print(" Фамилия: "+msg1);*

*// Еще одна приостановка потока:*

*Thread.sleep(2\*time);*

*// Вывод значения поля msg2:*

*System.out.println(" Имя: "+msg2);*

*}*

*catch(InterruptedException e){*

*// Обработка исключения*

*System.out.println("Прерывание потока: "+e);*

*}*

*}*

*}*

*// Класс, реализующий интерфейс Runnable:*

*class MakeThread implements Runnable{*

*// Поле объекта потока:*

*Thread t;*

*// Поле-объект MySource:*

*MySource src;*

*// Текстовые поля:*

*String name;*

*String surname;*

*int time;*

*// Конструктор:*

*MakeThread(String s1,String s2,int time, MySource obj){*

*surname=s1;*

*name=s2;*

*src=obj;*

*this.time=time;*

*// Создание потока:*

*t=new Thread(this);*

*// Запуск потока:*

*t.start();*

*}*

*// Определение метода run():*

*public void run(){*

*src.showName(surname,name,time);*

*}*

*}*

*class SynchThreads{*

*public static void main(String args[]){*

*// Объект "ресурса":*

*MySource obj=new MySource();*

*// Создание потоков:*

*MakeThread fellow1=new MakeThread("Иванов","Иван",1000,obj);*

*MakeThread fellow2=new MakeThread("Петров","Петр",450,obj);*

*MakeThread fellow3=new MakeThread("Сидоров","Сидор",1450,obj);*

*try{*

*// Ожидать завершения потоков*

*fellow1.t.join();*

*fellow2.t.join();*

*fellow3.t.join();*

*}*

*catch(InterruptedException e){*

*// Обработка исключения*

*System.out.println("Прерывание потока: "+e);*

*}*

*}*

*}*

**Пакет Java.net**

Пакет java.net состоит из классов, позволяющих создавать соединения в сети TCP/IP и осуществлять передачу данных между машинами сети.

Пакет java.net состоит из следующих классов:

|  |  |
| --- | --- |
| Класс | Описание |
| InetAddress | Используется для инкапсуляции как числового IP-адресса, так и его доменного имени. |
| Socket | Клиентский сокет. Позволяет соединяться с серверными сокетами и инициализировать обмен. |
| ServerSocket | Серверный сокет. Является «слушателем», который ожидает подключения клиентов. |
| URL | Является интерфейсом для доступа к информации в сети при помощи URL. |
| URI | Используется для инкапсуляции универсального идентификатора ресурса (URI). |
| URLConnection | Предназначен для доступа к атрибутам удаленного ресурса. |
| HttpURLConnection | Обеспечивает поддержку http соединений. Является подклассом класса URLConnection. |
| DatagramSocket,  DatagramPacket | Реализуют дейтаграммы поверх UDP. |
| CookieHandler, CookieManager, HttpCookie | Управляют файлами cookie. |

Дейтаграммы – это порции информации, передаваемые между машинами.

Пример использования класса URL:

*import java.net.\*;*

*class URLDemo{*

*publick static void main(string args[]) throws MalformedURLException {*

*URL hp = new URL(“http://www.ugrasu.ru/universitet/”);*

*System.out.println(“протокол: ”+hp.getProtocol());*

*System.out.println(“Порт: ”+hp.getPort());*

*System.out.println(“Хост: ”+hp.getHost());*

*System.out.println(“Файл: ”+hp.getFile());*

*System.out.println(“Целиком: ”+hp.toExternalForm ());*

*}*

*}*

Результат:

*протокол: http*

*Порт: -1*

*Хост: www.ugrasu.ru*

*Файл: /universitet/*

*Целиком: http://www.ugrasu.ru/universitet/*

# Лабораторная работа №12

## Тема: Работа с базами данных. Доступ к базам данных из программ на языке Java. JDBC-драйверы. Регистрация драйверов. Основные приемы программирования JDBC. Выполнение запросов. Построение модели данных для JTable на основе результирующего набора данных.

**Цели работы:** научиться работать с базами данных, отправлять запросы к БД и отображать полученные данные в объектах оконного приложения.

**Постановка задачи**

Подключить драйвер для работы с БД.

Получить доступ к БД.

Отправить запросы к БД и отобразить результаты запросов.

**Используемый метод**

JDBC – набор интерфейсов и классов, предназначенный для работы с базами данных. Эти интерфейсы и классы составили пакет java.sql, а также пакет javax.sql и его подпакеты, входящие в Java SE.

Кроме классов с методами доступа к базам данных для каждой СУБД необходим драйвер JDBC – промежуточная программа, реализующая интерфейсы JDBC методами данной СУБД.

Существуют четыре типа драйверов JDBC:

- драйвер, реализующий методы JDBC вызовами функций ODBC. Это так называемый *мост* (bridge) JDBC–ODBC. Непосредственную связь с базой при этом осуществляет драйвер ODBC, который должен быть установлен на той машине, на которой работает программа;

- драйвер, реализующий методы JDBC вызовами функций API самой СУБД. В этом случае на машине должен быть установлен клиент СУБД;

- драйвер, реализующий методы JDBC вызовами функций сетевого протокола, независимого от СУБД, например HTTP. Этот протокол должен быть, затем, реализован средствами СУБД;

- драйвер, реализующий методы JDBC вызовами функций сетевого протокола СУБД.

Перед обращением к базе данных следует установить нужный драйвер, например мост JDBC–ODBC:

*try{*

*Class dr = sun.jdbc.odbc.JdbcOdbcDriver.class;*

*}*

*catch(ClassNotFoundException e){*

*System.err.println("JDBC-ODBC bridge not found " + e);*

*}*

Далее необходимо связаться с базой данных. Методы связи описаны в интерфейсе Connection. Экземпляр класса, реализующего этот интерфейс, можно получить одним из статических методов getConnection() класса DriverManager, например:

*String url = "jdbc:odbc:mydb";*

*String login = "admin";*

*String password = "1nF4vb";*

*Connection con = DriverManager.getConnection(url, login, password);*

Обратите внимание на то, как формируется адрес базы данных url. Он начинается со строки "jdbc:", потом записывается подпротокол (subprotocol), в данном примере используется мост JDBC–ODBC, поэтому записывается "odbc:". Далее указывается адрес (subname) по правилам подпротокола, здесь просто имя локальной базы "mydb". Второй и третий аргументы – это имя и пароль для соединения с базой данных.

Связавшись с базой данных, можно посылать запросы. Запрос хранится в объекте, реализующем интерфейс Statement. Этот объект создается методом createStatement(), описанным в интерфейсе Connection. Например:

*Statement st = con.createStatement();*

Затем запрос заносится в этот объект методом execute() и потом выполняется методом getResultSet(). В простых случаях это можно сделать одним методом executeQuery(), например:

*ResultSet rs = st.executeQuery("SELECT name, code FROM tbl1");*

Здесь из таблицы tbl1 извлекается содержимое двух столбцов name и code и заносится в объект rs класса, реализующего интерфейс ResultSet.

SQL-операторы INSERT, UPDATE, DELETE, CREATE TABLE и др. в простых случаях выполняются методом executeUpdate().

Далее методом next() необходимо перебрать элементы объекта rs – строки полученной выборки – и извлечь данные многочисленными методами getX\_\_() интерфейса ResultSet, например:

*while (rs.next()){*

*emp[i] = rs.getString("name");*

*num[i] = rs.getInt("code");*

*i++;*

*}*

Ниже приведен пример запроса к базе Oracle через драйвер Oracle Thin. Апплет выводит в окно браузера четыре поля ввода для адреса базы, имени и пароля пользователя, и запроса. По умолчанию формируется запрос к стартовой базе Oracle, расположенной на локальном компьютере. Результат запроса выводится в окно браузера.

*import java.awt.\*;*

*import java.awt.event.\*;*

*import java.applet.\*;*

*import java.util.\*;*

*import java.sql.\*;*

*public class JdbcApplet extends Applet implements ActionListener, Runnable{*

*private TextField tf1, tf2, tf3;*

*private TextArea ta;*

*private Button b1, b2;*

*private String url = "jdbc:oracle:thin:@localhost:1521:ORCL",*

*login = "scott",*

*password = "tiger",*

*query = "SELECT \* FROM dept";*

*private Thread th;*

*private Vector results;*

*public void init(){*

*setBackground(Color.white);*

*try{*

*DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());*

*}*

*catch(SQLException e){*

*System.err.println(e);*

*}*

*setLayout(null);*

*setFont(new Font("Serif", Font.PLAIN, 14));*

*Label l1 = new Label("URL базы:", Label.RIGHT);*

*l1.setBounds(20, 30, 70, 25); add(l1);*

*Label l2 = new Label("Имя:", Label.RIGHT);*

*l2.setBounds(20, 60, 70, 25); add(l2);*

*Label l3 = new Label("Пароль:", Label.RIGHT);*

*l3.setBounds(20, 90, 70, 25); add(l3);*

*tf1 = new TextField(url, 30);*

*tf1.setBounds(100, 30, 280, 25); add(tf1);*

*tf2 = new TextField(login, 30);*

*tf2.setBounds(100, 60, 280, 25); add(tf2);*

*tf3 = new TextField(password, 30);*

*tf3.setBounds(100, 90, 280, 25); add(tf3);*

*tf3.setEchoChar('\*');*

*Label l4 = new Label("Запрос:", Label.LEFT);*

*l4.setBounds(10, 120, 70, 25); add(l4);*

*ta = new TextArea(query, 5, 50, TextArea.SCROLLBARS\_NONE);*

*ta.setBounds(10, 150, 370, 100); add(ta);*

*Button b1 = new Button("Отправить");*

*b1.setBounds(280, 260, 100, 30); add(b1);*

*b1.addActionListener(this);*

*}*

*public void actionPerformed(ActionEvent ae){*

*url = tf1.getText();*

*login = tf2.getText();*

*password = tf3.getText();*

*query = ta.getText();*

*if (th == null){*

*th = new Thread(this);*

*th.start();*

*}*

*}*

*public void run(){*

*try{*

*Connection con = DriverManager.getConnection(url, login, password);*

*Statement st = con.createStatement();*

*ResultSet rs = st.executeQuery(query);*

*ResultSetMetaData rsmd = rs.getMetaData();*

*// Узнаем число столбцов*

*int n = rsmd.getColumnCount();*

*results = new Vector();*

*while (rs.next()){*

*String s = " ";*

*// Номера столбцов начинаются с 1!*

*for (int i = 1; i <= n; i++)*

*s += " " + rs.getObject(i);*

*results.addElement(s);*

*}*

*rs.close();*

*st.close();*

*con.close();*

*repaint();*

*}*

*catch(Exception e){*

*System.err.println(e);*

*}*

*repaint();*

*}*

*public void paint(Graphics g){*

*if (results == null){*

*g.drawString("Can't execute the query", 5, 30);*

*return;*

*}*

*int y = 30, n = results.size();*

*for (int i = 0; i < n; i++)*

*g.drawString((String)results.elementAt(i), 5, y += 20);*

*}*

*}*

**Пакет java.sql**

Пакет java.sql содержит классы и интерфейсы JDBC.

Имеются следующие типы интерфейсов:

Driver - определяет методы как драйвера, так и менеджера драйверов JDBC;

Connection - выбирает методы для создания сеанса связи с базой данных;

DatabaseMetaData - устанавливает методы для получения сведений о базе данных, с которой работает приложение;

Statement - методы для передачи SQL-запроса базе данных;

PreparedStatement - аналогичен Statement, но сохраняет бинарные планы обслуживания запроса для дальнейшего использования;

CallableStatement - обеспечивает выполнение хранимых в базе данных SQL-процедур;

ResultSet - методы для обработки результата обращения к базе данных;

ResultSetMetaData - методы получения информации о типах данных, хранящихся в ResultSet;

В состав пакета входят следующие классы:

Date - спецификация SQL-типа DATE;

Numeric - спецификация SQL-типов DECIMAL и NUMERIC;

Time - спецификация SQL-типа TIME;

Timestamp - спецификация SQL-типа TIMESTAMP;

DriverManager - управление JDBC-драйверами;

DriverPropertyInfo - информация о JDBC-драйверах;

Types - управление дополнительными SQL-типами.

Кроме того, в состав пакета java.sql входят обработчики исключительных ситуаций DataTruncation, SQLException и SQLWarning.

**Класс JTable**

Входит в пакет javax.swing.table.

Объявляется:

*JTable (Object данные[][], Object заголовокСтолбцов[])*

Для создания простой таблицы класса JTable, отображающей данные, выполните следующие действия:

1. Создайте экземпляр класса JTable.
2. Создайте экземпляр класса JScrollPane, определяя таблицу в качестве объекта прокрутки.
3. Добавьте таблицу в панель с полосами прокрутки.
4. Добавьте панель с полосами прокрутки в панель содержимого.

Пример:

*// Инициализируем заголовок столбцов*

*String[] colHeads = {“Имя”, “Зарплата”, “Номер”};*

*//Инициализируем данные*

*Object[][] data = {*

*{“Иван”, “23000”,”95”},*

*{“Александр”,”15000”,”211”},*

*{“Андрей”,”11000”,”1455”},*

*{“Матвей”,”36000”,”13”},*

*{“Григорий”,”100”,”122”},*

*{“Дмитрий”,”100000”,”233”},*

*};*

*// создаем таблицу*

*JTable table = new JTable(data, colHeads);*

*//добавляем таблицу в панель с полосами прокрутки*

*JScrollPane jsp = new JScrollPane(table);*

*//добавляем панель с полосами прокрутки в панель содержимого*

*add(jsp);*

# Лабораторная работа №13

## Тема: Функциональная и компонентная декомпозиция проекта. Коллективная разработка проекта.

**Цели работы:** научиться работать в команде, интегрируя два и более программных решения в один проект.

**Постановка задачи**

Разбиться на группы по 2-3 человека (в зависимости от исполнителей в задаче). Каждый исполнитель выполняет свою часть проекта.

**Пример задачи**

Создать графический интерфейс написания запросов SELECT к БД. Запрос состоит из полей, которые надо получить и таблицы, содержащей эти поля. Результат запроса вывести в объект класса JTable.

Исполнитель\_1: создает графический интерфейс и пишет обработчики событий компонентов интерфейса.

Исполнитель\_2: создает соединение с БД, отправляет запрос и выводит в объект JTable результат.

Исполнитель\_3: пишет обработчики исключений и пишет отчет по проекту.

# Контрольная работа №2

**Задачи**

1. Создать окно с полем ввода информации, полем вывода и кнопкой. Требуется: по нажатию кнопки, вывести информацию из поля ввода в поле вывода.
2. Создать окно с полем выбора файла, полем вывода и кнопкой. Требуется: по нажатию кнопки, вывести информацию из файла в поле вывода.
3. Создать окно с полем ввода (URL), полем вывода и кнопкой. Требуется: по нажатию кнопки, вывести информацию об ресурсе в поле вывода.
4. Дано: таблица в БД, содержащая возраст студентов. Требуется: вывести всех студентов, возраст которых меньше введенного с клавиатуры целого числа.

**Задачи повышенной трудности**

1. Дано: таблица БД содержит информацию (id, ФИО, возраст, пол, инвестиции, прибыль). Создать окно с полем ввода, полем вывода (JTable) и кнопкой. Требуется: вывести все строки ФИО, инвестиции и прибыль, где ФИО начинается с символа, введенного в поле ввода.