Министерство образования Республики Беларусь

Учреждение образования
«Полоцкий государственный университет»

П. В. Ярошевич, Т. М. Глухова

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ И СТАНДАРТЫ ПРОЕКТИРОВАНИЯ

Методические указания
к лабораторным работам
для студентов специальности
1-40 01 01 «Программное обеспечение
информационных технологий»

Текстовое электронное издание

Новополоцк
Полоцкий государственный университет
2021

Об издании – 1, 2

1 – дополнительный титульный экран – сведения об издании

УДК 681.3(075.8)

Одобрено и рекомендовано к изданию методической комиссией
факультета информационных технологий (протокол № 6 от 26 июня 2020)

Кафедра вычислительных систем и сетей

РЕЦЕНЗЕНТЫ:

канд. техн. наук, доц., доц. кафедры технологий программирования
И. Б. БУРАЧЕНОК
канд. техн. наук, доц., доц. кафедры технологий программирования
А. Ф. ОСЬКИН

Пособие содержит краткие теоретические сведения для выполнения лабораторных работ, а также описание порядка выполнения каждой лабораторной работы и задания по вариантам.

Пособие предназначено для студентов специальности 1-40 01 01 «Программное обеспечение информационных технологий» очной формы обучения.

2 – дополнительный титульный экран – производственно-технические сведения

Для создания текстового электронного издания «Объектно-ориентированные технологии программирования и стандарты проектирования» использованы текстовый процессор Microsoft Word и программа Adobe Acrobat XI Pro для создания и просмотра электронных публикаций в формате PDF.

Павел Владимирович ЯРОШЕВИЧ
Татьяна Михайловна ГЛУХОВА

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ И СТАНДАРТЫ ПРОЕКТИРОВАНИЯ

Методические указания
к лабораторным работам
для студентов специальности
1-40 01 01 «Программное обеспечение
информационных технологий»

Редактор *А. А. Прадидова*

Подписано к использованию 29.03.2021.
Объем издания: 1,51 Мб. Заказ 182.

Издатель и полиграфическое исполнение:
учреждение образования «Полоцкий государственный университет».

Свидетельство о государственной регистрации
издателя, изготовителя, распространителя печатных изданий
№ 1/305 от 22.04.2014.

ЛП № 02330/278 от 08.05.2014.

211440, ул. Блохина, 29,
г. Новополоцк,
Тел. 8 (0214) 59-95-41, 59-95-44
http://www.psu.by

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 7
ЛАБОРАТОРНАЯ РАБОТА № 1. ОБЪЯВЛЕНИЕ
И ИСПОЛЬЗОВАНИЕ КЛАССОВ 8
Краткие теоретические сведения 8
Описание класса 8
Атрибуты класса 9
Методы класса 9
Класс Object 12
Уничтожение объектов 13
Порядок выполнения работы 14
ЛАБОРАТОРНАЯ РАБОТА № 2. ОСНОВНЫЕ ПРИНЦИПЫ
ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ 19
Краткие теоретические сведения. 19
Модификаторы доступа и инкапсуляция 19
Наследование 22
Полиморфизм 24
Абстрактные классы 25
Иерархия наследования и преобразование типов 26
Статические поля 28
Порядок выполнения работы 31
ЛАБОРАТОРНАЯ РАБОТА № 3. ИНТЕРФЕЙСЫ 34
Краткие теоретические сведения. 34
Интерфейсы 34
Интерфейсы в преобразованиях типов 36
Методы по умолчанию 36
Статические методы 37
Приватные методы 38
Константы в интерфейсах 38
Множественная реализация интерфейсов 39
Наследование интерфейсов 39
Вложенные интерфейсы 40
Интерфейсы как параметры и результаты методов 40
Интерфейсы в механизме обратного вызова 42
Порядок выполнения работы 44
ЛАБОРАТОРНАЯ РАБОТА № 4. ПРОГРАММИРОВАНИЕ
С ИСПОЛЬЗОВАНИЕМ ОБОБЩЕНИЙ 48
Краткие теоретические сведения. 48

Обобщение....48
Обобщение классов....48
Обобщение методов....50
Обобщение конструкторов....51
Обобщение интерфейсов....52
Иерархия классов....53
Метасимвол <?>....53
Ключевые понятия этой темы....54
Порядок выполнения работы....56
ЛАБОРАТОРНАЯ РАБОТА № 5. КОЛЛЕКЦИИ....58
Краткие теоретические сведения.....58
Коллекции....58
Итераторы....60
Список....62
Множество....63
Карта....64
Стек....66
Очередь....66
Интерфейсы Comparable и Comparator. Сортировка....66
Порядок выполнения работы....70
ЛАБОРАТОРНАЯ РАБОТА № 6. ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ....74
Краткие теоретические сведения.....74
Исключения....74
Модели обработки исключений....80
Порядок выполнения работы....81
ЛАБОРАТОРНАЯ РАБОТА № 7. ФАЙЛОВЫЙ ВВОД И ВЫВОД....83
Краткие теоретические сведения.....83
Потоки ввода-вывода....83
Закрытие потоков....85
Чтение и запись файлов с помощью FileInputStream и FileOutputStream....87
Классы для работы с потоками данных и текстами....89
Сериализация....91
Класс File для работы с файлами и каталогами....94
Порядок выполнения работы....95
ЛАБОРАТОРНАЯ РАБОТА № 8. ЛЯМБДА-ВЫРАЖЕНИЯ....98
Краткие теоретические сведения.....98

Лямбда 98
Лямбды как параметры и результаты методов..... 102
Порядок выполнения работы..... 107
СПИСОК РЕКОМЕНДОВАННОЙ ЛИТЕРАТУРЫ 111
Приложение А..... 112
Примеры на Java 112
ЛАБОРАТОРНАЯ РАБОТА № 1..... 112
ЛАБОРАТОРНАЯ РАБОТА № 2..... 114
ЛАБОРАТОРНАЯ РАБОТА № 3..... 118
ЛАБОРАТОРНАЯ РАБОТА № 4..... 124
Приложение В..... 128
Примеры на C++..... 128
ЛАБОРАТОРНАЯ РАБОТА № 1..... 128
ЛАБОРАТОРНАЯ РАБОТА № 2..... 130
ЛАБОРАТОРНАЯ РАБОТА № 3..... 136
ЛАБОРАТОРНАЯ РАБОТА № 4..... 141

ВВЕДЕНИЕ

Данные методические указания предназначены для студентов специальности 1-40 01 01 «Программное обеспечение информационных технологий» по курсу «Объектно-ориентированные технологии программирования и стандарты проектирования» для выполнения лабораторных работ.

Для правильного выполнения работ необходимы знания теоретического материала по данному курсу, языка программирования Java и навыков работы в средах разработки Eclipse, IntelliJ IDEA, NetBeans. Также для успешного выполнения задания необходимо изучение краткой теории к соответствующей лабораторной работе.

Для защиты лабораторной работы студенту необходимо:

a) предоставить отчет о выполнении лабораторной работы;
b) продемонстрировать программу;
c) выполнить дополнительное задание;
d) ответить на вопросы по теме текущей лабораторной работы;
e) ответить на вопросы по теме предыдущей лабораторной работы.

В отчет по выполненной лабораторной работе входят:

1. Формулировка задания на лабораторную работу.
2. Вариант задания на лабораторную работу.
3. Описание хода выполнения лабораторной работы: перечисление всех выполненных шагов.
4. Скриншоты. На каждый пункт должен быть один скриншот, не надо нарезать файл исходного кода на скриншоты, если что-то не помещается.
5. Листинги исходных кодов. В листинги исходный кодов должны быть добавлены файлы всех реализованных классов/методов. Файл включается в листинг тогда, когда в листинге уже присутствуют файлы/классы/методы, которые используются в этом файле.

В методических указаниях для каждой лабораторной работы отдельно отмечено, сколько и какие скриншоты должны быть включены в отчет.

По ходу изложения ключевые слова языка помечаются жирным шрифтом, а упоминаемые классы и члены классов – курсивом.

В Приложениях А и В приведены исходные коды примеров первых четырех лабораторных работ для облегченных вариантов.

ЛАБОРАТОРНАЯ РАБОТА № 1.
ОБЪЯВЛЕНИЕ И ИСПОЛЬЗОВАНИЕ КЛАССОВ

Цель: Освоить описание класса, атрибутов, конструкторов, методов. Ознакомиться с классом *Object* и его методами.

Краткие теоретические сведения

Классы – основной элемент абстракции, отвечающий за реализацию назначенного ему контракта и обеспечивающий сокрытие реализации. Классы объединяются в пакеты, которые связаны друг с другом только через ограниченное количество методов и классов, не имея никакого представления о процессах, происходящих внутри классов и методов других пакетов.

Классы позволяют провести декомпозицию поведения сложной системы до множества элементарных взаимодействий связанных объектов. Класс определяет структуру и/или поведение некоторого элемента предметной области.

Описание класса

В языке Java класс имеет следующий синтаксис:

[видимость] [модификаторы] **class** НазваниеКласса
extends [ИмяБазовогоКласса]
implements [СписокИменИнтерфейсов] {
атрибуты
Конструкторы
методы
}

– видимость: **public** – открытая видимость, не_указана – пакетная видимость;

– модификаторы: **abstract** – создаётся абстрактный класс, **final** – создаётся не наследуемый класс;

– **extends**: указывается класс-предок (в неявном виде идёт наследование от класса *Object*);

– **implements**: указывается перечень реализуемых интерфейсов.

Видимость членов класса

Атрибуты и методы представляют собой члены класса. Видимость может быть: **public**, **protected**, не_указанной, **private**.

Возможность класса использовать другой класс или члены другого класса зависит от их видимости, расположения класса и отношения класса к тому, к которому обращается.

Таблица 1. – Доступность члена класса в зависимости от видимости

Сам класс * Наследник + Другой класс -	Этот же пакет	**private**	'	**protected**	**public**
*	+	Да	Да	Да	Да
+	+	Нет	Да	Да	Да
-	+	Нет	Да	Да	Да
+	-	Нет	Нет	Да	Да
-	-	Нет	Нет	Нет	Да

Атрибуты класса

Синтаксис атрибута класса:

[видимость] [модификаторы] названиеАтрибута [= значение];

Модификаторы могут быть: **final** – атрибут, значение которого после инициализации нельзя изменить, **static** – статический атрибут класса, **volatile** – запрещение оптимизации для атрибута.

Объекты создаются динамически с помощью операции **new**.

Методы класса

Метод представляет собой набор операторов, которые выполняют определенные действия. Общее определение метода:

[видимость] [модификаторы] [возвращаемыйТип] названиеМетода ([параметры]) **throws** [СписокИсключений] {

Тело Метода

}

– модификаторы: **final** – метод, который не может быть переопределен, **static** – статический метод класса, **abstract** – абстрактный метод класса (может быть использован, если класс тоже абстрактный);

– возвращаемое значение: тип, массив, ссылка, **void** – если метод не возвращает значений;

```
int getInteger()
String getString()
double[] getDoubleArray()
Cat getCat()
```

– параметры: перечень простейших типов и ссылок (порядок определяется самостоятельно);

– **throws**: исключение или перечень исключений, обработка которых не осуществляется внутри метода или которые генерируются внутри метода.

Завершение выполнения метода

Методы могут возвращать некоторое значение. Для этого применяется оператор **return**. После оператора **return** указывается возвращаемое значение, которое является результатом метода. Возвращаемое значение всегда должно иметь тот же тип, что значится в определении функции.

Метод может использовать несколько вызовов оператора **return** для возвращения разных значений в зависимости от некоторых условий:

```
String dayTime(int hour) {
  if (0 > hour || hour > 24) {
    return "Invalid data!!!";
  } else if (hour > 21 || hour < 6) {
    return "Good night!";
  } else if (hour >= 15) {
    return "Good day!";
  } else {
    return "Good morning!";
  }
}
System.out.println(dayTime(1));
System.out.println(dayTime(6));
System.out.println(dayTime(11));
System.out.println(dayTime(16));
System.out.println(dayTime(21));
System.out.println(dayTime(26));
```

```
Good night!
Good morning!
Good morning!
Good day!
Good day!
Invalid data!!!
```

Оператор **return** применяется не только для возвращения значения из метода, но и для выхода из метода.

Перегрузка методов

В программе могут использоваться методы с одним и тем же именем, но с разными типами и/или количеством параметров и/или порядком параметров. Такой механизм называется перегрузкой методов (method overloading).

```
int Method(int i)
int Method(String s)
int Method(float f)
int Method(String s, int i)
int Method(int i, String s)
```

Методы с неопределенным числом параметров

Метод может принимать параметры переменной длины одного типа.

```
int sum(int...nums) {
  int result = 0;
  for (int n: nums) {
    result += n;
  }
  return result;
}
```

Параметр переменной длины может быть только один и должен быть последним параметром.

```
System.out.println(sum(1));
System.out.println(sum(1, 2, 3, 4, 5));
```

```
1
15
```

Конструктор класса

Кроме обычных методов классы могут определять специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Синтаксис конструктора:

[видимость] НазваниеКласса([параметры]) **throws** [СписокИсключений] {

Тело конструктора

}

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор без параметров. Если объявлен хотя бы один конструктор, то конструктор без параметров необходимо перегрузить самостоятельно. Конструктор класса может быть перегружен неограниченное число раз.

Ключевое слово **this** представляет ссылку на текущий экземпляр класса: можно обращаться к переменным и/или методам этого класса, вызывать перегруженный конструктор класса.

```
class Cat {
  protected String name;
  protected int age;
  public Cat() {
    this("Tom", 5);
  }
  public Cat(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public String toString() {
    return name + " is " + age + " years old.";
  }
}
```

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию (для переменных числовых типов это число 0, а для типа *String* и классов – это значение **null**).

Класс Object

Все классы наследуют от класса *Object*. Поэтому все типы и классы могут реализовать те методы, которые определены в классе *Object*.

Таблица 2. – Перечень методов класса *Object*

Метод	Назначение
Object clone()	создаёт новый объект, не отличающийся от клонируемого
boolean equals(Object o)	определяет, равен ли один объект другому
void finalize()	вызывается перед удалением неиспользуемого объекта
Class getClass()	получает класс объекта во время выполнения
int hashCode()	возвращает хеш-код, связанный с вызывающим объектом
void notify()	возобновляет поток, ожидающий объект
void notifyAll()	возобновляет все потоки, ожидающие объект
String toString()	возвращает строку, описывающую объект
void wait() **void** wait(**long** ms) **void** wait(**long** ms, **long** ns)	ожидает выполнения другого потока

Методы *getClass()*, *notify()*, *notifyAll()*, *wait()* являются финальными и их нельзя переопределять.

Метод equals

Метод *equals* сравнивает два объекта на равенство. Предыдущая реализация класса *Cat*, в которой не был переопределен метод *equals*, не позволит корректно сравнивать объекты класса на равенство.

```
class Cat {
  .....
  public boolean equals(Object object) {
    if (this == object) return true;
    if (!(object instance of Cat)) return false;
    Cat cat = (Cat) object;
    return this.name = cat.name && name.age == cat.age;
  }
  .....
}

Cat t0 = new Cat("Tom", 5);
Cat t1 = new Cat("Sam", 10);
System.out.println(p0.equals(p1));
true
```

Метод hashCode

Метод *hashCode* позволяет задать некоторое числовое значение, которое будет соответствовать данному объекту, или его хэш-код. По данному числу, например, можно сравнивать объекты. Следует переопределять метод *hashCode* в каждом классе, где переопределен метод *equals*.

Метод *hashCode* должен возвращать одинаковые и разные значения для двух объектов класса, если они равны или не равны соответственно. Если приложение остановить и запустить снова, *hashCode* может быть другим для одного и того же объекта.

Метод toString

Метод *toString* служит для получения представления данного объекта в виде строки. Метод *toString* вызывается автоматически, когда ваш объект передается методу *println*, оператору сцепления строк или *assert*.

Уничтожение объектов

Автоматическое освобождение памяти, занимаемой объектом, выполняется с помощью механизма «сборки мусора». Когда никаких ссылок на объект не существует, т.е. все ссылки на него вышли

из области видимости программы, предполагается, что объект больше не нужен, и память, занятая объектом, может быть освобождена. «Сборка мусора» происходит нерегулярно во время выполнения программы. Форсировать «сборку мусора» невозможно, можно лишь «рекомендовать» выполнить ее вызовом метода *System.gc()* или *Runtime.getRuntime().gc()*, но виртуальная машина выполнит очистку памяти тогда, когда сама посчитает это удобным. Вызов метода *System.runFinalization()* приведет к запуску метода *finalize()* для объектов, утративших все ссылки.

Порядок выполнения работы

В рамках данного задания необходимо реализовать класс, согласно варианту, для хранения информации из определенной предметной области.

Для демонстрации возможностей класса нужно разработать программу, в которой:

– создается массив для хранения объектов разработанного класса;

– массив заполняется объектами, созданными с помощью конструкторов разработанного класса;

– осуществляется вывод объектов массива в консоль;

– осуществляется две проверки объектов массива между собой на равенство.

В следующей последовательности в классе должны присутствовать:

– атрибуты класса (должны иметь область видимости **protected**);

– конструкторы класса (конструктор по умолчанию, конструктор с параметрами, конструктор копирования);

– mutator и accessor полей класса (пара методов *getAtribute* и *setAtribute* для каждого атрибута класса);

– метод, проверяющий текущий объект и выбранный объект на равенство (переопределение метода *equals*);

– метод формирования информации о текущем объекте в строку (переопределение метода *toString*).

Все наименования: класса, атрибутов класса, методов класса, создаваемых объектов, – должны быть на английском. Наименования должны быть без сокращений, аббревиатур или транслита. Наименования должны соответствовать нотации языка или выбран-

ной стилистике (наименование начинается с заглавной или прописной буквы, и как разделяются слова в наименовании, состоящем из нескольких слов). Файлы, в которых находятся реализованный класс и программа, не должны быть одним и тем же файлом.

Задания для лабораторной работы по вариантам

1. Car:
- номер (целое трехзначное число);
- марка – модель (строка);
- год выпуска (целое число);
- цвет (строка);
- цена (целое число);
- регистрационный номер (строка из семи символов).

2. Bus:
- ФИО водителя (строка);
- номер автобуса (целое число);
- номер маршрута (целое число);
- марка (строка);
- дата начала эксплуатации (строка в формате дд.мм.гггг);
- пробег (целое число).

3. Plain:
- серийный номер (целое восьмизначное число);
- марка – модель (строка);
- год выпуска (целое четырехзначное число);
- признак (Р – пассажирский, С – грузовой);
- время налета (целое число, в часах, меньше 15000).

4. House:
- номер (строка из шести символов);
- номер квартиры (целое число);
- площадь (целое число);
- этаж (целое число);
- количество комнат (целое число);
- улица (строка);
- тип здания (строка);
- срок эксплуатации (целое число).

5. Train:
- номер (целое трехзначное число);

– название (строка);
– станция отправления (строка);
– станция назначения (строка);
– количество вагонов (целое число меньше 30).

6. Patient:
– номер (целое шестизначное число);
– ФИО пациента (строка);
– адрес (строка);
– телефон (строка из 7 символов);
– номер медицинской карты (целое пятизначное число);
– диагноз (строка).

7. Customer:
– номер (целое пятизначное число);
– ФИО владельца фирмы (строка);
– адрес (строка);
– номер кредитной карточки (строка из 16 символов);
– номер банковского счета (строка).

8. File:
– имя файла (строка);
– размер файла (целое число);
– дата создания (в формате дд.мм.гггг);
– время создания (в формате чч:мм).

9. Book:
– код ISBN (строка)
– фамилия и инициалы автора (строка);
– название книги (строка);
– год издания (целое четырехзначное число);
– количество страниц (целое четырехзначное число).

10. Abiturient:
– номер (целое восьмизначное число);
– ФИО студента (строка);
– адрес (строка);
– оценка за ЦТ по математике (целое число);
– оценка за ЦТ по русскому (целое число);
– оценка за ЦТ по физике (целое число).

11. Product:
– номер (целое пятизначное число);

– наименование (строка);
– производитель (строка);
– цена (целое число);
– дата окончания срока хранения (строка в формате дд.мм.гггг);
– количество (целое число).

12. Message:
– номер устройства (целое трехзначное число);
– ID сообщения (целое восьмизначное число в 16-ричной системе);
– текст (строка);
– дата и время отправления (в формате дд.мм.гггг чч:мм).

13. Phone:
– номер (целое семизначное число);
– ФИО владельца (строка);
– дата подключения (в формате дд.мм.гггг);
– тарифный план (строка).

14. Company:
– название (строка);
– УНН (целое десятизначное число);
– ФИО владельца (строка);
– дата основания (в формате дд.мм.гггг).

15. Airlines:
– пункт назначения (строка);
– номер рейса (целое число);
– тип самолета (строка);
– время вылета (в формате чч:мм:сс).

Контрольные вопросы

1. Как осуществляется объявление класса?
2. Как осуществляется объявление атрибута класса?
3. Как осуществляется объявление метода класса?
4. Как осуществляется создание нового объекта класса?
5. Какие области видимости могут иметь члены класса?
6. Как осуществляется завершение выполнения метода?
7. Какой метод можно называть перегруженным?
8. Как осуществляется объявление конструктора класса?
9. Для чего предназначен класс Object?
10. Какие методы определенны в классе Object? Каково назначение каждого из них?

Содержание отчета

Скриншоты (3): разработанного класса, главного класса программы, результата выполнения программы.

Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрирование программы.
3. Выполнение дополнительного задания.
4. Ответ на вопросы по теме текущей лабораторной работы.

ЛАБОРАТОРНАЯ РАБОТА № 2. ОСНОВНЫЕ ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Цель: Ознакомиться с принципами ООП в языке программирования Java. Научиться их практическому применению.

Краткие теоретические сведения. Модификаторы доступа и инкапсуляция

Модификаторы доступа

Все члены класса в языке Java – поля и методы, свойства – имеют модификаторы доступа. В прошлых темах мы уже сталкивались с модификатором **public**. Модификаторы доступа позволяют задать допустимую область видимости для членов класса, то есть контекст, в котором можно употреблять данную переменную или метод.

В Java используются следующие модификаторы доступа:

1) **public**: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором **public**, видны другим классам из текущего пакета и из внешних пакетов;
2) модификатор по умолчанию. Отсутствие модификатора у поля или метода класса предполагает применение к нему модификатора по умолчанию. Такие поля или методы видны всем классам в текущем пакете;
3) **protected**: такой класс или член класса доступен из любого места в текущем классе или пакете, или в производных классах, даже если они находятся в других пакетах;
4) **private**: закрытый класс или член класса, противоположность модификатору **public**. Закрытый класс или член класса доступен только из кода в том же классе.

Рассмотрим модификаторы доступа на примере следующей программы:

```
class Person {
  String name;
  protected int age;
  public String address;
  private String phone;
  public Person(String name, int age, String address, String
phone) {
    this.name = name;
    this.age = age;
```

```
    this.address = address;
    this.phone = phone;
  }
  public void displayName() { System.out.printf("Name: %s\n",
name); }
  void displayAge() { System.out.printf("Age: %d\n", age); }
  private void displayAddress() { System.out.printf("Address:
%s\n", address); }
  void displayPhone() { System.out.printf("Phone: %s\n",
phone); }
}
public class Program {
  public static void main(String...args) {
    Person kate = new Person("Kate", 32, "Baker Street",
"+0123456789");
    kate.displayName();
    kate.displayAge();
    kate.displayAddress(); // compile error
    kate.displayPhone();
    System.out.println(kate.name);
    System.out.println(kate.age);
    System.out.println(kate.address);
    System.out.println(kate.phone); // compile error
  }
}
```

В данном случае оба класса расположены в одном пакете – пакете по умолчанию, поэтому в классе *Program* мы можем использовать все методы и переменные класса *Person*, которые имеют модификатор по умолчанию, **public** и **protected**. А поля и методы с модификатором **private** в классе *Program* не будут доступны. Если бы класс *Program* располагался в другом пакете, то ему были бы доступны только поля и методы с модификатором **public**. Модификатор доступа должен предшествовать остальной части определения переменной или метода.

Инкапсуляция

Казалось бы, почему не объявить все переменные и методы с модификатором **public**, чтобы они были доступны в любой точке программы вне зависимости от пакета или класса? Возьмем, например, поле age, которое представляет возраст. Если другой класс имеет прямой доступ к этому полю, то есть вероятность, что в процессе работы программы ему будет передано некорректное значе-

ние, например, отрицательное число. Подобное изменение данных не является желательным. Либо же мы хотим, чтобы некоторые данные были доступны напрямую, чтобы их можно было вывести на консоль или просто узнать их значение. В этой связи рекомендуется как можно больше ограничивать доступ к данным, чтобы защитить их от нежелательного доступа извне (как для получения значения, так и для его изменения). Использование различных модификаторов гарантирует, что данные не будут искажены или изменены ненадлежащим образом. Подобное сокрытие данных внутри некоторой области видимости называется инкапсуляцией. Вместо непосредственного использования полей, как правило, используют методы доступа.

```
class Person {
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public int getAge() {
    reutrn age;
  }
  public void setAge(int age) {
    if (age > 0 && age < 110) {
      this.age = age;
    }
  }
  public void display() {
    System.out.printf("Name: %s, Age: %d", name, age);
  }
  public String toString() {
    return String.format("%s is %d years old.", name, age);
  }
}
```

Вместо непосредственной работы с полями *name* и *age* в классе *Person* мы будем работать с методами, которые устанавливают и возвращают значения этих полей. Методы *setName*, *setAge* и наподобие еще называют мьютейтерами (mutator), так как они из-

меняют значения поля. А методы *getName*, *getAge* и наподобие называют аксессерами (accessor), так как с их помощью мы получаем значение поля.

Причем в эти методы мы можем вложить дополнительную логику. Например, в данном случае при изменении возраста производится проверка, насколько соответствует новое значение допустимому диапазону.

```
Person kate = new Person("Kate", 30);
System.out.println(kate);
kate.setAge(33);
System.out.println(kate);
kate.setAge(12345);
System.out.println(kate);
```

```
Kate is 30 years old.
Kate is 33 years old.
Kate is 33 years old.
```

Наследование

Одним из ключевых аспектов объектно-ориентированного программирования является наследование. С помощью наследования можно расширить функционал уже имеющихся классов за счет добавления нового функционала или изменения старого. Допустим, имеется класс *Person*, описывающий отдельного человека (предыдущий пример программы), и возникает необходимость добавить еще один класс, который описывает сотрудника предприятия – класс *Employee*. Так как этот класс реализует тот же функционал, что и класс *Person*, ведь сотрудник – это также и человек, то было бы рационально сделать класс *Employee* производным (наследником, подклассом) от класса *Person*, который, в свою очередь, называется базовым классом, родителем или суперклассом.

Чтобы объявить один класс наследником другого, надо использовать после имени класса-наследника ключевое слово **extends**, после которого идет имя базового класса. Для класса *Employee* базовым является *Person*, и поэтому класс *Employee* наследует все те же поля и методы, которые есть в классе *Person*.

Производный класс имеет доступ ко всем методам и полям базового класса (даже если базовый класс находится в другом пакете) кроме тех, которые определены с модификатором **private**. При этом производный класс также может добавлять свои поля и методы:

```
class Employee extends Person {
  private String company;
```

```
  public Employee(string name, int age, String company) {
    super(name, age);
    this.company = company;
  }
  public void work() {
    System.out.printf("Company: %s\n", company);
  }
  public String toString() {
    return super.toString() + " Working in " + company + ".";
  }
}
```

В данном случае класс *Employee* добавляет поле company, которое хранит место работы сотрудника, а также метод *work*.

Если в базовом классе определены конструкторы, то в конструкторе производного класса необходимо вызвать один из конструкторов базового класса с помощью ключевого слова **super**. Например, класс *Person* имеет конструктор, который принимает два параметра. Поэтому в классе *Employee* в конструкторе нужно вызвать конструктор класса *Person*. После слова **super** в скобках идет перечисление передаваемых аргументов. Таким образом, установка имени сотрудника делегируется конструктору базового класса. При этом вызов конструктора базового класса должен идти в самом начале в конструкторе производного класса.

```
Employee sam = new Employee("Sam", 20, "Microsoft");
sam.display();
sam.work();
System.out.println(sam);
```

```
Name: Sam, Age: 20
Company: Microsoft
Sam is 20 years old. Working in Microsoft.
```

Переопределение методов

Производный класс может определять свои методы, а может переопределять методы, которые унаследованы от базового класса. Например, переопределим в классе *Employee* метод *display*:

```
class Employee extends Person {
  private String company;
  public Employee(string name, int age, String company) {
    super(name, age);
    this.company = company;
  }
  @Override
  public void display() {
```

```
    super.display();
    System.out.printf(" Company: %s\n", company);
  }
  public String toString() {
    return super.toString() + " Working in " + company + ".";
  }
}
```

Перед переопределяемым методом указывается аннотация **@Override**. Данная аннотация необязательна. При переопределении метода он должен иметь уровень доступа не меньше, чем уровень доступа в базовом классе. Например, если в базовом классе метод имеет модификатор **public**, то и в производном классе метод должен иметь модификатор **public**. С помощью ключевого слова **super** мы также можем обратиться к реализации методов базового класса.

Запрет наследования

Хотя наследование очень интересный и эффективный механизм, в некоторых ситуациях его применение может быть нежелательным. В этом случае можно запретить наследование с помощью ключевого слова **final**. Кроме запрета наследования можно также запретить переопределение отдельных методов.

Полиморфизм

Способность ссылки динамически определять версию переопределенного метода в зависимости от переданного ссылке в сообщении типа объекта называется полиморфизмом. Полиморфизм является основой для реализации механизма динамического, или позднего, связывания.

При вызове переопределенного метода виртуальная машина динамически находит и вызывает именно ту версию метода, которая определена в подклассе. Данный процесс еще называется **dynamic method lookup,** или динамический поиск метода, или динамическая диспетчеризация методов.

Наследование и переопределение методов открывают нам большие возможности. Прежде всего мы можем передать переменной суперкласса ссылку на объект подкласса:

```
Person tom = new Person("Tom", 20);
Person sam = new Employeer("Sam", 20, "Oracle");
tom.display();
sam.display();
```

Так как *Employee* наследуется от *Person*, то объект *Employee* является в то же время и объектом *Person*. Грубо говоря, любой работник предприятия одновременно является человеком.

Однако несмотря на то, что переменная представляет объект *Person*, виртуальная машина видит, что в реальности она указывает на объект *Employee*. Поэтому при вызове методов у этого объекта будет вызывать та версия методов, которая определена в классе *Employee*, а не в *Person*.

```
Name: Tom, Age: 20
Name: Sam, Age: 20 Company: Oracle
```

Абстрактные классы

Кроме обычных классов в Java есть абстрактные классы. Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников, а производные классы уже реализуют этот функционал. При определении абстрактных классов используется ключевое слово **abstract**. Также:

– мы не можем использовать конструктор абстрактного класса для создания его объекта;

– кроме обычных методов абстрактный класс может содержать абстрактные методы. Такие методы определяются с помощью ключевого слова **abstract** и не имеют никакого функционала;

– производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Также следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный.

Зачем нужны абстрактные классы? Допустим, мы делаем программу для обслуживания банковских операций и определяем в ней три класса: *Person*, который описывает человека, *Employee*, который описывает банковского служащего, и класс *Client*, который представляет клиента банка. Очевидно, что классы *Employee* и *Client* будут производными от класса *Person*, так как оба класса имеют некоторые общие поля и методы. И так как все объекты будут представлять либо сотрудника, либо клиента банка, то напрямую мы от класса *Person* создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным.

```
abstract class Person {
  ...
  public abstract void display();
}
class Employeee extends Person {
  ...
  public void display() { ... }
}
class Client extends Person {
  ...
  public void display() { ... }
}
```

Иерархия наследования и преобразование типов

В прошлой главе говорилось о преобразованиях объектов простых типов. Однако с объектами классов все происходит немного по-другому. Допустим, у нас есть следующая иерархия классов:

```
class Person {
  ...
}
class Employeee extends Person {
  ...
}
class Client extends Person {
  ...
}
```

В этой иерархии классов можно проследить следующую цепь наследования: *Object* (все классы неявно наследуются от типа *Object*) *-> Person -> Employee* | *Client*.

Суперклассы обычно размещаются выше подклассов, поэтому на вершине наследования находится класс *Object*, а в самом низу – *Employee* и *Client*.

Объект подкласса также представляет объект суперкласса. Поэтому в программе мы можем написать следующим образом:

```
Object tom = new Person(...);
Object sam = new Employee(...);
Object kate = new Client(...);
Object bob = new Client(...);
Object alice = new Employee(...);
```

Это так называемое восходящее преобразование (от подкласса внизу к суперклассу вверху иерархии), или upcasting. Такое преобразование осуществляется автоматически.

Обратное не всегда верно. Например, объект *Person* не всегда является объектом *Employee* или *Client*. Поэтому нисходящее преобразование, или downcasting, от суперкласса к подклассу автоматически не выполняется. В этом случае нам надо использовать операцию преобразования типов.

```
Object sam = new Employee(...);
Employee eSam = (Employee) sam; // downcasting
eSam.display();
```

В данном случае переменная *sam* приводится к типу *Employee*, и затем через объект *eSam* мы можем обратиться к функционалу объекта *Employee*.

Мы можем преобразовать объект *Employee* по всей прямой линии наследования от *Object* к *Employee*.

```
Object sam = new Employee(...);
((Employee)sam).display();
```

Но рассмотрим еще одну ситуацию:

```
Object kate = new Client(...);
((Employee)kate).display();
```

В данном случае переменная типа *Object* хранит ссылку на объект *Client*. Мы можем без ошибок привести этот объект к типам *Person* или *Client*. Но при попытке преобразования к типу *Employee* мы получим ошибку во время выполнения, так как *kate* не представляет объект типа *Employee*.

Здесь мы явно видим, что переменная *kate* – это ссылка на объект *Client*, а не *Employee*. Однако нередко данные приходят извне, и мы можем точно не знать, какой именно объект эти данные представляют. Соответственно, возникает большая вероятность столкнуться с ошибкой. И перед тем как провести преобразование типов, мы можем проверить, возможно ли выполнить приведение с помощью оператора **instance of**:

```
Object kate = new Client(...);
if (kate instance of Employee) {
  ((Employee)kate).display();
} else {
  System.out.println("Conversion is invalid!");
}
```

Выражение *kate* **instance of** *Employee* проверяет, является ли переменная *kate* объектом типа *Employee*. Но так как в данном случае явно не является, то такая проверка вернет значение **false**, и преобразование не сработает.

Статические члены и модификатор static

Кроме обычных методов и полей класс может иметь статические поля, методы, константы и инициализаторы. Например, главный класс программы имеет метод *main*, который является статическим:

```
public static void main(String...args) {
}
```

Перед объявлением статических переменных, констант, методов и инициализаторов указывается ключевое слово **static**.

Статические поля

При создании объектов класса для каждого объекта создается своя копия нестатических обычных полей, а статические поля являются общими для всего класса. Поэтому они могут использоваться без создания объектов класса. Например, создадим статическую переменную:

```
class Person {
  public static int counter = 1;
  public Person() {
    counter++;
  }
}
```

Класс *Person* содержит статическую переменную *counter*, которая увеличивается в конструкторе. То есть при создании каждого нового объекта *Person* эта переменная будет увеличиваться.

Так как переменная *counter* статическая, то мы можем обратиться к ней в программе по имени класса:

```
Person tom = new Person();
Person bob = new Person();
System.out.println(Person.counter);
Person.counter = 10;
System.out.println(Person.counter);
```

```
3
10
```

Статические константы

Также статическими бывают константы, которые являются общими для всего класса.

```
class Math {
  public static final double PI = 3.14;
}
System.out.println(Math.PI);
```

```
3.14
```

Стоит отметить, что на протяжении всех предыдущих тем уже активно использовались статические константы. В частности, в выражении

```
System.out.println("hi");
```

```
hi
```

out как раз представляет статическую константу класса *System*. Поэтому обращение к ней идет без создания объекта класса *System*.

Статические инициализаторы

Статические инициализаторы предназначены для инициализации статических переменных, либо для выполнения таких действий, которые выполняются при создании самого первого объекта. Например, определим статический инициализатор:

```
class DemoClassWithInitializers {
  public static int counter;
  private String name;
  private int id;
  static {
    counter = 100;
    System.out.println("Static initializer.");
  }
  {
    this.name = "Bob_";
    System.out.println("Initializer.");
  }
  public DemoClassWithInitializers() {
    this.id = counter++;
    this.name += this.id;
    System.out.println("Contructor.");
  }
  public String toString() {
    return this.name;
  }
}
```

Статический инициализатор определяется как обычный, только перед ним ставится ключевое слово **static**. В данном случае в статическом инициализаторе мы устанавливаем начальное значение статического поля counter и выводим на консоль сообщение.

В самой программе создаются два объекта класса *Person*, поэтому консольный вывод будет выглядеть следующим образом:

```
DemoClassWithInitializers d0 = new
DemoClassWithInitializers();
DemoClassWithInitializers d1 = new
DemoClassWithInitializers();
```

```
Static initializer.
Initializer.
Contructor.
Initializer.
Contructor.
```

Стоит учитывать, что вызов статического инициализатора производится только перед созданием самого первого объекта класса или просто при обращении к классу.

Статические методы

Статические методы также относятся ко всему классу в целом. В примере выше статическая переменная *counter* была доступна извне, и мы могли изменить ее значение вне класса *Person*. Сделаем ее недоступной для изменения извне, но доступной для чтения. Для этого используем статический метод:

```
class Person {
  private static int counter = 0;
  ...
  public static int getCounter() {
    return counter;
  }
  ...
}
```

Теперь статическая переменная недоступна извне, она приватная. Для обращения к статическому методу используется имя класса *Person.counter*.

```
System.out.print(Person.getCounter() + " ");
Person p0 = new Person();
Person p1 = new Person();
System.out.println(Person.getCounter() + " ");
Person.counter = 10;
System.out.println(Person.getCounter());
```

```
0 2 10
```

При использовании статических методов надо учитывать ограничения: в статических методах мы можем вызывать только другие статические методы и использовать только статические переменные.

Методы определяются как статические, когда они не затрагивают состояние объекта, то есть его нестатические поля и константы, и для вызова метода нет смысла создавать экземпляр класса. Например:

```
class Operation {
  public static int sum(int x, int y) { x + y; }
```

```
  public static int substract(int x, int y) { return x - y; }
  public static int multiply(int x, int y) { return x * y; }
}

System.out.println(Operation.sum(45, 23));
System.out.println(Operation.substract(45, 23));
System.out.println(Operation.multiply(45, 23));
System.out.println(Operation.sum(45, 23));
68
22
1035
```

В данном случае для методов *sum*, *subtract*, *multiply* не имеет значения, какой именно экземпляр класса *Operation* используется. Эти методы работают только с параметрами, не затрагивая состояние класса. Поэтому их можно определить как статические.

Порядок выполнения работы

Согласно варианту задания, выбирается базовый класс, а остальные классы будут его наследниками. Для демонстрации возможностей реализованной иерархии классов необходимо разработать программу, в которой для хранения объектов используется массив, в массив добавляется 10 объектов (объекты создаются с помощью конструкторов с параметрами) и в цикле осуществляется вывод объектов в консоль.

К реализации каждого класса предъявляются требования, аналогичные требованиям к реализации класса из лабораторной работы № 1.

Базовый класс должен:

1) содержать 3–5 атрибутов;
2) быть абстрактным;
3) содержать статическую переменную, отвечающую за количество созданных объектов.

Класс наследник должен:

1) содержать 2–4 атрибута (то есть дочерний класс должен описываться 5–9 атрибутами, с учетом атрибутов базового класса);
2) использовать часть функционала базового класса при создании объекта (должен вызывать конструктор базового класса), проверке объектов на равенство и формировании информации о текущем объекте в строку.

Классы должны быть размещены по пакетам следующим образом (здесь «]» означает, что название должно быть дано согласно варианту):

src

[classes]

[BaseClass].java

[ChildClass1].java

[ChildClass2].java

[ChildClass3].java

Program.java

В базовом пакете проекта должен располагаться главный класс программы *Program*/*Main*. В пакете [*classes*] должны располагаться базовый класс и дочерние классы. Пакет [*classes*] должен быть переименован в соответствии с предметной областью согласно варианту задания (например, если у нас иерархия геометрических фигур: фигура, квадрат, треугольник, круг, – то пакет может называться *figures*).

Задания для лабораторной работы по вариантам

1. Служащий, персона, рабочий, инженер.
2. Млекопитающие, парнокопытные, птицы, животные.
3. Тест, экзамен, выпускной экзамен, испытание.
4. Организация, страховая компания, судостроительная компания, завод.
5. Журнал, книга, печатное издание, учебник.
6. Квитанция, накладная, документ, чек.
7. Место, область, город, мегаполис.
8. Двигатель, двигатель внутреннего сгорания, дизель, турбореактивный двигатель.
9. Автомобиль, поезд, транспортное средство, экспресс.
10. Рабочий, кадры, инженер, администрация.
11. Республика, монархия, королевство, государство.
12. Корабль, пароход, парусник, корвет.
13. Студент, преподаватель, персона, завкафедрой.
14. Деталь, механизм, изделие, узел.
15. Игрушка, продукт, товар, молочный продукт.

Контрольные вопросы

1. Какие существуют модификаторы доступа? Какие области видимости они предоставляют?

2. Какие существуют основные принципы объектно-ориентированного программирования?

3. Что такое инкапсуляция? Что такое mutators и accessors? Каково их предназначение?

4. Что такое наследование? Как осуществляется наследование класса?

5. Что такое переопределение метода базового класса? Когда метод может быть переопределен и когда не может быть переопределен?

6. Что такое полиморфизм?

7. Как работает динамическая диспетчеризация методов?

8. Что такое абстрактный класс? Что такое абстрактные методы? Для чего используются абстрактные классы?

9. Что такое восходящее и нисходящее преобразование типов?

10. Когда используются статические атрибуты и методы класса? Чем они отличаются от нестатических?

Содержание отчета

Скриншоты (6): базового класса, каждого дочернего класса, главного класса программы и консоли.

Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответ на вопросы по теме текущей лабораторной работы.
5. Ответ на вопросы по теме предыдущей лабораторной работы.

ЛАБОРАТОРНАЯ РАБОТА № 3. ИНТЕРФЕЙСЫ

Цель: Научиться создавать и реализовывать интерфейсы. Освоить создание программы, состоящей из нескольких пакетов.

Краткие теоретические сведения.
Интерфейсы

Механизм наследования очень удобен, но он имеет свои ограничения. В частности, мы можем наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.

В языке Java подобную проблему частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. Один класс может применить множество интерфейсов.

Чтобы определить интерфейс, используется ключевое слово **interface**. Например:

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

Данный интерфейс называется *Printable*. Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов. Так, в данном случае объявлен один метод, который не имеет реализации.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию у них доступ **public**, так как цель интерфейса – определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Чтобы класс применил интерфейс, надо использовать ключевое слово **implements**:

```
interface Printable {
  void print();
}
class Book implements Printable {
  private String name;
  private String author;
  public Book(String name, String author) {
    this.name = name;
```

```
    this.author = author;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getAuthor() {
    return author;
  }
  public void setAuthor(String author) {
    this.author = author;
  }
  public void print() {
    System.out.printf("%s (%s)\n", name, author);
  }
}
```

В данном случае класс *Book* реализует интерфейс *Printable*. При этом надо учитывать, что если класс применяет интерфейс, то он должен реализовать все методы интерфейса, как в случае выше реализован метод *print*. Потом в методе *main* мы можем создать объект класса *Book* и вызвать его метод *print*.

```
Book book = new Book("Java. Complete Referense.", "H.
Shildt");
book.print();
```

```
Java. Complete Referense. (H. Shildt)
```

Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут эти методы реализовать. В тоже время мы не можем напрямую создавать объекты интерфейсов, поэтому следующий код не будет работать:

```
Printable pr = new Printable();
pr.print();
```

Одним из преимуществ использования интерфейсов является то, что они позволяют добавить в приложение гибкости. Например, в дополнение к классу *Book* определим еще один класс, который будет реализовывать интерфейс *Printable*:

```
class Journal implements Printable {
  private String name;
  public Journal(String name) {
    this.name = name;
  }
```

```
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public void print() {
    System.out.println(name);
  }
}
```

Класс *Book* и класс *Journal* связаны тем, что они реализуют интерфейс *Printable*. Поэтому мы динамически в программе можем создавать объекты *Printable* как экземпляры обоих классов:

```
Printable printable = new Book("Java. Complete Referense.",
"H. Shildt");
printable.print();
printable = new Journal("Foreign Policy");
printable.print();
```

```
Java. Complete Referense. (H. Shildt)
Foreign Policy
```

Интерфейсы в преобразованиях типов

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Например, так как класс *Journal* реализует интерфейс *Printable*, то переменная типа *Printable* может хранить ссылку на объект типа *Journal*:

```
Printable p = new Journal("Foreign Policy");
p.print();
String name = ((Journal)p).getName();
System.out.println(name);
```

И если мы хотим обратиться к методам класса *Journal*, которые определены не в интерфейсе *Printable*, а в самом классе *Journal*, то нам надо явным образом выполнить преобразование типов.

Методы по умолчанию

Ранее, до JDK 8, при реализации интерфейса мы должны были обязательно реализовать все его методы в классе, а сам интерфейс мог содержать только определения методов без конкретной реализации. В JDK 8 была добавлена такая функциональность как методы по умолчанию. И теперь интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая исполь-

зуется, если класс, реализующий данный интерфейс, не реализует метод. Например, создадим метод по умолчанию в интерфейсе *Printable*:

```
interface Printable {
  default void print() {
    System.out.println("Undefined printable!");
  }
}
```

Метод по умолчанию – это обычный метод без модификаторов, который помечается ключевым словом **default**. Затем в классе *Journal* нам необязательно этот метод реализовать, хотя мы можем его и переопределить:

```
class Journal implements Printable {
  private String name;
  public Journal(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
Printable p = new Journal("Foreign Policy");
p.print();
```

```
Undefined printable!
```

Статические методы

Начиная с JDK 8, в интерфейсах доступны статические методы – они аналогичны методам класса:

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

```
read printable
```

Чтобы обратиться к статическому методу интерфейса, также, как и в случае с классами, пишут название интерфейса и метод.

Приватные методы

По умолчанию все методы в интерфейсе фактически имеют модификатор **public**. Однако начиная с Java 9, мы также можем определять в интерфейсе методы с модификатором **private**. Они могут быть статическими и нестатическими, но они не могут иметь реализации по умолчанию.

Подобные методы могут использоваться только внутри самого интерфейса, в котором они определены. То есть, к примеру, нам надо выполнять в интерфейсе некоторые повторяющиеся действия, и в этом случае такие действия можно выделить в приватные методы:

```
interface Calculatable {
  default int sum(int a, int b) {
    return sumAll(a, b);
  }
  default int sum(int a, int b, int c) {
    return sumAll(a, b, c);
  }
  private int summAll(int...values) {
    int result = 0;
    for (int v: values) {
      result += v;
    }
    return result;
  }
}

class Calculation implements Calculatable {
}

Calculatable c = new Calculation();
System.out.println(c.sum(1, 2));
System.out.println(c.sum(1, 2, 3));
```

```
3
6
```

Константы в интерфейсах

Кроме методов в интерфейсах могут быть определены статические константы:

```
interface Stateable {
  int OPEN = 1;
  int CLOSED = 0;
  void printState(int n);
}
```

Хотя такие константы также не имеют модификаторов, но по умолчанию они имеют модификатор доступа **public static final**, и поэтому их значение доступно из любого места программы.

Применение констант:

```
interface Stateable {
    int OPEN = 1;
    int CLOSED = 0;
    void printState(int n);
}
class WaterPipe implements Stateable {
    public void printState(int n) {
      if (n == OPEN) {
        System.out.println("Water is opened!");
      } else if (n == CLOSED) {
        System.out.println("Water is closed!");
      } else {
        System.out.println("State is invalid!");
      }
    }
}
WaterPipe pipe = new WaterPipe();
pipe.printState(1);
```

```
Water is opened!
```

Множественная реализация интерфейсов

Если нам надо применить в классе несколько интерфейсов, то они все перечисляются через запятую после слова **implements**:

```
interface Printable {
  void print();
}
interface Searchable {
  void search();
}
class Book implements Printable, Searchable {
  public void print() {
    ...
  }
  public void search() {
    ...
  }
}
```

Наследование интерфейсов

Интерфейсы, как и классы, могут наследоваться:

```
interface Printable {
```

```
  void print();
}
interface Searchable {
  void search();
}
interface BookStuff extends Printable, Searchable {
  void reference();
}

class Book implements BookStuff {
  public void print() {
    ...
  }
  public void search() {
    ...
  }
  public void reference() {
    ...
  }
}
```

При применении этого интерфейса класс *Book* должен будет реализовать как методы интерфейса *BookStuff*, так и методы базовых интерфейсов *Printable* и *Searchable*.

Вложенные интерфейсы

Как и классы, интерфейсы могут быть вложенными, то есть могут быть определены в классах или других интерфейсах. Например:

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

Использование интерфейса будет аналогично предыдущим случаям:

```
Printer.Printable p = new Journal("Foreign Affairs");
```

Интерфейсы как параметры и результаты методов

Так же, как и в случае с классами, интерфейсы могут использоваться в качестве типа параметров метода или в качестве возвращаемого типа:

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

```
class Book implements Printable {
  private String name;
  private String author;
  public Book(String name, String author) {
    this.name = name;
    this.author = author;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getAuthor() {
    return author;
  }
  public void setAuthor(String author) {
    this.author = author;
  }
  public void print() {
    System.out.printf("%s (%s)\n", name, author);
  }
}

class Journal implements Printable {
  private String name;
  public Journal(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public void print() {
    System.out.println(name);
  }
}

public class Program {
  public static void read(Printable p) {
    p.print();
  }
  public static Printable createPrintable(boolean option,
String name) {
```

```
    if (option) {
      return new Book(name, "Undefined");
    } else {
      return new Journal(name);
    }
  }
  public static void main(String...args) {
    Printable p = createPrintable(false, "Foreign Affairs");
    p.print();
    read(new Book("Java for impatients", "Cay Horstmann"));
    read(new Journal("Java Daily News"));
  }
}
```

```
Foreign Affairs
Java for impatients (Cay Horstmann)
Java Daily News
```

Метод *read()* в качестве параметра принимает объект интерфейса *Printable*, поэтому в этот метод мы можем передать как объект *Book*, так и объект *Journal*. Метод *createPrintable()* возвращает объект *Printable*, поэтому мы также можем возвратить как объект *Book*, так и *Journal*.

Интерфейсы в механизме обратного вызова

Одним из распространенных способов использования интерфейсов в Java является создание обратного вызова. Суть обратного вызова состоит в том, что мы создаем действия, которые вызываются при других действиях. Стандартный пример – нажатие на кнопку. Когда мы нажимаем на кнопку, мы производим действие, но в ответ на это нажатие запускаются другие действия. Например, нажатие на значок принтера запускает печать документа на принтере и т.д.

```
interface EventHandler {
  void execute();
}
class ButtonClickHandler implements EventHandler {
  public void execute() {
    System.out.println("Button is pressed!");
  }
}
class Button {
  private EventHandler handler;
  public Button(EventHandler action) {
    this.handler = action;
  }
```

```
  public void click() {
    handler.execute();
  }
}
Button b = new Button(new ButtonClickHandler());
b.click();
b.click();
Button is pressed!
Button is pressed!
```

Итак, здесь у нас определен класс *Button*, который в конструкторе принимает объект интерфейса *EventHandler* и в методе *click* (имитация нажатия) вызывает метод *execute* этого объекта.

Далее определяется реализация *EventHandler* в виде класса *ButtonClickHandler,* и в основной программе объект этого класса передается в конструктор Button. Таким образом, через конструктор мы устанавливаем обработчик нажатия кнопки, и при каждом вызове метода *button.click()* будет вызываться этот обработчик.

Но, казалось бы, зачем нам выносить все действия в интерфейс, его реализовывать. Почему бы не написать проще сразу в классе *Button*:

```
class Button {
  public void click() {
    System.out.println("Button is pressed!");
  }
}
```

Дело в том, что на момент определения класса нам не всегда бывают точно известны те действия, которые должны производиться. Особенно если класс *Button* и класс основной программы находятся в разных пакетах, библиотеках и могут проектироваться разными разработчиками. К тому же у нас может быть несколько кнопок – объектов *Button*, и для каждого объекта надо определить свое действие. Например, изменим главный класс программы:

```
interface EventHandler {
  void execute();
}

Button tvButton = new Button(new EventHandler() {
  private boolean on = false;
  public void execute() {
    if (on) {
      on = false;
      System.out.println("TV is off!");
```

```
    } else {
      on = true;
      System.out.println("TV is on!");
    }
  }
});

Button printButton = new Button(new EventHandler() {
  public void execute() {
    System.out.println("Printer is print documents...");
  }
});
tvButton.click();
printButton.click();
tvButton.click();
printButton.click();
TV is on!
Printer is print documents...
TV is off!
Printer is print documents...
```

Здесь у нас две кнопки – одна для включения-выключения телевизора, а другая для печати на принтере. Вместо того, чтобы создавать отдельные классы, реализующие интерфейс *EventHandler*, здесь обработчики задаются в виде анонимных объектов, которые реализуют интерфейс *EventHandler*. Причем обработчик кнопки телевизора хранит дополнительное состояние в виде логической переменной *on*.

И в завершение надо сказать, что интерфейсы в данном качестве особенно широко используются в различных графических API: AWT, Swing, JavaFX, где обработка событий объектов – элементов графического интерфейса, особенно актуальна.

Порядок выполнения работы

В реализованную лабораторную работу № 2 нужно внести следующие изменения:

– добавить возможность сравнивать объекты между собой по умолчанию и по любому выбранному полю базового класса;

– ввод объекта с консоли должен осуществляться в отдельном методе для каждого дочернего класса;

– для хранения объектов должен использоваться список.

Для демонстрации нужно разработать программу *Program*, в которой:

– для хранения объектов используется список;

– в список добавляется 15 объектов (объекты создаются с помощью конструкторов с параметрами);

– осуществляется вывод объектов списка в консоль;

– осуществляется сортировка и повторный вывод объектов списка в консоль.

За сравнение объектов дочерних классов между собой по умолчанию отвечает интерфейс *[BaseClass]Comparable*. Этот интерфейс содержит метод *compareTo,* который в качестве аргумента принимает ссылку на объект базового класса, а возвращает целочисленное значение – результат сравнения текущего объекта и объекта, переданного в аргументе метода (< 0 – меньше, == 0 – равно, > 0 – больше). Этот интерфейс может быть наследником *Comparable<[BaseClass]>*.

Базовый класс должен принадлежать интерфейсу *[BaseClass]Comparable*. Базовый класс может реализовать интерфейс, а дочерние классы должны это сделать таким образом, чтобы массив или список объектов можно было упорядочить сначала по классам, а затем по любому из атрибутов дочернего класса.

За сравнение объектов дочерних классов по полям базового класса отвечает интерфейс *[BaseClass]Comparator*. Этот интерфейс содержит метод *compare,* который в качестве аргументов принимает ссылки на два объекта базового класса, а возвращает целочисленное значение – результат их сравнения (< 0 – меньше, == 0 – равно, > 0 – больше). Этот интерфейс может быть наследником *Comparator<[BaseClass]>*.

Классы *[BaseClass]Comparator** реализуют интерфейс *[BaseClass] Comparator* – реализация метода *compare* определяет порядок элементов.

Интерфейсы *[BaseClass]Comparator* и *[BaseClass]Comparable* располагаются в пакете *[classes]*.

В классе *Input[BaseClass]* должны быть реализованы статические методы *Input[ChildClass1]*, *Input[ChildClass2]*, *Input[ChildClass3]* – по одному на каждый из дочерних классов. В каждом из этих методов сперва осуществляется ввод атрибутов соответствующего класса, а затем метод возвращает объект дочернего класса, созданный с помощью конструктора с параметрами соответствующего класса. Класс *Input[BaseClass]* должен располагаться в пакете *utils*.

Класс список *[BaseClass]List* может быть реализован на основе массива, ссылок или быть наследником от *ArrayList<[BaseClass]>*. Класс список *[BaseClass]List* должен располагаться в пакете *list*. У класса списка должны быть следующие методы:

- добавление элемента в конец списка;
- добавление элемента в произвольное положение в списке;
- удаление элемента по индексу;
- удаление всех элементов из списка;
- получение элемента по индексу;
- изменение элемента по индексу;
- сортировка списка по порядку, предусмотренному по умолчанию;
- сортировка списка с помощью выбранного компаратора;
- вывод списка в консоль.

Классы должны быть размещены по пакетам, следующими образом:

```
src
    list
        [BaseClass]List.java
    [classes]
        [BaseClass].java
        [BaseClass]Comparable.java
        [BaseClass]Comparator.java
        [BaseClass]Comparator1.java
        [BaseClass]Comparator2.java
        [BaseClass]Comparator3.java
        [ChildClass1].java
        [ChildClass2].java
        [ChildClass3].java
    utils
        Input[BaseClass].java
    Program.java
```

Контрольные вопросы

1. Что такое интерфейсы? Чем интерфейсы отличаются от классов?
2. Как осуществляется описание интерфейса?
3. Как осуществляется описание членов интерфейса?

4. Как осуществляется реализация интерфейса?
5. Как сделать метод интерфейса методом по умолчанию?
6. Как используются статические члены интерфейса?
7. Что нужно сделать для того, чтобы в классе можно было реализовать больше одного интерфейса?
8. Как определить метод интерфейса, который может быть использован только внутри него самого?
9. Как один интерфейс может наследовать другой интерфейс? Может ли он в этом случае наследовать больше одного интерфейса?
10. Каким образом интерфейсы используются в механизме обратного вызова?

Содержание отчета

Скриншоты (5): каждого интерфейса, класса списка, главного класса программы, результата выполнения программы.

Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответ на вопросы по теме текущей лабораторной работы.
5. Ответ на вопросы по теме предыдущей лабораторной работы.

ЛАБОРАТОРНАЯ РАБОТА № 4. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ОБОБЩЕНИЙ

Цель: Освоить описание и реализацию параметризированных методов и классов, научиться использовать шаблоны методов и классов.

Краткие теоретические сведения. Обобщение

Обычные классы и методы работают с конкретными типами: либо с примитивами, либо с классами. При создании программы, которая должна работать с разными типами, могут возникнуть проблемы.

Одним из механизмов обеспечения универсальности кода в объектно-ориентированных языках является полиморфизм. Например, метод, который получает в аргументе объект базового класса, а затем использует этот метод с любым классом, производным от него. Метод становится более универсальным, а область его применения расширяется. Это относится и к классам – использование базового класса вместо производного обеспечивает дополнительную гибкость. Если в аргументе метода передается интерфейс вместо класса, то ограничения ослабляются и в них включается все, что реализует данный интерфейс, – в том числе и классы, которые еще не были созданы. Это дает возможность реализовать интерфейс, чтобы соответствовать требованиям класса или метода. Таким образом, если присутствует возможность создать новый класс, то интерфейсы позволяют выходить за рамки иерархий классов.

Обобщение позволяет указать то, что код будет работать не с конкретным интерфейсом или классом, а с некоторым незаданным типом, что придает программе ещё более общий характер. Обобщение позволяет создавать компоненты, которые могут легко использоваться с разными типами.

Обобщение классов

Одной из причин появления обобщения стало создание классов-контейнеров. Контейнер предназначен для хранения объектов, используемых в программе. Контейнеры обладают большей гибкостью и отличаются по своим характеристикам от простых массивов. Контейнеры являются одной из самых часто используемых библиотек, так как необходимость хранения групп объектов возникает едва ли не в каждой программе.

Рассмотрим пример создания класса-контейнера для хранения объекта произвольного класса, реализованный с использованием обычных классов и с использованием обобщений.

```
class Container {
  private Object a;
  public Container(Object a) {
    this.a = a;
  }
  public void set(Object a) {
    this.a = a;
  }
  public Object get() {
    return a;
  }
}
```

В этом случае требуется приводить значение, возвращаемое функцией *get()*, к соответствующему типу вручную:

```
Container c = new Container(new Point(10, 10));
Point p = (Point) c.get();
System.out.println(p);
c.set("Point");
String s = (String) c.get();
System.out.println(s);
c.set(1.0);
Double d = (Double) c.get();
System.out.println(d);
```

```
java.awt.Point[x=10,y=10]
Point
1.0
```

При использовании обобщенной реализации класса-контейнера сперва требуется указать тип объекта, хранящегося в контейнере, в угловых скобках. В дальнейшем в контейнер можно будет помещать объекты только этого типа (или производного, так как принцип заменимости работает и для параметризованных типов), а при извлечении вы автоматически получаете объект нужного типа. Создание обобщенного класса осуществляется так же, как и обычного класса, но после имени класса указывается параметризированный тип.

```
class Container<T> {
  private T a;
  public Container(T a) {
    this.a = a;
  }
```

```
  public void set(T a) {
    this.a = a;
  }
  public T get() {
    return a;
  }
}
```

В этом случае описание ссылки на объект обобщенного класса:

```
Container<Integer> c = null;
```

А создание объекта обобщенного класса:

```
c = new Container<Integer>(10);
```

Получение содержимого класса-контейнера:

```
Integer i = c.get();
```

При объявлении обобщенного класса можно указывать больше одного параметризированного типа. Концепция нескольких объектов, «упакованных» в один объект, называется кортежем. Получатель объекта может читать элементы, но не может добавлять их (эта концепция еще называется объектом передачи данных). Обычно кортеж может иметь произвольную длину, а все объекты кортежа могут относиться к разным типам. Возможно задать тип каждого объекта и при этом гарантировать, что при чтении значения будет получен правильный тип.

```
class Test<A, B> {
  A a;
  B b;
  Test(A a, B b) {
    this.a = a;
    this.b = b;
  }
}
```

Основная идея обобщенных типов в Java: указывается, какой тип должен использоваться, а механизм обобщения берет на себя все подробности. При создании экземпляра обобщенного типа, преобразования приведения типа выполняются автоматически, а правильность типов проверяется на этапе компиляции.

Обобщение методов

Обобщение может применяться не только к целым классам, но и к отдельным методам. Сам класс при этом может быть обобщенным, а может и не быть – это не зависит от наличия обобщенных методов.

Обобщенный метод может изменяться независимо от класса. В общем случае обобщенные методы следует использовать «по

мере возможности». Иначе говоря, если возможно обобщить метод вместо целого класса, вероятно, стоит выбрать именно этот вариант. Кроме того, статические методы не имеют доступа к параметрам типа обобщенных классов; если такие методы должны использовать обобщение, это должно происходить на уровне метода, а не на уровне класса.

Чтобы определить обобщенный метод, следует указать список параметров перед возвращаемым значением.

```
class GenericMethod {
  <T> void printNameOfClass(T value) {
    System.out.println(value.getClass().getName());
  }
}
GenericMethod gm = new GenericMethod();
gm.printNameOfClass("");
gm.printNameOfClass(1);
gm.printNameOfClass((1.0));
gm.printNameOfClass((1.0f));
gm.printNameOfClass('c');
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
```

Класс *GenericMethod* не обобщен, хотя и класс, и его методы могут быть обобщенными одновременно. Но в данном случае только метод *printNameOfClass()* имеет параметр типа, обозначаемый списком параметров перед возвращаемым значением метода.

При использовании обобщенного класса параметры типов должны указываться во время созданиия экземпляра. Но при использовании обобщенного метода указывать параметры типа не обязательно, потому что это определяется на этапе компиляции. Таким образом, вызов *printNameOfClass()* выглядит как обычный вызов метода; можно сказать, что метод *printNameOfClass()* существует в бесконечном количестве перегруженных версий. Для вызовов *printNameOfClass()*, использующих примитивные типы, в действие вступает механизм автоматической упаковки.

Обобщение конструкторов

Даже если класс не является обобщенным, его конструктор может быть таковым:

```
class Container {
  double value;
```

```
  <T extends Number> Contauriner(T value) {
    this.value = value.doubleValue();
  }
  public String toString() { return "Value = " + value; }
}
```

В описанном конструкторе тип значения *value* ограничивается классами наследниками *Number*. Помимо ограничения типа параметра классами наследниками, также возможно его ограничение суперклассами от указанного *(<? super …>*).

```
Container[] c = new Container[4];
c[0] = new Container(100);
c[1] = new Container(010);
c[2] = new Container(1.0);
c[3] = new Container(0x1);
for (Container container : c) {
    out.println(container);
}
```

```
Value = 100.0
Value = 8.0
Value = 1.0
Value = 1.0
```

Обобщение интерфейсов

Обобщение работает и с интерфейсами. Рассмотрим пример обобщенного интерфейса, содержащий функцию нахождения максимума:

```
interface Max <T extends Comparable<T>> {
  T max();
}
```

Как и при описании интерфейса, ограничим тип параметра *T*, указав то, что он должен реализовывать интерфейс *Comporable<T>*.

```
interface Max <T extends Comparable<T>> {
  T max();
}

class Container <T extends Comparable<T>> implements Max {
  T[] array;
  Container(T[] array) {
    this.array = array;
  }
  public T max() {
    T value = array[0];
    for (int i = 1; i < array.length; i++) if
(value.compareTo(array[i]) < 0) value = array[i];
```

```
    return value;
  }
}

Float[] farray = {0.1f, 0.2f, 0.0f, 0.5f, 0.8f, 0.6f, 0.4f,
0.3f, 0.7f};
Container<Float> cf = new Container<Float>(farray);
System.out.println(cf.max());
Character[] carray = {'T', 'Z', 'U', 'a', '5', 'M'};
Container<Character> cc = new Container<Character>(carray);
System.out.println(cc.max());
```
```
0.8
a
```

Иерархия классов

В иерархии классов как суперкласс, так и подклассы могут быть обобщенными. Пример обобщенного суперкласса:

```
class Base<T> {
  T value;
  Base(T value) { this.value = value; }
  T getValue() { return value; }
}
class Child<T> extends Base<T> {
  Child(T value) { super(value); }
}
```

Пример обобщенного подкласса:

```
class Base {
  int value;
  Base(int value) { this.value = value; }
  int getValue() { return value; }
}
class Child<T> extends Base {
  T value;
  Child(int value0, T value1) { super(value0); this.value =
value1;}
}
```

Метасимвол <?>

Метасимвол <?> используется при указании области используемого типа.

Например, полиморфное присвоение применяется только к базовому типу, а не к параметру обобщенного типа, и когда следующее объявление является верным:

```
ArrayList<Animal> list = new ArrayList<Animal>();
```

Этот код выдаст ошибку на этапе компиляции:

```
ArrayList<Animal> list = new ArrayList<Dog>();
```

В тоже время, описание ссылки, создание объекта и установку ссылки на созданный объект можно представить как:

```
ArrayList<? extends Animal> list = new ArrayList<Dog>();
```

Метасимвол <?> применим как к классам, так и к интерфейсам. Он позволяет получить доступ к подтипу или супертипу объявленного в аргументе типа. Рассмотрим более подробный пример.

Пусть реализована следующая иерархия классов:

```
class Animal { }
class Dog extends Animal { }
class Pitbull extends Dog { }

public class Program
  public static void wildcardMethod(List<? extends Dog> list)
{
    System.out.println(list);
  }
  public static void main(String...args) {
    ArrayList<Animal> list_a = new ArrayList<Animal>();
    ArrayList<Dog> list_d = new ArrayList<Dog>();
    ArrayList<Pitbull> list_p = new ArrayList<Pitbull>();
    ArrayList<? extends Dog> list_wp = new ArrayList<Pitbull>
();
    ArrayList<? super Dog> list_wa= new ArrayList<Animal>();
    // TODO: PASTE LINE HERE!
  }
}
```

Определите правильность компиляции метода *wildcarMethod*, при вставке следующих строчек:

```
wildcardMethod(list_a);
wildcardMethod(list_d);
wildcardMethod(list_p);
wildcardMethod(list_wp);
wildcardMethod(list_wa);
```

Как изменится результат компиляции, если в методе *wildcardMethod*, **extends** заменить на **super**?

Ключевые понятия этой темы

– действия с обобщенным классом или методом отслеживаются на этапе компиляции, предотвращается неверное использование обобщения;

– во время использования обобщенной коллекции приведение типа не обязательно при доступе к элементам коллекции; иначе приведение типа обязательно;

– позволяют использовать одинаковые подходы при обработке разных данных;

– информация об обобщенных типах не существует во время исполнения, она предназначена лишь для безопасности компиляции; смешивание обобщений с устаревшим кодом может вызвать ошибку во время исполнения;

– полиморфное присвоение применяется только к базовому типу, а не к параметру обобщенного типа;

– *ArrayList<Animal>* может принимать ссылки типа *Dog*, *Cat* или любой другой подтип *Animal* (будь то подкласс или реализация интерфейса, если *Animal* является таковым);

– существует возможность передать обобщенной коллекции метод, который принимает не обобщенные коллекции; компилятор не может предотвратить метод от вставки неправильного типа в коллекцию с ранее безопасным типом;

– метасимвол *<?>* позволяет получить доступ к подтипу или супертипу объявленного в аргументе типа;

– метасимвол *<?>* хотя и использует ключевое слово **extends**, но применим как к классам, так и к интерфейсам;

– *List<Object>* ссылается только на *List<Object>*, в то время как *List<?* **extends** *Object>* может предоставить доступ к любому типу списка;

– использование метасимвола *<?>* позволяет получить доступ к коллекции без возможности её модификации;

– можно использовать более одного параметризованного типа в обобщенном классе;

– можно объявлять обобщенные методы, используя тип, не объявленный в классе;

– примитивные типы не могут быть параметрами обобщений (вместо них следует использовать классы обертки);

– внутри обобщенного класса нельзя создавать объекты типа параметра обобщения;

– в обобщении нельзя создавать массив объектов типа параметра обобщения;

– в обобщениях не может быть статических атрибутов обобщенного типа;

– статические методы не могут использовать значение обобщенного типа.

Порядок выполнения работы

Разработать обобщённую структуру данных согласно варианту задания и дополнительно реализовать указанную в варианте функцию. Созданная структура данных должна быть реализована на основе массива или ссылок (при реализации нельзя использовать никакие другие коллекции).

Для демонстрации возможностей класса создать две программы: *ProgramA* – использование разработанной структуры для хранения элементов примитивного типа данных, *ProgramB* – использование разработанной структуры для хранения элементов класса из лабораторной работы № 1.

Задания для лабораторной работы по вариантам

1. Структура данных «Стек». Дополнительно реализовать функцию проверки структур на равенство.
2. Структура данных «Словарь». Дополнительно реализовать функцию сортировки.
3. Структура данных «Очередь». Дополнительно реализовать функцию проверки структур на равенство.
4. Структура данных «Хэш-таблица». Дополнительно реализовать функцию проверки структур на равенство.
5. Структура данных «Стек». Дополнительно реализовать функцию сортировки.
6. Структура данных «Множество». Дополнительно реализовать функции дополнения множества и объединения двух множеств.
7. Структура данных «Словарь». Дополнительно реализовать функцию проверки структур на равенство.
8. Структура данных «Очередь с двумя концами». Дополнительно реализовать функцию сортировки.
9. Структура данных «Хэш-таблица». Дополнительно реализовать объединения двух хэш-таблиц.
10. Структура данных «Очередь с двумя концами». Дополнительно реализовать функцию поиска элемента.

11. Структура данных «Бинарное дерево». Дополнительно реализовать функцию проверки структур на равенство.

12. Структура данных «Очередь». Дополнительно реализовать функцию поиска элемента.

13. Структура данных «Множество». Дополнительно реализовать функции разности и симметрической разности множеств.

14. Структура данных «Двунаправленный связный список». Дополнительно реализовать функцию сортировки.

15. Структура данных «Бинарное дерево». Дополнительно реализовать функцию поиска элемента.

Контрольные вопросы

1. Что такое обобщение? Какие причины возникновения обобщений?
2. Как осуществляется обобщение классов?
3. Как осуществляется обобщение интерфейсов?
4. Как осуществляется обобщение методов?
5. Какие правила обобщения статических полей и методов класса?
6. Как создаются элементы, массивы элементов параметризованного типа в обобщенном классе?
7. Как осуществляется создание объектов обобщенного класса?
8. Как создать коллекцию определенного типа?
9. Как осуществляется приведение к типу в коллекции?
10. Какое назначение метасимвола <?>?

Содержание отчета

Скриншоты (3): обобщенной структуры данных, примера использования реализованной структуры данных с примитивными типами данных, примера использования реализованной структуры данных с классом из лабораторной работы № 1.

Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответ на вопросы по теме текущей лабораторной работы.
5. Ответ на вопросы по теме предыдущей лабораторной работы.

ЛАБОРАТОРНАЯ РАБОТА № 5. КОЛЛЕКЦИИ

Цель: Ознакомиться с контейнерами, научиться их использовать для хранения и работы с объектами.

Краткие теоретические сведения.
Коллекции

Библиотека утилит Java (java.util.*) содержит большой набор классов коллекций/контейнеров. Контейнеры обладают разнообразными возможностями для работы с объектами, с их помощью удаётся решить огромное количество задач.

В Java для хранения объектов предоставляется два интерфейса:

1. Коллекция. Класс *List* (список) хранит элементы в порядке вставки. Класс *Set* (множество) хранит неповторяющиеся элементы. Класс *Queue* (очередь) выдаёт элементы в порядке, определяемом спецификой очереди (обычно это порядок вставки элементов в очередь).
2. Карта. Класс *Map* (карта/словарь/ассоциативный массив) хранит наборы пар объектов «ключ = значение», с возможностью выбора значения по ключу.

Коллекции в Java различаются тем, сколько в одной ячейке коллекции «помещается» элементов. Коллекции (*Collection*) содержат только один элемент в каждой ячейке. В коллекции *Map* (карта) хранятся два объекта: ключ и связанное с ним значение.

Коллекции *ArrayList* и *LinkedList* принадлежат к семейству *List*. Элементы в этих коллекциях хранятся в порядке добавления. Однако они различаются не только скоростью выполнения тех или иных операций, но и количеством операций (*LinkedList* содержит больше операций, чем *ArrayList*).

В множествах *Set* (*HashSet*, *TreeSet* и *LinkedHashSet*) каждый элемент хранится только в одном экземпляре, а разные реализации *Set* используют разный порядок хранения элементов. В *HashSe*t порядок элементов определяется по значению функции хеширования хранимого в контейнере класса, хотя на первый взгляд порядок следования элементов выглядит хаотично. В *TreeSet* объекты хранятся отсортированными в зависимости от условия сравнения. В *LinkedHashSet* обеспечивается хранение элементов в порядке добавления.

Коллекция *Map* позволяет искать объекты по ключу, как несложная база данных. Объект, ассоциированный с ключом, называется значением.

В идеале весь программный код должен писаться в расчёте на взаимодействие с этими интерфейсами, а точный тип – указываться только в точке создания. Объект *List* может быть создан как:

```
List<Apple> apples = new ArrayList<Apple>();
```

Если позднее возникнет необходимость изменить реализацию, достаточно сделать это в точке создания:

```
List<Apple> apples = new LinkedList<Apple>();
```

Такой подход работает не всегда, потому что некоторые классы обладают дополнительной функциональностью. Например, *LinkedList* содержит дополнительные методы, не входящие в интерфейс *List*, а *TreeMap* – методы, не входящие в *Map*. Если такие методы используются в программе, восходящее преобразование к обобщённому интерфейсу невозможно.

Добавление групп элементов в коллекцию

Для включения групп элементов в коллекцию классы *Arrays* и *Collections* содержат вспомогательные методы. Метод *Arrays.asList()* получает либо массив, либо список элементов, разделённых запятыми, и преобразует его в объект *List*. Метод *Collections.addAll* получает объект *Collection* и либо массив, либо список, разделённый запятыми, и добавляет элементы в *Collection*.

```
List<Integer> list_0 = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> list_1 = new ArrayList<Integer>(Arrays.asList(6,
7, 8, 9, 10));
Integer[] array_0 = new Integer[]{11, 12, 13, 14, 15};

Collection<Integer> collection = new ArrayList<Integer>();
System.out.println(collection);

collection = new ArrayList<Integer>(list_0);
System.out.println(collection);

collection.addAll(list_1);
System.out.println(collection);

Collections.addAll(collection, array_0);
System.out.println(collection);

Collections.addAll(collection, 16, 17, 18, 19, 20);
System.out.println(collection);
```

```
[]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20]
```

Вывод содержимого контейнеров

Для получения печатного представления массива необходимо использовать метод *Arrays.toString*, но контейнеры отлично выводятся и без посторонней помощи. Следующий пример демонстрирует использование основных типов контейнеров:

```
Collection fill(Collection<String> collection) {
  collection.add("cat");
  collection.add("dog");
  collection.add("rat");
  collection.add("rat");
  return collection;
}
Map fill(Map<String, String> map) {
  map.put("cat", "Fuzzy");
  map.put("dog", "Rags");
  map.put("rat", "Bosco");
  map.put("rat", "Spot");
  return map;
}

System.out.println(fill(new ArrayList<String>()));
System.out.println(fill(new HashSet<String>()));
System.out.println(fill(new TreeMap<String, String>()));
[cat, dog, rat, rat]
[rat, cat, dog]
{cat=Fuzzy, dog=Rags, rat=Spot}
```

Итераторы

Итератор – это объект, обеспечивающий перемещение по последовательности объектов с выбором каждого объекта этой последовательности без углубления в её структуру. Вдобавок, итератор является «легковесным» (lightweight) объектом: его создание обходится без заметных затрат ресурсов. Из-за этого итераторы часто имеют ограничения; например, *Iterator* в Java поддерживает перемещение только в одном направлении, он предоставляет возможность:

– запросить у контейнера итератор вызовом метода *iterator()*. Полученный итератор готов вернуть начальный элемент последовательности при первом вызове своего метода *next()*;

– получить следующий элемент последовательности вызовом метода *next()*;

– проверить, остались ли еще объекты в последовательности (метод *hasNext()*);

– удалить из последовательности последний элемент, возвращённый итератором, методом *remove()*.

```
List<String> words = Arrays.asList("The weather is wonderful
today !".split(" "));
Iterator<String> iterator = words.iterator();

while(iterator.hasNext()) {
  System.out.print("\'" + iterator.next() + "\'" + " " );
}
System.out.println();

for (iterator = words.iterator(); iterator.hasNext(); itera-
tor.next()) {
  System.out.print(iterator.next() + " - ");
}
System.out.println();

for (String word: words) {
  System.out.print(word + "#");
}
System.out.println();
'The' 'weather' 'is' 'wonderful' 'today' '!'
The - is - today -
The#weather#is#wonderful#today#!#
```

При использовании *Iterator* можно не беспокоиться о количестве элементов в последовательности. Проверка осуществляется методами *hasNext()* и *next()*. При переборе элементов списка в одном направлении, без его модификации, можно использовать более компактную запись – «синтаксис foreach».

ListIterator

ListIterator – более мощная разновидность *Iterator*, поддерживаемая только классами *List*. Если *Iterator* поддерживает перемещение только вперёд, *ListIterator* является двусторонним. Кроме того, он может выдавать индексы следующего и предыдущего элементов по отношению к текущей позиции итератора в списке и заменять последний посещённый элемент методом *set()*. Вызов *listIterator()* возвращает *ListIterator*, указывающий в начало *List*, а для создания итератора *ListIterator*, изначально установленного на элемент с индексом *n*, используется вызов *listiterator(n)*. Все перечисленные возможности продемонстрированы в следующем примере:

```
List<Integer> numbers = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7);
System.out.println(numbers);
```

```
ListIterator<Integer> it = numbers.listIterator();
while(it.hasNext()) {
  int value = it.next();
  System.out.print(it.previousIndex() + " " + value + " " +
it.nextIndex() + ", ");
}

while(it.hasPrevious()) {
  System.out.print(it.previous() + " ");
}
System.out.println();

it = numbers.listIterator(5);
while(it.hasNext()) {
  it.next();
  it.set(0);
}
System.out.println(numbers);
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
0 0 1, 1 1 2, 2 2 3, 3 3 4, 4 4 5, 5 5 6, 6 6 7, 7 7 8
7 6 5 4 3 2 1 0
[0, 1, 2, 3, 4, 0, 0, 0]
```

Список

Существует две основные разновидности коллекции *List*: *ArrayList* и *LinkedList*.

ArrayList оптимизирована для произвольного доступа к элементам, однако операции вставки (удаления) элементов в середину списка работают относительно медленно.

LinkedList оптимизирована для последовательного доступа к элементам, с быстрыми операциями вставки (удаления) в середину списка. Произвольный доступ к элементам *LinkedList* выполняется относительно медленно, но по широте возможностей он превосходит *ArrayList*.

LinkedList

LinkedList реализует базовый интерфейс *List*. Класс *LinkedList* также содержит методы, позволяющие использовать его в качестве стека, очереди (*Queue*) или двухсторонней очереди (дека).

Методы *getFirst()* и *element()* идентичны – они возвращают начало (первый элемент) списка без его удаления и выдают исключение *NoSuchElementException* для пустого списка. Метод *peek()* возвращает первый элемент списка или *null* для пустого списка. Метод *addFirst()* вставляет элемент в начало списка. Метод *offer()* де-

лает то же, что *add()* и *addLast()* – он добавляет элемент в конец списка. Метод *removeLast()* удаляет и возвращает последний элемент списка.

Следующий пример демонстрирует схожие и различающиеся аспекты этих методов:

```
LinkedList<String> words = new LinkedList<String>();
Collections.addAll(words, "one", "two", "three", "four",
"five", "six", "seven", "eight", "nine", "ten");
System.out.println(words);

System.out.println(words.getFirst() + " " + words.element() +
" " + words.peek());
System.out.println(words.remove() + " " + words.removeFirst()
+ " " + words.poll());
System.out.println(words.removeLast());

words.addFirst("0");
words.add("10");
words.addLast("11");
words.offer("12");
System.out.println(words);
```

```
[one, two, three, four, five, six, seven, eight, nine, ten]
one one one
one two three
ten
[0, four, five, six, seven, eight, nine, 10, 11, 12]
```

Множество

В множествах (*Set*) каждое значение может храниться только в одном экземпляре. Попытки добавить новый экземпляр эквивалентного объекта блокируются. Множества часто используются для проверки принадлежности объекта заданному множеству. Следовательно, важнейшей операцией *Set* является операция поиска, поэтому на практике обычно выбирается реализация *HashSet*, оптимизированная для быстрого поиска.

Set имеет такой же интерфейс, что и *Collection*. В сущности, *Set* и является *Collection*, но обладает несколько иным поведением (кстати, идеальный пример использования наследования и полиморфизма: выражение разных концепций поведения). В следующем примере в множество *HashSet* включаются пятьдесят случайных чисел от 0 до 25, однако в результате каждое число присутствует в коллекции только в одном экземпляре.

```
Random rand = new Random(System.nanoTime());
Set<Integer> set = new HashSet<Integer>();
```

```
for (int i = 0; i < 50; i++) {
  set.add(rand.nextInt(25));
}
System.out.print(set);
```
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 22, 23, 24]
```

Предсказуемый порядок следования чисел в выводе объясняется тем, что *HashSet* использует хеширование для ускорения выборки. Порядок, поддерживаемый *HashSet*, отличается от порядка *TreeSet* или *LinkedHashSet*, поскольку каждая реализация упорядочивает элементы по-своему. Если требуется, чтобы результат был отсортирован, следует воспользоваться *TreeSet* вместо *HashSet*. Одной из наиболее распространённых операций со множествами является проверка принадлежности методом *contains()*, но существуют и другие операции:

```
Set<String> setA = new HashSet<String>();
Collections.addAll(setA, "A B C D E F G H I J K L".split("
"));
// setA.add("M");
System.out.println("Does set A contains \'H\'? - " +
setA.contains("H"));
System.out.println("Does set A contains \'N\'? - " +
setA.contains("N"));

Set<String> setB = new HashSet<String>();
Collections.addAll(setB, "H I J K L".split(" "));
System.out.println(setA.containsAll(setB));
setA.removeAll(setB);
System.out.println(setA);
```
```
Does set A contains 'H'? - true
Does set A contains 'N'? - false
true
[A, B, C, D, E, F, G]
```

Карта

Возможность отображения одних объектов на другие (ассоциация) чрезвычайно полезна при решении широкого класса задач программирования. В качестве примера рассмотрим программу, анализирующую качество распределения класса Java *Random*. В идеале класс *Random* должен выдавать абсолютно равномерное распределение чисел, но чтобы убедиться в этом, необходимо сгенерировать большое количество случайных чисел и подсчитать

их количество в разных интервалах. Множества упрощают эту задачу. Ключом в данном случае является число, сгенерированное при помощи *Random*, а значением – количество его вхождений:

```
for (int i = 0; i < 10000; i++) {
  int r = rand.nextInt(25);
  Integer freq = m.get(r);
  if (freq == null) {
    m.put(r, 1);
  } else {
    m.put(r, freq + 1);
  }
}
System.out.println("Keys are: " + m.keySet());
System.out.println("Values are: " + m.values());
System.out.println("Map is:");
for (Integer key: m.keySet()) {
  System.out.print("(" + key + ", " + m.get(key) + ") ");
}
```

```
Keys are: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
Values are: [383, 385, 388, 434, 430, 395, 424, 419, 396, 361,
414, 405, 444, 366, 418, 397, 425, 386, 414, 389, 396, 395,
391, 384, 361]
Map is:
(0, 383) (1, 385) (2, 388) (3, 434) (4, 430) (5, 395) (6, 424)
(7, 419) (8, 396) (9, 361) (10, 414) (11, 405) (12, 444) (13,
366) (14, 418) (15, 397) (16, 425) (17, 386) (18, 414) (19,
389) (20, 396) (21, 395) (22, 391) (23, 384) (24, 361)
```

В *main()* механизм автоматической упаковки преобразует случайно сгенерированное целое число в ссылку на *Integer*, которая может использоваться с *HashMap* (контейнеры не могут использоваться для хранения примитивов). Метод *get()* возвращает **null**, если элемент отсутствует в контейнере, то есть если число было сгенерировано впервые. В противном случае метод *get()* возвращает значение *Integer*, связанное с ключом, и последнее увеличивается на 1 (автоматическая упаковка снова упрощает вычисления, но в действительности при этом выполняются преобразования к *Integer* и обратно).

Map может вернуть *Set* своих ключей, *Collection* значений или множество *Set* всех пар «ключ=значение». Метод *keySet()* создает множество всех ключей, которое затем используется в синтаксисе **foreach** для перебора *Map*.

Стек

Стек часто называют контейнером, работающим по принципу «последним вошел, первым вышел» (LIFO). То есть элемент, последним занесённый в стек, будет получен первым при извлечении из стека. В классе *LinkedList* имеются методы, напрямую реализующие функциональность стека.

Очередь

Очередь обычно представляет собой контейнер, работающий по принципу «первым вошел, первым вышел» (FIFO). Иначе говоря, элементы заносятся в очередь с одного «конца» и извлекаются с другого в порядке их поступления. Очереди часто применяются для реализации надежной передачи объектов между разными областями программы.

Класс *LinkedList* содержит методы, поддерживающие поведение очереди, и реализует интерфейс *Queue*, поэтому *LinkedList* может использоваться в качестве реализации *Queue*.

Ключевые методы в Map, Set, List

Таблица 3. – Ключевые методы коллекций Список, Множество и Карта

Method	List	Set	Map
boolean add(elemant) boolean add(index, elemnt)	+ +	+	
boolean contains(object) **boolean** containsKey(key) **boolean** containsValue(value)	+	+	 + +
object get(index) object get(key)	+		 +
int indexOf(object)	+		
Iterator interator()	+	+	
Set keySet()			+
put(key, value)			+
remove(index) remove(object) remove(key)	+ +	 +	 +
int size()	+	+	
Object[] toArray()+	+	+	

Интерфейсы Comparable и Comparator. Сортировка. Интерфейс Comparable

При добавлении новых элементов объект *TreeSet* автоматически проводит сортировку, помещая новый объект на правильное для

него место. Однако со строками все понятно. А что, если бы мы использовали не строки, а свои классы, например, следующий класс *Person*:

```
class Person {
  private String name;
  private int age;
  public Person() {
    this.name = "Undefined";
    this.age = 18;
  }
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
  public String toString() {
    return String.format("%s is %d years old.", name, age);
  }
}
```

Объект *TreeSet* мы не сможем типизировать данным классом, поскольку в случае добавления объектов *TreeSet* не будет знать, как их сравнивать, и следующий код не будет работать:

```
TreeSet<Person> people = new TreeSet<Person>();
people.add(new Person("Tom", 20));
```

При выполнении этого кода мы столкнемся с ошибкой, которая скажет, что объект *Person* не может быть преобразован к типу *Comparable*.

Для того чтобы объекты *Person* можно было сравнить и сортировать, они должны применять интерфейс *Comparable<E>*. При применении интерфейса он типизируется текущим классом. Применим его к классу *Person*:

```
class Person implements Comparable<Person> {
  …
  public int compareTo(Person person) {
```

```
    return name.compareTo(person.name);
  }
}
```

Интерфейс *Comparable* содержит один единственный метод *int compareTo(E item)*, который сравнивает текущий объект с объектом, переданным в качестве параметра. Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны.

В данном случае мы не возвращаем явным образом никакое число, а полагаемся на встроенный механизм сравнения, который есть у класса *String*. Но мы также можем определить и свою логику. Например, сравнивать по длине имени:

```
public int compareTo(Person person) {
  return name.length() - person.name.length();
}
```

Теперь мы можем типизировать *TreeSet* типом *Person* и добавлять в дерево соответствующие объекты.

Интерфейс Comparator

Однако перед нами может возникнуть проблема: что, если разработчик не реализовал в своем классе, который мы хотим использовать, интерфейс *Comparable*, либо реализовал, но нас не устраивает его функциональность, и мы хотим ее переопределить? На этот случай есть еще более гибкий способ, предполагающий применение интерфейса *Comparator<E>*.

Интерфейс *Comparator* содержит ряд методов, ключевым из которых является метод *compare()*:

```
public interface Comparator<E> {
  int compare(E a, E b);
}
```

Метод *compare* также возвращает числовое значение: если оно отрицательное, то объект *a* предшествует объекту *b*, иначе – наоборот, а если метод возвращает ноль, то объекты равны. Для применения интерфейса нам вначале надо создать класс компаратора, который реализует этот интерфейс:

```
class PersonComparator implements Comparator<Person> {
  public int compare(Person a, Person b) {
    return a.getName().compareTo(b.getName());
  }
}
```

Здесь опять же проводим сравнение по строкам. Теперь используем класс компаратора для создания объекта *TreeSet*:

```
PersonComparator pcomp = new PersonComparator();
TreeSet<Person> people = new TreeSet<Person>();
people.add(new Person("Tom", 20));
people.add(new Person("Nick", 19));
people.add(new Person("Alice", 21));
people.add(new Person("Bill", 18));
for (Person p: people) {
  System.out.println(p);
}
```

```
Alice is 21 years old.
Bill is 18 years old.
Nick is 19 years old.
Tom is 20 years old.
```

Для создания *TreeSet* здесь используется одна из версий конструктора, которая в качестве параметра принимает компаратор. Теперь вне зависимости от того, реализован ли в классе *Person* интерфейс *Comparable*, логика сравнения и сортировки будет использоваться та, которая определена в классе компаратора.

Сортировка по нескольким критериям

Начиная с JDK 8 в механизм работы компараторов были внесены некоторые дополнения. В частности, теперь мы можем применять сразу несколько компараторов по принципу приоритета.

Допустим, нам надо отсортировать пользователей Person по имени и по возрасту. Для этого определим два компаратора *PersonNameComparator* и *PersonAgeComparator*. Интерфейс компаратора определяет специальный метод по умолчанию *thenComparing*, который позволяет использовать цепочки компараторов для сортировки набора:

```
class PersonNameComparator implements Comparator<Person> {
  public int compare(Person a, Person b) {
    return a.getName().compareTo(b.getName());
  }
}
class PersonAgeComparator implements Comparator<Person> {
  public int compare(Person a, Person b) {
    return a.getAge() - b.getAge();
  }
}
Comparator<Person> pcomp = new
PersonNameComparator().thenComparing(new
PersonAgeComparator());
TreeSet<Person> people = new TreeSet(pcomp);
```

```
people.add(new Person("Tom", 20));
people.add(new Person("Nick", 19));
people.add(new Person("Alice", 21));
people.add(new Person("Tom", 18));
for (Person p: people) {
  System.out.println(p);
}
```

```
Alice is 21 years old.
Nick is 19 years old.
Tom is 18 years old.
Tom is 20 years old.
```

В данном случае сначала применяется сортировка по имени, а потом по возрасту.

Порядок выполнения работы

В задании (A) требуется разработать контейнер, выбрав структуру данных самостоятельно на основе уже существующей структуры данных. То есть контейнер наследует выбранную структуру данных, которая согласуется с темой задания, с типом согласно варианту задания. Решить указанное в условии варианта задание требуется посредством переопределения или добавления методов в контейнер. При выполнении задания следует использовать коллекции и итераторы.

В задании (B) требуется разработать консольное меню для работы со списком объектов из лабораторной работы № 3.

Для демонстрации задания (A) и задания (B) нужно разработать программы *ProgramA* и *ProgramB* соответственно.

Задание (A) для лабораторной работы по вариантам

1. Пользователь вводит информацию о прямоугольных треугольниках (длины катетов). Фигура добавляется в коллекцию в том случае, если её площадь отлична от площадей имеющихся в коллекции фигур. Вывести фигуры в порядке убывания площади.

2. Пользователь осуществляет ввод целых положительных чисел. Число не может быть добавлено в коллекцию, если в ней уже присутствует число с таким же количеством бит, установленных в единицу.

3. Осуществляется ввод чисел. В зависимости от знака числа оно заносится в одну из двух коллекций. Вывести элементы коллекции с наибольшим количеством элементов в порядке добавления значения.

4. Вводятся вещественные числа. Отсортировать их по убыванию дробной части.

5. Пользователь осуществляет ввод строк, которые добавляются в коллекцию. Строка не может быть добавлена в коллекцию, если в ней уже присутствует строка такого же размера.

6. Вводится последовательность целых положительных чисел. Отсортировать полученные значения по убыванию количества нулевых бит.

7. Пользователь вводит строки в формате «ключ=значение». Ключ – целое число, значение – целое число. Если при попытке добавления пары «ключ=значение» в коллекции уже существует запись с таким ключом, то необходимо увеличить величину значения на значение из добавляемой пары. Вывести коллекцию, упорядоченную по возрастанию значений.

8. Пользователь вводит строки в формате «ключ=значение». Ключ и значение являются строками. Если при попытке добавления пары «ключ=значение» в коллекции уже существует запись с таким ключом, то значение определяется как максимум длины строки текущего значения и значения из добавляемой пары. Вывести коллекцию, упорядоченную по возрастанию длины ключа.

9. Пользователь вводит строки в формате «ключ=значение». Ключ – целое число, значение – строка. Вывести коллекцию, упорядоченную по убыванию длины значения.

10. Осуществляется ввод слов. Если пользователь вводит строку, то каждое слово строки добавляется в коллекцию. Определить наличие введённого слова в коллекции.

11. Пользователь вводит строки в формате «ключ=значение». Ключ – строка, значение – целое число. Если при попытке добавления пары «ключ=значение в коллекции уже существует запись с таким ключом, то значение определяется как максимум текущего значения и значения из добавляемой пары. Вывести коллекцию, упорядоченную по убыванию значений.

12. Пользователь вводит вещественные числа. Элемент добавляется в коллекцию лишь в том случае, если в коллекции не содержится элемента с такой же целой частью.

13. Пользователь вводит строки в формате «ключ=значение». Ключ – целое число, значение – вещественное число. Вывести коллекцию, упорядочив пары по убыванию суммы цифр дробных частей значений.

14. Осуществляется ввод строк. Продублировать строки, длина которых меньше средней длины всех строк.

15. Осуществляется ввод слов или строк, символы которых добавляются в коллекцию. Вывести список символов без повторений.

Задание (В) для лабораторной работы по вариантам

Копии исходных кодов лабораторной работы № 3 должны быть модифицированы следующим образом:

– *[BaseClass]* должен принадлежать интерфейсу *Comparable<[BaseClass]>*;

– классы компараторы *[BaseClass]Comparator** – интерфейсу *{Comparator<[BaseClass]>*;

– класс список должен быть реализован на основе *ArrayList<[BaseClass]>*;

– в методе *main* главного класса программы создается список для хранения объектов базового класса и пользователю предоставляется возможность работать с этим списком посредством консольного меню.

Для работы со списком в меню предоставляется выбор одной из следующих операций:

– добавление подготовленных записей в список (15 записей);
– добавление элемента в список;
– изменение элемента по индексу;
– удаление элемента по индексу;
– поиск элемента в списке;
– сортировка списка;
– сортировка списка с помощью выбора компаратора;
– вывод списка в консоль;
– выход из меню, завершение работы программы.

Контрольные вопросы

1. Что такое коллекции?
2. Как иерархия интерфейсов коллекций представлена в Java?
3. Какие классы коллекций присутствуют в Java?
4. Что из себя представляет структура данных *List*? Какие основные операции присутствуют в *List*?
5. В чем заключается отличие класса *ArrayList* от класса *LinkedList*?

6. Что такое структура данных *Set*? Какие основные операции присутствуют в *Set*?

7. В чем заключаются отличия между классами *TreeSet*, *HashSet* и *LinkedHashSet*?

8. Что такое структура данных *Map*? Чем она отличается от структур данных *List*, *Set*?

9. Чем отличаются между собой классы *TreeMap*, *HashMap*, *LinkedHashMap*?

10. Для чего используется библиотека *Collections*? Какие операции над коллекциями она позволяет выполнять?

Содержание отчета

Скриншоты (3): исходные коды разработанной коллекции и главного класса программы для задания (А) и исходные коды главного класса программы для задания (В).

Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответ на вопросы по теме текущей лабораторной работы.
5. Ответ на вопросы по теме предыдущей лабораторной работы.

ЛАБОРАТОРНАЯ РАБОТА № 6.
ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ

Цель: Научиться описывать, реализовывать, перехватывать и обрабатывать исключительные ситуации. Создавать исключения, классы исключительных ситуаций.

Краткие теоретические сведения.
Исключения

Исключения являются неотъемлемой частью программирования на Java. Одно из преимуществ обработки исключений состоит в том, что она позволяет сосредоточиться на решаемой проблеме, а затем обработать все ошибки в описанном коде в другом месте. Исключения описываются как средство передачи информации и восстановления после ошибок на стадии выполнения. Ценность исключений – в передаче информации. Java фактически настаивает, чтобы программа сообщала обо всех ошибках в виде исключений, и именно это обстоятельство обеспечивает Java большое преимущество перед языками вроде C++, где программа может сообщать об ошибках разными способами (или не сообщать вовсе).

Механизм исключений Java помогает справиться с ошибками на стадии исполнения программы, которые невозможно предусмотреть заранее, но при этом данный механизм помогает выявлять многие ошибки программирования, выходящие за пределы возможностей компилятора.

Иерархия классов исключений в языке программирования Java представляет собой:

Throwable;
- Error;
- Exception;
- IOException;
- RuntimeException.

Типы исключений

Класс Java *Throwable* описывает все объекты, которые могут возбуждаться как исключения. Существует две основные разновидности объектов *Throwable* (то есть ветви наследования). Класс *Error* представляет системные ошибки и ошибки времени компиляции, которые обычно не перехватываются (кроме нескольких особых слу-

чаев). Класс *Exception* может быть возбуждён из любого метода стандартной библиотеки классов Java или пользовательского метода в случае неполадок при исполнении программы. Таким образом, для программистов интерес представляет прежде всего класс *Exception*.

В Java присутствует целая группа исключений, члены которой всегда возбуждаются в Java автоматически, и совсем не обязательно включать их в спецификацию исключений. Все они унаследованы от одного базового класса *RuntimeException*. Исключения, возбуждённые методом *RuntimeException* (или любым унаследованным от него исключением), относятся к непроверяемым (unchecked). Такие исключения означают ошибки в программе, и фактически их никогда не придется перехватывать – это делается автоматически. Проверка *RuntimeException* привела бы к излишнему загромождению программы.

Непроверяемые исключения обнаруживаются в ходе работы программы. Непроверяемые исключения: *ArithmeticException*, *ArrayStoreException*, *BufferOverflowException*, *ClassCastException*, *EnumConstantNotPresentException*, *FileSystemAlreadyExistsException*, *IllegalArgumentException*, *IllegalStateException*, *IndexOutOfBoundsException*, *NegativeArraySizeException*, *NullPointerException*, *SecurityException*, *SystemException*, *TypeNotPresentException*, *UnsupportedOperationException*.

Проверка наличия обработки проверяемых исключений осуществляется на этапе компиляции. Проверяемые исключения: *ClassNotFoundException*, *CloneNotSupportedException*, *IllegalAccessException*, *InstantiationException*, *InterruptedException*, *NoSuchFieldException*, *NoSuchMethodException*.

Класс Throwable

Любой класс исключений в языке Java является потомком *Throwable*. Далее приведен список методов класса *Throwable*:

```
Throwable()
Throwable(String message)
Throwable(String message, Throwable cause)
Throwable(String message, Throwable cause,  oolean
enableSuppression,  oolean writableStackTrace)
Throwable(Throwable cause)
void addSuppressed(Throwable exception)
Throwable fillInStackTrace()
Throwable getCause()
String getLocalizedMessage()
String getMessage()
```

```
StackTraceElement[] getStackTrace()
Throwable[] getSuppressed()
Throwable initCause(Throwable cause)
void printStacktTrace()
void printStackTrace(PrintStream s)
void printStacktrace(PrintWriter s)
void setStackTrace(StackTraceElement[] strackTrace)
String toString()
```

Область применения исключений

Исключения могут быть использованы для того, чтобы:

– обработать ошибку на текущем уровне;

– исправить проблему и снова вызвать метод, возбудивший исключение;

– предпринять все необходимые действия и продолжить выполнение без повторного вызова метода;

– попытаться найти альтернативный результат вместо того, который должен был бы произвести вызванный метод;

– сделать все возможное в текущем контексте и заново возбудить это же исключение, перенаправив его на более высокий уровень;

– сделать все, что можно в текущем контексте, и возбудить новое исключение, перенаправив его на более высокий уровень;

– завершить работу программы;

– упростить программу (если используемая вами схема обработки исключений делает все только сложнее, значит, она никуда не годится);

– добавить библиотеке и программе безопасность (сначала это поможет в отладке программы, а в дальнейшем окупится её надёжностью).

Операторы исключений

Для осуществления создания, передачи и обработки исключений используются следующие операторы (ключевые слова языка Java):

– **try** – определяет блок кода, в котором ожидается появление исключений;

– **catch** – определяет блок кода, в котором выполняется обработка некоторого исключения;

– **finally** – определяет блок кода, выполнение которого осуществляется независимо от наличия исключения в блоке **try**;

– **throw** – используется для того, чтобы создать исключение;

– **throws** – определяет, какие исключения могут быть сгенерированы в результате работы метода. Обработка данных исключений возлагается на методы, вызвавшие их.

Рассмотрим пример использования всех ключевых слов, связанных с исключениями:

```
void function() throws ExceptionA, ExceptionB, … {
  try {
    …
    try {
      …
    } finally {
      …
    }
    …
  } catch(SomeException ex) {
    throw new ExceptionA();
  } …
  …
  } catch(LastException ex) {
    throw new Exception…();
  }finally {
    …
  }
}
```

Из примера видно, что в Java допускается обработка нескольких исключений в одном блоке **try**. В этом случае порядок исключений должен быть от частных к более общим, и исключение *Exception* должно идти последним. Также допускается вложение блоков **try** друг в друга, и в этом случае исключения, которые не обрабатываются во вложенном блоке **try**, передаются во внешний блок **try**. За блоком, отслеживающим появление исключений, достаточно идти блоку обработки исключения (**try** { … } **catch**(…) { … }) или финальному блоку (**try** { … } **finally** { … }).

При отсутствии обработки непроверяемого исключения в случае его возникновения выполнение программы прерывается и выводится сообщение о месте возникновения исключения.

```
Int division(int a, int b) {
  int result;
  result = a / b;
  return result;
}

int result = division(100, 0);
java.lang.ArithmeticException: / by zero
```

В этом случае будет выведена информация об ошибке и приведён стек методов, через которые ошибка будет передаваться до первоначальной точки входа.

Для того чтобы избежать прерывания программы, необходимо обработать исключительную ситуацию. Это можно сделать следующим образом:

```
int division(int a, int b) {
  int result;
  try {
    result = a / b;
  } catch(ArithmeticException ex) {
    System.out.println("Division by zero!!!");
    return Integer.MAX_VALUE;
  }
  return result;
}

int result = division(100, 0);
System.out.println(result);
```

Генерация исключений

Для генерации исключений используется оператор **throw** как в показанном ранее примере:

```
throw new Exception("Some exception happend here!");
java.lang.Exception: Some exception happend here!
```

Создание класса исключения

В языке Java, если в методе возможна генерация исключения, то это должно быть указано в описании метода с помощью ключевого слова **throws**. В спецификации метода после ключевого слова **throws** должен быть приведён список исключений, которые этот метод генерирует, но не обрабатывает.

```
Class MyException extends Exception {
  MyException() {}
  MyException(String msg) {super(msg);}
}

void generator() throws MyException {
    throw new MyException();
}

try{
  generator();
```

```
} catch (MyException ex) {
}
System.out.println("Exception is handled!");
Exception is handled!
```

Если при вызове метода *generator()* в *main* не обрабатывается исключение *MyException*, то компилятор выдаст сообщение об ошибке. А в случае когда класс *MyException* наследует одного из потомков класса *Exception*, подобную исключительную ситуацию не обязательно обрабатывать:

```
class MyException extends ArithmeticException {
  …
}
```

Генерация исключений при наследовании

При переопределении в производном классе метода, который в базовом классе генерировал исключение, допускается, что этот метод больше не будет генерировать исключение.

```
Class A {
  void generator() throws MyException {
    throw new MyException();
  }
}
class B extends A {
  void generator() {
  }
}
```

Однако компилятор не позволит вызвать этот метод через ссылку на базовый класс.

Также при переопределении в производном классе метода, который в базовом классе не генерировал исключение, этот метод не может генерировать исключение. Следующий код некорректен:

```
class A {
  void generator() {
  }
}
class B extends A {
  void generator() throws MyException {
    throw new MyException();
  }
}
```

Таким образом, если в производном классе переопределяется метод базового класса, то они оба должны генерировать исключения или не генерировать таковых.

Блок finally

Содержимое этого блока выполняется независимо от того, была отловлена и обработана ошибка внутри блока **try** или нет.

```
Void generator() throws Exception {
    throw new Exception();
}

Random rand = new Random(System.currentTimeMillis());
while (true) {
  try {
    int value = rand.nextInt(5);
    System.out.println("value = " + value);
    if (value == 1) {
      generator();
    }
  } catch(Exception ex) {
    System.out.println("Exception");
    break;
  } finally {
    System.out.println("finally block");
  }
}
```

```
value = 2
finally block
value = 3
finally block
value = 1
Exception
finally block
```

Модели обработки исключений

В теории обработки исключений имеется две основные модели. Модель прерывания, которая используется в Java и C++, предполагает следующее: ошибка настолько серьёзна, что при возникновении исключения продолжить исполнение невозможно. Кто бы ни возбудил исключение, сам факт его выдачи означает, что исправить ситуацию «на месте» невозможно и возвращать управление обратно не нужно.

Альтернативная модель называется возобновлением. Она подразумевает, что обработчик ошибок сделает что-то для исправления ситуации, после чего предпринимается попытка повторить неудавшуюся операцию в надежде на успешный исход. В таком случае, чтобы применить модель возобновления в Java, вам придётся не возбуждать исключение, а вызвать метод, способный решить проблему. Также можно создать блок **try** внутри цикла **while**, кото-

рый станет снова и снова обращаться к этому блоку, пока не будет достигнут нужный результат.

Исторически сложилось так, что программисты, использующие операционные системы с поддержкой возобновления, со временем переходили к модели прерывания, забывая другую модель. Хотя идея возобновления выглядит привлекательно, она не настолько полезна на практике. Основная причина кроется в обратной связи: обработчик ошибки часто должен знать, где произошло исключение, и содержать специальный код для каждого отдельного места ошибки. А это усложняет написание и поддержку программ, особенно для больших систем, где исключения могут быть сгенерированы во многих различных местах.

Порядок выполнения работы

В рамках этой лабораторной работы требуется модифицировать копии исходных кодов лабораторной работы № 5, для того чтобы обеспечить возобновляемую модель обработки исключений. Пользователю предоставляется возможность работать со списком элементов посредством консольного меню. При неправильном или некорректном вводе данных с консоли программа не должна завершаться, а в случае ошибки работы программы пользователю выводится сообщение об ошибке, в котором говорится о причине ее возникновения. После этого пользователя возвращают на выбор действия из консольного меню.

Для этого нужно:

– создать класс исключений *[BaseClass]Exception* и добавить его в пакет *utils*;

– в каждом методе *next[ChildClass*]* класса *[BaseClass]Input* требуется осуществить перехват и обработку всех непроверяемых исключений, которые могут возникнуть при вводе, обращении к элементам или приведении к типу;

– здесь же для каждого атрибута дочернего класса нужно осуществить проверку на валидность введенного значения: формат, диапазон значений;

– здесь же в случае возникновения исключения или не прохождения проверки значения на правильность нужно сгенерировать сообщение об ошибке созданного класса исключений и показать, что этот метод может генерировать подобного рода исключения;

– в методе *main* главного класса программы требуется осуществить перехват и обработку всех возможных исключений.

Контрольные вопросы

1. Что такое исключительные ситуации? Каково их предназначение, область применения?
2. Расскажите про иерархию исключительных ситуаций. Какое место в ней занимает класс *Throwable*?
3. Какие ключевые слова используются для обработки исключительных ситуаций?
4. Каковы причины возникновения непроверяемых и проверяемых исключений?
5. Какие непроверяемые исключения вы знаете? Назовите пример, приводящий к непроверяемому исключению.
6. Какие проверяемые исключения вы знаете? Назовите пример, приводящий к проверяемому исключению.
7. Как осуществляется обработка нескольких исключений?
8. Как осуществляется генерация исключений?
9. Какие существуют правила использования исключений при наследовании?
10. Какие модели обработки исключений вы знаете?

Содержание отчета

Скриншоты (2): созданного класса исключения; класса, осуществляющего чтение/запись объектов.

Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответ на вопросы по теме текущей лабораторной работы.
5. Ответ на вопросы по теме предыдущей лабораторной работы.

ЛАБОРАТОРНАЯ РАБОТА № 7.
ФАЙЛОВЫЙ ВВОД И ВЫВОД

Цель: Научиться использовать классы ввода и вывода для записи и чтения объектов в бинарные/текстовые файлы.

Краткие теоретические сведения.
Потоки ввода-вывода

Отличительной чертой многих языков программирования является работа с файлами и потоками. В Java основной функционал работы с потоками сосредоточен в классах из пакета *java.io*.

Ключевым здесь является понятие потока, хотя в программировании оно довольно перегружено и может обозначать множество различных концепций. В данном случае применительно к работе с файлами и вводом-выводом мы будем говорить о потоке (stream) как об абстракции, которая используется для чтения или записи информации (файлов, сокетов, текста консоли и т.д.).

Поток связан с реальным физическим устройством с помощью системы ввода-вывода Java. У нас может быть определен поток, который связан с файлом и через который мы можем вести чтение или запись файла. Это также может быть поток, связанный с сетевым сокетом, с помощью которого можно получить или отправить данные в сети. Все эти задачи: чтение и запись различных файлов, обмен информацией по сети, ввод-ввывод в консоли – мы будем решать в Java с помощью потоков.

Объект, из которого можно считать данные, называется потоком ввода, а объект, в который можно записывать данные, – потоком вывода. Например, если надо считать содержание файла, то применяется поток ввода, а если надо записать в файл – то поток вывода.

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: *InputStream* (представляющий потоки ввода) и *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) и *Writer* (для записи потоков символов).

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

Классы InputStream и OutputStream

Класс *InputStream* является базовым для всех классов, управляющих байтовыми потоками ввода. Рассмотрим его 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 байтов, которое равно *number*.

Класс *OutputStream* является базовым 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, item **int** length): записывает в выходной поток некоторое число байтов, равное *length*, из массива *buffer*, начиная со смещения *offset*, то есть с элемента *buffer[offset]*.

Абстрактные классы Reader и Writer

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 символы из потока в объект *CharBuffer*. Возвращает количество успешно считанных символов. При достижении конца файла возвращает –1;

– **absract int** read(**char**[] buffer, **int** offset, **int** count): считывает в массив *buffer* из потока символы, количество которых равно *count,* начиная со смещения *offset*;

– **long** skip(**long** count): пропускает количество символов, равное *count*. Возвращает число успешно пропущенных символов.

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 append(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*;

– Функционал, описанный классами *Reader* и *Writer*, наследуется непосредственно классами символьных потоков, в частности классами *FileReader* и *FileWriter* соответственно, предназначенными для работы с текстовыми файлами.

Закрытие потоков

При завершении работы с потоком его надо закрыть с помощью метода *close()*, который определен в интерфейсе *Closeable*.

Метод *close* имеет следующее определение:

```
void close() throws IOException
```

Этот интерфейс уже реализуется в классах *InputStream* и *OutputStream*, а через них и во всех классах потоков.

При закрытии потока освобождаются все выделенные для него ресурсы, например, файл. В случае если поток окажется не закрыт, может происходить утечка памяти.

Есть два способа закрытия файла. Первый – традиционный, заключается в использовании блока **try**..**catch**..**finally**. Например, считаем данные из файла:

```
FileInputStream fin = null;
try {
  fin = new FileInputStream("file.txt");
  int i = -1;
  while ((i=fin.read()) != -1) {
    System.out.print((char)i + " ");
  }
} catch (IOException ex) {
  System.out.println(ex.getMessage());
} finally {
  try {
    if (fin != null) {
      fin.close();
    }
  } catch (IOException ex) {
    System.out.println(ex.getMessage());
  }
}
```

Поскольку при открытии или считывании файла может произойти ошибка ввода-вывода, то код считывания помещается в блок **try**. Чтобы быть уверенным, что поток в любом случае закроется, даже если при работе с ним возникнет ошибка, вызов метода **close()** помещается в блок **finally**. И так как метод **close()** также в случае ошибки может генерировать исключение *IOException*, то его вызов помещается во вложенный блок **try**..**catch**.

Начиная с Java 7 можно использовать еще один способ, который автоматически вызывает метод *close*. Этот способ заключается в использовании конструкции *try-with-resources* (*try-с-ресурсами*). Данная конструкция работает с объектами, которые реализуют интерфейс *AutoCloseable*. Так как все классы потоков реализуют интерфейс *Closeable*, который в свою очередь наследуется от *AutoCloseable*, то их также можно использовать в данной конструкции.

Итак, перепишем предыдущий пример с использованием конструкции *try-with-resources*:

```
try (FileInputStream fin = new FileInputStream("file.txt")) {
  int i = -1;
  while ((i=fin.read()) != -1) {
    System.out.print((char)i + " ");
  }
} catch (IOException ex) {
  System.out.println(ex.getMessage());
}
```

Синтаксис конструкции следующий:

```
try( [ ClassName variableName = ConstructorOfClass ] ).
```

Данная конструкция также не исключает использования блоков **catch**. После окончания работы в блоке **try** у ресурса (в данном случае у объекта *FileInputStream*) автоматически вызывается метод **close()**. Если нам надо использовать несколько потоков, которые после выполнения надо закрыть, то мы можем указать объекты потоков через точку с запятой.

Чтение и запись файлов с помощью FileInputStream и FileOutputStream

Запись файлов и класс FileOutputStream

Класс *FileOutputStream* предназначен для записи байтов в файл. Он является производным от класса *OutputStream*, поэтому наследует всю его функциональность. Через конструктор класса *FileOutputStream* задается файл, в который производится запись. Класс поддерживает несколько конструкторов:

```
FileOutputStream(String path)
FileOutputStream(File file)
FileOutputStream(String path,  oolean append)
FileOutputStream(File file,  oolean append)
```

Файл задается либо через строковый путь, либо через объект *File*. Второй параметр – *append*, задает способ записи: если он равен **true**, то данные дозаписываются в конец файла, а при **false** файл полностью перезаписывается. Например, запишем в файл строку “Hello World!”:

```
String text = "Hello World!";
try (FileOutputStream fos = new FileOutputStream("file.txt"))
{
  byte[] buffer = text.getBytes();
```

```
  fos.write(buffer, 0, buffer.length);
} catch (IOException ex) {
  System.out.println(ex.getMessage());
}
```

Для создания объекта *FileOutputStream* используется конструктор, принимающий в качестве параметра путь к файлу для записи. Если такого файла нет, то он автоматически создается при записи. Так как здесь записываем строку, то ее надо сначала перевести в массив байтов. С помощью метода *write* строка записывается в файл. Для автоматического закрытия файла и освобождения ресурса объект *FileOutputStream* создается с помощью конструкции **try**…**catch**.

Чтение файлов и класс FileInputStream

Для считывания данных из файла предназначен класс *FileInputStream*, который является наследником класса *InputStream* и поэтому реализует все его методы. Если файл не может быть открыт, например, по указанному пути такого файла не существует, то генерируется исключение *FileNotFoundException*.

Считаем данные из ранее записанного файла и выведем на консоль:

```
try (FileInputStream fin = new FileInputStream("file.txt")) {
  System.out.printf("File size: %d bytes\n", fin.available());
  int i = -1;
  while ((i = fin.read()) != -1) {
    System.out.print((char)i + " ");
  }
} catch (IOException ex) {
  System.out.println(ex.getMessage());
}
```

```
File size: 12 bytes
H e l l o   W o r l d !
```

В данном случае мы считываем каждый отдельный байт в переменную *i*. Когда в потоке больше нет данных для чтения, метод возвращает число –1. Затем каждый считанный байт конвертируется в объект типа **char** и выводится на консоль.

Совместим оба примера и выполним чтение из одного и запись в другой файл:

```
try (FileInputStream fin = new
FileInputStream("file.txt");FileOutputStream fos = new
FileOutputStream("another_file.txt")) {
  byte[] buffer = new byte[fin.available()];
```

```
  fin.read(buffer, 0, buffer.length);
  fos.write(buffer, 0, buffer.length);
} catch (IOException ex) {
  System.out.println(ex.getMessage());
}
```

Классы *FileInputStream* и *FileOutputStream* предназначены, прежде всего, для записи двоичных файлов, то есть для записи и чтения байтов. И хотя они также могут использоваться для работы с текстовыми файлами, но все же для этой задачи больше подходят другие классы.

Классы для работы с потоками данных и текстами

Для чтения из файлов и записи в файлы определен ряд классов, ознакомимся с некоторыми из них. Для получения более подробной информации требуется самостоятельно изучить документы по адресу docs.oracle.com/~java/io/$.

Классы ByteArrayInputStream и ByteArrayOutputStream

Для работы с массивами байтов – их чтения и записи, используются классы *ByteArrayInputStream* и *ByteArrayOutputStream*. Как и для объектов *ByteArrayInputStream*, для *ByteArrayOutputStream* не надо явным образом закрывать поток с помощью метода *close*.

Буферизованные потоки BufferedInputStream и BufferedOutputStream

Для оптимизации операций ввода-вывода используются буферизуемые потоки. Эти потоки добавляют к стандартным специальный буфер в памяти, с помощью которого повышается производительность при чтении и записи потоков. Класс *BufferedInputStream* накапливает вводимые данные в специальном буфере без постоянного обращения к устройству ввода. Класс *BufferedOutputStream* аналогично создает буфер для потоков вывода. Этот буфер накапливает выводимые байты без постоянного обращения к устройству, и когда буфер заполнен, производится запись данных.

Классы DataOutputStream и DataInputStream

Классы *DataOutputStream* и *DataInputStream* позволяют записывать и считывать данные примитивных типов. Класс *DataOutputStream* представляет поток вывода и предназначен для записи данных примитивных типов, таких, как **int**, **double** и т.д. Для записи каждого из примитивных типов предназначен свой метод. Класс *DataInputStream* действует противоположным образом – он считывает из потока данные примитивных типов.

Форматируемый вывод, классы PrintStream и PrintWriter

Класс *PrintStream* – это именно тот класс, который используется для вывода на консоль. Выводя на консоль некоторую информацию с помощью вызова *System.out.println()*, мы задействуем *PrintStream*, так как переменная *out* в классе *System* как раз и представляет объект класса *PrintStream*, а метод *println()* – это метод класса *PrintStream*.

Но *PrintStream* полезен не только для вывода на консоль. Мы можем использовать данный класс для записи информации в поток вывода.

Для вывода информации в выходной поток *PrintStream* использует следующие методы:

- *println()* – вывод строковой информации с переводом строки;
- *print()* – вывод строковой информации без перевода строки;
- *printf()* – форматированный вывод.

На *PrintStream* похож другой класс *PrintWriter*. Его можно использовать как для вывода информации на консоль, так и в файл или любой другой поток вывода. *PrintWriter* реализует интерфейсы *Appendable*, *Closable* и *Flushable*, и поэтому после использования представляемый им поток надо закрывать. Для записи данных в поток он также использует методы *printf()* и *println()*.

Чтение и запись текстовых файлов с помощью FileReader и FileWriter

Хотя с помощью ранее рассмотренных классов можно записывать текст в файлы, но они предназначены, прежде всего, для работы с бинарными потоками данных, и их возможностей для полноценной работы с текстовыми файлами недостаточно. Для этой цели служат совсем другие классы, которые являются наследниками абстрактных классов *Reader* и *Writer*. Класс *FileWriter* является производным от класса *Writer*. Он используется для записи текстовых файлов. Класс *FileReader* наследуется от абстрактного класса *Reader* и предоставляет функциональность для чтения текстовых файлов.

Буферизация символьных потоков с помощью BufferedReader и BufferedWriter

Класс *BufferedWriter* записывает текст в поток, предварительно буферизируя записываемые символы, тем самым снижая количество обращений к физическому носителю для записи данных. Класс

BufferedReader считывает текст из символьного потока ввода, буферизируя прочитанные символы. Использование буфера призвано увеличить производительность чтения данных из потока.

Сериализация

Сериализация представляет процесс записи состояния объекта в поток, соответственно процесс извлечения или восстановления состояния объекта из потока называется десериализацией. Сериализация очень удобна, когда идет работа со сложными объектами.

Сразу надо сказать, что сериализовать можно только те объекты, которые реализуют интерфейс *Serializable*. Этот интерфейс не определяет никаких методов, он просто служит указателем системе, что объект, реализующий его, может быть сериализован. В файл без проблем можно записать как сериализованные, так и не сериализованные объекты, однако проблемы возникнут при десериализации.

Класс ObjectOutputStream

Для сериализации объектов в поток используется класс *ObjectOutputStream*. Он записывает данные в поток.

Для записи данных ObjectOutputStream использует ряд методов, среди которых можно выделить следующие:

– **void** close() – закрывает поток;

– **void** flush() – очищает буфер и сбрасывает его содержимое в выходной поток;

– **void** write(**byte**[] buf) – записывает в поток массив байтов;

– **void** write(**int** val) – записывает в поток один младший байт из *val*;

– **void** writeBoolean(**boolean** val) – записывает в поток значение **boolean**;

– **void** writeByte(**int** val) – записывает в поток один младший байт из *val*;

– **void** writeChar(**int** val) – записывает в поток значение типа **char**, представленное целочисленным значением;

– **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** writeObject(Object obj) – записывает в поток отдельный объект.

Эти методы охватывают весь спектр данных, которые можно сериализовать.

```
Class Person implements Serializable {
  private String name;
  private int age;
  public Person() {
    this.name = “Undefined”;
    this.age = 18;
  }
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
  public String toString() {
    return String.format(“%s is %d years old.”, name, age);
  }
}
try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(“file.data”))) {
  Person p = new Person(“Sam”, 33);
  oos.writeObject(p);
} catch (Exception ex) {
  System.out.println(ex.getMessage());
}
```

Класс ObjectInputStream

Класс *ObjectInputStream* отвечает за обратный процесс – чтение ранее сериализованных данных из потока.

Функционал *ObjectInputStream* сосредоточен в методах, предназначенных для чтения различных типов данных. Рассмотрим основные методы этого класса:

– **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;

– **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;

– Object readObject() – считывает из потока объект.

Например, извлечем выше сохраненный объект *Person* из файла:

```
try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("file.data"))) {
  Person p = (Person) ois.readObject();
  System.out.println(p)
} catch (Exception ex) {
  System.out.println(ex.getMessage());
}
```

```
Sam is 33 years old.
```

Теперь совместим сохранение и восстановление из файла на примере списка объектов:

```
ArrayList<Person> people = new ArrayList<Person>();
people.add(new Person("Tom", 20));
people.add(new Person("Nick", 19));
people.add(new Person("Alice", 21));
people.add(new Person("Bill", 18));
```

```
try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("people.data"))) {
  oos.writeObject(people);
} catch (Exception ex) {
  System.out.println(ex.getMessage());
}
try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("people.data"))) {
  ArrayList<Person> p = (ArrayList<Person>) ois.readObject();
  System.out.println(p);
} catch (Exception ex) {
  System.out.println(ex.getMessage());
}
```

```
[Tom is 20 years old., Nick is 19 years old., Alice is 21
years old., Bill is 18 years old.]
```

Исключение данных из сериализации

По умолчанию сериализуются все переменные объекта. Однако, возможно, мы хотим, чтобы некоторые поля были исключены из сериализации. Для этого они должны быть объявлены с модификатором **transient**. Например, исключим из сериализации объекта *Person* переменную *name*:

```
class Person implements Serializable {
  private transient String name;
  private int age;
  …
}
oos.writeObject(people);
System.out.println((ArrayList<Person>) ois.readObject());
```

```
[null is 20 years old., null is 19 years old., null is 21
years old., null is 18 years old.]
```

Класс File для работы с файлами и каталогами

Класс *File*, определенный в пакете *java.io*, не работает напрямую с потоками. Его задачей является управление информацией о файлах и каталогах. Хотя на уровне операционной системы файлы и каталоги отличаются, но в Java они описываются одним классом *File*.

Класс *File* имеет ряд методов, которые позволяют управлять файлами и каталогами. Рассмотрим некоторые из них:

– **boolean** createNewFile() – создает новый файл по пути, который передан в конструктор. В случае удачного создания возвращает **true**, иначе – **false**;

– **boolean** delete() – удаляет каталог или файл по пути, который передан в конструктор. При удачном удалении возвращает **true**;

– **boolean** exists() – проверяет, существует ли по указанному в конструкторе пути файл или каталог. И если файл или каталог существует, то возвращает **true**, иначе возвращает **false**;

– String getAbsolutePath() – возвращает абсолютный путь для файла, переданного в конструктор объекта;

– String getName() – возвращает краткое имя файла или каталога;

– String getParent() – возвращает имя родительского каталога;

– **boolean** isDirectory() – возвращает значение **true**, если по указанному пути располагается каталог;

– **boolean** isFile() – возвращает значение **true**, если по указанному пути находится файл;

– **boolean** isHidden() – возвращает значение **true**, если каталог или файл являются скрытыми;

– **long** length() – возвращает размер файла в байтах;

– **long** lastModified() – возвращает время последнего изменения файла или каталога. Значение представляет количество миллисекунд, прошедших с начала эпохи Unix;

– String[] list() – возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге;

– File[] listFiles() – возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге;

– **boolean** mkdir() – создает новый каталог и при удачном создании возвращает значение **true**;

– **boolean** renameTo(File dest) – переименовывает файл или каталог.

Порядок выполнения работы

Требуется выполнить задание (A), используя классы файлового ввода и вывода. В задании (B) необходимо модифицировать копию лабораторной работы № 6 для чтения списка из файла и записи списка в файл.

Для демонстрации задания (A) и задания (B) нужно разработать программы *ProgramA* и *ProgramB* соответственно.

Задание (A) для лабораторной работы по вариантам

1. Разделить текст файла на предложения. Для каждого предложения вывести длину и используемые в нем знаки пунктуации.

2. Файл содержит целые числа. При чтении файла, в зависимости от чётности или нечётности числа, заносить его в один из двух списков.

3. В указанном каталоге удалить файлы, в названии которых содержится введённая подстрока.

4. Из находящихся в файле слов составить предложения, содержащие введённое пользователем количество слов.

5. Для указанного файла составить словарь: вывести список слов и частоту их встречаемости.

6. Посчитать количество файлов введённого пользователем разрешения в указанной директории.

7. Инвертировать названия файлов (не изменяя расширения) в указанной директории.

8. В бинарном файле записаны по очереди целые числа (**int**) и вещественные числа (**double**). Выведите **int** или **double**, в зависимости от того, сумма которого типа больше.

9. Вывести файлы с размером больше указанного в каталоге и вложенных подкаталогах.

10. В текстовом файле с учётом регистра инвертировать слова (если слово начиналось с большой буквы, то после инвертирования оно тоже должно начинаться с большой буквы, хотя это будет последняя буква слова).

11. Определить суммарный объём файлов в указанной директории.

12. В бинарном файле записаны строки. Вывести их в отсортированном по алфавиту порядке.

13. Для текстового файла посчитать количество слов одинаковой длины.

14. Вывести целые числа (**int**), составленные из считанных в текстовом файле байтов.

15. Посчитать сумму встречаемых в текстовом файле чисел (за разделитель принимать любой не цифровой знак).

Задание (В) для лабораторной работы по вариантам

Для выполнения задания в классе списка могут быть созданы методы *read* и *write* соответственно для чтения и записи списка. В методе *main* главного класса программы в меню также должны быть добавлены пункты *read* и *write*. Для хранения записей используется бинарный файл, в котором размещён сразу весь список или идёт количество элементов в списке и сами элементы списка.

Контрольные вопросы

1. Какие существуют основные классы для работы с данными в виде потока байтов и текста?
2. Для чего используются классы *InputStream*, *OutputStream*? Как осуществляется чтение/запись с помощью них?
3. Для чего используются классы *Reader*, *Writer*? С помощью каких конструкторов можно создать объекты этого класса?
4. Как осуществляется закрытие потоков? Для каких классов закрытие потока выполнять не обязательно?
5. Как осуществляется чтение/запись с помощью классов *FileInputStream*, *FileOutputStream*?
6. Что такое буферизированные потоки? Какие буферизированные потоки вы знаете?
7. Какие классы используются для форматируемого вывода?
8. Какие классы используются для работы с текстовыми файлами?
9. Что такое сериализация? Как осуществляется сериализация классов?
10. Как осуществляется работа с файлами и директориями?

Содержание отчета

Скриншоты (3): главного класса программы для задания (A), метода записи списка объектов в файл для задания (B), метода чтения списка объектов из файла для задания (B).

Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответ на вопросы по теме текущей лабораторной работы.
5. Ответ на вопросы по теме предыдущей лабораторной работы.

ЛАБОРАТОРНАЯ РАБОТА № 8.
ЛЯМБДА-ВЫРАЖЕНИЯ

Цель: Ознакомиться с лямбда-выражениями, научиться их использовать в программе.

Краткие теоретические сведения.
Лямбда

Лямбда представляет набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку ->. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая собственно представляет тело лямбда-выражения, где выполняются все действия. Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе. При этом важно, что функциональный интерфейс должен содержать только один единственный метод без реализации. Рассмотрим пример:

```
interface Operationable {
  int calculate(int x, int y);
}
Operationable operation;
operation = (x, y)-> x + y;
int result = operation.calculate(10, 20);
System.out.println(result);
```

```
30
```

В роли функционального интерфейса выступает интерфейс *Operationable*, в котором определен один метод без реализации – метод *calculate*. Данный метод принимает два параметра – два целых числа, и возвращает некоторое целое число.

По факту лямбда-выражения являются в некотором роде сокращенной формой внутренних анонимных классов, которые ранее применялись в Java. В частности, предыдущий пример мы можем переписать следующим образом:

```
interface Operationable {
  int calculate(int x, int y);
}
Operationable operation = new Operationable() {
```

```
  public int calculate(int x, int y) {
    return x + y;
  }
};
int result = operation.calculate(10, 20);
```
```
30
```

Чтобы объявить и использовать лямбда-выражение, основная программа разбивается на ряд этапов:

1. Определение ссылки на функциональный интерфейс:

```
Operationable operation;
```

2. Создание лямбда-выражения:

```
operation = (x, y)-> x + y;
```

Причем параметры лямбда-выражения соответствуют параметрам единственного метода интерфейса *Operationable*, а результат соответствует возвращаемому результату метода интерфейса. При этом нам не надо использовать ключевое слово **return** для возврата результата из лямбда-выражения. Так, в методе интерфейса оба параметра представляют тип int, значит, в теле лямбда-выражения мы можем применить к ним сложение. Результат сложения также представляет тип int, объект которого возвращается методом интерфейса.

3. Использование лямбда-выражения в виде вызова метода интерфейса:

```
int result = operation.calculate(10, 20);
```

Так как в лямбда-выражении определена операция сложения параметров, результатом метода будет сумма чисел 10 и 20.

При этом для одного функционального интерфейса мы можем определить множество лямбда-выражений. Например:

```
Operationable operation1 = (x, y) -> x + y;
Operationable operation2 = (x, y) -> x - y;
Operationable operation3 = (x, y) -> x * y;
Operationable operation4 = (x, y) -> x / y;

System.out.println(operation1.calculate(20, 10));
System.out.println(operation2.calculate(20, 10));
System.out.println(operation3.calculate(20, 10));
System.out.println(operation4.calculate(20, 10));
```
```
30
10
200
2
```

Отложенное выполнение

Одним из ключевых моментов в использовании лямбд является отложенное выполнение (*deferred execution*). То есть мы определяем в одном месте программы лямбда-выражение и затем можем его вызывать при необходимости неопределенное количество раз в различных частях программы. Отложенное выполнение может потребоваться, к примеру, в следующих случаях:

– выполнение кода в отдельном потоке;

– выполнение одного и того же кода несколько раз;

– выполнение кода в результате какого-то события;

– выполнение кода только в том случае, когда он действительно необходим и если он необходим.

Передача параметров в лямбда-выражение

Параметры лямбда-выражения должны соответствовать типам параметров метода из функционального интерфейса. При написании самого лямбда-выражения тип параметров писать необязательно, хотя в принципе это можно сделать, например:

```
operation = (int x, int y) -> x + y;
```

Если метод не принимает никаких параметров, то пишутся пустые скобки, например:

```
() -> 30 + 20;
```

Если метод принимает только один параметр, то скобки можно опустить:

```
n -> n * n;
```

Терминальные лямбда-выражения

Выше мы рассмотрели лямбда-выражения, которые возвращают определенное значение. Но также могут быть и терминальные лямбды, которые не возвращают никакого значения. Например:

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

Printable printer = s -> System.out.println(s);
printer.print("Hello Java!");
```

```
Hello Java!
```

Лямбды и локальные переменные

Лямбда-выражение может использовать переменные, которые объявлены во вне, в более общей области видимости, – на уровне

класса или метода, в котором лямбда-выражение определено. Однако в зависимости от того, как и где определены переменные, могут различаться способы их использования в лямбдах.

Рассмотрим первый пример – использования переменных уровня класса:

```
interface Operationable {
  int calculate();
}

class Program {
  static int x = 20;
  static int y = 10;
  public static void main(String…args) {
    Operationable op = () -> {x = 30; return x + y;};
    System.out.println(op.calculate());
    System.out.println(x);
  }
}
```

```
40
30
```

Переменные *x* и *y* объявлены на уровне класса, и в лямбда-выражении мы их можем получить и даже изменить. Так, в данном случае после выполнения выражения изменяется значение переменной *x*.

Теперь рассмотрим другой пример – локальные переменные на уровне метода:

```
int m = 70, n = 80;
Operation op = () -> {return m + n;};
System.out.println(op.calculate());
```

```
150
```

Локальные переменные уровня метода мы также можем использовать в лямбдах, но изменять их значение мы уже не сможем. Если мы попробуем это сделать, то среда разработки (Netbeans) может нам высветить ошибку и то, что такую переменную надо пометить с помощью ключевого слова **final**, то есть сделать константой: ***final int*** *n=70;*. Однако это необязательно.

Более того, мы не сможем изменить значение переменной, которая используется в лямбда-выражении, вне этого выражения. То есть даже если такая переменная не объявлена как константа, по сути она является константой.

Блоки кода в лямбда-выражениях

Существуют два типа лямбда-выражений: однострочное выражение и блок кода. Примеры однострочных выражений демонстрировались выше. Блочные выражения обрамляются фигурными скобками. В блочных лямбда-выражениях можно использовать внутренние вложенные блоки, циклы, конструкции **if**, **switch**, создавать переменные и т.д. Если блочное лямбда-выражение должно возвращать значение, то явным образом применяется оператор **return**:

```
interface Operationable {
  int calculate(int x, int y);
}
Operationable operation = (x, y) -> {
  if (y == 0) {
    return 0;
  } else {
    return x / y;
  }
};
System.out.println(operation.calculate(20, 10));
System.out.println(operation.calculate(20, 0));
```

```
2
0
```

Обобщенный функциональный интерфейс

```
interface Operationable<T> {
  T calculate(T x, T y);
}
Operationable<Integer> operation1 = (x, y) -> x + y;
Operationable<String> operation2 = (x, y) -> x + y;
System.out.println(operation1.calculate(20, 10));
System.out.println(operation2.calculate("10", "20"));
```

```
30
1020
```

Таким образом, при объявлении лямбда-выражения ему уже известно, какой тип параметры будут представлять и какой тип они будут возвращать.

Лямбды как параметры и результаты методов. Лямбды как параметры методов

Одним из преимуществ лямбд в Java является то, что их можно передавать в качестве параметров в методы. Рассмотрим пример:

```
interface Expression {
  boolean isEqual(int n);
}
```

```
int sum(int[] numbers, Expression expression) {
  int result = 0;
  for (int i: numbers) {
    if (expression.isEqual(i)) {
      result += i;
    }
  }
  return result;
}

Expression expression = (n) -> n%2 == 0;
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
System.out.println(sum(numbers, expression));
20
```

Функциональный интерфейс *Expression* определяет метод *isEqual()*, который возвращает **true**, если в отношении числа n действует какое-нибудь равенство.

В основном классе программы определяется метод *sum()*, который вычисляет сумму всех элементов массива, соответствующих некоторому условию, а само условие передается через параметр *Expression expression*. Причем на момент написания метода *sum* мы можем абсолютно не знать, какое именно условие будет использоваться. Само же условие определяется в виде лямбда-выражения:

```
Expression expression = (n) -> n%2 == 0;
```

То есть в данном случае все числа должны быть четными или остаток от их деления на 2 должен быть равен 0. Затем это лямбда-выражение передается в вызов метода *sum*. При этом можно не определять переменную интерфейса, а сразу передать в метод лямбда-выражение:

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int x = sum(numbers, (n) -> n > 5);
System.out.println(x);
```

Ссылки на метод как параметры методов

Начиная с JDK 8, в Java можно в качестве параметра в метод передавать ссылку на другой метод. В принципе данный способ аналогичен передаче в метод лямбда-выражения.

Ссылка на метод может передаваться первым или вторым способом:

1) имя_класса::имя_статического_метода (если метод статический);
2) объект_класса::имя_метода (если метод нестатический).

Рассмотрим на примере:

```
interface Expression {
  boolean isEqual(int n);
}
class Expressions {
  static boolean isEven(int n) {
    return n%2 == 0;
  }
  static boolean isPositive(int n) {
    return n > 0;
  }
}

int sum(int[] numbers, Expression expression) {
  int result = 0;
  for (int i: numbers) {
    if (expression.isEqual(i)) {
      result += i;
    }
  }
  return result;
}
int[] numbers = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
System.out.println(sum(numbers, Expressions::isEven));
System.out.println(sum(numbers, Expressions::isPositive));
```

```
0
15
```

Здесь также определен функциональный интерфейс *Expression*, который имеет один метод. Кроме того, определен класс *Expressions*, который содержит два статических метода. В принципе, их можно было определить и в основном классе программы. В программе определен метод *sum()*, который возвращает сумму элементов массива, соответствующих некоторому условию. Условие передается в виде объекта функционального интерфейса *Expression*. В методе *main* два раза вызываем метод *sum*, передавая в него один и тот же массив чисел, но разные условия. Первый вызов метода sum:

```
System.out.println(sum(numbers, Expressions::isEven));
```

На место второго параметра передается *Expressions::isEven*, то есть ссылка на статический метод *isEven()* класса *Expressions*. При этом методы, на которые идет ссылка, должны совпадать по параметрам и результату с методом функционального интерфейса.

Использование ссылок на методы в качестве параметров аналогично использованию лямбда-выражений.

Если нам надо вызвать нестатические методы, то в ссылке вместо имени класса применяется имя объекта этого класса:

```
interface Expression {
  boolean isEqual(int n);
}
class Expressions {
  boolean isEven(int n) {
    return n%2 == 0;
  }
}
int[] numbers = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
Expressions expressions = new Expressions();
System.out.println(sum(numbers, expressions::isEven));
```

Ссылки на конструкторы

Подобным образом мы можем использовать конструкторы: название_класса::**new**. Например:

```
class Person {
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public String toString() {
    return name + " " + age;
  }
}
interface PersonBuilder {
  Person create(String name, int age);
}
PersonBuilder personBuilder = Person::new;
Person person = personBuilder.create("Tom", 20);
System.out.println(person);
```

```
Tom 20
```

При использовании конструкторов методы функциональных интерфейсов должны принимать тот же список параметров, что и конструкторы класса, и должны возвращать объект данного класса.

Лямбды как результат методов

Также метод в Java может возвращать лямбда-выражение. Рассмотрим следующий пример:

```
interface Operation {
  int execute(int x, int y);
}
```

```
Operation action(int number) {
  switch (number) {
    case 1:
      return (x, y) -> x + y;
    case 2:
      return (x, y) -> x - y;
    case 3:
      return (x, y) -> x * y;
    case 4:
      return (x, y) -> x / y;
  }
  return (x, y) -> 0;
}
Operation operation = action(1);
int a = operation.execute(6, 5);
System.out.println(a);
int b = action(2).execute(8, 2);
System.out.println(b);
```

```
11
6
```

В данном случае определен функциональный интерфейс *Operation*, в котором метод *execute* принимает два значения типа *int* и возвращает значение типа *int*.

Метод *action* принимает в качестве параметра число и в зависимости от его значения возвращает то или иное лямбда-выражение. Оно может представлять либо сложение, либо вычитание, либо умножение, либо просто возвращает 0. Стоит учитывать, что формально возвращаемым типом метода *action* является интерфейс *Operation*, а возвращаемое лямбда-выражение соответствует этому интерфейсу.

В методе *main* мы можем вызвать данный метод *action*. Например, сначала получить его результат – лямбда-выражение, которое присваивается переменной *Operation*, а затем через метод *execute* выполнить это лямбда-выражение:

```
Operation operation = action(1);
int a = operation.execute(6, 5);
System.out.println(a);
```

Либо можно сразу получить и тут же выполнить лямбда-выражение:

```
int b = action(2).execute(8, 2);
System.out.println(b);
```

Порядок выполнения работы

В задании (А), согласно варианту задания, требуется создать интерфейс и класс, при необходимости. В методе *main* главного класса программы *ProgramA* требуется создать лямбда-выражение, согласно варианту, и продемонстрировать его работу. В задании (В) требуется модифицировать копию лабораторной работы № 7, заменив поиск и сортировку лямбда-выражениями.

Для демонстрации Задания (А) и Задания (В) нужно разработать программы *ProgramA* и *ProgramB* соответственно.

Задание (А) для лабораторной работы по вариантам

1. Написать интерфейс *Expression*, который содержит метод. Этот метод принимает в качестве параметров два вектора и возвращает вектор. Создать класс *Expressions*, содержащий статические методы: сложение векторов, разность векторов. Создать метод *operation*, который принимает в качестве параметров ссылку на элемент интерфейса *Expression* (выражение) и два вектора и который выполняет данное выражение. В главном классе программы вызвать методы класса *Expressions*, передав выражение и вектора методу *operation* в качестве параметров.

2. Написать интерфейс, который содержит метод. Этот метод принимает в качестве параметра строку и возвращает строку. Создать метод, возвращающий лямбда-выражение различных операций над строкой: удаление подряд идущих одинаковых символов, инвертирование содержимого строки, инвертирование регистра строки. В главном классе программы продемонстрировать выполнение каждой операции.

3. Написать интерфейс, который содержит обобщенный метод. Метод принимает в качестве параметра массив значений и возвращает значение указанного типа. Создать два лямбда-выражения: нахождение минимального элемента массива, сумма положительных и отрицательных элементов вещественного массива.

4. Написать интерфейс, который содержит метод. Метод принимает в качестве параметра массив строк и возвращает сроку. Создать лямбда-выражение, находящее строку с наибольшим количеством символов в верхнем регистре.

5. Написать интерфейс *Expression*, который содержит метод. Метод принимает в качестве параметров две строки и возвращает

строку. Создать класс *Expressions*, содержащий статические методы: объединения строк, удаление из первой строки всех символов, содержащихся во второй строке. Создать метод *operation*, принимающий в качестве параметров ссылку на элемент интерфейса *Expression*, массив строк, и выполняющий выражение для соседних элементов массива. В главном классе программы объявить и инициализировать массив строк и вызвать методы класса *Expressions*, передав метод и массив методу *operation* в качестве параметров.

6. Написать интерфейс, который содержит метод. Метод принимает в качестве параметров два целых числа и возвращает целое число. Создать метод, возвращающий лямбда-выражение различных операций между двумя числами. В главном классе программы продемонстрировать выполнение каждой операции.

7. Написать интерфейс, который содержит обобщенный метод для целочисленного типа. Метод принимает в качестве параметра массив значений и возвращает значение указанного типа. Создать два лямбда-выражения: подсчет суммы по модулю два элементов целочисленного массива, вычисление произведения элементов вещественного массива.

8. Написать интерфейс, который содержит метод. Метод принимает в качестве параметров два вектора целых чисел и возвращает целое число. Создать лямбда-выражение, вычисляющее сумму элементов, образованных в результате произведения первого транспонированного вектора на второй.

9. Написать интерфейс, который содержит метод. Метод принимает в качестве параметров матрицу целых чисел, целое число и возвращает целое число. Создать лямбда-выражение, вычисляющее сумму чисел из столбца матрицы с номером, переданным в качестве второго параметра.

10. Написать интерфейс, который содержит обобщенный метод. Метод принимает в качестве параметра массив значений и возвращает значение указанного типа. Создать два лямбда-выражения: объединение строк массива в одну, нахождение самой короткой строки.

11. Написать интерфейс, который содержит метод. Метод принимает в качестве параметра массив целых чисел и возвращает истину или ложь. Создать лямбда-выражение, определяющее принадлежность массива последовательности Фибоначчи.

12. Написать интерфейс, который содержит метод. Метод принимает в качестве параметра матрицу целых чисел и возвращает матрицу целых чисел. Создать лямбда-выражение, осуществляющее сортировку строк матрицы в зависимости от значения произведения элементов строки.

13. Написать интерфейс, который содержит метод. Метод принимает в качестве параметров два массива строк (оба массива одинакового размера) и возвращает словарь. Создать лямбда-выражение, сопоставляющее слову перевод.

14. Написать интерфейс, который содержит метод. Метод принимает в качестве параметров массив строк, строку и возвращает строку. Создать лямбда-выражение, объединяющее массив строк в одну, используя в качестве разделителя другую строку.

15. Написать интерфейс, который содержит метод. Метод принимает в качестве аргументов массив целых чисел и возвращает массив вещественных чисел. Создать лямбда-выражение, инвертирующее порядок элементов массива и приводящее их к вещественному типу.

Задание (В) для лабораторной работы по вариантам

Для выполнения задания нужно используемые компараторы заменить на лямбда-выражения и модифицировать поиск для нахождения элемента по полю базового или дочернего класса.

Контрольные вопросы

1. Что такое лямбда-выражения?
2. Какие требования предъявляются к интерфейсу для использования его в качестве лямбда-выражения?
3. Как осуществляется передача параметров в лямбда-выражение?
4. Как локальные переменные могут быть использованны в лямбда-выражениях?
5. Как используется интерфейс с обобщенными типами в лямбда-выражениях?
6. Как лямбда-выражение может быть использованно в качестве параметра метода?
7. Как лямбда-выражение может быть использованно в качестве ссылки на метод в параметрах метода?

8. Как с помощью лямбда-выражения можно создавать объекты класса?

9. Как лямбда-выражения могут быть использованы в качестве возвращаемых значений метода?

10. Какие примеры использования лямбда-выражений вы можете привести?

Содержание отчета

Скриншоты (2): реализованного интерфейса для задания, метода *main* главного класса программы для задания.

Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответ на вопросы по теме текущей лабораторной работы.
5. Ответ на вопросы по теме предыдущей лабораторной работы.

СПИСОК РЕКОМЕНДОВАННОЙ ЛИТЕРАТУРЫ

1. Bates, B. SCJP Sun Certified Programmer for Java 6 Study Guide / B. Bates, S. Kathy. – O'Reilly Media, 2008. – 890 p.
2. Boyarsky, J. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide: Exam 1Z0-809 / J. Boyarsky, S. Selikoff. – John Wiley & Sons, 2015. – 723 p.
3. Блинов, И. Н. Java. Методы программирования : учеб.-метод. пособие / И. Н. Блинов, В. С. Романчик. – Минск : Четыре четверти, 2013. – 896 с.
4. Программирование на языке Java. Конспект лекций / А. В. Гаврилов [и др.]. – СПб : Ун-т ИТМО, 2015. – 126 с.
5. МакГрат, М. Программирование на Java для начинающих / М. МакГрат. – М. : Эксмо, 2016. – 192 с.
6. Нимейер, П. Программирование на Java / П. Нимейер, Д. Леук. – М. : Эксмо, 2014. – 1216 с.
7. Пол, А. Объектно-ориентированное программирование на C++ / А. Пол. – М. : Бином, 2001. – 476 с.
8. Шилдт, Г. Java. Полное руководство / Г. Шилдт. – Изд. 8-е. – М. : ООО «И.Д. Вильямс», 2012. – 1104 с.
9. Шилдт, Г. Java 8: руководство для начинающих / Г. Шилдт. – Изд. 6-е. – М. : ООО «И.Д. Вильямс», 2015. – 720 с.
10. Эккель, Б. Философия Java. Библиотека программиста / Б. Эккель. – Изд. 4-е. – СПб. : Питер, 2009. – 640 с.

Примеры на Java

В качестве примера приведены исходные коды для лабораторных работ 1, 2, 3 и 4 с упрощенными вариантами (в классе меньше атрибутов и в лабораторной работе № 4 нет примера с классами). Варианты на лабораторные работы 1, 2 и 4 (в лабораторной работе № 3 у всех одно и тоже задание):

1. Домашний питомец.
2. Домашний питомец, кошка, собака, попугай.
4. Односвязный список.

Для запуска примера лабораторной работы следует расположить исходные коды лабораторной работы в пакетах согласно названиям и использовать «makefile» для запуска программы. Команды в «makefile» позволяют:

a) build – рядом с папкой «src» создать папку «build», содержащую скомпилированную программу;

b) run – запустить уже скомпилированную программу;

c) clear – удалить папку «build»;

d) all – последовательно выполнить команды «make build», «make run», «make clear».

ЛАБОРАТОРНАЯ РАБОТА № 1

makefile

```
MAKEFLAGS += --silent

build:
    rm -f -R build
    javac --source-path src -d build src/*.java

run:
    java --class-path build Main

clear:
    rm -f -R build

all:
    make clear
    make build
    make run
    make clear
```

src/Pet.java

```
public class Pet {

  protected String name;
  protected int age;

  public Pet() {
    this.name = "noname";
    this.age = 0;
  }

  public Pet(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public Pet(Pet pet) {
    this.name = pet.name;
    this.age = pet.age;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getAge() {
    return this.age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  public boolean equals(Object obj) {
    if (obj==null || !(obj instanceof Pet)) {
      return false;
    }
    Pet pet = (Pet) obj;
    return this.name==pet.name && this.age==pet.age;
  }

  public String toString() {
    return name + " is " + age + " years old";
  }

}
```

src/Main.java

```
public class Main {
  public static void main(String...args) {
    Pet[] pets = new Pet[5];
    pets[0] = new Pet("Alex", 6);
    pets[1] = new Pet("Bob", 2);
    pets[2] = new Pet("Cyan", 7);
    pets[3] = new Pet("Dillan", 4);
    pets[4] = new Pet("Elan", 9);
    for (Pet pet: pets) {
      System.out.println(pet);
    }
    Pet p = new Pet("Alex", 6);
    for (Pet pet: pets) {
      if (pet.equals(p)) {
        System.out.println("Pet found!");
      }
    }
  }
}
```

```
Alex is 6 years old
Bob is 2 years old
Cyan is 7 years old
Dillan is 4 years old
Elan is 9 years old
Pet found!
```

ЛАБОРАТОРНАЯ РАБОТА № 2

src/pets/Pet.java

```
package pets;

public class Pet {
  …
}
```

src/pets/Cat.java

```
package pets;

public class Cat extends Pet {

  protected int tailLength;

  public Cat() {
    super();
    this.tailLength = 15;
  }
```

```
  public Cat(String name, int age, int tailLength) {
    super(name, age);
    this.tailLength = tailLength;
  }

  public Cat(Cat cat) {
    super(cat);
    this.tailLength = cat.tailLength;
  }

  public int getTailLength() {
    return tailLength;
  }

  public void setTailLength(int tailLength) {
    this.tailLength = tailLength;
  }

  public boolean equals(Object obj) {
    if (obj==null || !(obj instanceof Cat)) {
      return false;
    }
    Cat cat = (Cat) obj;
    return super.equals(cat) &&
this.tailLength==cat.tailLength;
  }

  public String toString() {
    return super.toString() + " and his tail is " + tailLength
+ " cm";
  }

}
```

src/Dog.java

```
package pets;

public class Dog extends Pet {

  protected short maxSpeed;

  public Dog() {
    super();
    this.maxSpeed = 10;
  }

  public Dog(String name, int age, short maxSpeed) {
    super(name, age);
    this.maxSpeed = maxSpeed;
  }
```

```
  public Dog(Dog dog) {
    super(dog);
    this.maxSpeed = dog.maxSpeed;
  }

  public short getMaxSpeed() {
    return maxSpeed;
  }

  public void setMaxSpeed(short maxSpeed){
    this.maxSpeed = maxSpeed;
  }

  public boolean equals(Object obj) {
    if (obj==null || !(obj instanceof Dog)) {
      return false;
    }
    Dog dog = (Dog) obj;
    return super.equals(dog) && this.maxSpeed==dog.maxSpeed;
  }

  public String toString() {
    return super.toString() + " and run at " + maxSpeed + "
m/s";
  }

}
```

src/Parrot.java

```
package pets;

public class Parrot extends Pet {

  protected int numOfColors;

  public Parrot() {
    super();
    this.numOfColors = 1;
  }

  public Parrot(String name, int age, int numOfColors) {
    super(name, age);
    this.numOfColors = numOfColors;
  }
```

```
  public Parrot(Parrot parrot) {
    super(parrot);
    this.numOfColors = parrot.numOfColors;
  }

  public int getNumOfColors() {
    return numOfColors;
  }

  public void setNumOfColors(int numOfColors) {
    this.numOfColors = numOfColors;
  }

  public boolean equals(Object obj) {
    if (obj==null || !(obj instanceof Parrot)) {
      return false;
    }
    Parrot parrot = (Parrot) obj;
    return super.equals(parrot) &&
this.numOfColors==parrot.numOfColors;
  }

  public String toString() {
    return super.toString() + " and pictured in " +
numOfColors + " colors";
  }

}
```

src/Main.java

```
import pets.*;


public class Main {
  public static void main(String...args) {
    Pet[] pets = new Pet[6];
    pets[0] = new Cat("Alex", 6, 10);
    pets[1] = new Dog("Bob", 2, (short) 11);
    pets[2] = new Parrot("Cyan", 7, 10);
    pets[3] = new Cat("Dillan", 4, 15);
    pets[4] = new Dog("Elan", 9, (short) 12);
    pets[5] = new Parrot("Felix", 10, 3);

    for (Pet pet: pets) {
      System.out.println(pet);
```

```
    }
    Pet p = new Cat("Alex", 6, 12);
    for (Pet pet: pets) {
      if (pet.equals(p)) {
        System.out.println("Pet found!");
      }
    }
  }
}
```

```
Alex is 6 years old and his tail is 10 cm
Bob is 2 years old and run at 11 m/s
Cyan is 7 years old and pictured in 10 colors
Dillan is 4 years old and his tail is 15 cm
Elan is 9 years old and run at 12 m/s
Felix is 10 years old and pictured in 3 colors
```

ЛАБОРАТОРНАЯ РАБОТА № 3

src/list/PetList.java

```
package list;

import java.util.ArrayList;

import pets.Pet;
import pets.PetComparator;


public class PetList extends ArrayList<Pet> {

  public void sort() {
    for (int i=0; i<this.size(); i++) {
      for (int j=i+1; j<this.size(); j++) {
        if (this.get(i).compareTo( this.get(j) ) > 0) {
          Pet pet = this.get(i);
          this.set(i, this.get(j));
          this.set(j, pet);
        }
      }
    }
  }

  public void sort(PetComparator petComparator) {
    for (int i=0; i<this.size(); i++) {
      for (int j=i+1; j<this.size(); j++) {
```

```
        if (petComparator.compare( this.get(i), this.get(j) )
> 0) {
          Pet pet = this.get(i);
          this.set(i, this.get(j));
          this.set(j, pet);
        }
      }
    }
  }

  public String toString() {
    String str = "";
    for (Pet pet: this) {
      str += pet.toString() + "\n";
    }
    return str;
  }

}
```

src/Pets/PetComparable.java

```
package pets;


public interface PetComparable extends Comparable<Pet> {
  int compareTo(Pet pet);
}
```

src/Pets/Pet.java

```
package pets;


public abstract class Pet implements PetComparable {
  …
  public int compareTo(Pet pet) {
    return
this.getClass().getSimpleName().compareTo(pet.getClass().getSi
mpleName());
  }
  …
}
```

src/Pets/Cat.java

```
package pets;

public class Cat extends Pet {
  …
```

```
  public int compareTo(Pet pet) {
    if (pet instanceof Cat) {
      Cat cat = (Cat) pet;
      return this.tailLength - cat.tailLength;
    }
    if (pet instanceof Dog) {
      return -1;
    }
    if (pet instanceof Parrot) {
      return +1;
    }
    return 0;
  }
  …
}
```

src/Pets/Dog.java

```
package pets;


public class Dog extends Pet {
…
  public int compareTo(Pet pet) {
    if (pet instanceof Cat) {
      return +1;
    }
    if (pet instanceof Dog) {
      Dog dog = (Dog) pet;
      return this.maxSpeed - dog.maxSpeed;
    }
    if (pet instanceof Parrot) {
      return +1;
    }
    return 0;
  }
…
}
```

src/Pets/Parrot.java

```
package pets;


public class Parrot extends Pet {
…
  public int compareTo(Pet pet) {
```

```
    if (pet instanceof Cat) {
      return -1;
    }
    if (pet instanceof Dog) {
      return -1;
    }
    if (pet instanceof Parrot) {
      Parrot parrot = (Parrot) pet;
      return this.numOfColors-parrot.numOfColors;
    }
    return 0;
  }
…
}
```

src/Pets/PetComparator.java

```
package pets;


public interface PetComparator {
  public int compare(Pet p0, Pet p1);
}
```

src/Pets/PetComparatorName.java

```
package pets;


public class PetComparatorName implements PetComparator {
  public int compare(Pet p0, Pet p1) {
    return p0.getName().compareTo(p1.getName());
  }
}
```

src/Pets/PetComparatorAge.java

```
package pets;


public class PetComparatorAge implements PetComparator {
  public int compare(Pet p0, Pet p1) {
    return p0.getAge() - p1.getAge();
  }
}
```

src/utils/PetInput.java

```
package utils;

import java.io.PrintStream;
import java.util.Scanner;
```

```
import pets.Cat;
import pets.Dog;
import pets.Parrot;


public class PetInput {

  private static PrintStream out = System.out;
  private static Scanner in = new Scanner(System.in);

  public static Cat nextCat() {
    out.println("Cat: \'Name\' \'Age\' \'Tail Length\'");
    String name = in.next();
    int age = in.nextInt();
    int tailLength = in.nextInt();
    return new Cat(name, age, tailLength);
  }

  public static Dog nextDog() {
    out.println("Dog: \'Name\' \'Age\' \'Max Speed\'");
    String name = in.next();
    int age = in.nextInt();
    short maxSpeed = in.nextShort();
    return new Dog(name, age, maxSpeed);
  }

  public static Parrot nextParrot() {
    out.println("Parrot: \'Name\' \'Age\' \'Num of Colors\'");
    String name = in.next();
    int age = in.nextInt();
    int numOfColors = in.nextInt();
    return new Parrot(name, age, numOfColors);
  }

}
```

src/Main.java

```
import list.PetList;
import pets.Cat;
import pets.Dog;
import pets.Parrot;
import utils.PetInput;


public class Main {
  public static void main(String...args) {
```

```
    PetList pl = new PetList();

    pl.add(new Cat("Alex", 6, 10));
    pl.add(new Dog("Bob", 2, (short) 11));
    pl.add(new Parrot("Cyan", 7, 10));
    pl.add(new Cat("Dillan", 4, 15));
    pl.add(new Dog("Elan", 9, (short) 12));
    pl.add(new Parrot("Felix", 10, 3));

    pl.add(PetInput.nextCat());
    pl.add(PetInput.nextDog());
    pl.add(PetInput.nextParrot());

    System.out.println("Before sort:");
    System.out.println(pl);

    System.out.println("After sort:");
    pl.sort();
    System.out.println(pl);
  }
}
```

```
Cat: 'Name' 'Age' 'Tail Length'
Alex 5 15
Dog: 'Name' 'Age' 'Max Speed'
Bob 5 30
Parrot: 'Name' 'Age' 'Num of Colors'
Cyan 1 5
Before sort:
Alex is 6 years old and his tail is 10 cm
Bob is 2 years old and run 11 m/s
Cyan is 7 years old and pictured in 10 colors
Dillan is 4 years old and his tail is 15 cm
Elan is 9 years old and run 12 m/s
Felix is 10 years old and pictured in 3 colors
Alex is 5 years old and his tail is 15 cm
Bob is 5 years old and run 30 m/s
Cyan is 1 years old and pictured in 5 colors

After sort:
Felix is 10 years old and pictured in 3 colors
Cyan is 1 years old and pictured in 5 colors
Cyan is 7 years old and pictured in 10 colors
Alex is 6 years old and his tail is 10 cm
Alex is 5 years old and his tail is 15 cm
Dillan is 4 years old and his tail is 15 cm
Bob is 2 years old and run 11 m/s
Elan is 9 years old and run 12 m/s
Bob is 5 years old and run 30 m/s
```

src/SingleLinkedList.java

```
public class SingleLinkedList<T> {

  private class Node<T> {
    public T data;
    public Node<T> next;
    public Node(T data) {
      this.data = data;
      this.next = null;
    }
    public String toString() {
      return data.toString();
    }
  }

  private Node<T> head;
  private int size;

  public SingleLinkedList() {
    head = null;
    size = 0;
  }

  public boolean add(T t) {
    Node<T> newT = new Node<T>(t);
    if (head == null) {
      head = newT;
    } else {
      Node<T> last = head;
      while (last.next != null) {
        last = last.next;
      }
      last.next = newT;
    }
    size++;
    return true;
  }

  public boolean add(int i, T t) {
    if (i==0) {
      Node<T> node = new Node<T>(t);
      node.next = head;
      head = node;
```

```
      return true;
    } else if (i<size) {
      Node<T> prev = head, curr = new Node<T>(t), next =
head.next;
      for (int c=0; c<i-1; c++) {
        prev = prev.next;
        next = next.next;
      }
      prev.next = curr;
      curr.next = next;
      return true;
    }
    size++;
    return false;
  }

  public void clear() {
    Node<T> last = head;
    while (last.next != null) {
      last = last.next;
    }
    last.next = head;
    head = null;
    size = 0;
  }

  public T get(int i) {
    if (i<0 || i>=size) {
      return null;
    }
    Node<T> curr = head;
    for (int c=0; c<i; c++) {
      curr = curr.next;
    }
    return curr.data;
  }

  public boolean isEmpty() {
    return size == 0;
  }

  public T remove(int i) {
    T t = null;
    if (i==0) {
      Node<T> curr = head;
      head = head.next;
```

```
      t = curr.data;
      curr.next = null;
    } else if (i<size) {
      Node<T> prev = head, curr = head.next, next =
head.next.next;
      for (int c=0; c<i-1; c++) {
        prev = prev.next;
        curr = curr.next;
        next = next.next;
      }
      prev.next = next;
      t = curr.data;
      curr.next = null;
    }
    size--;
    return t;
  }

  public boolean set(int i, T t) {
    if (i<0 || i>=size) {
      return false;
    }
    Node<T> curr = head;
    for (int c=0; c<i; c++) {
      curr = curr.next;
    }
    curr.data = t;
    return true;
  }

  public int size() {
    return size;
  }

  public String toString() {
    String s = "";
    for (Node<T> curr=head; curr!=null; curr=curr.next) {
      s += curr.toString() + "\n";
    }
    return s;
  }
}
```

src/Main.java

```
public class Main {
  public static void main(String...args) {
```

```
    SingleLinkedList<Integer> arr = new
SingleLinkedList<Integer>();
    arr.add(0);
    arr.add(2);
    arr.add(4);
    arr.add(1);
    arr.add(3);
    System.out.println(arr);
    System.out.println("# " + arr.set(2, 10));
    System.out.println(arr);
    arr.clear();
  }
}
```

```
0
2
4
1
3

# true
0
2
10
1
3
```

Примеры на C++

В качестве примера приведены исходные коды для лабораторных работ 1, 2, 3 и 4 с упрощенными вариантами (в классе меньше атрибутов и в лабораторной работе № 4 нет примера с классами). Варианты на лабораторные работы 1, 2 и 4 (в лабораторной работе № 3 у всех одно и тоже задание):

1. Домашний питомец.
2. Домашний питомец, кошка, собака, попугай.
4. Односвязный список.

В C++ для сравнения двух объектов класса и вывода элемента класса в консоль требуется перегрузить соответствующие операторы.

ЛАБОРАТОРНАЯ РАБОТА № 1

pet.h

```
#pragma one

#include <iostream>
#include <string>
using namespace std;


class Pet {
protected:
  string name;
  int age;
public:
  Pet();
  Pet(string name, int age);
  Pet(const Pet &pet);
  string get_name() const;
  int get_age() const;
  void set_name(const string name);
  void set_age(const int age);
  friend bool operator == (const Pet &pet0, const Pet &pet1);
  friend ostream& operator << (ostream &out, const Pet &pet);
};
```

pet.cpp

```
#include "pet.h"


Pet::Pet() {
  this->name = "noname";
```

```
  this->age = 0;
}
Pet::Pet::Pet(string name, int age) {
  this->name = name;
  this->age = age;
}
Pet::Pet(const Pet &pet) {
  this->name = pet.name;
  this->age = pet.age;
}

string Pet::get_name() const {
  return this->name;
}
int Pet::get_age() const {
  return this->age;
}
void Pet::set_name(const string name) {
  this->name = name;
}
void Pet::set_age(const int age) {
  this->age = age;
}

bool operator == (const Pet &pet0, const Pet &pet1) {
  return pet0.name==pet1.name && pet0.age==pet1.age;
}
ostream& operator << (ostream &out, const Pet &pet) {
  out << pet.name << " is " << pet.age << " years old ";
  return out;
}
```

main.cpp

```
#include <iostream>
using namespace std;

#include "pet.h"


int main(int argc, char *argv[]) {
  Pet **pets = new Pet *[5];
  pets[0] = new Pet("Alex", 6);
  pets[1] = new Pet("Bob", 2);
  pets[2] = new Pet("Cyan", 7);
  pets[3] = new Pet("Dillan", 4);
  pets[4] = new Pet("Elan", 9);
```

```
  for (int i=0; i<5; i++) {
    cout << *pets[i] << endl;
  }
  Pet *p = new Pet("Alex", 6);
  for (int i=0; i<5; i++) {
    if (*p == *pets[i]) {
      cout << "Pet found!" << endl;
    }
  }
  return 0;
}
```

```
Alex is 6 years old
Bob is 2 years old
Cyan is 7 years old
Dillan is 4 years old
Elan is 9 years old
Pet found!
```

ЛАБОРАТОРНАЯ РАБОТА № 2

pets/pet.h

```
#pragma once

#include <iostream>
#include <string>
using namespace std;


class Pet {

…

  friend bool operator == (const Pet &pet0, const Pet &pet1);
  friend ostream& operator << (ostream &fout, const Pet &pet);
protected:
  virtual bool equal(const Pet &pet) const;
  virtual void print(ostream &fout) const;
};
```

pets/pet.cpp

```
#include "pet.h"

bool operator == (const Pet &pet0, const Pet &pet1) {
  return pet0.equal(pet1);
}
```

```
ostream& operator << (ostream &fout, const Pet &pet) {
  pet.print(fout);
  return fout;
}

bool Pet::equal(const Pet &pet) const {
  return this->name==pet.name && this->age==pet.age;
}
void Pet::print(ostream &fout) const {
  fout << this->name << " is " << this->age << " years old ";
}
…
```

pets/cat.h

```
#pragma once

#include <iostream>
#include <string>
using namespace std;

#include "cat.h"
#include "dog.h"
#include "parrot.h"
#include "pet.h"


class Cat: public Pet {
protected:
  int tail_length;
public:
  Cat();
  Cat(string name, int age, int tail_length);
  Cat(const Cat &cat);
  int get_tail_length() const;
  void set_tail_length(const int tail_length);
protected:
  bool equal(const Pet &pet) const override;
  void print(ostream &fout) const override;
};
```

pets/cat.cpp

```
#include "cat.h"


Cat::Cat(): Pet() {
  this->tail_length = 15;
}
```

```
Cat::Cat(string name, int age, int tail_length): Pet(name,
age) {
  this->tail_length = tail_length;
}
Cat::Cat(const Cat &cat): Pet(cat) {
  this->tail_length = cat.tail_length;
}

int Cat::get_tail_length() const {
  return tail_length;
}
void Cat::set_tail_length(const int tail_length) {
  this->tail_length = tail_length;
}

bool Cat::equal(const Pet &pet) const {
  if (dynamic_cast<const Cat *>(&pet)==0) {
    return false;
  }
  const Cat *cat = dynamic_cast<const Cat *>(&pet);
  return Pet::equal(pet) && this->tail_length==cat-
>tail_length;
}
void Cat::print(ostream &fout) const {
  Pet::print(fout);
  fout << "and his tail is " << this->tail_length << " cm";
}
```

pets./dog.h

```
#pragma once

#include <iostream>
#include <string>
using namespace std;

#include "cat.h"
#include "dog.h"
#include "parrot.h"
#include "pet.h"


class Dog: public Pet {
protected:
  double max_speed;
public:
  Dog();
```

```
  Dog(string name, int age, double max_speed);
  Dog(const Dog &dog);
  double get_max_speed() const;
  void set_max_speed(const double max_speed);
protected:
  bool equal(const Pet &pet) const override;
  void print(ostream &fout) const override;
};
```

pets/dog.cpp

```
#include "dog.h"


Dog::Dog(): Pet() {
  this->max_speed = 15;
}
Dog::Dog(string name, int age, double max_speed): Pet(name,
age) {
  this->max_speed = max_speed;
}
Dog::Dog(const Dog &dog): Pet(dog) {
  this->max_speed = dog.max_speed;
}

double Dog::get_max_speed() const {
  return max_speed;
}
void Dog::set_max_speed(const double max_speed) {
  this->max_speed = max_speed;
}

bool Dog::equal(const Pet &pet) const {
  if (dynamic_cast<const Dog *>(&pet)==0) {
    return false;
  }
  const Dog *dog = dynamic_cast<const Dog *>(&pet);
  return Pet::equal(pet) && this->max_speed==dog->max_speed;
}
void Dog::print(ostream &fout) const {
  Pet::print(fout);
  fout << "and run at " << this->max_speed << " m/s";
}
```

pets/parrot.h

```
#pragma once

#include <iostream>
```

```
#include <string>
using namespace std;

#include "pet.h"


class Parrot: public Pet {
protected:
  int num_of_colors;
public:
  Parrot();
  Parrot(string name, int age, int num_of_colors);
  Parrot(const Parrot &parrot);
  int get_num_of_colors() const;
  void set_num_of_colors(const int num_of_colors);
protected:
  bool equal(const Pet &pet) const override;
  void print(ostream &fout) const override;
};
```

pets/parrot.cpp

```
#include "parrot.h"


Parrot::Parrot(): Pet() {
  this->num_of_colors = 15;
}
Parrot::Parrot(string name, int age, int num_of_colors):
Pet(name, age) {
  this->num_of_colors = num_of_colors;
}
Parrot::Parrot(const Parrot &parrot): Pet(parrot) {
  this->num_of_colors = parrot.num_of_colors;
}

int Parrot::get_num_of_colors() const {
  return num_of_colors;
}
void Parrot::set_num_of_colors(const int num_of_colors) {
  this->num_of_colors = num_of_colors;
}

bool Parrot::equal(const Pet &pet) const {
  if (dynamic_cast<const Parrot *>(&pet)==0) {
    return false;
  }
```

```
  const Parrot *parrot = dynamic_cast<const Parrot *>(&pet);
  return Pet::equal(pet) && this->num_of_colors==parrot-
>num_of_colors;
}
void Parrot::print(ostream &fout) const {
  Pet::print(fout);
  fout << "and pictured in " << this->num_of_colors << "
colors";
}
```

main.cpp

```
#include <iostream>
using namespace std;

#include "pets/cat.h"
#include "pets/dog.h"
#include "pets/parrot.h"
#include "pets/pet.h"


int main(int argc, char *argv[]) {
  Pet **pets = new Pet *[6];
  pets[0] = new Cat("Alex", 6, 10);
  pets[1] = new Dog("Bob", 2, (short) 11);
  pets[2] = new Parrot("Cyan", 7, 10);
  pets[3] = new Cat("Dillan", 4, 15);
  pets[4] = new Dog("Elan", 9, (short) 12);
  pets[5] = new Parrot("Felix", 10, 3);
  for (int i=0; i<6; i++) {
    cout << *pets[i] << endl;
  }
  Pet *p = new Cat("Alex", 6, 12);
  for (int i=0; i<6; i++) {
    if (*p == *pets[i]) {
      cout << "Pet found!" << endl;
    }
  }
  return 0;
}
```

```
Alex is 6 years old and his tail is 10 cm
Bob is 2 years old and run at 11 m/s
Cyan is 7 years old and pictured in 10 colors
Dillan is 4 years old and his tail is 15 cm
Elan is 9 years old and run at 12 m/s
Felix is 10 years old and pictured in 3 colors
```

pets/petcomparable.h

```
#pragma once


class Pet;

class PetComparable {
public:
  virtual int compare_to(const Pet &pet)=0;
};
```

pets/petcomparator.h

```
#pragma once
#include "pet.h"


class PetComparator {
public:
  virtual int compare(const Pet &pet0, const Pet &pet1)=0;
};
```

pets/petcomparatorname.h

```
#pragma once

#include "petcomparator.h"


class PetComparatorName: public PetComparator {
public:
  int compare(const Pet &pet0, const Pet &pet1) override {
    return pet0.get_name().compare(pet1.get_name());
  }
};
```

pets/petcomparatorage.h

```
#pragma once

#include "petcomparator.h"


class PetComparatorAge: public PetComparator {
public:
  int compare(const Pet &pet0, const Pet &pet1) override {
    return pet0.get_age()-pet1.get_age();
  }
};
```

pets/pet.h

```
#pragma once
…
class Pet: public PetComparable {
…
  friend istream& operator >> (istream &fin, Pet &pet);
…
  virtual void input(istream &fin);
…
};
```

pets/pet.cpp

```
#include "pet.h"
…
istream& operator >> (istream &fin, Pet &pet) {
  pet.input(fin);
  return fin;
}
…
void Pet::input(istream &fin) {
  fin >> this->name >> this->age;
}
…
```

pets/cat.h

```
#pragma once
…
class Cat: public Pet {
…
  void input(istream &fin) override;
…
};
```

pets/cat.cpp

```
#include "cat.h"
…
void Cat::input(istream &fin) {
Pet::input(fin);
fin >> this->tail_length;
}
…
```

pets./dog.h

```
#pragma once
…
class Dog: public Pet {
…
```

```
  void input(istream &fin) override;
…
};
```

pets/dog.cpp

```
#include "dog.h"
…
void Dog::input(istream &fin) {
  Pet::input(fin);
  fin >> this->max_speed;
}
…
```

pets/parrot.h

```
#pragma once
…
class Parrot: public Pet {
…
  void input(istream &fin) override;
…
};
```

pets/parrot.cpp

```
#include "parrot.h"
…
void Parrot::input(istream &fin) {
  Pet::input(fin);
  fin >> this->num_of_colors;
}
…
```

list/petlist.h

```
#pragma once

#include <iostream>
#include <vector>
using namespace std;

#include "../pets/pet.h"
#include "../pets/petcomparator.h"


class PetList: vector<Pet *> {
public:
  void add(Pet *pet);
```

```
  bool is_contains(Pet *pet);
  void sort();
  void sort(PetComparator &pet_comparator);
  friend ostream& operator << (ostream &fout, const PetList
&petlist);
};
```

list/petlist.cpp

```
#include "petlist.h"


void PetList::add(Pet *pet) {
     this->push_back(pet);
}

bool PetList::is_contains(Pet *pet) {
     for (auto it = this->begin(); it != this->end(); it++) {
          if (*pet == *(*it)) {
               return true;
          }
     }
     return false;
}

void PetList::sort() {
  for (int i = 0; i < (*this).size(); i++) {
    for (int j = i+1; j < (*this).size(); j++) {
      if ( (*this)[i]->compare_to( *(*this)[j] ) > 0) {
        Pet* tmp = (*this)[i];
        (*this)[i] = (*this)[j];
        (*this)[j] = tmp;
      }
    }
  }
}

void PetList::sort(PetComparator &pet_comparator) {
  for (int i = 0; i < (*this).size(); i++) {
    for (int j = i+1; j < (*this).size(); j++) {
      if (pet_comparator.compare(*(*this)[i],*(*this)[j]) > 0)
{
        Pet* tmp = (*this)[i];
        (*this)[i] = (*this)[j];
        (*this)[j] = tmp;
      }
    }
  }
}
```

```
ostream& operator << (ostream &out, const PetList &list) {
          out << "This list contains next animals:" << endl;
          for (auto it = list.begin(); it != list.end(); it++)
{
               out << *(*it) << endl;
          }
          return out;
}
```

main.cpp

```
#include <iostream>
#include "list/petlist.h"
#include "pets/cat.h"
#include "pets/dog.h"
#include "pets/parrot.h"
#include "pets/pet.h"
#include "pets/petcomparatorname.h"
#include "pets/petcomparatorage.h"
using namespace std;


int main(int argc, char *argv[]) {
  PetList pl;

  pl.add(new Cat("Alex", 6, 10));
  pl.add(new Dog("Bob", 2, (short) 11));
  pl.add(new Parrot("Cyan", 7, 10));
  pl.add(new Cat("Dillan", 4, 15));
  pl.add(new Dog("Elan", 9, (short) 12));
  pl.add(new Parrot("Felix", 10, 3));

  Pet *p0 = new Cat(), *p1 = new Dog(), *p2 = new Parrot();
  cin >> *p0 >> *p1 >> *p2;
  pl.add(p0);
  pl.add(p1);
  pl.add(p2);

  cout << "Before sort:" << endl;
  cout << pl << endl;

  pl.sort(*(new PetComparatorName()));

  cout << "After sort:" << endl;
  cout << pl << endl;

  return 0;
}
```

```
>> Georg 5 14
>> Henry 7 5.5
>> Imp 3 0
Before sort:
This list contains next animals:
Alex is 6 years old and his tail is 10 cm
Bob is 2 years old and run at 11 m/s
Cyan is 7 years old and pictured in 10 colors
Dillan is 4 years old and his tail is 15 cm
Elan is 9 years old and run at 12 m/s
Felix is 10 years old and pictured in 3 colors
Georg is 5 years old and his tail is 14 cm
Henry is 7 years old and run at 5.5 m/s
Imp is 3 years old and pictured in 0 colors

After sort:
This list contains next animals:
Alex is 6 years old and his tail is 10 cm
Georg is 5 years old and his tail is 14 cm
Dillan is 4 years old and his tail is 15 cm
Henry is 7 years old and run at 5.5 m/s
Bob is 2 years old and run at 11 m/s
Elan is 9 years old and run at 12 m/s
Imp is 3 years old and pictured in 0 colors
Felix is 10 years old and pictured in 3 colors
Cyan is 7 years old and pictured in 10 colors
```

ЛАБОРАТОРНАЯ РАБОТА № 4

singlelinkedlist.h

```
#pragma once

#include <iostream>
using namespace std;


template<class T> class SingleLinkedList {
private:
  template<class U> class Node {
  public:
    U t;
    Node<U> *next;
    Node<U>(U t) {
      this->t = t;
      this->next = NULL;
```

```
    }
  };
  Node<T> *head;
  int size;
public:
  SingleLinkedList<T>() {
    head = NULL;
    size = 0;
  }
  bool add(T t) {
    if (head == NULL) {
      head = new Node<T>(t);
      size = 1;
    } else {
      Node<T> *curr = head;
      while (curr->next != NULL) {
        curr = curr->next;
      }
      curr->next = new Node<T>(t);
      size = size + 1;
    }
    return true;
  }
  bool add(int i, T t) {
    if (i == 0) {
      Node<T> *node = new Node<T>(t);
      node->next = head;
      head = node;
    } else if (i<size) {
      Node<T> *prev=head, *curr=new Node<T>(t), *next=head-
>next;
      for (int c=0; c<i-1; c++) {
        prev = prev->next;
        next = next->next;
      }
      prev->next = curr;
      curr->next = next;
    }
    size++;
    return true;
  }
  T get(int i) {
    if (i<0 || i>=size) {
      return NULL;
    }
    Node<T> *curr = head;
```

```
    for (int c=0; c<i; c++) {
      curr = curr->next;
    }
    return curr->t;
  }
  void clear() {
    while (head != NULL) {
      Node<T> *curr = head;
      head = head->next;
      curr->next = NULL;
      curr = NULL;
      size--;
    }
  }
  T remove(int i) {
    T t = NULL;
    if (i == 0) {
      Node<T> *curr = head;
      head = head->next;
      curr->next = NULL;
      t = curr->t;
    } else if (i < size) {
      Node<T> *prev=head, *curr=head->next, *next=head->next-
>next;
      for (int c=0; c<i-1; c++) {
        prev = prev->next;
        curr = curr->next;
        next = next->next;
      }
      prev->next = next;
      curr->next = NULL;
      t = curr->t;
    } else {
      return NULL;
    }
    size--;
    return t;
  }
  bool set(int i, T t) {
    if (i<0 || i>=size) {
      return false;
    }
    Node<T> *curr = head;
    for (int c=0; c<i; c++) {
      curr = curr->next;
    }
```

```
    curr->t = t;
    return true;
  }
  T& operator[] (int i) {
    return get(i);
  }
  friend ostream& operator << (ostream &fout, const
SingleLinkedList<T> &singlelinkedlist) {
    for (Node<T> *curr=singlelinkedlist.head; curr!=NULL;
curr=curr->next) {
      cout << curr->t << endl;
    }
    return fout;
  }
};
```

main.cpp

```
#include <iostream>
using namespace std;

#include "singlelinkedlist.h"


int main() {
  SingleLinkedList<int> a;
  a.add(0);
  a.add(2);
  a.add(4);
  a.add(1);
  a.add(3);
  cout << a << endl;
  cout << "# " << a.set(2, 10) << endl;
  cout << a << endl;
  a.clear();
}
```

```
0
2
4
1
3

# 1
0
2
10
1
3
```