Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный университет
имени П.О. Сухого»

Кафедра «Информатика»

В.Н. Шибеко

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ JAVA

Практикум по выполнению лабораторных работ по дисциплине
«Объектно-ориентированное программирование»
для студентов специальности 1-40 04 01
«Информатика и технологии программирования»
дневной формы обучения

Гомель 2020

УДК 004.438
ББК 32.973.2
П80

Рецензенты доцент кафедры «Информационные технологии»
ГГТУ им. П.О. Сухого, к.т.н., доцент В.И. Токочаков

Шибеко В.Н.
Программирование на языке Java: практикум / В.Н. Шибеко; М-во образования Респ. Беларусь, Гомель. гос. техн. ун-т им. П. О. Сухого. – Гомель: ГГТУ им. П. О. Сухого, 2020. – xx с.

Излагаются основные темы, изучаемые в курсе «Объектно-ориентированное программирование». Приводятся сведения об объектном подходе к разработке на основе языка программирования Java.

Для студентов специальности 1-40 04 01 «Информатика и технологии программирования»

УДК 004.438
ББК 32.973.2

Оглавление

ВВЕДЕНИЕ

Целью методического пособия является формирование практических навыков программирования на популярном языке программирования Java. Годом рождения» Java считается 1991, когда был выпущен портативный пульт с сенсорным экраном. Сначала Java предназначался для программирования бытовых электронных устройств, таких как сотовые телефоны и другие мобильные устройства. Потом Java стала применяться для программирования браузеров, серверов, полноценных приложения. Таким образом, язык нашёл свое применение не только во встраивыемых устройствах, но и в интернете. Причиной этого являются принципы и особенности языка, заложенные в него изначально. Это:

Простота. Синтаксис языка Java изначально представлял собой упрощённый вариант синтаксиса языка C++.

Объектная-ориентированность. В центре внимания находятся объекты и их типы. Java более строгий объектно-ориентированный язык, чем C++ и Object Pascal.

Распределённость. В Java изначально присутствовали средства создания распределённых приложений. Причём это не только низкоуровневые средства работы с сокетами (на основе протоколов TCP и UDP), но и API для работы с протоколами более высоких уровней (например, класс URL позволяет передавать данные по протоколу HTTP).

Надежность. Значительное внимание в Java уделяется раннему обнаружению возможных ошибок, контролю в процессе выполнения программы, а также устранению ситуаций, которые могут вызвать ошибки.

Безопасность. Язык Java предназначен для создания программ, работающих в сети. По этой причине большое внимание при его создании было уделено безопасности: в язык встроена настраиваемая система обеспечения безопасности при выполнении кода. Например, эта система предотвращает намеренное переполнение стека выполняемой программы (один из распространенных способов атаки, используемых вирусами), повреждение участков памяти, находящихся за пределами пространства, выделенного процессу, несанкционированное чтение файлов и их модификация.

Независимость от архитектуры компьютера (также называемая «кросс-платформенность»). Компилятор генерирует объектный файл, формат которого не зависит от архитектуры компьютера. Скомпилированная программа может выполняться на любых процессорах, для ее работы необходима лишь исполняющая система Java.

Переносимость. В отличии от языков C и C++, ни один из аспектов спецификации Java не зависит от реализации конкретной исполняющей системы.

И размер основных типов данных, и арифметические операции над ними строго определены. Например, тип int в языке Java всегда означает 32-х разрядное целое число. В языках C и C++ тип int может означать как 16-разрядное, так и 32-х разрядное целое число. Фиксированный размер числовых типов позволяет избежать многих неприятностей, связанных с выполнением программ на разных компьютерах. Бинарные данные хранятся и передаются в неизменном формате, что также позволяет избежать недоразумений, связанных с разным порядком следования байтов на разных платформах. Строковые данные сохраняются в стандартном формате Unicode.

Пособие состоит из 7 разделов. Каждый раздел содержит краткие теоретческие сведения и пример задания на лабораторную работу. При выполнении лабораторной работы вариант задания выдает преподаватель.

Первый раздел является вводным в программирование на язske Java. Рассматриваются примитивные типы данных, операторы и массивы. Второй раздел содержит вопросы обработки строк. Третий раздел является одним из основных и вводит в понятие также класса, описывает его элементы, называемые членами класса, определяет модификаторы доступа и другие понятия, необходимые для работы с классом в Java. Четвертый раздел описывает коллекции Java. Пятый раздел кратко описывает реализацию принципов ООП в Java, включая реалицию наследования, которое базируется на таких понятиях как:

- суперкласс, подкласс;
- абстрактный класс;
- интерфейс.

Шестой раздел касается вопросов постоянства или хранения данных во внешних источниках. В рамках темы рассмотрены вопросы потоков, сериализации. Последний раздел является завершающим и касается темы модульности, основой реализации которого в Java явдяется понятие пакета.

1. ОСНОВЫ ЯЗЫКА

1.1. Типы данных Java

Java — строго типизированный язык[1]:

- каждая переменная обладает типом, каждое выражение имеет тип, и каждый тип строго определен.
- все присваивания, как явные, так и посредством передачи параметров в вызовах методов, проверяются на соответствие типов.

Компилятор Java проверяет все выражения и параметры на предмет совместимости типов. Любые несоответствия типов являются ошибками, которые должны быть исправлены до завершения компиляции класса.
Типы данных в языке Java, делятся на три группы:

- примитивные типы (primitive types),
- ссылочные типы (reference types),
- тип – null.

Переменную типа **null** создать невозможно, но можно присвоить значение null (только) ссылочному типу данных.

Примитивные типы данных Java. Существует восемь примитивных типов данных в Java:

- Числовые данные (numeric types):
  - ✓ Целые типы (integral types) – **byte**, **short**, **int**, **long**, **char**
  - ✓ Вещественные типы (floating-point types) – **float**, **double**
- Логический тип – boolean

Примитивные типы хранят значение. В Java возможны преобразования между целыми значениями и значениями с плавающей точкой. Кроме того, можно преобразовывать значения целых типов и типов с плавающей точкой в значения типа char и наоборот, поскольку каждый символ соответствует цифре в кодировке Unicode. Фактически тип boolean является единственным примитивным типом в Java, который нельзя преобразовать в другой примитивный тип. Кроме того, любой другой примитивный тип нельзя преобразовать в boolean.

Преобразование типов. Преобразование типов в Java бывает двух видов: неявное и явное. Неявное преобразование типов выполняется в случае если выполняются условия:

- Оба типа совместимы
- Длина целевого типа больше или равна длине исходного типа

Во всех остальных случаях должно использоваться явное преобразование типов.

Ссылочные типы данных Java. Существует четыре типа ссылочных данных в Java:

- Классы (class types)
- Интерфейсы (interface types)
- Переменные типов (type variables)
- Массивы (array types)

Ссылочные типы хранят ссылку на объект, или же тип данных **null**, то есть нулевую (пустую) ссылку. В Java все является объектом, за исключением примитивных типов. Примитивные типы передаются по значению, а ссылочные по ссылке.

1.2. Операторы

Операторы сравнения[3]. *С*остоят из операторов равенства, которые проверяют равенство или неравенство значений (== , **!**=), и операторов отношения (<,>,<=,>=), используемых с упорядоченными типами (числами и символами) при проверке соотношения больше/меньше. Операторы обоих типов возвращают значение типа boolean, поэтому их обычно используют с условными операторами if и циклами while и for для выбора ветви или проверки условия выполнения цикла.

Операторы отношения. В Java предусмотрены следующие операторы отношения:

- меньше (<), возвращает true, если первый операнд меньше второго;
- меньше или равно (<=), возвращает true, если первый операнд меньше или равен второму;
- больше (>), возвращает true, если первый операнд больше второго;
- больше или равно (>=), возвращает true, если первый операнд больше или равен второму.

Булевы операторы.

- *Условное И (&&)*. Данный оператор выполняет логическую операцию **И** над операндами. Он возвращает **true** тогда и только тогда, когда оба операнда истинны. Если один или оба операнда ложны, он возвращает **false**. Например:
- *Условное ИЛИ (||)*. Данный оператор выполняет логическую операцию **ИЛИ** на двух операндах типа boolean. Он возвращает **true**, если один или оба операнда истинны. Если оба операнда ложны, он возвращает **false**.

- *Логическое НЕ (!)*. Этот унарный оператор меняет boolean значение операнда. Он возвращает **false**, если применяется к **true** значению, и **true**, если задано **false** значение.
- *Логическое И (&)*. С операндами типа boolean поведение оператора **&** аналогично поведению оператора **&&**, но он всегда вычисляет оба операнда, каким бы ни было значение первого операнда.
- *Логическое ИЛИ (|)*. Данный оператор выполняет логическую операцию **ИЛИ** над двумя операндами типа boolean.
- *Логическое исключающее ИЛИ (^)*. Для операндов типа boolean данный оператор вычисляет **исключающее ИЛИ**. Он возвращает **true**, если только один из двух операндов истинен.

Особенности выполнения целочисленных операций. Результат арифметической операции имеет тип **int**, кроме того случая, когда один из операндов типа **long**. В этом случае результат будет типа **long**. Перед выполнением арифметической операции всегда происходит повышение (*promotion*) типов **byte**, **short**, **char**. Они преобразуются в тип **int**, а может быть, и в тип **long**, если другой операнд типа **long**. Операнд типа **int** повышается до типа **long**, если другой операнд типа **long**. При делении на ноль возникает исключительная ситуация, которая при отсутствии обработки этой исключительной ситуации (ArithmeticException) приводит к аварийному завершению работы программы. Следует иметь в виду, что переполнение не является исключительной ситуацией, лишние старшие биты просто отбрасываются.

Особенности выполнения операций с вещественными типами. Вещественные числа в Java представлены типами данных float и double. Количество бит отведенные под представление этих чисел смотрите в таблице ниже.

Таблица 1.1

Типы данных и размеры занимаемой ими памяти

Тип	**По умолчанию**	**Размер**	**Диапазон**	**Классы -обертки *(wrapper classes)***
float	0.0	32 bits	от 1.4E−45 до 3.4028235E+38	Float
double	0.0	64 bits	от 4.9E−324 до 1.7976931348623157E+308	Double

К обычным вещественным числам добавляются еще четыре значения:

- положительная бесконечность, выражаемая константой POSITIVE_INFINITY и возникающая при переполнении положительного значения, например в результате операции умножения 3.0*6e307 или при делении на нуль;

- отрицательная бесконечность NEGATIVE_INFINITY, возникающая при переполнении отрицательного значения, например в результате операции умножения -3.0*6e307 или при делении на нуль отрицательного числа;
- "не число", записываемое константой **NaN** (Not a Number) и возникающее, например, при умножении нуля на бесконечность.
- кроме того, стандарт различает положительный и отрицательный нуль, возникающий при делении на бесконечность соответствующего знака, хотя сравнение 0.0 == -0.0 дает в результате истину, true.

Операции с бесконечностями выполняются по обычным математическим правилам. Во всем остальном вещественные типы — это обычные вещественные значения, к которым применимы все арифметические операции и операции сравнения.

Все математические функции из библиотеки java.lang.Math работают с числами типа double.

Классы-обертки. Примитивные типы, в отличие от объектов, используются для таких значений из соображений производительности. Применение объектов для этих значений добавляет нежелательные накладные расходы, даже в случае простейших вычислений. Несмотря на то что примитивные типы обеспечивают выигрыш производительности, бывают случаи, когда вам может понадобиться объектное представление. Например, вы не можете передать в метод примитивный тип по ссылке. Кроме того, многие из стандартных структур данных, реализованных в Java, оперируют с объектами, что означает, что вы не можете применять эти структуры данных для сохранения примитивных типов. Чтобы справиться с такими (и подобными) ситуациями, Java предлагает обертки примитивных типов, которые представляют собой классы, помещающие примитивный тип в объект. Обертки для примитивных типов — это **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character** и **Boolean**. Эти классы предоставляют большой диапазон ***методов***, позволяющий в полной мере интегрировать примитивные типы в иерархию объектных типов Java.

Так как для каждого примитивного типа есть его не примитивный, а настоящий класс, с полями и методами, то для каждой такой пары возможно автоматическое преобразование в виде операций boxing и unboxing. Boxing и unboxing это преобразование примитивных типов в объекты обертки и обратно.

Переменные и константы в Java. В Java для обозначения констант служит ключевое слово **final**. Ключевое слово **final** означает, что присвоить данной переменной какое-нибудь значение можно лишь один раз, после чего изменить его уже нельзя. Рекомендуется использовать для именования констант прописные буквы, хоть это и не является обязательным, но такой стиль способствует удобочитаемости кода. Пример объявления константы:

final double CM_PER_INCH = 2.54;

Переменная — основной компонент хранения данных в Java-программе. Переменная определяется комбинацией идентификатора, типа и необязательного начального значения. Кроме того, все переменные имеют область определения, которая задает их видимость для других объектов и время существования. В Java все переменные должны быть объявлены до их использования. Объявить переменную можно в любом месте программы. Основная форма объявления переменных выглядит следующим образом:
тип идентификатор [=значение][, *идентификатор* [=значение] ...] , где

тип — это один из элементарных типов Java либо имя класса или интерфейса, а *идентификатор* — это имя переменной.
Переменной можно присвоить начальное значение (инициализировать ее), указывая знак равенства и значение. Следует помнить, что выражение инициализации должно возвращать значение того же (или совместимого) типа, который указан для переменной. Для объявления более одной переменной указанного типа можно использовать список с разделителями-запятыми:
int a, **b**, **c**; // объявление трех переменных типа int: a, b и c
int d = 3, **e**, **f** = 5;// объявление еще трех переменных типа int с инициализацией d и f
byte z = 22; // инициализация переменной z
double pi = 3.14159;// объявление приблизительного значения переменной pi
char x = 'x'; // присваивание значения 'x' переменной x
Обратите внимание, что некоторые объявления осуществляют инициализацию переменных.

1.3. Массивы

Одномерные массивы[1]. Одномерные массивы, по сути, представляют собой список однотипных переменных. Многомерные массивы представляют собой массивы массивов. При объявлении переменной многомерного массива для указания каждого дополнительного индекса используют отдельный набор квадратных скобок: **int twoD**[][] = **new int**[4][5];
В стандартной библиотеке JDK существует класс **java.util.Arrays** содержащий методы для работы с массивами. Основные методы работы с массивами:

- **Arrays.equals** сравнивает содержимое одномерных массивов;
- **Arrays.copyOf** копирует одномерный массив;
- **Arrays.toString** и **Arrays.deepToString** преобразуют одномерные и многомерные массивы в строку, более менее удобную для вывода на консоль.
- **Arrays.sort** сортирует одномерный массив, при этом метода для сортировки многомерных массивов в стандартной библиотеке Java нет;
- **Arrays.binarySearch** ищет в одномерном массиве заданное значение и возвращает его индекс;
- **Arrays.deepEquals** выполняет сравнение многомерных массивов.

Многомерные массивы[1]. В Java **многомерные массивы** представляют собой **массивы массивов**. При объявлении переменной многомерного массива для указания каждого дополнительного индекса используют отдельный набор квадратных скобок. Например, следующий код объявляет переменную двумерного массива x.

```
int x[][] = new int[2][3];
```

Этот оператор распределяет память для массива размерностью 2×3 и присваивает ссылку на него переменной x. Внутренне эта матрица реализована как массив массивов значений типа int. Каждая пара квадратных скобок представляет одно измерение, поэтому данный массив является двухмерным. Чтобы получить доступ к одиночному элементу int двухмерного массива, нужно указать два значения индекса, по одному для каждого измерения. На стадии создания массив ссылок заполняется значениями по умолчанию, то есть значениями **null**. Важно знать, что первый индекс содержит ссылки на строки, доступ к элементам которых дает второй индекс. Подобно одномерным массивам, многомерные массивы можно инициализировать при помощи массива-литерала. Но необходимо вложить массивы в другие массивы, применяя множество вложенных фигурных скобок. Пример ниже создает массив x:

```
int[][] x = { { 1, 2, 3 },
              { 4, 5, 6 }};
```

Для сравнения многомерных массивов есть метод **Arrays.deepEquals**. Например, объявленные ниже два массива при сравнении дают значение **true**.

```
int[][] x = { {1,2,3}, {4,5,6}};
int[][] y = { {1,2,3}, {4,5,7}};

boolean b = Arrays.deepEquals(x,y);
```

1.4. Лабораторная работа № 1

Цель: Знакомство со средой программирования и основными конструкциями языка Java.

Варианты заданий взять у преподавателя. Пример задания.

Задание 1. Табулирование функций. В соответствии с вариантом задания выполнить табулирование функции. Результаты разместить в массиве, обработать и выдать на печать.

Задание 2. В одномерном массиве, подготовленном в задании 2, вычислить:

- сумму положительных элементов массива;
- произведение элементов массива, расположенных между максимальным по модулю и минимальным по модулю элементами.

Упорядочить элементы массива по убыванию.

Задание 3. Дана квадратная матрица. Определить:

- сумму элементов в тех столбцах, которые не содержат отрицательных элементов;
- минимум среди сумм модулей элементов диагоналей, параллельных побочной диагонали матрицы.

2. СТРОКИ

2.1. Классы для работы со строками

Работа со строками является одной из самых распространённых задач в программировании[2]. Для этого предназначены классы:

- ✓ String.
- ✓ StringBuilder
- ✓ StringBuffer

Для простых операций со строками активно не использующими редактирование, применяется класс String. Для редактирования строк используются классы StringBuilder и StringBuffer. Класс StringBuilder идентичен StringBuffer за исключением одного важного отличия: он не синхронизирован, что означает, что он не является безопасным в отношении потоков. Выгода от применения StringBuilder связана с более высокой производительностью. Однако в случае разработки многопоточных программ вы должны использовать StringBuffer, а не StringBuilder.

2.2. Класс String.

Объект класса String является последовательностью символов Unicode в кодировке UTF-16. Символы в строках хранятся в кодировке Unicode, в которой каждый символ занимает два байта. Тип каждого символа — char. Эта последовательность может быть произвольной длины. Внутри класса String символы строки хранятся в простом массиве, но класс ревностно оберегает этот массив и доступ к нему возможен только через API класса, то есть через его методы. Это необходимо для поддержки идеи о том, что объекты класса String являются неизменяемыми (immutable). То есть, как только объект String создан, вы не можете изменить символы, образующие строку. Вы можете осуществлять любые операции над строками. Особенность в том, что всякий раз, когда вам нужна измененная версия существующей строки, создается новый объект String, включающий все модификации. Оригинальная строка остается неизменной. Этот подход используется потому, что фиксированная, неизменная строка может быть реализована более эффективно, нежели изменяемая. Неизменяемые строки имеют одно большое преимущество: компилятор может делать строки совместно используемыми. Для тех случаев, когда нужны модифицируемые строки, Java предлагает два выбора в виде двух классов: StringBuffer и StringBuilder. Оба класса содержат строки, которые могут быть изменены после того, как созданы. Класс StringBuilder введен в стандартную библиотеку Java, начиная с версии Java 5, для ускорения работы с текстом в одном процессе. В многопоточной среде вместо класса StringBuilder, не обеспечивающего синхронизацию, следует использовать класс StringBuffer. Есть два способа создания строки (объекта класса String):

- при помощи оператора присваивания (=)
- при помощи оператора **new**

Если строка создатся на основе оператора присваивания, то если существует два или более одинаковых строковых литерала, то для них в памяти выделается место только для одного (так как нет смысла хранить несколько одинаковых строк), но ссылки на этот объект могут быть присвоены любому количеству строковых переменных. Это позволяет уменьшить использование памяти виртуальной машины и в какой-то степени оптимизировать работу с часто используемыми строками (как вы помните строки в объектах класса String неизменяемые). Когда же строка создается при помощи оператора **new**, то для каждой строки, даже если они одинаковые, выделяется своя область памяти в куче (heap), так как оператор **new** создает новый объект и выделяет место для него в памяти виртуальной машины. Класс String предоставляет множесмтво конструкторов для создания строк, перечислим лишь некоторые:

- String() — создается объект с пустой строкой;
- String(String str) — конструктор копирования: из одного объекта создается его точная копия, поэтому данный конструктор используется редко;
- String(StringBuffer str) — преобразованная копия объекта класса StringBuffer;
- String(StringBuilder str) — преобразованная копия объекта класса StringBuilder;
- String(byte[] byteArray) — объект создается из массива байтов byteArray;
- String(char[] charArray) — объект создается из массива charArray символов Unicode;
- String(byte[] byteArray, int offset, int count) — объект создается из части массива байтов byteArray, начинающейся с индекса offset и содержащей count байтов;
- String(char[] charArray, int offset, int count) — то же, но массив состоит из символов Unicode;
- String(int[] intArray, int offset, int count) — то же, но массив состоит из символов Unicode, записанных в массив целого типа, что позволяет использовать символы Unicode, занимающие больше двух байтов;
- String(byte[] byteArray, String encoding) — символы, записанные в массиве байтов, задаются в Unicode-строке с учетом кодировки encoding;
- String(byte[] byteArray, int offset, int count, String encoding) — то же самое, но только для части массива;
- String(byte[] byteArray, Charset charset) — символы, записанные в массиве байтов, задаются в Unicode-строке с учетом кодировки, заданной объектом charset;
- String(byte[] byteArray, int offset, int count, Charset charset) — то же самое, но только для части массива.

Класс String содержит ряд методов для работы со строками, в первую очередь это операции сравнения, извлечения и поиска. Следует помнить, что в Java, все кроме примитивных типов являются ссылочными типами. Поэтому:

- если просто сравниваются две строки s1 и s2, то хотя их содержимое может быть равно, но при сравнении s1==s2 мы можем получить false, так как это разные объекты и при использовании оператора сравнения (==) для строк, равно как и для любых других объектов, происходит сравнение ссылок, а данном случае они разные.
- если же для сравнения использовать метод equals() класса String, то происходит сравнение содержимого объектов (строк). Так же надо учитывать что метод сравнения строк equals() регистро-зависимый, то есть строки "Hello" и "HELLO" для него будут разными.

Для того чтобы сравнивать строки независимо от регистра символов существует метод equalsIgnoreCase(). Для сравнения строк существуют еще методы compareTo(String *str*) и compareToIgnoreCase(String *str*). Данные методы сравнивают лексическое значение строки со строкой заданной в параметре *str*, определяя сортируются ли она в алфавитном порядке раньше или позже строки в параметре str. Методы возвращают целое число типа **int**, которое может быть меньше, равно или больше нуля. Данное число определяется по следующим правилам:

- Сравниваются символы данной строки *this* и строки *str* с одинаковым индексом (т.е. посимвольное сравнение строк), пока не встретятся различные символы с индексом, допустим, k или пока одна из строк не закончится.
- В первом случае возвращается значение this.charAt(k) — str.charAt(k) , т. е. разность кодировок Unicode первых несовпадающих символов.
- Во втором случае возвращается значение this.length() — str.length() , т. е. разность длин строк.
- Если строки совпадают, возвращается 0, т.е. в той же ситуации, в которой метод equals() возвращает true.
- Если значение str равно null, возникает исключительная ситуация.

Сравнение символов происходит строго по их расположению в таблице кодировки стандарта Unicode, то есть эти методы не учитывают алфавитное расположение символов в локальной кодировке.

Метод isEmpty() проверяет является ли строка пустой, то есть не содержащей ни одного символа. Не стоит это путать если переменная типа String содержит null ссылку, то есть не ссылается ни на какой объект.

Выбрать символ с индексом *ind* можно методом charAt(int *ind*). Если индекс *ind* отрицателен или больше, чем длина строки, возникает исключительная ситуация. Все символы строки в виде массива символов можно получить методом toCharArray(). Если же надо включить в массив символов *dst*, начиная с индекса *ind* массива, подстроку от индекса *begin* включительно до индекса *end*, то используйте метод getChars(int *begin*, int *end*, char[] *dst*, int *ind*), в массив будет записано *end-begin* символов, которые займут элементы массива,

начиная с индекса *ind* до индекса *ind*+(*end-begin*)-1. Этот метод создает исключительную ситуацию в следующих случаях:

- ссылка *dst* == null;
- индекс *begin* отрицателен;
- индекс *begin* больше индекса *end*;
- индекс *end* больше длины строки;
- индекс *ind* отрицателен;
- *ind*+(*end-begin*) больше dst.length.

Если надо получить массив байтов, содержащий все символы строки в байтовой кодировке ASCII, то используйте метод getBytes(). Этот метод при переводе символов из Unicode в ASCII использует локальную кодовую таблицу.

Если же надо получить массив байтов не в локальной кодировке, а в какой-то другой, применяйте метод getBytes(String *encoding*) или
метод getBytes(Charset *encoding*).

Метод regionMatches() сравнивает указанную часть строки с другой частью строки. Существует также перегруженная форма, которая игнорирует регистр символов при сравнении. Оба метода возвращают результат boolean. Общая форма этих двух методов:

- regionMatches(int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)
- regionMatches(boolean *ignoreCase*, int *startIndex*,
  String *str2*, int *str2StartIndex*, int *numChars*)

В обеих версиях *startIndex* задает индекс начала диапазона строки вызывающего объекта String. Строка, подлежащая сравнению, передается в *str2*. Индекс символа, начиная с которого нужно выполнять сравнение в *str2*, передается в *str2StartIndex*, а длина сравниваемой подстроки — в *numChars*. Во второй версии, если *ignoreCase* равно true, регистр символов игнорируется. В противном случае регистр учитывается.

Метод startWith() определяет, начинается ли заданный объект String с указанной строки.

Метод endsWith() определяет, завершается ли объект String заданным фрагментом. Эти методы возвращают результат типа boolean. Так как методы достаточно просты, то я думаю достаточно примера в коде, чтобы понять как они работают.

Класс String предлагает два метода, которые позволяют вам выполнять поиск в строке определенного символа или подстроки.

- indexOf() — ищет первое вхождение символа или подстроки.
- lastIndexOf() — ищет последнее вхождение символа или подстроки.

Эти два метода перегружены несколькими разными способами. Во всех случаях эти методы возвращают позицию в строке (индекс), где символ или подстрока была найдена, либо -1 в случае неудачи.

Метод contains() проверяет содержит ли проверяемая строка, строку переданную как аргумент метода и если содержит возвращает true, если нет – false.

Для формирования новых строк на основе редактирования есть также набор методов:

- replace() имеет две формы. Первая заменяет в исходной строке все вхождения одного символа другим. Вторая форма replace() заменяет одну последовательность символов на другую.
- join() объединяет строки переданные как параметры и разделяет их разделителем, переданным в качестве первого параметра.
- toLowerCase() преобразует все символы строки из верхнего регистра в нижний. Метод toUpperCase() преобразует все символы строки из нижнего регистра в верхний. Небуквенные символы, такие как десятичные цифры, остаются неизменными.
- trim() возвращает новую строку, в которой удалены начальные и конечные символы с кодами, не превышающими ' \u0020'.

2.3. Класс StringBuilder.

Объекты класса StringBuilder — это строки переменной длины. Только что созданный объект имеет буфер пределенной *емкости* (*capacity*), по умолчанию достаточной для хранения 16 символов. Емкость можно задать в конструкторе объекта. Как только буфер начинает переполняться, его емкость автоматически увеличивается, чтобы вместить новые символы. В любое время емкость буфера можно увеличить, обратившись к методу ensureCapacity(int *minCapacity*). Этот метод изменит емкость, только если *minCapacity* будет больше длины хранящейся в объекте строки. Емкость будет увеличена по следующему правилу. Пусть емкость буфера равна N. Тогда новая емкость будет равна Max(2 * N + 2, minCapacity). Таким образом, емкость буфера нельзя увеличить менее чем вдвое. Методом setLength(int *newLength*) можно установить любую длину строки. Если она окажется больше текущей длины, то дополнительные символы будут равны ' \u0000'. Если она будет меньше текущей длины, то строка окажется обрезанной, последние символы потеряются, точнее, будут заменены символом ' \u0000' . Емкость при этом не изменится. Если число *newLength* окажется отрицательным, возникнет исключительная ситуация. Количество символов в строке можно узнать так же, как и для объекта класса String, методом length() , а емкость — методом capacity().

В классе StringBuilder определены четыре конструктора:

- StringBuilder() — создает пустой объект с емкостью 16 символов;
- StringBuilder(int capacity) — создает пустой объект заданной емкости capacity;
- StringBuilder(String str) — создает объект емкостью str.length() + 16, содержащий строку str;
- StringBuilder(CharSequence str) — создает объект, содержащий строку str.

В классе StringBuilder более десяти методов append(), добавляющих подстроку в конец строки. Они не создают новый экземпляр строки, а возвращают ссылку на ту же самую, но измененную строку.

- append(String) присоединяет строку в конец данной строки. Если присоединяемая строка равна null, то добавляется строка "null". Два аналогичных метода работают с параметром типа StringBuffer и CharSequence.
- шесть методов append(type) добавляют примитивные типы boolean, char,
    - int, long, float, double, преобразованные в строку.
- append(char[]) и append(char[], int, int) - присоединяют к строке массив char и подмассив символов, преобразованные в строку:
- append(char[]) и append(char[], int, int) - добавляет к строке кодовую точку (code point).
- append(Object obj), добавляет просто объект. Перед этим объект obj преобразуется в строку своим методом toString().

Множество методов insert() предназначены для вставки строки, указанной вторым параметром метода, в данную строку. Место вставки задается первым параметром метода, индексом символа строки, перед которым будет сделана вставка. Он должен быть неотрицательным и меньше длины строки, иначе возникнет исключительная ситуация. Строка раздвигается, емкость буфера при необходимости увеличивается. Методы возвращают ссылку на ту же самую, но преобразованную строку:

- insert(int, String) вставляет строку в данную строку перед ее символом с индексом int. Если ссылка str == null, вставляется строка "null".
- insert(sb.length(), "xxx") будет работать так же, как метод append("xxx").
- Шесть методов insert(int, type elem) вставляют примитивные типы boolean, char, int, long, float, double, преобразованные в строку.
- Два метода вставляют массив char и подмассив char символов, преобразованные в строку.

Удаление символов из строки выполняется методами delete.

- delete(int beging, int end) удаляет из строки символы, начиная с индекса begin включительно до индекса end исключительно; если end больше длины строки, то до конца строки. Если begin отрицательно, больше длины строки или больше end, возникает исключительная ситуация. Если begin == end, удаление не происходит.
- deleteCharAt(int ind) удаляет символ с указанным индексом ind. Длина строки уменьшается на единицу. Если индекс ind отрицателен или больше длины строки, возникает исключительная ситуация.

Метод reverse() меняет порядок расположения символов в строке на обратный.

Для поиска подстроки в строке применяются методы indexOf() и lastIndexOf()

- indexOf(String *str*) выполняет поиск первого вхождения *str*. Возвращает индекс позиции совпадения или -1 в случае неудачи.
- indexOf(String *str*, int *startIndex*) выполняет поиск первого вхождения *str*, начиная с *startIndex*. Возвращает индекс позиции совпадения или -1 в случае неудачи.
- lastIndexOf(String *str*) выполняет поиск последнего вхождения *str*. Возвращает индекс позиции совпадения или -1 в случае неудачи.
- lastIndexOf(String *str*, int *startIndex*) выполняет поиск последнего вхождения str, начиная с startIndex. Возвращает индекс позиции совпадения или -1 в случае неудачи.

Метод substring возвращает строку класса String и имеет две следующие формы:
- substring(int *startIndex*)
- substring(int *startIndex,* int *endIndex*)

Первая форма возвращает подстроку, которая начинается от *startIndex* и продолжается до конца объекта StringBuffer. Вторая форма возвращает подстроку от позиции *startIndex* до *endIndex-1*.

Метод trimToSize() который уменьшает размер символьного буфера объекта с тем, чтобы он соответствовал текущему содержимому.

2.4. Лабораторная работа № 2

Цель: Знакомство с классами обработки строк.

Вариант задания взять у преподавателя. Пример задания:

Необходимо использовать два класса строк: String и StringBuilder.

Текстовые сообщения часто печатаются строчными буквами, но многие сотовые телефоны имеют встроенные средства преобразования строчной буквы в прописную после символа пунктуации, как точка или знак вопроса. Составить программу, которая будет вводить сообщение в переменную String (на одной строке), а затем обрабатывать его с получением новой строки с прописными буквами в соответствующих местах.

- Составить программу, которая будет вводить строку в переменную String. Подсчитать, сколько различных символов встречаются в ней. Вывести их на экран.
- Дана строка, содержащая зашифрованный русский текст. Каждая буква заменяется на следующую за ней (буква я заменяется на а). Получить новую расшифрованную строку.

3. КЛАССЫ

3.1. Общие сведения.

Класс – это коллекция данных, хранимых в именованных полях, и кода, организованного в именованные методы, оперирующие этими данными. Поля и методы называются членами класса (members). В Java классы могут содержать другие классы. Эти классы-члены, или внутренние классы, предоставляют дополнительную функциональность, которую мы обсудим чуть позже. Сейчас мы рассмотрим только поля и методы. Члены класса делятся на два различных типа: члены класса, связанные непосредственно с классом (статические), и члены экземпляра, связанные с каждым экземпляром класса (то есть с объектом). Не принимая во внимание классы-члены, мы получаем четыре типа членов:

- Поля класса (статические поля – static fields)
- Методы класса (статические методы – static methods)
- Поля экземпляра (instance fields)
- Методы экземпляра (instance methods)

Поля класса связаны с классом, в котором они определены, а не с экземпляром класса. То же касается и методов класса, как мы это уже отмечали. Поскольку статические методы класса могут вызываться без создания объектов класса, то они не могут работать с полями и методами экземпляров класса, так как таковых может просто не существовать. Поэтому при попытке обращения из статического метода к обычному методу или полю экземпляра, компилятор выдаст ошибку. То есть надо понять, что статические поля и методы класса могут существовать и использоваться без наличия в памяти объектов данного класса. В тоже время обычные методы класса могут обращаться к статическим полям и статическим методам класса. Обращение к полям и методам класса происходит через имя класса, в то время как обращение к полям и методам экземпляра класса происходит через имя этого экземпляра.

Объявление класса выполняется путем указания данных, которые он содержит, и кода, воздействующего на эти данные. Для объявления класса служит ключевое слово class. Объявление полей класса похоже на объявление переменных, и если вы помните, то при создании экземпляра класса, все его поля инициализируются значениями по умолчанию. Для объектов значением по умолчанию является null. Для создания объектов используется оператор new, который выделяет в heap'е (куче) память под объект, где так же размещаются все поля (данные) класса. При выделении памяти происходит инициализация полей. Если нет конструкторов в классе (о которых мы поговорим чуть позже) которые инициализируют эти поля какими-то значениями, то поля инициализируются значениями по умолчанию. Объявление методов в классе, обычно делают после объявления переменных. Модификаторы так же как для классов и полей могут быть, а могут и не присутствовать. Кроме того, метод может возвращать какое-

либо значение, как в нашем примере возвращает значение типа int через оператор return, а может не возвращать, и тогда тип возвращаемого значения указывается как void. Общая форма объявления метода выглядит следующим образом:

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

Здесь *тип* указывает тип данных, возвращаемых методом. Он может быть любым допустимым типом, в том числе типом класса, созданным программистом. Если метод не возвращает значение, типом его возвращаемого значения должен быть void. *Имя* служит для указания имени метода. Оно может быть любым допустимым идентификатором, кроме тех, которые уже используются другими элементами в текущей области определения. *Список_параметров* — последовательность пар “тип-идентификатор”, разделенных запятыми. По сути, параметры — это переменные, которые принимают значения аргументов, переданных методу во время его вызова. Если метод не имеет параметров, список параметров будет пустым. Методы, тип возвращаемого значения которых отличается от void, возвращают значение вызывающей процедуре с помощью следующей формы оператора return:

```
return <возвращаемое значение>
```

Оператор return может использоваться и без *возвращаемого значения* если тип метода void**,** в этом случае оператор return просто прерывает выполнение метода. Конструкторы, это по существу, те же самые методы, но которые автоматически вызываются при создании объекта, если они были объявлены в классе и используются при создании объекта. Конструктор инициализирует объект непосредственно во время создания. Его имя совпадает с именем класса, в котором он находится, а синтаксис аналогичен синтаксису метода. Как только он определен, конструктор автоматически вызывается непосредственно после создания объекта, перед завершением выполнения операции new.

```
class Sample {
    int number;
    Sample() {number=1;}
}
```

Модификаторы, как и в предыдущих случаях, могут отсутствовать, а имя конструктора совпадает с именем класса.

3.2. Перегрузка

В Java существуют два вида перегрузки:

- метода
- конструктора

Java разрешает определение внутри одного класса двух или более методов с одним именем, если только они содержат разные параметры. В этом случае методы называют перегруженными, а процесс — перегрузкой (**overloading**) методов. При вызове перегруженного метода для определения нужной версии Java использует тип и/или количество аргументов метода. Следовательно, перегруженные методы должны различаться по типу и/или количеству их параметров. Хотя возвращаемые типы перегруженных методов могут быть различны, самого возвращаемого типа недостаточно для различения двух версий метода. Когда Java встречает вызов перегруженного метода, она просто выполняет ту его версию, параметры которой соответствуют аргументам, использованным в вызове. Методы могут возвращать значения. Значениями могут быть как примитивные типы, так и объекты.

Наряду с перегрузкой обычных методов можно также выполнять перегрузку конструкторов. У каждого Java класса есть по крайней мере один конструктор, который является методом с таким же именем, как и имя класса. Его назначение – выполнить всю необходимую инициализацию нового объекта. Это конструктор по умолчанию, поскольку у него нет аргументов. Но если мы определяем в классе свои конструкторы, то если хотим использовать конструктор по умолчанию, то должны определить и его.

3.3. Передача аргументов в метод.

Во время вызова метода в стеке создается для него блок памяти, где сохраняются все переменные используемые в методе, в том числе и те, что объявлены как аргументы. После выполнения метода этот блок со всеми переменными уничтожается. При вызове методов значения копируются в переменные метода объявленные как аргументы. Для примитивных типов аргументы метода получают их копии, поэтому метод никак не может влиять на переменную переданную в него как аргумент, но может работать только с её копией. При передаче объектов значения передаются по ссылке, и, поскольку, значение содержит ссылку на объект, то метод может изменять состояние переданного ему таким образом объекта.

3.4. Модификаторы

final. Для переменных и полей значение должно быть присвоено ровно один раз к моменту завершения инициализации экземпляра. Модификатор final можно использовать и при объявлении аргументов в методе.

static. Статические поля и методы относятся не к экземпляру класса, а ко всему классу. Для доступа и работы с ними нет необходимости создавать экземпляр класса. На методы, объявленные как **static**, накладывается ряд ограничений:

- Они могут вызывать только другие статические методы.
- Они могут получать доступ только к статическим переменным.
- Они ни коим образом не могут ссылаться на члены типа **this** или **super**.
- Статические методы не могут быть абстрактными
- Статические методы переопределяются в подклассах только как статические
- При переопределении статических методов полиморфизм не действует, ссылки всегда направляются на методы класса, а не объекта

Статические поля так же могут использовать и модификатор **final**, что имеет тот же смысл что и для обычных полей – после инициализации, значение такого поля не может быть изменено.

private. Модификатор private делает поля и методы доступными только для методов внутри данного класса и не доступными из вне. Это означает, что на экземпляре класса нет возможности вызвать методы объявленные как private, или получить доступ к private полям класса. Это один из механизмов инкапсуляции в Java. В Java существует такое понятие как сеттеры и геттеры (set, get). Это методы, соответственно, для установки или получений значений полей класса.

3.5. Ключевое слово – this.

Каждый раз при вызове обычного (не статического) метода, кроме явных аргументов, если они имеются, туда еще передается и не явный аргумент – this. this хранит ссылку на объект из которого был вызван метод this может использоваться внутри любого метода для ссылки на текущий объект. То есть this всегда служит ссылкой на объект, из которого был вызван метод. Ключевое слово this можно использовать везде, где допускается ссылка на текущий объект. Поскольку this указывает на текущий объект, то его можно использовать для явного указания обращений к полям текущего экземпляра класса, например:

```
this.age = 5;
```

this можно использовать для вызова одних конструкторов класса из других. Вызвать один метод класса из другого метода очень просто – по имени. И даже если имена одинаковые, то работает правило перегрузки, а вот как вызывать

конструкторы? У них же имя совпадает с именем класса. Один метод из другого можно вызывать в любом месте метода, но вызов одного конструктора из другого может быть только в первой строке вызывающего конструктора.

3.6. Метод finalize()

Иногда при уничтожении объект должен будет выполнять какое-либо действие. Например, если объект содержит какой-то ресурс, отличный от ресурса Java (вроде файлового дескриптора или шрифта), может требоваться гарантия освобождения этих ресурсов перед уничтожением объекта. Для подобных ситуаций Java предоставляет механизм, называемый финализацией. Используя финализацию, можно определить конкретные действия, которые будут выполняться непосредственно перед удалением объекта сборщиком мусора.

Чтобы добавить в класс средство выполнения финализации, достаточно определить метод finalize(). Среда времени выполнения Java вызывает этот метод непосредственно перед удалением объекта данного класса. Внутри метода finalize() нужно указать те действия, которые должны быть выполнены перед уничтожением объекта. Сборщик мусора запускается периодически, проверяя наличие объектов, на которые отсутствуют ссылки как со стороны какого-либо текущего состояния, так и косвенные ссылки через другие ссылочные объекты. Непосредственно перед освобождением ресурсов среда времени выполнения Java вызывает метод finalize() по отношению к объекту. Важно понимать, что метод finalize() вызывается только непосредственно перед сборкой мусора.

3.7. Внутренние классы

Статические вложенные классы. Статический вложенный класс или интерфейс определен как **static** член окружающего класса, что делает его аналогом поля и метода класса, которые так же объявлены как **static**. Статический вложенный класс не связан ни с одним экземпляром внешнего класса, то есть может использоваться без создания экземпляра внешнего класса. Тем не менее статический вложенный класс имеет доступ ко всем **static** членам окружающего класса, включая любые другие статические вложенные классы и интерфейсы. Статический вложенный класс может использовать любой статический член окружающего класса без указания имени этого окружающего класса. Статический вложенный класс имеет доступ ко всем статическим членам окружающего класса, включая **private** члены. Обратное тоже верно: методы окружающего класса имеют доступ ко всем **static** членам статического вложенного класса, включая его **private static** члены, однако при обращении к ним необходимо указывать имя статического вложенного класса, которому они принадлежат. Поскольку статические вложенные классы сами являются членами класса, то статический вложенный класс может быть объявлен с любыми модификаторами доступа. Эти модификаторы имеют одно и то же значение для статических вложенных классов и для остальных членов класса. Статический вложенный класс не может называться так же, как называется любой из окружающих классов. Кроме того, статические вложенные классы и интерфейсы могут быть объявлены только в классах верхнего уровня и других статических вложенных классах и интерфейсах. В действительности это часть более общего запрета на использование **static** членов любого вида во внутренних, локальных и анонимных классах. Из статических вложенных классов вы не можете обращаться к нестатическим членам внешнего класса. Обратное тоже верно – из внешних классов вы не можете обращаться к нестатическим членам вложенного статического класса. Статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать его объект. Обратное так же верно – для доступа к нестатическим членам вложенного класса, внешний должен создать его объект. Если в статический вложенный класс вкладывается еще один класс, то это не делает его автоматически статическим, он будет вложенным *inner* классом, даже если его окружающий класс является вложенным в интерфейс. Чтобы класс вложенный в статический класс был статическим необходимо это явно указать при помощи ключевого слова static.

Вложение классов в интерфейсы особенно удобно при создании общего кода, который должен использоваться со всеми реализациями этого интерфейса.

Статические вложенные классы можно так же использовать для проверки работоспособности классов, то есть для их тестирования. Поскольку в них вы можете расположить метод main(). Как вы уже знаете при компиляции будет скомпилирован отдельный .class файл для вложенного класса, который вы можете запускать для тестирования работы отдельного класса не запуская всю программу. Кроме того, в окончательную сборку программы можно не включать файлы .class которые были сгенерированы для тестирования, их можно просто удалить и собрать программу, таким образом в ней не будет лишнего кода, который вы создали для тестирования.

Статический вложенный класс способен наследовать другие классы, реализовывать интерфейсы и являться объектом наследования для любого класса, обладающего необходимыми правами доступа.

Внутренние нестатические классы. Внутренний класс определяется внутри окружающего класса, но он объявляется без модификатора **static**. Этот тип внутренних классов является аналогом методов и полей экземпляра. Экземпляр внутреннего класса всегда связан с экземпляром окружающего класса, а код внутреннего класса имеет доступ ко всем полям и методам (как статическим, так и нестатическим) окружающего класса. Есть несколько особенностей синтаксиса языка Java, проявляющихся при работе с экземпляром класса, содержащего внутренний класс. Вы не можете создать экземпляр внутреннего класса без привязки к экземпляру внешнего класса. То есть сперва должен быть создан экземпляр внешнего класса, а только затем уже вы можете создать экземпляр внутреннего класса. Как правило внешний класс содержит метод, возвращающий ссылку на внутренний класс. У внутреннего класса, как и у любого члена класса, может быть установлен один из трех уровней видимости: public, protected или private. Если ни один из этих модификаторов не указан, то по умолчанию применяется пакетная видимость. Внутренний класс не может иметь имя, совпадающее с именем окружащего класса или пакета. Правило не распространяется ни на поля, ни на методы. Внутренний класс не может иметь полей, методов или классов, объявленных как static (за исключением полей констант, объявленных как static и final**)**. Создать объект внутреннего класса можно непосредственно при помощи особого синтаксиса оператора **new**. Чтобы создать объект внутреннего класса обязательно должен присутствовать объект внешнего класса, поскольку каждый экземпляр внутреннего класса связан с экземпляром своего внешнего класса. Для создания объекта внутреннего класса указывается не имя его внешнего класса, как можно было бы ожидать, а объект внешнего класса, за которым, через точку, следует new. Внутренние классы имеют право наследовать другие классы,

реализовывать интерфейсы и выступать в роли объектов наследования. Когда класс затеняет или замещает члены родительского класса, для ссылки на скрытый член можно применить ключевое слово super. Так же, данное ключевое слово можно задействовать и для работы во внутренних классах. Для этого используется имя текущего класса, затем точка, ключевое слово super, опять точка и имя затененного поля или метода. Если внутренний класс наследуется обычным образом, то он теряет доступ к private членам своего внешнего класса, в котором он был объявлен. Но доступ к членам protected, public и пакетного доступа сохраняется. Все то же самое справедливо при наследовании класса любой глубины вложенности. Это означает что любой внутренний класс любой глубины вложенности имеет доступ ко всем членам внешних классов без дополнительного уточнения в именах этих членов. Наследник внутреннего класса (который наследуется обычным образом), так же может обращаться к этим членам без дополнительного уточнения их имени. Исключение составляют только члены внешних классов объявленные как private.

Локальные классы. Локальный класс – это класс, определенный в блоке Java-кода. Как и локальная переменная, локальный класс виден только внутри блока. Локальным классам присущи многие особенности внутренних классов. Кроме того, локальные классы могут работать с любыми **final** переменными или параметрами, которые доступны в блоке, где класс был определен. Интерфейсы нельзя определить локально. Обычно локальный класс определяется в методе, но он также может быть определен в статическом инициализаторе или инициализаторе экземпляра класса и вообще в любом блоке кода. Поскольку все блоки Java кода находятся внутри определения класса, то все локальные классы вложены в окружающие классы. По этой причине локальные классы имеют много общего с внутренними классами. Но чаще их рассматривают как отдельный вид внутренних классов. Видимость локального класса регулируется областью видимости того блока, в котором он объявлен. Локальный класс может обращаться к любым полям и аргументам метода объявленным в текущем блоке кода, даже если они не объявлены как final, но только в том случае если их значение не изменяется после инициализации, по существу те же яйца только вид с боку. Локальные классы никогда не объявляются с помощью модификаторов доступа (т.е. public, protected и т.п.), поскольку их область видимости всегда ограничивается блоком, в котором они объявлены. Локальный класс виден только внутри блока, который его определяет, его нельзя применять вне этого блока. Как и внутренние классы, и по тем же причинам, локальные классы не могут содержать static поля, методы или классы. Единственное исключение составляют константы, объявленные как static и final. Подобно внутреннему классу, локальный класс не может называться так же, как и

окружающий его класс. Экземпляры локальных классов, как и экземпляры внутренних классов, имеют окружающий экземпляр, ссылка на который неявно передается всем конструкторам локальных классов. Локальные классы могут применять такой же синтаксис this, какой используют внутренние классы для явной ссылки на объект окружающего класса или члены. Так как локальные классы не видимы вне блока, в котором они определены, то нет необходимости использовать ключевое слово new, применяемое к внутренним классами для явного указания окружающего экземпляра при создании экземпляра локального класса. Локальные классы могут наследовать любые другие классы и естественно получать доступ к членам этих классов если они не объявлены как private, что в принципе естественно и не нарушает общих правил, поскольку унаследованные члены становятся членами класса наследника, а private члены не могут быть унаследованы.

Анонимный класс. Анонимный класс – это локальный класс без имени. Можно объявить анонимный (безымянный) класс, который может расширить (extends) другой класс или реализовать (implements) интерфейс. Объявление такого класса выполняется одновременно с созданием его объекта посредством оператора new. Такие классы не являются ни одним из типов внутренних классов. Анонимные классы эффективно используются, как правило, для реализации (переопределения) нескольких методов и создания собственных методов объекта. Анонимный класс не может иметь конструкторов, поскольку имя конструктора должно совпадать с именем класса, а в данном случае класс не имеет имени. Любые аргументы, которые вы укажете в круглых скобках, стоящих за именем родительского класса в определении анонимного класса, неявно передаются конструктору родительского класса. Чаще всего анонимные классы применяются для расширения родительских классов простыми классами, которые не требуют аргументов конструктора, поэтому скобки в определении анонимного класса зачастую пусты.

3.8. Лабораторная работа № 3

Цель: Освоить создание классов, использование экземпляров классов обработке. Задание взять у преподавателя.

Пример задания:

Создать класс «Прямоугольник», описывающий объекты – прямоугольники со сторонами, параллельными осям координат. Класс должен содержать указанные ниже элементы.

– Закрытые поля для хранения координат левого верхнего и правого нижнего углов прямоугольника.

- Конструктор без параметров для создания прямоугольника с левым верхним углом в начале координат и правым нижним углом в точке (1,-1).
- Конструктор с параметрами для создания прямоугольника с произвольными координатами углов. Предусмотреть проверку на корректность введенных данных.
- Геттеры для доступа к полям класса.
- Геттер для определения площади прямоугольника.
- Метод, результатом которого является **true**, если прямоугольник квадрат, и **false** в противном случае.
- Метод для перемещения прямоугольника по вертикали вверх или по горизонтали вправо (в зависимости от значения соответствующего параметра) на заданную величину.
- Метод для поворота прямоугольника вокруг левого верхнего угла на 90^0 против часовой стрелки.
- Статический метод для проверки, располагается ли один прямоугольник внутри другого (входные параметры – объекты класса, результат **true** или **false).**

Разработать программу, выполняющую следующие действия:

− Создает три объекта класса «**Прямоугольник**» (один с помощью конструктора без параметров и два произвольных);

Определяет располагается ли какой-нибудь прямоугольник внутри другого;

− Осуществляет перемещение или поворот (по выбору пользователя) для второго прямоугольника и выводит новую информацию о нем

4. КОЛЛЕКЦИИ

4.1. Объекты – контейнеры

Контейнерные классы – это классы для хранения данных, организованных определенным образом.

– Библиотеки контейнерных классов Java содержат практически все основные контейнеры, реализующие сложные структуры данных, а с учетом наличия параметрических типов практически исчезает необходимость создания своих контейнерных классов.
– Выбор контейнера связан с тем, что требуется делать в программе, например, если включать в конец, а брать с начала – это очередь – и контейнер Queue.
– Вместе с контейнером нам достается и алгоритм, а значит не надо думать о реализации
– Кроме того, классы контейнеров сами заботятся об управлении собственное памятью. (capacity - свойство)

Два больших вида контейнеров: последовательные и ассоциативные.

Последовательные контейнеры:

– очереди, двусторонние очереди (деки)
– списки (простые), двусторонние списки
– стеки

Особенность последовательных контейнеров – быстрое добавление и удаление, но последовательный поиск.

Ассоциативные контейнеры:

– Словари
– Множества
– Бинарные деревья
– Списковые массивы

Особенность - обеспечивают быстрый доступ к данным по ключу, но операции включения и исключения – медленнее, чем в последовательных списках.

Преимущества использования контейнеров:

– Снижают количество написанного кода. Предоставляя полезную структуру данных и алгоритмов, коллекции освобождают от низкоуровневого программирования и позволяют сосредоточится на других частях программы;
– Повышают быстродействие и качество программы. Коллекции обеспечивают высокую производительность, высокое качество реализации полезных структур данных и алгоритмов. Различные реализации каждого интерфейса являются взаимозаменяемыми, так что

программы могут быть легко настроены на переключение разных реализаций коллекций.
- Позволяет взаимодействовать не связанным между собой API. Коллекции - общеупотребляемые типы данных, поэтому могут легко использоваться в качестве аргументов функций и их возвращаемых значений.
- Уменьшает количество усилий, приложенных для изучения новых API. Многие API получают коллекции на вход и отдают другие коллекции на выходе. В недавнем прошлом каждое API имело свою собственную реализацию своих коллекций. Из-за этого приходилось изучать структуры данных, свойственные каждому API.
- Способствует повторному использованию программного обеспечения. Новые структуры данных, унаследованные от существующих реализаций коллекций или интерфейсов коллекций, легко могут быть использованы где-угодно.

Происходит процесс стандартизации интерфейсов для универсальных типов, которыми являются контейнеры.

4.2. Интерфейсы и коллекции Java

Java предлагает большой набор разнообразных контейнерных типов данных, хранящих списки объектов. При этом, при разработке структур, возникли вопросы реализации обобщений алгоритмов обработки данных. И как было принято в современных подходах, при решении этих вопросов, было выполнено разделение на интерфейсы и реализацию [5].

Collection: базовый интерфейс для всех коллекций и других интерфейсов коллекций
Этот интерфейс определяет основные методы для манипуляции с данными, такие как:
- вставка (add, addAll),
- удаление (remove, removeAll, clear),
- поиск (contains)

От Collection наследуются интерфейсы:
- List
- Set
- Queue
- Deque

Queue: наследует интерфейс Collection и представляет функционал для структур данных в виде очереди. Deque наследует интерфейс Queue и представляет

функционал для двунаправленных очередей. List наследует интерфейс Collection и представляет функциональность простых списков. Set также расширяет интерфейс Collection и используется для хранения множеств уникальных объектов.

Реализации интерфейса List представляют собой упорядоченные коллекции. Кроме того, разработчику предоставляется возможность доступа к элементам коллекции по индексу и по значению (так как реализации позволяют хранить дубликаты, результатом поиска по значению будет первое найденное вхождение). Интерфейс List содержит классы:

- Vector
- Stack
- ArrayList

Vector — (устарел, не рекомендуется использовать) реализация динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Vector появился в JDK версии Java 1.0, но как и Hashtable, эту коллекцию не рекомендуется использовать, если не требуется достижения потокобезопасности. Потому как в Vector, в отличии от других реализаций List, все операции с данными являются синхронизированными. В качестве альтернативы часто применяется аналог — ArrayList.

Stack — (устарел, не рекомендуется использовать) данная коллекция является расширением коллекции Vector. Была добавлена в Java 1.0 как реализация стека LIFO (last-in-first-out). Является частично синхронизированной коллекцией (кроме метода добавления push()). После добавления в Java 1.6 интерфейса Deque, рекомендуется использовать для стека именно реализации этого интерфейса.

ArrayList — как и Vector является реализацией динамического массива объектов.

- Позволяет хранить любые данные, включая null в качестве элемента.
- Как можно догадаться из названия, его реализация основана на обычном массиве.
- Данную реализацию следует применять, если в процессе работы с коллекцией предполагается частое обращение к элементам по индексу.
- Из-за особенностей реализации поиндексное обращение к элементам выполняется за константное время O(1).
- Но данную коллекцию рекомендуется избегать, если требуется частое удаление/добавление элементов в середину коллекции.

Интерфейс Queue описывает коллекции с предопределённым способом вставки и извлечения элементов, а именно — очереди FIFO (first-in-first-out). Помимо методов, определённых в интерфейсе Collection, определяет дополнительные методы для извлечения и добавления элементов в очередь. Содержит:

- LinkedList,
- ArrayDeque/

LinkedList — ещё одна реализация List.

- Позволяет хранить любые данные, включая null.
- Особенностью реализации данной коллекции является то, что в её основе лежит двунаправленный связный список (каждый элемент имеет ссылку на предыдущий и следующий).
- Благодаря этому, добавление и удаление из середины, доступ по индексу, значению происходит за линейное время O(n), а из начала и конца за константное O(1).
- Так же, ввиду реализации, данную коллекцию можно использовать как стек или очередь.

ArrayDeque — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out). Интерфейс Deque и реализация ArrayDeque были добавлены в Java 1.6.

- Эта коллекция представляет собой реализацию с использованием массивов, подобно ArrayList, но не позволяет обращаться к элементам по индексу и хранение null.
- Как заявлено в документации, коллекция работает быстрее чем Stack, если используется как LIFO коллекция, а также быстрее чем LinkedList, если используется как FIFO.

Интерфейс Map предоставляет разработчику базовые методы для работы с данными вида «ключ — значение». Также как и Collection, он был дополнен generic – типами в версии Java 1.5, а в версии Java 8 появились дополнительные методы для работы с лямбда-выражениями. Интерфейс содержит:

- Hashtable,
- HashMap,
- LinkedHashMap,

- TreeMap,
- WeakHashMap.

Hashtable (устарел, не рекомендуется использовать) реализация такой структуры данных, как хэш-таблица. Она не позволяет использовать null в качестве значения или ключа. Hashtable является синхронизированной (почти все методы помечены как synchronized).

HashMap — коллекция является альтернативой Hashtable.

Двумя основными отличиями от Hashtable являются то, что HashMap не синхронизирована и HashMap позволяет использовать null как в качестве ключа, так и значения. Так же как и Hashtable, данная коллекция не является упорядоченной: порядок хранения элементов зависит от хэш-функции. Добавление элемента выполняется за константное время O(1), но время удаления, получения зависит от распределения хэш-функции. В идеале является константным, но может быть и линейным O(n).

LinkedHashMap — это упорядоченная реализация хэш-таблицы. Здесь, в отличии от HashMap, порядок итерирования равен порядку добавления элементов. Данная особенность достигается благодаря двунаправленным связям между элементами (аналогично LinkedList). Но это преимущество имеет также и недостаток — увеличение памяти, которое занимает коллекция.

TreeMap — реализация Map основанная на деревьях.

- Как и LinkedHashMap является упорядоченной.
- По-умолчанию, коллекция сортируется по ключам с использованием принципа "natural ordering", но это поведение может быть настроено под конкретную задачу при помощи объекта Comparator, которые указывается в качестве параметра при создании объекта TreeMap.

WeakHashMap — реализация хэш-таблицы, которая организована с использованием weak references. Garbage Collector автоматически удалит элемент из коллекции при следующей сборке мусора, если на ключ этого элемента нет жёстких ссылок.

Интерфейс Set (рис.4.5) представляет собой неупорядоченную коллекцию, которая не может содержать дублирующиеся данные. Является программной моделью математического понятия «множество». Интерфейс содержит:

- HashSet,
- LinkedHashSet,
- LinkedHashMap,
- TreeSet.

HashSet — реализация интерфейса Set, базирующаяся на HashMap. Внутри использует объект HashMap для хранения данных.

В качестве ключа используется добавляемый элемент, а в качестве значения — объект-пустышка (new Object()).

Из-за особенностей реализации порядок элементов не гарантируется при добавлении.

LinkedHashSet — отличается от HashSet только тем, что в основе лежит LinkedHashMap вместо HashSet. Благодаря этому отличию порядок элементов при обходе коллекции является идентичным порядку добавления элементов.

TreeSet — аналогично другим классам-реализациям интерфейса Set содержит в себе объект NavigableMap, что и обуславливает его поведение. Предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием "natural ordering".

4.3. Лабораторная работа № 4

Пример задания.

- На основе описания класса стандартной коллекции разработать собственный класс коллекций.
- Коллекция должна быть реализованы без использования стандартных коллекций и хранить элементы обобщенного типа <T>.
- Реализовать интерфейсы Comparable, Collection
- Реализовать основные методы добавления, замены и удаления элементов коллекции, соответствующие методам стандартной коллекции
- Реализовать методы экспорта элементов коллекции в массив и список.
- Разработать класс для хранения данных коллекции во внешнем источнике. Обеспечить чтение и запись данных.
- Разработать класс предметной области в соответствии с вариантом задания из файла «Классы предметной области» (см. таблицу 4.1).
- Реализовать методы доступа к элементу коллекции на основе ключа, указанного в графе «Доступ к элементу» описания класса пред метной области.
- Разработать класс для обработки и отображения данных элементов коллекции. Операции обработки: редактирование, сортировка, фильтрация

Таблица 4.1

Пример класса предметной области

Вариант	Наименование полей	Ключ сортировки	Доступ к элементу	Вид поиска
Ведомость начисления зарплаты	• Цех,участок • Ф.И.О. • Объем выполнен.работы • Расценки на единицу продукции • Начисляемый заработок	Объем выполнен. работы	По • Цех • ФИО	По вхождению в строку Ф.И.О.

5. РЕАЛИЗАЦИЯ ПРИНЦИПОВ ООП В JAVA

5.1. Принципы ООП

К принципам объектно-ориентированного программирования относятся [5]:

- абстракция,
- инкапсуляция,
- наследование
- полиморфизм

Абстракция - это описание общих характеристик (модели) объекта, без лишней детализации, но достаточной для его представления в программе.

Абстракция - это описание общих характеристик (модели) объекта, без лишней детализации, но достаточной для его представления в программе.

Инкапсуляция – это сокрытие деталей реализации за внешним интерфейсом, т.е. это механизм, который связывает код и данные, которыми он манипулирует, защищая оба эти компонента от внешнего вмешательства и злоупотреблений. Один из возможных способов представления инкапсуляции — представление в виде защитной оболочки, которая предохраняет код и данные от произвольного доступа со стороны другого кода, находящегося снаружи оболочки. Доступ к коду и данным, находящимся внутри оболочки, строго контролируются тщательно определенным интерфейсом.

Наследование – это создание производных классов, наследующих свойства базового. Кроме того, унаследованные классы могут иметь и свои дополнительные свойства, кроме тех что унаследовали и могут переопределить некоторые унаследованные свойства, что по существу уже является полиморфизмом.

Полиморфизм – это разная обработка сообщений в разных классах. Например, унаследованный метод может выполнять другие действия отличные от базового метода.

В основе всех четырех принципов лежит понятие класса и объектов, как его представителей. В рамках ООП данные принципы реализуются через реализацию описания классов, механизмов наследования и модификаторов доступа. Концепция иерархии реализуется также через абстрактные классы и интерфейсы. Концепция инкапсуляции в Java опирается на модицикаторы доступа и пакеты.

5.2. Наследование.

Наследование — одно из фундаментальных понятий объектно-ориентированного программирования, поскольку оно позволяет создавать иерархические классификации. Используя наследование, можно создать общий класс, который определяет характеристики, общие для набора связанных элементов. Затем этот класс может наследоваться другими, более специализированными классами, каждый из которых будет добавлять свои уникальные характеристики. В терминологии Java наследуемый класс называют *суперклассом*. Наследующий класс носит название *подкласса*. Следовательно, *подкласс* — это специализированная версия *суперкласса*. Он наследует все переменные экземпляра и методы, определенные *суперклассом*, и добавляет собственные, уникальные элементы.

Объявление класса-наследника. Чтобы наследовать класс, достаточно просто вставить определение одного класса в другой с использованием ключевого слова **extends**.

```
class Child extends Parent {

  // Код класса Child

}
```

В данном примере объявляется что класс Child является наследником класса Parent. У каждого определяемого вами класса есть родительский класс. Если вы не указали родительский класс в операторе extends, то его родительским классом будет класс java.lang.Object. Класс Object – это:

- единственный класс в Java, у которого нет родителя;
- Класс, от которого наследуются все классы Java.

В Java не поддерживается множественное наследование. То есть у класса потомка может быть только один родительский класс. Конструкторы не наследуются, но подкласс может вызывать конструктор, определенный его суперклассом, с помощью следующей формы ключевого слова super:

```
super(список_аргументов);
```

Список_аргументов определяет любые аргументы, требуемые конструктору в суперклассе, он может быть пустым для вызова конструктора суперкласса по умолчанию. Оператор super всегда должен быть первым выполняемым внутри конструктора подкласса. Java гарантирует, что конструктор класса будет вызываться при каждом создании экземпляра класса. Конструктор супер-класса будет вызываться всякий раз при создании экземпляра подкласса. Чтобы гарантировать второе утверждение, Java обеспечивает порядок, согласно которому каждый конструктор будет вызывать родительский конструктор. Поэтому, если первый оператор в конструкторе не вызывает другой конструктор

с помощью this или super(), то Java неявно вставляет вызов super(), то есть вызывает родительский конструктор без аргументов. Все это означает, что вызовы конструкторов объединяются в цепочку; при каждом создании объекта вызывается последовательность конструкторов: конструктор подкласса, конструктор родительского класса и далее вверх по иерархии классов до конструктора класса Object. Так как конструктор родительского класса всегда вызывается первым оператором конструктора подкласса, то операторы конструктора класса Object всегда выполняются первыми. Затем выполняются операторы конструктора подкласса и далее вниз по иерархии классов вплоть до конструктора класса, объект которого создается. Здесь есть важное следствие: когда вызван конструктор, он может положиться на то, что поля родительского класса уже проинициализированы. Если конструктор не вызывает конструктор родительского класса, то Java делает это неявно. Если класс не объявляет ни одного конструктора, то для него по умолчанию создается конструктор без аргументов. Классы, объявленные с указанием модификатора public, получают конструкторы с модификатором public. Все остальные классы получают конструктор по умолчанию, который объявляется без каких бы то ни было модификаторов доступа.

5.3. Полиморфизм.

Переопределение методов в классах наследниках. Когда подкласс определяет метод экземпляра с таким же именем, типом возвращаемого значения и параметрами, что и метод его родительского класса, то данный метод за мещает (overrides) метод родительского класса. Когда этот метод выполняется для экземпляра класса, то вызывается новое определение, а не старое определение родительского класса. При переопределении метода его сигнатура и тип возвращаемого значения должны полностью сохраняться. Если в подклассе мы изменим тип, количество или порядок следования параметров, то получим новый метод, не переопределяющий метод суперкласса. Проверку соответствия сигнатуры переопределяемого метода можно возложить на компилятор, записав перед методом подкласса аннотацию @Override. В этом случае компилятор пошлет на консоль сообщение об ошибке, если сигнатура помеченного метода не будет соответствовать сигнатуре ни одного метода суперкласса с тем же именем. При переопределении метода права доступа к нему можно только расширить, но не сузить. Открытый метод public должен остаться открытым, защищенный protected может стать public, но не может стать private.
Модификатор final для методов и классов. Мы уже рассматривали ключевое слово final для полей и переменных. Но модификатор final может так же применятся к классам и методам. Смысл ключевого слова final зависит от контекста, но в основном оно означает: "Это нельзя изменить" Теперь мы рассмотри модификатор final в контексте применения к методам и классам

поскольку в данном случае это связано с наследованием и полиморфизмом, которые мы сейчас изучаем.

Если модификатор final стоит перед определением метода, то это означает, что этот метод не может быть переопределен (overrided) в классах наследниках.

Если модификатор final стоит перед определением класса, то это означает, что от этого класса невозможно унаследоваться.

Стоит отметить что метод объявленный
как private, косвенно является final методом. Так как вы не можете получить доступ к private методу, то не можете его и переопределить.

Поля и статические методы. Полиморфизм поддерживается только для вызова обычных методов. А, например, прямое обращение к полю или статическому методу будет обработано на стадии компиляции. Статические методы не поддерживают полиморфного поведения.

5.4. Абстрактные классы

Абстрактный класс – это класс, экземпляр которого нельзя создать. Такой класс определяет сущность методов, которые должны реализовать подклассы. Например, такая ситуация может возникать, когда суперкласс не в состоянии создать полноценную реализацию метода. Абстрактные классы объявляются при помощи модификатора abstract. И как можно догадаться они могут содержать абстрактные методы, которые объявляются при помощи этого же модификатора. Java позволяет определить метод без реализации, объявив его с модификатором abstract. У абстрактного метода нет тела; у него есть только заголовок, заканчивающийся точкой с запятой.

Вот правила для абстрактных методов и абстрактных классов, которые их содержат:

- Любой класс с абстрактным методом автоматически становится абстрактным и должен быть объявлен как abstract.
- Нельзя создать ни одного экземпляра абстрактного класса.
- Экземпляры подклассов абстрактного класса, можно создавать только в том случае, если все методы, объявленные как abstract, замещены и реализованы (то есть имеют тело). Такие классы часто называют реальными, чтобы подчеркнуть тот факт, что они не абстрактные.
- Если подкласс абстрактного класса, не реализует всех методов, объявленных как abstract, то этот подкласс сам является абстрактным.
- Методы, объявленные как static, private и final, не могут быть объявлены как abstract, поскольку такие методы не могут быть замещены

подклассами. Точно так же класс, объявленный как final, не может содержать методов, объявленных как abstract.

– Класс может быть объявлен как abstract, даже если он не содержит ни одного абстрактного метода. Такое объявление означает, что реализация класса все еще не закончена и класс будет служить родителем для одного или нескольких подклассов, которые завершат реализацию. Экземпляр такого класса не может быть создан.
– Можно создавать объектные переменные абстрактных классов, однако такие переменные должны ссылаться на объект неабстрактного подкласса.

5.5. Интерфейсы

Интерфейсы. Чтобы было проще понять интерфейсы в Java, можно их представлять, как полностью абстрактные классы, то есть классы все методы которых не имеют реализации, а только лишь объявлены. Но так было до Java 8, в которой появилась возможность создавать в интерфейсе реализацию метода по умолчанию. Интерфейсы так же могут содержать поля, но все они будут объявлены как final и static**,** то есть по существу являются константами. Кроме того, интерфейсы служат для реализации множественного наследования в Java, которое в чистом виде в Java не поддерживается. Интерфейс (interface), в отличие от класса, содержит только константы, заголовки методов, а также реализацию методов по умолчанию. Описание интерфейса начинается со слова interface, перед которым может стоять модификатор public, означающий, как и для класса, что интерфейс доступен всюду. Если же модификатора public нет, интерфейс будет виден только в своем пакете. После слова interface записывается имя интерфейса, потом может стоять слово extends и список интерфейсов-предков через запятую. Таким образом, одни интерфейсы могут порождаться от других интерфейсов, образуя свою, независимую от классов, иерархию, причем в ней допускается множественное наследование интерфейсов. В этой иерархии нет корня, общего предка. Затем в фигурных скобках записываются в любом порядке константы, заголовки методов и реализации методов по умолчанию. Можно сказать, что в интерфейсе все методы абстрактные (те у которых нет реализации по умолчанию), но слово abstract писать не надо. Константы всегда статические, поэтому клюючевые слова static и final указывать не нужно. Все эти модификаторы принимаются по умолчанию. Все константы и методы в интерфейсах всегда открыты, не обязательно даже указывать модификатор public. Вот какую схему можно предложить для иерархии личность-студент:

interface Person { . . . }
interface Student extends Automobile{ . . . }
Для интерфейсов невозможно создать экземпляр объекта, но можно создать объектную ссылку интерфейсного типа на объект класса, реализующего данный интерфейс. При реализации интерфейса в заголовке класса после его имени или после имени его суперкласса, если он есть, записывается слово implements и, через запятую, перечисляются имена интерфейсов. Ссылочная переменная интерфейса располагает сведениями только о тех методах, которые объявлены в этом интерфейсе. Если класс реализует более одного интерфейса, то он должен предоставить реализацию каждого метода каждого интерфейса либо он должен быть объявлен как abstract**.** Методы, которые реализуют интерфейс, должны быть объявлены как public**.** Интерфейсы можно применять для импорта совместно используемых констант в несколько классов посредством простого объявления интерфейса, который содержит переменные, инициализированные нужными значениями. Если интерфейс не содержит никаких методов, любой класс, который включает в себя такой интерфейс, в действительности ничего не реализует. Это равносильно тому, что класс импортировал бы постоянные поля в пространство имен класса в качестве переменных типа final.

Интерфейсы – наследование в интерфейсах. Ключевое слово extends позволяет одному интерфейсу наследовать другой. Синтаксис определения такого наследования аналогичен синтаксису наследования классов, но в отличие от них интерфейсы могут наследоваться от любого множества интерфейсов, а не от одного как классы.
Интерфейс, который расширяет (наследуется) более одного интерфейса, наследует все методы и константы от каждого родителя и может определять собственные дополнительные абстрактные методы и константы.
Когда класс реализует интерфейс, который наследует другие интерфейсы, он должен предоставлять реализации всех методов, определенных внутри цепочки наследования интерфейса.

- Если класс реализует интерфейсы в которых есть методы с одинаковой сигнатурой и возвращаемым значением, но сам не переопределяет реализацию этих одинаковых методов, то тогда возникнет ошибка компиляции. Если же он переопределяет эти методы, то тогда все нормально.
- Если в интерфейсах наследуемых один от другого есть одинаковые методы с реализациями по умолчанию, то предпочтение отдается реализации самого нижнего метода по умолчанию в цепочке, так как действует правило переопределения методов.
- Но если в классе переопределена реализация этого метода, то предпочтение отдается реализации метода в классе.
- Благодаря ключевому слову super, можно обратиться к реализации метода по умолчанию родительского интерфейса из метода интерфейса

наследника. Эта форма ключевого слова super выглядит следующим образом: имя_интерфейса.super.имя_метода()

У интерфейсов есть еще одна возможность: определять в нем один или несколько статических методов. Аналогично статическим методам в классе, метод, объявляемый как static в интерфейсе, можно вызывать независимо от любого объекта. Для этого метод должен быть определен в интерфейсе. Для вызова статического метода достаточно указать имя интерфейса и через точку имя самого метода: имя_интерфейса.имя_статического_метода.

Статические методы из интерфейсов не наследуются ни реализующими их классами, ни интерфейсами наследниками. Интерфейс может быть объявлен членом класса или другого интерфейса. Такой интерфейс называется интерфейсом-членом или вложенным нтерфейсом. Вложенный в класс интерфейс может быть объявлен как **public, private** или **protected**. Это отличает его от интерфейса верхнего уровня или интерфейса вложенного в другой интерфейс, который должен быть либо объявлен как public, либо, как уже было отмечено, должен использовать уровень доступа, заданный по умолчанию. Когда вложенный в класс интерфейс используется вне содержащей его области определения, он должен определяться именем класса, членом которого он является. То есть вне класса, в котором объявлен вложенный интерфейс, его имя должно быть полностью определено. К интерфейсу вложенному в другой интерфейс таких требований нет, так как они всегда имеют модификатор public.

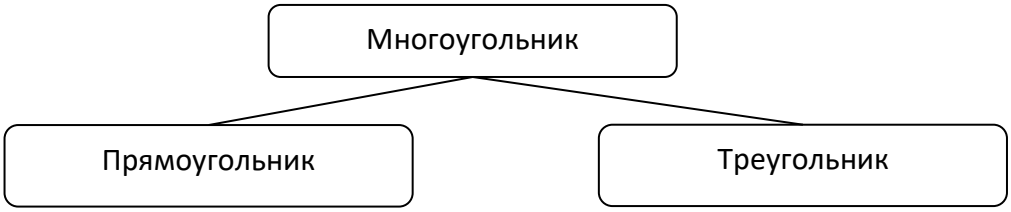
Вложенный в класс интерфейс объявленный с модификатором private не может быть имплементирован каким-либо классом. Он может использоваться только внутри того класса где был объявлен. Иногда полезно определить пустой интерфейс. Класс может реализовать этот интерфейс, указав его в секции implements. При этом нет необходимости реализовывать методы. Любой экземпляр класса становится экземпляром интерфейса. С помощью оператора instanceof Java код может проверить, является ли объект экземпляром интерфейса. Таким образом, эта техника полезна для предоставления дополнительной информации об объекте. Интерфейс Cloneable из пакета java.lang является примером интерфейса-маркера (*marker interface*). Он не определяет методов, но идентифицирует класс, внутреннее состояние которого можно клонировать методом clone() класса Object.

5.6. Лабораторная работа № 5

Цель: Освоить механизм наследования, получить навыки разработки и реализации иерархии классов.

Пример задания.

Создать иерархию классов:



Класс многоугольников должен быть *абстрактным*, содержать следующие элементы: поля (массив, содержащий длины сторон; цвет фигуры); абстрактный метод вычисления площади; метод вывода информации об объекте. Класс многоугольников должен реализовывать интерфейс **Comparable.**

Классы прямоугольников и треугольников должны содержать переопределенные методы для вычисления площади.

Разработать программу, которая выполняет следующие действия:

- считывает информацию из текстового файла, каждая строка которого содержит длины сторон прямоугольника или треугольника и цвет фигуры, например: **1 3 3 White;**
- формирует на основании этой информации массив объектов базового класса иерархии;
- выводит на экран всю информацию в виде:

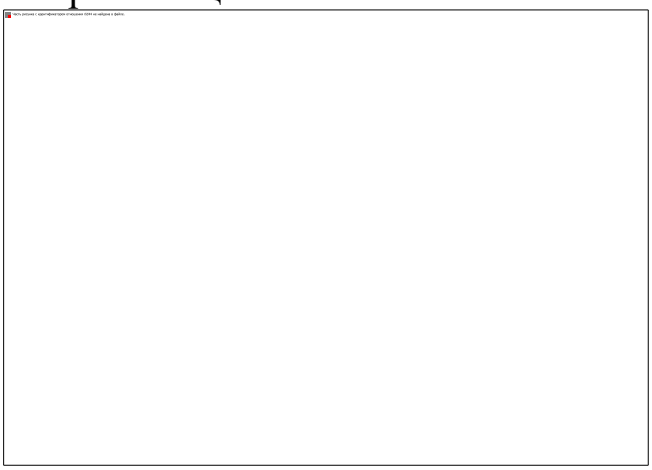
Номер	Вид фигуры	Площадь	Цвет
1	треугольник	15,23	Red

- сортирует массив в порядке возрастания площадей многоугольников и выводит отсортированный массив;
- вычисляет периметры всех прямоугольных треугольников красного цвета.

6. ПОТОКИ И СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ.

6.1. Постоянство объектов

Созданный в приложении информационный объект прекратит свое существование, когда выполнение приложения завершится: объект «живет» в рамках этого приложения. Когда выполнение приложения завершается, объект оказывается вне области видимости. Если на объект, ранее созданный нет более ссылок, то он также будет удален. Чтобы объект продолжил «жить», он должен быть записан в какое-то хранилище.



Объект создается в приложении 1, которое затем записывает его на «запоминающее устройство», скажем в базу данных. Поскольку в итоге этот объект располагается в постоянном хранилище, другие приложения могут получить к нему доступ.

Если мы хотим сохранить состояние некоторого объекта, то нам потребуется предпринять для этого определенные действия. Концепция сохранения состояния объекта с целью его использования позднее называется концепцией постоянства

Для сохранения объектов используются:

- Файлы
- Бинарные потоки (файловые потоки с прямым доступом)
- Сериализация объектов:
- Хранилище данных

Виды современных хранилищ данных:

- Реляционные СУБД
- Объектные СУБД
- NoSql
- Базы знаний

6.2. Потоки ввода-вывода

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса:

- InputStream (представляющий потоки ввода)
- OutputStream (представляющий потоки вывода)

Но поскольку работать с байтами не очень удобно, то для работы с потоками символов были добавлены абстрактные классы:

- Reader (для чтения потоков символов)
- Writer (для записи потоков символов)

Все остальные классы, работающие с потоками, являются наследниками этих абстрактных классов

Потоки байтов.

Абстрактный класс InputStream является базовым для всех классов, управляющих байтовыми потоками ввода. Основные методы:

- int available(): возвращает количество байтов, доступных для чтения в потоке
- void close(): закрывает поток
- int read(): возвращает целочисленное представление следующего байта в потоке. Когда в потоке не останется доступных для чтения байтов, данный метод возвратит число -1
- int read(byte[] buffer): считывает байты из потока в массив buffer. После чтения возвращает число считанных байтов. Если ни одного байта не было считано, то возвращается число -1
- int read(byte[] buffer, int offset, int length): считывает некоторое количество байтов, равное length, из потока в массив buffer. При этом считанные байты помещаются в массиве, начиная со смещения offset, то есть с элемента buffer[offset]. Метод возвращает число успешно прочитанных байтов.
- long skip(long number): пропускает в потоке при чтении некоторое количество байт, которое равно number

Абстрактный класс OutputStream. Класс OutputStream является базовым классом для всех классов, которые работают с бинарными потоками записи. Свою функциональность он реализует через следующие методы:

- void close(): закрывает поток
- void flush(): очищает буфер вывода, записывая все его содержимое
- void write(int b): записывает в выходной поток один байт, который представлен целочисленным параметром b
- void write(byte[] buffer): записывает в выходной поток массив байтов buffer.
- void write(byte[] buffer, int offset, int length): записывает в выходной поток некоторое число байтов, равное length, из массива buffer, начиная со смещения offset, то есть с элемента buffer[offset].

Класс FileOutputStream предназначен для записи байтов в файл. Он является производным от класса OutputStream, поэтому наследует всю его функциональность.

Для считывания данных из файла предназначен класс FileInputStream, который является наследником класса InputStream и поэтому реализует все его методы.

Пример побайтног чтения данных из файла.

```
class FilesApp1 {
   public static void main(String[] args) {
      try (fin = new FileInputStream("C://SomeDir//note.txt")) {
         System.out.println("Размер файла: " +
               fin.available() + “байт(а)");
         int i = -1;
         while ((i = fin.read()) != -1) {System.out.print((char) i);          }
      } catch (IOException ex) {
         System.out.println(ex.getMessage());
      }
   }
}
```

Подобным образом можно считать данные в массив байтов и затем производить с ним манипуляции.

Потоки символов

Абстрактный класс Reader предоставляет функционал для чтения текстовой информации. Рассмотрим его основные методы:

- absract void close(): закрывает поток ввода
- int read(): возвращает целочисленное представление следующего символа в потоке. Если таких символов нет, и достигнут конец файла, то возвращается число -1
- int read(char[] buffer): считывает в массив buffer из потока символы, количество которых равно длине массива buffer. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1
- int read(CharBuffer buffer): считывает в объект CharBuffer из потока символы. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1
- absract int read(char[] buffer, int offset, int count): считывает в массив buffer, начиная со смещения offset, из потока символы, количество которых равно count
- long skip(long count): пропускает количество символов, равное count. Возвращает число успешно пропущенных символов

Абстрактный Класс Writer определяет функционал для всех символьных потоков вывода. Его основные методы:

- Writer append(char c): добавляет в конец выходного потока символ c. Возвращает объект Writer
- Writer (CharSequence chars): добавляет в конец выходного потока набор символов chars. Возвращает объект Writer
- abstract void close(): закрывает поток
- abstract void flush(): очищает буферы потока
- void write(int c): записывает в поток один символ, который имеет целочисленное представление
- void write(char[] buffer): записывает в поток массив символов
- absract void write(char[] buffer, int off, int len) : записывает в поток только несколько символов из массива buffer. Причем количество символов равно len, а отбор символов из массива начинается с индекса off
- void write(String str): записывает в поток строку
- void write(String str, int off, int len): записывает в поток из строки некоторое количество символов, которое равно len, причем отбор символов из строки начинается с индекса off

6.3. Сериализация объектов в JAVA

Сериализация представляет процесс записи состояния объекта в поток, **Десериализация** - процесс извлечения или восстановления состояния объекта из потока.

Сериализация очень удобна, когда идет работа со сложными объектами. Интерфейс программиста используется для записи объекта в файл. Для вас важно лишь следующее:

- вы сможете записать файл как единый блок;
- вы сможете восстановить объект точно так же, как вы его сохранили.

интерфейс Serializable. Сериализовать можно только те объекты, которые реализуют интерфейс Serializable. Этот интерфейс не определяет никаких методов, просто он служит указателем системе, что объект, реализующий его, может быть сериализован.

Класс ObjectOutputStream. Для сериализации объектов в поток используется класс ObjectOutputStream. Он записывает данные в поток. При создании объекта ObjectOutputStream в конструктор передается поток, в который производится запись.

Пример. Дан класс User1.

```
public class User1 implements Serializable {
public String Name="";
private String LastName = "";
```

```
public int Age=0;

public User1() { }
public User1(String newName, String newLastName, int newAge) {
Name = newName; LastName = newLastName;
                Age = newAge; }
public String ToString() {
 return "{Name=" + Name+ ",Lastname=" + LastName
   +", Age=" + Age + "}";
}
```

Сериализация объектов данног класса

```
static void Test01W(){
try(ObjectOutputStream oos создаем объект сериализации= new
ObjectOutputStream(подаем поток как параметр
        new FileOutputStream("d:/-/user.dat")))
   { создаем объект p
      User1 p = new User1("Петр","Петров",25);
      oos.writeObject(p);//Сериализация и запись
   }
   catch(Exception ex){
      System.out.println(ex.getMessage());
   }
}
```

Десериализация объектов

```
static void Test01R(){
 try(ObjectInputStream ois = new ObjectInputStream(
      new FileInputStream("d:/-/user.dat")))
 {   //Чтение и десериализация
    User1 p = (User1)ois.readObject();
    System.out.printf("Имя: %s \t Возраст: %d \n",
                                        p.Name, p.Age);
   }
   catch(Exception ex){
      System.out.println(ex.getMessage());
   }
}
```

Схема сериализации и обратная операции просты:

- Создаем объект на основе сериализуемого класса
- Сохраняем объект в потоке, используя метод writeObject()
- Читаем объект на основе метода readObject()

Приведем пример сериализации объектов в поток

```
import java.io.*;
import java.util.ArrayList;
public class FileApp {

        public static void test11() {
        String filename = "d:/-/people.dat";
        // создадим список объектов
        ArrayList<User1> people = new ArrayList();
        people.add(new User1("Том", "", 22));
        people.add(new User1("Джон", "", 21));

                try { ObjectOutputStream oos =

                        new ObjectOutputStream(

                          new FileOutputStream(filename));
                    //Сериализация и запись коллекции
                    oos.writeObject(people);
                    System.out.println("Запись произведена");
                } catch (Exception ex) {
                    System.out.println(ex.getMessage());
                }

        }

}
```

А также десериазизации объектов из потока

```
// десериализация в новый список
ArrayList<User1> newPeople = new ArrayList<>();
try { ObjectInputStream ois =

       new ObjectInputStream(

        new FileInputStream(filename));

        newPeople=(ArrayList<User1>);

         ois.readObject();
} catch (Exception ex) {
```

```
        System.out.println(ex.getMessage());
}

for(User1 p : newPeople) {
 System.out.printf("Имя: %s \t Возраст: %d \n",  p.Name, p.Age);
}
```

Схема:

- Запись в поток коллекции как объекта
- Чтение из потока коллекции как объекта

Методы класса ObjectOutputStream

- void writeDouble(double val): записывает в поток значение типа double
- void writeFloat(float val): записывает в поток значение типа float
- void writeInt(int val): записывает целочисленное значение int
- void writeLong(long val): записывает значение типа long
- void writeShort(int val): записывает значение типа short
- void writeUTF(String str): записывает в поток строку в кодировке UTF-8
- void close(): закрывает поток
- void flush(): очищает буфер и сбрасывает его содержимое в выходной поток
- Эти методы охватывают весь спектр данных, которые можно сериализовать

Методы класса ObjectInputStream

Object readObject(): считывает из потока объект

- int read(): считывает из потока один байт и возвращает его целочисленное представление
- boolean readBoolean(): считывает из потока одно значение boolean
- byte readByte(): считывает из потока один байт
- char readChar(): считывает из потока один символ char
- double readDouble(): считывает значение типа double
- float readFloat(): считывает из потока значение типа float
- int readInt(): считывает целочисленное значение int
- long readLong(): считывает значение типа long

- short readShort(): считывает значение типа short
- String readUTF(): считывает строку в кодировке UTF-8
- void close(): закрывает поток
- int skipBytes(int len): пропускает при чтении несколько байт, количество которых равно len
- int available(): возвращает количество байт, доступных для чтения

6.4. Лабораторная работа № 6

Цель: получить навыки разработки программ, использующих потоки, научиться выполнять сериализацию и десериализацию объектов. Вариант задания и класс предметной области взять у преподавателя.

Разработать приложение, выполняющее следующие функции:

- Потоковая запись внесенной в таблицу информации об объектах предметной области в структурированный файл.
- Потоковое считывание информации из файла с помощью класса и вывод в виде таблицы.
- Изменение любой записи в таблице и сохранение измененной строки.
- Выполнить сериализацию и десериализацию объектов предметной области.

7. ПАКЕТЫ.

7.1. Основные сведения о пакетах

Пакет (*package*) представляет собой именованную совокупность классов (и, возможно, подпакетов). Пакеты группируют классы и определяют пространства имен для классов, которые в них входят [3]. Пакеты позволяют:

- Задать пространства имен, предотвращение коллизий имен классов
- Логическая группировка связанных классов
- Инкапсуляция - сокрытие деталей реализации за счет модификаторов доступа

Платформа Java содержит пакеты, имена которых начинаются на com, java, javax и org. Посмотрите это в сырцах, не поленитесь. Основные классы языка входят в пакет java.lang. Различные вспомогательные классы находятся в java.util. Классы для ввода и вывода входят в java.io, а классы для работы в сети – в java.net. Некоторые их этих пакетов содержат подпакеты. Например, java.lang содержит два специализированных пакета java.lang.reflect и java.lang.ref, а java.util содержит подпакет java.util.zip, который в свою очередь содержит классы для работы с ZIP-архивами.

Каждый класс имеет как простое имя, данное ему в определении, так и полное имя, включающее имя пакета, в который он входит. Например, класс String является частью пакета java.lang, а его полное имя – java.lang.String.

В стандартную библиотеку Java API входят сотни классов. Каждый программист в ходе работы добавляет к ним десятки своих классов. Множество классов растет и становится необозримым. Уже давно принято отдельные классы, решающие какую-то одну определенную задачу, объединять в библиотеки классов. Но библиотеки классов, кроме стандартной библиотеки, не являются частью языка.

Разработчики Java включили в язык дополнительную конструкцию — пакеты (*packages*). Все классы Java распределяются по пакетам. Кроме классов пакеты могут содержать интерфейсы и вложенные подпакеты (*subpackages*). Образуется древовидная структура пакетов и подпакетов.

Эта структура в точности отображается на структуру файловой системы. Все файлы с расширением class (содержащие байт-коды), образующие один пакет, хранятся в одном каталоге файловой системы. Подпакеты образуют подкаталоги этого каталога.

Каждый пакет создает одно пространство имен (*namespace*). Это означает, что все имена классов, интерфейсов и подпакетов в пакете должны быть уникальны. Имена в разных пакетах могут совпадать, но это будут разные программные

единицы. Таким образом, ни один класс, интерфейс или подпакет не может оказаться сразу в двух пакетах. Если надо в одном месте программы использовать два класса с одинаковыми именами из разных пакетов, то имя класса уточняется именем пакета: *пакет.Класс*. Такое уточненное имя называется полным именем класса (*fully qualified name*).

7.2. Определение пакета

Чтобы определить пакет для какого-либо класса, следует использовать директиву package. Если в Java-коде присутствует ключевое слово package, оно должно быть первой лексемой кода в файле Java, то есть должно следовать сразу после комментариев и пробелов. После ключевого слова должно стоять имя требуемого пакета и точка с запятой. Например:

package pro.java.pkg001;

Все файлы с исходными текстами и тем-более откомпилированные файлы с байт-кодом (расширение .calss) должны находится в соответствующих под-каталогах, как, например в данном случае в подкаталоге pro\java\pkg001. Все современные IDE при создании пакета, сразу же создают соответствующую файловую структуру в каталоге проекта, так что об этом можно особо не беспокоиться, но знать необходимо. Строка *package имя* - только одна и это обязательно первая строка файла, каждый класс попадает только в один пакет или подпакет. Классы, определяемые файлом Java-кода без директивы package, входят в пакет без имени, существующий по умолчанию. Классы одного пакета получают специальный доступ друг к другу. Именно поэтому, когда мы использовали в программе несколько классов, мы могли получать доступ из одного класса к методам или полям другого. Компилятор всегда создает для таких классов безымянный пакет (unnamed package), которому соответствует текущий каталог (current working directory) файловой системы. Безымянный пакет служит обычно хранилищем небольших пробных или промежуточных классов. Большие проекты лучше хранить в пакетах. Более того, некоторые программные продукты Java вообще не работают с безымянным пакетом. Поэтому в технологии Java рекомендуется все классы помещать в пакеты. Имя пакета может быть любым, но важно, чтобы оно было уникальным. Одной из важных функций пакетов является разделение пространства имен Java и предотвращение конфликта имен между классами. Например, классы java.util.List и java.awt.List можно различить только по именам их пакетов. Однако важно, чтобы различались и имена самих пакетов.

Использование класса из пакета:

- Классы текущего пакета и пакета java.lang всегда видны
- Классы других пакетов доступны по полному имени с пакетом, но если они имеют соответствующие модификаторы доступа

- Можно использовать директиву import

Методы и поля классов в текущем пакете доступны другим классам из этого же пакета, если у них нет ни каких модификаторов доступа (о которых мы поговорим чуть позже).

А вот классы и их поля и методы принадлежащих другим пакетам, если у них нет ни каких модификаторов доступа, уже не доступны даже по полному имени пакета и класса.

7.3. Импорт классов и пакетов

Класс пакета pro.java.pkg001 может ссылаться на любой другой класс в pro.java.pkg001 по его простому имени. А поскольку классы пакета java.lang являются базовыми для языка Java, любой Java-код может ссылаться на любой класс этого пакета по его простому имени. Значит, всегда можно набирать String вместо java.lang.String. Однако по умолчанию для всех других классов нужно указывать полные имена. Поэтому, чтобы применить класс File пакета java.io, следует набрать java.io.File.

Директива import бывает двух видов. Чтобы определить отдельный класс, на который можно ссылаться по его простому имени, следом за ключевым словом import введите имя класса и точку с запятой:

import java.io.File; // Теперь можно набирать File вместо java.io.File

Чтобы импортировать целый пакет классов, введите после import имя пакета, символы .,* и точку с запятой. Например, чтобы использовать класс File вместе с несколькими другими классами пакета java.io, нужно просто импортировать целый пакет:

import java.io.*; // Теперь простые имена можно применять для всех классов из java.io

Этот синтаксис импорта пакета не относится к подпакетам. Даже если я импортирую пакет java.util, мне все еще придется обращаться к классу java.util.zip.ZipInputStream по его полному имени. Если два класса, импортированные из разных пакетов, называются одинаково, к ним нельзя обращаться по простому имени; во избежание двусмысленности к обоим классам следует обращаться по их полным именам.

В Java введена еще одна форма оператора import, предназначенная для импорта статических полей и методов класса — оператор import static. Например, можно написать оператор: import static java.lang.Math.*. После этого все статические поля и методы класса Math можно использовать без указания имени класса.

Вместо записи: double r = Math.*cos*(Math.*PI* * alpha) можно записать просто double r = *cos*(*PI* * alpha).

7.4. Структура исходного файла

Теперь можно описать структуру исходного файла с текстом программы на языке Java.

- В первой строке файла может быть необязательный оператор package.
- В следующих строках могут быть необязательные операторы import.
- Далее идут описания классов и интерфейсов.

Данные элементы могут перемежаться с комментариями, но они должны идти именно в такой последовательности. Существует еще несколько важных ограничений по файлам Java.

- Во-первых, каждый файл может содержать максимум один класс, объявленный как public. Такой класс предназначен для использования другими классами в других пакетах. Более подробно public классы и связанные с ними модификаторы мы рассмотрим в следующей статье. Данное ограничение по открытым классам относится только к классам верхнего уровня; как вы скоро узнаете, класс может содержать любое количество вложенных, или внутренних, классов, объявленных как public.
- Второе ограничение касается имени файла в Java. Если файл Java содержит класс public, то имя файла должно представлять собой имя public класса, к которому присоединено расширение .java. Например, если Point является public классом, его исходный код должен находиться в файле Point.java.

Отсюда следует, что если в проекте есть несколько открытых классов, то они должны находиться в разных файлах. Даже если ваши классы не являются public, полезно определять только по одному классу на файл и давать файлу имя класса. Для технологии Java характерно записывать исходный текст каждого класса в отдельном файле. В конце концов, компилятор всегда создает class-файл для каждого класса.

7.5. Модификаторы доступа.

Классы и пакеты используемы совместно с модификаторами доступа служат средствами инкапсуляции, то есть средствами сокрытия деталей реализации за простым интерфейсом.

Модификаторы доступа могут применяться как к классам, так и их членам – полям и методам. Всего существует четыре модификатора доступа и тут приведем их краткое описание, потом рассмотрим каждый подробно.

- public – любой компонент, объявленный как public, доступен из любого кода
- protected – разрешает доступ к компоненту в пределах пакета и классам наследникам
- private – разрешает доступ к компонентам в пределах класса
- по умолчанию (нет ключевого слова) – разрешает доступ к компонентам в пределах пакета

7.6. Доступ к классам

По умолчанию классы верхнего уровня доступны в том пакете, в котором они определены. Впрочем, если класс верхнего уровня объявлен как public, то он доступен везде (или везде, где доступен сам пакет). Мы ограничили это утверждение классами верхнего уровня, потому что классы могут быть объявлены как члены других классов. Так как эти внутренние классы являются членами класса, то они подчиняются правилам контроля доступа к членам класса. Члены класса всегда доступны внутри тела класса. По умолчанию члены класса также доступны в пакете, в котором класс определен. Для класса, не являющегося вложенным, может быть указан только один из двух возможных уровней доступа: заданный по умолчанию и public. Когда класс объявлен как public, он должен быть единственным public классом, объявленным в файле, и имя файла должно совпадать с именем класса. Как public могут быть объявлены классы, поля, методы и конструкторы. Элементы, объявленные как private, доступны только внутри этого же класса. Как private могут быть объявлены поля, методы, конструкторы, вложенные классы и вложенные интерфейсы. Классы и интерфейсы верхнего уровня не могут быть объявлены как private. Модификатор private рекомендуется применять к полям классов, дабы их можно было изменить только с помощью методов класса.

7.7. Лабораторная работа № 7

Тема: Построение приложения с разделением на модули.

Цель: Освоить инструменты построения пакетов в Java

Задание. Используя задание на лабораторную работу №6 выделить:

- Основной компонент, подключающий пакеты;
- Пакет чтения и записи данных в сериализованный файл;
- Пакет обработки данных;
- Пакет ввода данных и отображения результатов.

ЛИТЕРАТУРА

1. Гаврилов А.В. Учебное пособие по языку Java/ А.В. Гаврилов, Дегтярева О.А., Лезин И.А., Лезина И.В. – Самара: СНЦ РАН 2012. – 224с.

2. Хабибуллин И.Ш. Java 7 / И.Ш.Хабибуллин. – СПб: БХВ-Петербург, 2012. – 768 с.

3. Васильев А. Н. Самоучитель Java с примерами и задачами/Васильев А. Н., СПб: Наука и Техника, 2012. – 368 с.

4. Хорстман К. Java. Основы/ Хортсман К. С., Корнелл Г. – СПб: «И.Д.Вильямс», 2014. – 864с.

5. Сеттер Р.В. Изучаем Java на примерах и задачах / Сеттер Р.В. СПб: Наука и Техника, 2016. – 240 с.