Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерного проектирования

Кафедра проектирования информационно-компьютерных систем

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ И ПРОГРАММИРОВАНИЕ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальности 1-58 01 01 «Инженерно-психологическое обеспечение информационных технологий»; направлений специальности 1-40 05 01-09 «Информационные системы и технологии (в обеспечении промышленной безопасности)», 1-40 05 01-10 «Информационные системы и технологии (в бизнес-менеджменте)»

Минск БГУИР 2025

УДК 004.42(076.5)
ББК 32.973.3я73
О-29

А в т о р ы:

В. В. Хорошко, А. П. Горбач, Н. М. Бруй, А. Д. Ларькин,
Н. И. Куприянов, А. Ю. Писарчик, В. В. Листратенко,
П. И. Горох, А. С. Гриб, Д. В. Проходский

Р е ц е н з е н т ы:

кафедра «Программное обеспечение информационных систем и технологий»
Белорусского национального технического университета
(протокол № 9 от 19.04.2023);

директор ГЦ «Белмикроанализ»
ОАО «ИНТЕГРАЛ» – управляющая компания холдинга «ИНТЕГРАЛ»
кандидат физико-математических наук, доцент
А. Н. Петлицкий

Объектно-ориентированное проектирование и программирование.
О-29 Лабораторный практикум : пособие / В. В. Хорошко, А. П. Горбач,
Н. М. Бруй [и др.]. – Минск : БГУИР, 2025. – 116 с. : ил.
ISBN 978-985-543-760-5.

Приведено описание девяти лабораторных работ с рассмотрением теоретических сведений по представленным темам и даны примеры программ, поясняющие выполнение работ. Представлены варианты индивидуальных заданий и контрольные вопросы.

УДК 004.42(076.5)
ББК 32.973.3я73

ISBN 978-985-543-760-5

СОДЕЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1.
ИЕРАРХИЯ КЛАССОВ

1.1 Цель работы

Изучить основные принципы и механизмы ООП. Изучить структуру класса, принцип работы конструкторов и создания объектов класса. Научиться проектировать иерархии классов, осуществлять взаимодействие между классами.

1.2 Теоретическая часть

1.2.1 Введение в классы и объекты

Java является объектно-ориентированным языком, поэтому ключевыми понятиями являются «класс» и «объект», а любую программу на *Java* можно представить как набор взаимодействующих между собой объектов. Класс является шаблоном или чертежом, объект – это экземпляр класса.

Класс определяется с помощью ключевого слова *class*:

```
class Person {
}
```

В данном случае класс называется *Person*. Все содержимое класса, т. е. тело класса, помещается в фигурные скобки. Класс может содержать поля и методы: поля – это свойства объекта, описание объекта, «как объект выглядит»; методы – это действия объекта, поведение объекта.

Например, класс *Person*, который представляет человека, мог бы иметь следующее определение:

```
class Person {
    String name;
    int age;
    void displayInfo(){
        System.out.println("Name: " + name ", age: " + age);
    }
}
```

В классе *Person* определены два поля: *name* представляет имя человека, а *age* – его возраст и метод *displayInfo*, который выводит эти данные на консоль. Класс представляет новый тип, поэтому можно определить переменную класса с помощью ключевого слова *new* и конструктора по умолчанию:

```
Person p = new Person();
```

Кроме обычных методов, классы могут определять специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса и выполняют инициализацию объекта. Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор без параметров, т. е. конструктор по умолчанию.

Определенный выше класс *Person* не имеет никаких конструкторов, поэтому для него автоматически создается конструктор по умолчанию, который можно использовать для создания объекта *Person*.

Для создания объекта *Person* используется выражение *new Person()*. Оператор *new* выделяет память для объекта *Person*, затем вызывается конструктор по умолчанию, который не принимает никаких параметров. После выполнения данного выражения в памяти выделяется участок, где хранятся все данные объекта типа *Person*, переменная *p* хранит ссылку на объект.

Если необходимо, чтобы при создании объекта производилась какая-то логика, например, чтобы поля класса получали какие-то определенные значения, можно определить в классе свои конструкторы, например:

```
public class Program{
    public static void main(String[] args) {
        Person p1 = new Person();      // Вызов первого кон-
структора без параметров
        p1.displayInfo();
        Person p2 = new Person("Clara"); // Вызов второго
конструктора с одним параметром
        p2.displayInfo();
        Person p3 = new Person("Kate", 25); // Вызов третьего
конструктора с двумя параметрами
        p3.displayInfo();
    }
}
class Person {
    String name;
    int age;
    Person()
    {
        this.name = "name";
        this.age = 10;
    }
    Person(String name)
    {
        this.name = name;
       this age = 0;
    }
    Person(String name, int age)
    {
```

```
        this.name = name;
        this.age = age;
    }
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса.

1.2.2 Наследование

Одним из ключевых аспектов объектно-ориентированного программирования является наследование. С помощью наследования можно расширить функциональность уже имеющихся классов за счет добавления новой функциональности или изменения старой. Например, есть следующий класс *Person*, описывающий отдельного человека:

```
class Person {
    String name;
    public String getName(){ return name; }
    public Person(String name){
        this.name=name;
    }

    public void display() {
        System.out.println("Name: " + name);
    }
}
```

В дальнейшем может потребоваться добавить еще один класс, который описывает сотрудника предприятия, – класс *Employee*. Так как этот класс реализует тот же функционал, что и класс *Person*, поскольку сотрудник – это также и человек, то было бы рационально сделать класс *Employee* производным (дочерним, наследником, подклассом) от класса *Person*, который, в свою очередь, называется базовым классом, родителем, материнским, суперклассом:

```
class Employee extends Person{
    public Employee(String name){
        super(name);
    }
}
```

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*, поэтому класс *Employee* наследует все те же поля и методы, которые есть в классе *Person*.

В *Java* запрещено множественное наследование, т. е. дочерний класс может наследовать лишь один родительский класс.

Если в базовом классе определены конструкторы, то для инициализации объекта класса-наследника в конструкторе класса-наследника необходимо вызвать один из конструкторов класса-родителя с помощью ключевого слова *super*. Например, класс *Person* имеет конструктор, который принимает один параметр, поэтому в классе *Employee* в конструкторе нужно вызвать конструктор класса *Person* с помощью ключевого слова *super*. При вызове конструктора после слова *super* в скобках идет перечисление передаваемых аргументов. При этом вызов конструктора базового класса должен идти в самом начале в конструкторе производного класса, т. е. строчка с ключевым словом *super* идет первой. Таким образом, установка параметров, передаваемых с помощью ключевого слова *super*, делегируется конструктору базового класса.

Причем даже если производный класс никакой другой работы не производит в конструкторе, как в примере выше, все равно необходимо вызвать конструктор базового класса:

```
public class Program{
    public static void main(String[] args) {
        Person p = new Person("Clara");
        p.display();
        Employee p2 = new Employee("Kate");
        p2.display();
    }
}
class Person {
    String name;
    public String getName(){ return name; }
    public Person(String name){
        this.name=name;
    }
    public void display(){
        System.out.println("Name: " + name);
    }
}
class Employee extends Person{
    public  Employee(String name){
        super(name);
    }
}
```

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:

```
public class Program{
    public static void main(String[] args) {
        Employee p = new Employee("Clara", "Microsoft");
        p.display();  // Clara
        p.work();     // Clara works in Microsoft
    }
}
class Person {
    String name;
    public String getName(){ return name; }
    public Person(String name){
        this.name=name;
    }
    public void display(){
        System.out.println("Name: " + name);
    }
}
class Employee extends Person{
    String company;
    public Employee(String name, String company) {
        super(name);
        this.company=company;
    }
    public void work(){
        System.out.printf("%s works in %s \n", getName(),
company);
    }
}
```

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

Переопределение методов

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

```
public class Program{
          public static void main(String[] args) {
        Employee p = new Employee("Pam", "Microsoft");
        p.display();
    }
}
```

```
class Person {
    String name;
    public String getName(){ return name; }
    public Person(String name){
        this.name=name;
    }
    public void display(){
        System.out.println("Name: " + name);
    }
}
class Employee extends Person{
    String company;
    public Employee(String name, String company) {
        super(name);
        this.company=company;
    }
    @Override
    public void display(){
        System.out.printf("Name: %s \n", getName());
        System.out.printf("Works in %s \n", 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*.

Однако в данном случае мы видим, что часть метода *display()* в *Employee* повторяет действия из метода *display()* базового класса. Поэтому мы можем сократить класс *Employee*:

```
class Employee extends Person{
    String company;
    public Employee(String name, String company) {
        super(name);
        this.company=company;
    }
    @Override
    public void display(){
        super.display();
        System.out.printf("Works in %s \n", company);
    }}
```

С помощью ключевого слова *super* также можно обратиться к реализации методов базового класса.

1.2.3 Запрет наследования

В некоторых ситуациях применение наследования может быть нежелательным, тогда можно запретить наследование с помощью ключевого слова *final*. Например:

```
public final class Person {
}
```

Если бы класс *Person* был определен таким образом, то следующий код был бы ошибочным и не сработал, так как мы тем самым запретили наследование:

```
class Employee extends Person{
}
```

Кроме запрета наследования, можно также запретить переопределение отдельных методов. Например, в примере выше переопределен метод *display()*, запретим его переопределение:

```
public class Person {
    public final void display(){
        System.out.println("Имя: " + name);
    }
}
```

В этом случае класс *Employee* не сможет переопределить метод *display*.

1.2.4 Динамическая диспетчеризация методов

Наследование и возможность переопределения методов позволяют передать переменной суперкласса ссылку на объект подкласса:

```
Person sam = new Employee("Sam", "Oracle");
```

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 вызываться та версия метода, которая определена в классе *Employee*, а не в *Person*. Например:

```
public class Program{
    public static void main(String[] args) {
        Person p = new Person("Clara");
        p.display();
        Person p2 = new Employee("Pam", "Oracle");
        p2.display();
    }
}
class Person {
    String name;
    public String getName() { return name; }
    public Person(String name){
        this.name=name;
    }
    public void display(){
        System.out.printf("Person %s \n", name);
    }
}

class Employee extends Person{
    String company;
    public Employee(String name, String company) {
        super(name);
        this.company = company;
    }
    @Override
    public void display(){
             System.out.printf("Employee %s works in %s \n",
     super.getName(), company);
    }
}
```

Консольный вывод данной программы:

```
Person Clara
Employee Pam works in Oracle
```

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

1.3 Практическая часть

1 Изучить описание к лабораторной работе.

2 Получить у преподавателя вариант задания из приложения А.

3 В соответствии с полученной предметной областью реализовать иерархии классов (абстрактных классов, интерфейсов). Каждый класс должен иметь

отражающее смысл название и информативный состав. Для полей классов использовать инкапсуляцию. Определить конструкторы, методы *setTun(), getTun(), toString(), equals(), hashcode()*, а также другие методы с учетом назначения классов.

4 Предусмотреть консольное меню с использованием конструкции *switch-case*. Пункты меню должны отражать возможности программы. Необходимо применять ввод данных с использованием *Scanner*. Если вариант задания предполагает деление на роли, то необходимо реализовать возможность авторизации пользователей.

1.4 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Иллюстрация решения задачи.
5 Выводы.
6 Список использованных источников.

1.5 Контрольные вопросы

1 Дайте определение понятию «класс». Как можно описать поля и методы класса?

2 В чем разница между классом и объектом?

3 Какое ключевое слово используется при наследовании классов?

4 Как по-другому называются классы-наследники и классы, от которых наследуют?

5 В чем смысл наследования?

6 Разрешено ли в языке программирования *Java* наследование дочерним классом нескольких родительских классов?

7 Какое ключевое слово дает запрет на наследование?

8 В каком случае возможно переопределение методов?

9 Какая аннотация используется для обозначения того, что метод переопределен? Является ли она обязательной?

1.6 Список использованных источников

1 Шилдт, Г. Java. Полное руководство / Г. Шилдт. – 10-е изд. – М. : Диалектика, 2018. – 1488 с.

2 Java Documentation [Электронный ресурс]. – Режим доступа: https://docs.oracle.com/en/java.

3 Монахов, В. Язык программирования Java и среда NetBeans / В. Монахов. – 3-е изд. – СПб. : БХВ-Петербург, 2011. – 704 с.

ЛАБОРАТОРНАЯ РАБОТА № 2.
ГЕНЕРАЦИЯ И ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

2.1 Цель работы

Изучить возможности обработки исключительных ситуаций в программе и создания собственных классов исключений. Научиться разрабатывать программы с использованием механизмов обработки исключительных ситуаций.

2.2 Теоретическая часть

Введение

Обработка ошибок является важной задачей, поскольку пользователь может «отвернуться» от программы, если он потеряет результаты своей работы из-за возникшей ошибки или непредвиденных обстоятельств. В этом случае необходимо предоставить пользователю возможность завершить программу, сохранить результат и сообщить об обнаруженной ошибке.

Предположим, что во время выполнения программы на *Java* возникла ошибка, которая могла быть вызвана неверными данными в файле, ненадежным сетевым соединением, выходом за границы массива или попыткой использовать неназначенную ссылку. Пользователи обычно ожидают, что программа автоматически справится с проблемами и, если операция не может быть завершена успешно из-за ошибки, программа либо вернется в «безопасное» состояние, чтобы пользователь мог выполнить другие команды, либо даст пользователю возможность сохранить результаты и завершить работу.

Однако, чтобы добиться этого, необходимо передать данные обработчику исключений из того места, где возникла ошибка, что может быть сложно, так как фрагмент кода, где возникла ошибка, может быть далеко от кода, который может восстановить данные и сохранить результаты. Поэтому при разработке программы с обработкой исключений необходимо предусмотреть возможные ошибки и связанные с ними проблемы. Например, ошибки ввода, сбои оборудования, физические ограничения и ошибки программирования могут привести к ошибкам в программе.

2.2.1 Классификация исключений

В языке *Java* исключения представляются объектами, которые всегда являются экземплярами класса, производного от класса *Throwable*. Если не хватает стандартных классов исключений, можно создавать свои собственные. На рисунке 2.1 показана упрощенная иерархия наследования исключений в *Java*.

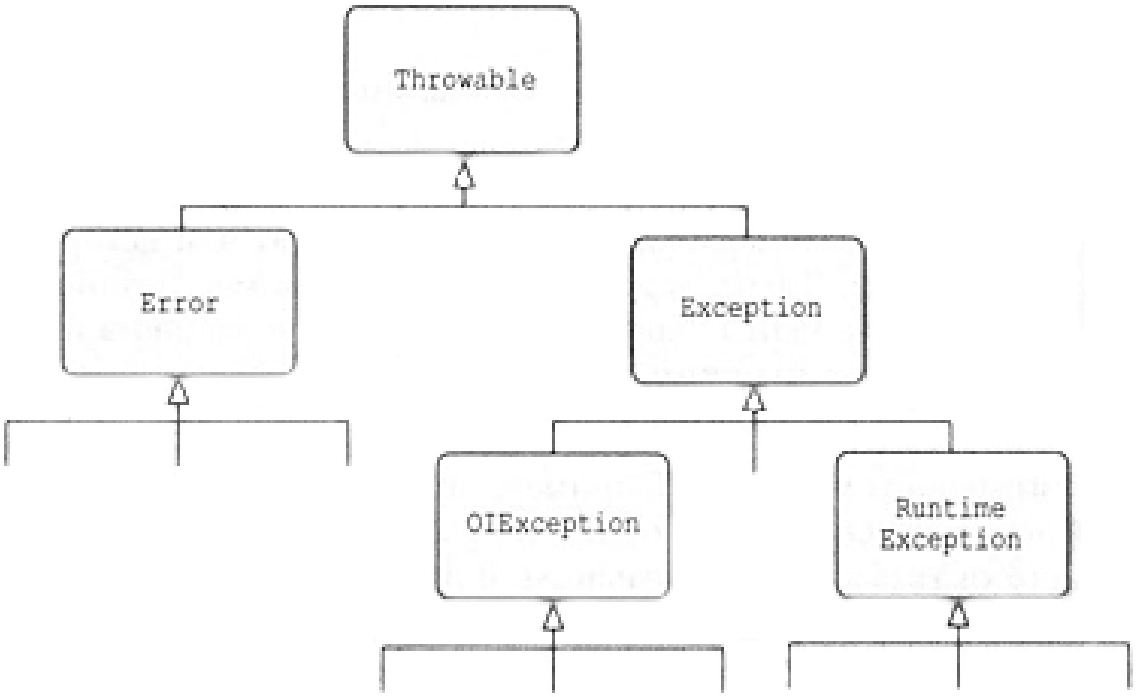

Рисунок 2.1 – Иерархия наследования исключений в *Java*

Иерархия наследования исключений делится на две ветви: *Error* и *Exception*, и хотя класс *Throwable* является общим предком для всех исключений, класс *Error* описывает внутренние ошибки и ситуации, связанные с нехваткой ресурсов в системе *Java*. Объекты этого класса нельзя создать самостоятельно. Когда в системе возникают внутренние ошибки, возможности разработчика ограничены. Ему остается только уведомить пользователя и попытаться аккуратно прервать выполнение программы, хотя такие ситуации встречаются нечасто.

При программировании на языке *Java* нужно уделить внимание иерархии класса *Exception*. Эта иерархия разделяется на две ветви: *RuntimeException* и все остальные исключения. Исключения типа *RuntimeException* возникают из-за ошибок программирования, таких как неверное приведение типов, выход за пределы массива или обращение к объекту по пустой ссылке *null*. Остальные исключения происходят из-за непредвиденных обстоятельств, например ошибок ввода-вывода или попыток открыть несуществующий файл. *RuntimeException*-исключения, как правило, являются ошибками программиста. Например, *ArrayIndexOutOfBoundsException* можно избежать, проверяя индексы массива, а *NullPointerException* – проверяя, содержит ли переменная пустое значение *null*, перед ее использованием.

В *Java* любое исключение, происходящее от класса *Error* или *RuntimeException*, считается непроверяемым. Все остальные исключения являются проверяемыми. Компилятор проверяет наличие соответствующих обработчиков для всех проверяемых исключений.

2.2.2 Объявление проверяемых исключений

Метод может бросать исключения, если возникает ситуация, с которой он не может справиться. Суть заключается в том, что метод должен сообщить компилятору о том, какие значения он может вернуть, а также какие ошибки могут возникнуть в ходе выполнения. Например, если метод предназначен для чтения данных из файла, то можно предположить, что такой файл не существует или он пуст. Следовательно, метод должен указать, что он может бросить исключение типа *IOException*.

Объявление о том, что метод может бросить исключение, делается в его заголовке. Приведенный ниже пример показывает объявление одного из конструкторов класса *FileInputStream* из стандартной библиотеки *Java*:

```
public FileInputStream(String name) throws
FileNotFoundException
```

Это означает, что данный конструктор создает объект типа *FileInputStream* на основе параметра *name* типа *String*, но также может бросить исключение типа *FileNotFoundException*, если что-то пойдет не так. Если это произойдет, система выполнения начнет поиск обработчика событий, который будет предназначен для обработки объектов типа *FileNotFoundException*.

Для метода, который вы создаете, необязательно объявлять все возможные исключения, которые могут возникнуть внутри метода. Чтобы определить, когда следует использовать оператор *throws*, важно знать, что исключение может возникнуть в следующих четырех случаях.

1 Вызывается метод, который генерирует проверяемое исключение, например конструктор класса *FileInputStream*.

2 Ошибка обнаружена и проверяемое исключение явным образом генерируется с помощью оператора *throw*.

3 Программист допустил ошибку, например в программе есть выражение *a*[-1] = 0, в результате чего генерируется непроверяемое исключение, например *ArrayIndexOutOfBoundsException*.

4 Возникает внутренняя ошибка в виртуальной машине или библиотеке исполняющей системы.

В первых двух случаях вам нужно сообщить пользователям метода о возможных исключениях. Это необходимо, потому что любой метод, который может генерировать исключение, является потенциально опасным местом в приложении. Если не предусмотреть обработку этого исключения в программе, то выполнение текущего потока может быть прервано.

Методы являются важными компонентами классов и могут генерировать исключения, которые могут привести к остановке приложения. Чтобы объявить, что метод может сгенерировать проверяемое исключение, нужно указать его в заголовке, как показано ниже:

```
class MyAnimation {
  public Image loadImage(String s) throws IOException {
    ...
  }
}
```

Если метод может сгенерировать несколько проверяемых исключений, все они должны быть перечислены в заголовке через запятую, например:

```
class MyAnimation {
  public Image loadImage(String s) throws
FileNotFoundException, EOFException {
    ...
  }}
```

Но исключения, которые происходят из-за внутренних ошибок, т. е. ошибки, производные от класса *Error*, объявлять не нужно. Эти исключения могут возникнуть в любом методе и не поддаются контролю программиста.

Точно так же необязательно объявлять непроверяемые исключения, производные от класса *RuntimeException*, как в примере ниже:

```
class MyAnimation{
  void drawImage(int i) throws ArrayIndexOutOfBoundsException
{
    // Не рекомендуется!
    ...
  }}
```

Ответственность за подобные ошибки лежит на разработчике приложения. Если существует риск выхода индекса за пределы массива, необходимо уделить больше внимания предотвращению этой ситуации, а не объявлять о возможности ее возникновения.

Таким образом, при объявлении метода необходимо учитывать все проверяемые исключения, которые могут возникнуть в ходе его работы. В то же время непроверяемые исключения, которые могут возникнуть из-за ошибок разработчика или класса *Error*, находятся вне контроля. Если в объявлении метода не указаны все проверяемые исключения, компилятор выдаст ошибку.

Вместо объявления исключений можно использовать обработку исключений при помощи блока *try-catch*. В этом случае исключение не будет сгенерировано, и объявление исключения в операторе *throws* необязательно.

Если метод объявляет возможность генерирования исключения определенного класса, то он может сгенерировать исключения и его подклассов. Например, если конструктор класса *FileInputStream* объявляет о возможности генерирования исключения типа *IOException*, то это может быть и простое исключение класса *IOException*, и объект одного из производных от него классов, в том числе класса *FileNotFoundException*.

2.2.3 Порядок генерирования исключений

Предположим, в приложении возникла критическая ситуация. К примеру, есть метод *readData()*, который считывает данные из файла с заголовком *Content-length: 1024*, но после чтения 733 символов происходит конец файла. Это некорректная ситуация, которая должна привести к генерации исключения. Теперь необходимо определить, какой тип исключения следует генерировать. В этом случае подойдет одно из исключений типа *IOException*. Документация описывает исключение *EOFException* как «сигнализирующее о неожиданном достижении конца файла во время чтения данных». Генерирование этого исключения может быть выполнено следующим образом:

```
throw new EOFException();
```

или так:

```
EOFException e = new EOFException();
throw e;
```

Код, который непосредственно генерирует это исключение, показан ниже:

```
String readData(Scanner in) throws EOFException {
   ...
   while (…) {
        if (!in.hasNext()) // Достигнут конец файла
        //(признак EOF)
        {
        if (n < len)
        throw new EOFException();
        }
   ...
   }
   return s;
}
```

В классе *EOFException* есть второй конструктор, который принимает строку в качестве параметра. Это можно использовать для более подробного описания исключения, как показано ниже:

```
String gripe = "Content-length: " + len + ", Received: " + n;
throw new EOFException(gripe);
```

Если существующий класс исключений подходит для конкретных целей приложения, то генерация исключения не составит труда. Для этого необходимо:

1 Найти подходящий класс.

2 Создать экземпляр этого класса.

3 Сгенерировать исключение.

После того как исключение было сгенерировано в методе, управление уже не будет возвращено в вызывающую часть программы. Это означает, что не нужно беспокоиться о возвращении значения по умолчанию или коде ошибки.

2.2.4 Создание классов исключений

Возможна ситуация в прикладной программе, когда необходимо создать свой класс исключения, так как ни один из стандартных классов не соответствует необходимым требованиям. Чтобы создать свой класс исключения, необходимо унаследовать его от класса *Exception* или его подклассов, таких как *IOException*. Класс исключения можно снабдить конструктором по умолчанию или конструктором, содержащим подробное сообщение, которое может быть полезным при отладке программы. Пример создания собственного класса исключения представлен ниже, где класс *FileFormatException* является подклассом *IOException*:

```
class FileFormatException extends IOException {
    public FileFormatException() {}
    public FileFormatException(String message) {
        super(message);
    }
}
```

Чтобы сгенерировать исключение собственного типа, необходимо вызвать оператор *throw* с новым экземпляром созданного класса исключения в методе, где возможно возникновение ошибки. Пример генерации исключения *FileFormatException*:

```
String readData(BufferedReader in) throws FileFormatException
{
    ...
    while () {
        if (ch == -1) {
// Достигнут конец файла (признак EOF)
            if (n < len)
                throw new FileFormatException();
        }
        ...
    }
    return s;}
```

2.2.5 Перехват одного исключения

Если исключение возникает и не перехватывается, выполнение программы прерывается и на консоль выводится сообщение о типе исключения и содержимом стека. Для программ с графическим интерфейсом исключения перехватываются и сообщение выводится на экран, после чего программа возвращается к обработке событий пользовательского интерфейса. В программировании графических интерфейсов не рекомендуется сворачивать консольное окно при отладке.

Для перехвата исключений используется блок *try/catch*. Простейшая форма блока выглядит следующим образом:

```
try {
// Код
}
catch (тип_исключения e) {
// Обработчик исключения данного типа
}
```

Если фрагмент кода в блоке *try* генерирует исключение указанного типа, то программа пропускает оставшуюся часть кода в блоке *try* и выполняет код обработчика в блоке *catch*. Если исключение не генерируется, блок *catch* пропускается. Если какой-то оператор в блоке *try* генерирует исключение другого типа, выполнение кода немедленно прекращается, и программа завершается, если исключение не перехватывается в вызывающей части программы.

Для иллюстрации принципов, описанных выше, рассмотрим обычный код для ввода данных:

```
public void read(String filename) {
    try {
        InputStream in = new FileInputStream(filename);
        int b;
        while ((b = in.read()) != -1) {
            processInputData(b);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Большая часть кода в блоке *try* выполняет простые операции: посимвольный ввод и обработка данных до конца файла. Метод *read()* может сгенерировать исключение типа *IOException*. В случае возникновения этого исключения весь оставшийся код в блоке *try* пропускается, программа переходит в блок *catch* и выводит трассировку стека. Для небольших программ это является разумным подходом обработки исключений. Но существуют и другие способы.

Часто бывает лучше не обрабатывать исключение внутри метода, а просто передать его вызывающей части программы. Если метод *read()* встретит ошибку ввода, то ответственность за ее обработку должна лежать на вызывающей части программы. При таком подходе достаточно указать, что метод *read()* может генерировать исключение типа *IOException* с помощью оператора *throws*, как показано в следующем коде:

```
public void read(String filename) throws IOException {
   InputStream in = new FileInputStream(filename);
   int b;
```

```
while ((b = in.read()) != -1){
        // Обработать введенные данные
}}
```

Важно помнить, что компилятор обращает внимание на операторы *throws*. При вызове метода, генерирующего проверяемое исключение, его нужно обработать в этом методе или передать обработку вызывающей части программы. В целом, стоит перехватывать только те исключения, которые можно обработать самостоятельно, а все остальные передавать дальше с помощью оператора *throws*, чтобы вызывающая часть программы могла обработать их.

Исключением из этого правила является создание метода, который переопределяет метод суперкласса и не генерирует никаких исключений (например, метод *paintComponent()* из класса *JComponent*), где каждое проверяемое исключение необходимо обрабатывать внутри этого метода. В таком случае нельзя добавлять новые операторы *throws*, отсутствующие в переопределяемом методе из суперкласса, в метод из подкласса.

2.2.6 Перехват нескольких исключений

В *try*-блоке можно обрабатывать несколько типов исключений по отдельности, каждый с помощью своего оператора *catch*.

Для каждого типа исключения следует предусмотреть свой блок оператора *catch* следующим образом:

```
try {
// Код, способный генерировать исключения
catch (FileNotFoundException e) {
// Чрезвычайные действия, если отсутствуют нужные файлы
}
catch (UnknownHostException e) {
// Чрезвычайные действия, если хосты неизвестны
}
catch (IOException e) {
// Чрезвычайные действия во всех остальных случаях появления
ошибок ввода-вывода
}
```

Если возникает ошибка, то объект исключения содержит информацию о ней, которую можно получить через вызов метода *getMessage()*. Также можно получить конкретный тип исключения, вызвав метод *getClass().getName()*.

В версии *Java* 7 и выше разнотипные исключения можно обрабатывать в одном операторе *catch*, если для них требуются одинаковые действия. Однако это следует делать только в тех случаях, когда перехватываемые исключения не являются подклассами друг для друга. При возникновении ошибок ввода-вывода, не связанных с отсутствующими файлами или неизвестными хостами, следует использовать отдельный оператор *catch*.

```
try {
   // Код, способный генерировать исключения
}
catch (FileNotFoundException | UnknownHostException e){
// Чрезвычайные действия, если нужные файлы отсутствуют или
неизвестны хосты
catch (IOException e){
// Чрезвычайные действия во всех остальных случаях появления
ошибок ввода-вывода
}
```

2.2.7 Блок оператора finally

Если в методе возникает исключение, то оставшаяся часть метода не будет выполнена. Если при этом метод использует локальные ресурсы, которые неизвестны внешнему коду, то их освободить не удастся. Хотя можно попытаться перехватить исключения и повторно их сгенерировать, это решение неэффективно, потому что придется освобождать ресурсы в двух местах: в обычном коде и в коде исключения. Однако блок оператора *finally* позволяет решить эту проблему.

Код в блоке оператора *finally* выполняется независимо от того, возникло исключение или нет. Так, в приведенном ниже примере кода графический контекст освобождается при любых условиях:

```
InputStream in = new FileinputStream(...);
try{
// 1 Код, способный генерировать исключения
// 2
}
catch (IOException e){
// 3 Вывести сообщение об ошибке
// 4
}
finally{
// 5
in.close();
}
// 6
```

Рассмотрим три возможные ситуации, в которых блок оператора *finally* выполняется в программе.

1 Если в коде не происходит никаких исключений, то сначала полностью выполняется блок оператора *try*, а затем блок оператора *finally*. Таким образом, выполнение программы проходит последовательно через точки 1, 2, 5 и 6.

2 Если в блоке оператора *try* генерируется исключение, которое перехватывается в блоке оператора *catch*, то программа выполняет блок оператора *try* до той точки, где возникло исключение, и затем переходит к блоку оператора *catch*. После этого выполняется блок оператора *finally*. Если в блоке оператора *catch* не

генерируются исключения, то выполнение программы продолжается после блока оператора *finally*, проходя через точки 1, 3, 4, 5 и 6. Если же в блоке оператора *catch* генерируется исключение, то управление передается вызывающей части программы и выполнение программы проходит только через точки 1, 3 и 5.

3 Если исключение не перехватывается в блоке оператора *catch*, то программа выполняет блок оператора *try* до той точки, где возникло исключение, и затем переходит к блоку оператора *finally*. После этого исключение передается обратно вызывающей части программы. Таким образом, выполнение программы проходит только через точки 1 и 5.

Оператор *finally* может использоваться независимо от оператора *catch*, например:

```
InputStream in = …;
try{
  // Код, способный генерировать исключения
}
finally{
in.close();
}
```

Метод *in.close()* из блока *finally* будет выполнен независимо от того, возникло ли исключение в блоке *try*. Если исключение будет сгенерировано, то оно будет обработано в соответствующем блоке *catch*.

2.3 Практическая часть

1 Изучить описание к лабораторной работе, ознакомиться с примерами реализации обработки исключений.

2 Исходя из варианта задания и исходного кода лабораторной работы № 1:

а) реализовать обработку исключительных ситуаций посредством блоков кода *try() {} catch{}*. Программа не должна экстренно прерываться при возникновении исключительных ситуаций;

б) добавить собственное исключение, унаследованное от класса *Exception*. Логическая составляющая исключения определяется исходя из ошибок, которые могут возникнуть в исходном коде лабораторной работы № 1.

2.4 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Иллюстрация решения задачи.
5 Выводы.
6 Список использованных источников.

2.5 Контрольные вопросы

1 Как реализовать обработку исключений?
2 Для чего нужно создавать собственные классы исключений?
3 В чем различия между *throw* и *throws?*
4 Как перехватить несколько исключений?
5 Для чего нужен оператор *finally*?

2.6 Список использованных источников

1 Хорстманн, К. С. Java. Библиотека профессионала. В 2 т. Т. 1 : Основы / К. С. Хорстманн. – 11-е изд. – СПб. : Диалектика, 2019. – 864 с.

2 Блох, Д. Java: эффективное программирование / Д. Блох. – 3-е изд. – СПб. : Диалектика, 2019. – 464 с.

3 Исключения в Java [Электронный ресурс]. – Режим доступа: https://javarush.com/groups/posts/isklyucheniya-java. – Дата доступа: 06.03.2023.

4 Исключения в Java. Часть I (try-catch-finally) [Электронный ресурс]. – Режим доступа: https://habr.com/ru/company/golovachcourses/blog/223821. – Дата доступа: 06.03.2023.

5 Обработка исключений [Электронный ресурс]. – Режим доступа: https://metanit.com/java/tutorial/4.1.php. – Дата доступа: 06.03.2023.

ЛАБОРАТОРНАЯ РАБОТА № 3.
КОЛЛЕКЦИИ. РАБОТА С КОЛЛЕКЦИЯМИ

3.1 Цель работы

Ознакомиться с коллекциями в *Java*. Научиться разрабатывать программы с использованием таких коллекций, как *ArrayList, TreeSet, HashSet, HashMap*.

3.2 Теоретическая часть

Коллекции в *Java* – это наборы объектов, которые могут быть использованы для хранения, поиска, сортировки и манипулирования данными. Они представляют собой абстрактные структуры данных, которые могут использоваться для организации и управления данными [1]. Они представляют собой более высокий уровень абстракции, чем простые массивы, и позволяют использовать их в более простой и гибкой форме.

Интерфейс *Collection* в *Java* предоставляет множество методов для работы с коллекциями данных. Он предоставляет стандартизированный подход к операциям над коллекциями, таким как добавление, удаление, поиск и итерирование. Данный интерфейс является первичным в определении функционала для коллекций и расширяет другой интерфейс, *Iterable,* который позволяет работать с сущностями с перечислениями. Можно отметить, что все объекты наследуемых коллекций, которые будут рассмотрены позже, также могут перебираться с помощью конструкции *for-each*. Среди методов интерфейса *Collection* можно выделить следующие базовые:

– *boolean isEmpty()* – возвращает *true,* если вызывающая коллекция пуста, или *false* при наличии элементов;

– *int Size()* – возвращает количество объектов вызывающей коллекции;

– *boolean add(E, E)* – добавляет *E* к вызывающей коллекции и возвращает *true*, если элемент добавлен;

– *boolean addAll(Collection <& extends E> C)* – добавляет все элементы в вызывающую коллекцию и возвращает *true* при добавлении всех элементов, а если ни один элемент не был добавлен или не все элементы были добавлены, возвращает *false*;

– *boolean contains(Object O)* – возвращает *true*, если объект *O* содержится в вызывающей коллекции, и *false* в противном случае;

– *boolean remove(Object O)* – возвращает *true,* при удачном удалении одного экземпляра *O* из вызывающей коллекции или *false* в противном случае;

– *boolean removeAll(Collection <&> C)* – удаляет все элементы из вызывающей коллекции и возвращает *true*, если по итогу коллекция изменяется, т. е. элементы удаляются, или *false* в противном случае;

– *boolean clear()* – удаляет все элементы из вызывающей коллекции;

– *Iterator <E> iterator()* – возвращает итератор для вызывающей коллекции;

– *boolean equals(Object O)* – возвращает *true* при равенстве вызывающей коллекции и объекта *O*.

В основе всех коллекций лежит применение того или иного интерфейса, который определяет базовый функционал. Среди этих интерфейсов можно выделить следующие [2]:

– ***Collection*** – базовый интерфейс для коллекций и других интерфейсов;

– ***Queue*** наследует интерфейс ***Collection*** и представляет функционал для структур данных в виде обычной очереди;

– ***Deque*** наследует интерфейс ***Queue*** и представляет функционал для реализации двунаправленных очередей;

– ***List*** наследует интерфейс ***Collection*** и представляет функциональность простых списков;

– ***Set*** расширяет интерфейс ***Collection*** и используется для хранения множеств уникальных объектов;

– ***SortedSet*** расширяет интерфейс ***Set*** для создания сортированных коллекций;

– ***NavigableSet*** расширяет интерфейс ***SortedSet*** для создания коллекций с возможностью поиска по соответствию;

– ***Map*** не наследуется от интерфейса ***Collection***, используется для хранения пар «ключ – значение». Он предоставляет методы для добавления, удаления и поиска элементов в карте. Также стоят упоминания-интерфейсы, расширяющие этот интерфейс, такие как *Sortedmap* и *NavigableMap*. Первый расширяет непосредственно *Map* и предоставляет методы для сортировки, второй наследует *SortedMap* и добавляет методы для навигации по данным.

Примечание – Интерфейсы и рассматриваемые далее классы располагаются в пакете *java.util*, поэтому перед применением следует подключить данный пакет.

Приведенные выше интерфейсы реализуются следующими абстрактными классами:

– ***AbstractCollection*** является базовым абстрактным классом для остальных коллекций, реализует интерфейс ***Collection****;*

– ***AbstractList*** расширяет класс ***AbstractCollection*** и реализует интерфейс ***List***, предназначен для создания коллекций в виде списков;

– ***AbstractSet*** расширяет класс ***AbstractCollection***, реализует интерфейс ***Set*** для создания коллекций в виде множеств;

– ***AbstractQueue*** расширяет класс ***AbstractCollection***, реализует интерфейс ***Queue***, предназначен для создания коллекций в виде очередей и стеков;

– ***AbstractSequentialList*** реализует класс ***AbstractList***, использует интерфейс ***List***, предназначен для создания связанных списков;

– ***AbstractMap*** реализует интерфейс ***Map***, предназначен для создания наборов по типу книги с объектами в виде пары «ключ – значение».

Рассмотрев интерфейсы и абстрактные классы, можно перейти непосредственно к коллекциям, из которых можно выделить следующие [3]:

– ***ArrayList*** применяет интерфейс ***List***, представляет простой список объектов динамического объема;

– ***LinkedList*** наследует класс ***AbstractSequentialList***, реализует интерфейсы ***List***, ***Queue*** и ***Dequeue*** и представляет собой связанный список, в котором совмещены функциональность списков и очереди;

– ***ArrayDeque*** реализует интерфейсы ***Queue*** и **Dequeue**, представляет двунаправленную очередь, где возможны вставка и удаление элементов в начале и конце коллекции;

– ***HashSe*t** реализует интерфейс ***Set*** и наследуется от ***AbstractSet***, представляет из себя набор объектов, хеш-множество, в котором каждый элемент имеет хеш-код – уникальный ключ, который однозначно определяет элементы таблицы;

– ***TreeSet*** реализует ***SortedSet***, представляет из себя набор отсортированных в порядке возрастания или в алфавитном порядке объектов в виде дерева;

– ***LinkedHashSet*** расширяет класс ***HashSet***, является связанным хеш-множеством, что представлено упорядочиванием объектов набора в порядке их вставки;

– ***PriorityQueue*** реализует интерфейс ***Queue***, является очередью приоритетов, в которой элементы хранятся в порядке приоритета, где наименьший элемент имеет наивысший приоритет;

– ***HashMap*** реализует интерфейс ***Map***, представляет собой структуру данных, хранящую пары «ключ – значение», что позволяет хранить одинаковые данные в качестве значений в этой паре;

– ***TreeMap*** реализует интерфейс ***NavigableMap***, используется для хранения данных в структурированном виде с возможностью навигации по данным, а порядок хранения данных задается компаратором.

Таким образом, почти для каждой задачи хранения объектов в том или ином виде существует большое разнообразие готовых классов, умелое пользование которыми поможет сохранить время разработчика и увеличить быстродействие программы.

Рассмотрим методы класса *ArrayList*, которых нет в интерфейсе *Collection*:

– *set(int index, E element)* заменяет элемент *index* на переданный *element*;

– *get(int index)* возвращает элемент с указанным индексом из вызывающей коллекции;

– *clone()* возвращает копию массива;

– *indexOf(Object o)* возвращает индекс первого вхождения элемента в списке. Возвращает -1 при отсутствии элемента в списке;

– *lastIndexOf(Object o)* возвращает индекс последнего элемента в списке;

– *toArray()* превращает объект в фиксированный массив с возможностью приведения списка в массив объектов определенного типа с помощью передачи параметра в метод;

– *ensureCapacity(int minCap)* увеличивает размер внутреннего массива.

Класс *ArrayList* позволяет работать с простыми списками объектов и удовлетворяет базовым потребностям в хранении данных. Но на этом функционал коллекций не ограничивается. Далее будут более подробно рассмотрены такие классы, как *LinkedList*, *HashSet*, *TreeSet*, *HashMap*, *TreeMap* и их методы. Но для начала необходимо рассмотреть итератор.

Итераторы используются в *Java* для обхода элементов в коллекциях. Они представляют собой объекты, которые позволяют последовательно получать доступ к элементам коллекции, причем без предварительного знания количества элементов или их порядка [4].

Для использования итератора в *Java* необходимо выполнить следующие шаги:

1) инициализировать итератор коллекции:

```
Iterator<T> iterator = collection.iterator();
```

2) применить методы итератора для обхода элементов коллекции:

```
while (iterator.hasNext()) {
    T element = iterator.next(); // Действия с элементом коллекции
}
```

Метод *hasNext()* возвращает *true*, если в коллекции еще есть элементы, и *false*, если достигнут конец коллекции. Метод *next()* возвращает следующий элемент коллекции и перемещает указатель на следующий элемент.

Пример использования итератора для списка строк:

```
List<String> list = new ArrayList<>();
list.add("apple");
list.add("banana");
list.add("orange");
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String fruit = iterator.next();
    System.out.println(fruit);
}
```

Этот код выведет на экран объекты в порядке их добавления.

Итераторы также позволяют удалять элементы из коллекции во время обхода. Для этого нужно вызвать метод *remove()* после метода *next()*:

```
iterator.remove();
```

Необходимый для удаления определенного элемента фрагмент кода представлен ниже:

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
        String fruit = iterator.next();
    if (fruit.equals("banana")) {
        iterator.remove();
    }
}
```

Далее будут рассмотрены различные коллекции.

LinkedList в *Java* является реализацией двусвязного списка [5]. Двусвязный список представляет собой коллекцию элементов, каждый из которых имеет связь с предыдущим и следующим элементами списка. Его удобно использовать в случае необходимости работы с элементами по всему объему. Хотя *LinkedList* имеет более низкую производительность, чем *ArrayList*, в большинстве случаев это незаметно, и дает больше удобства в навигации.

Пример создания списка:

```
LinkedList<String> list = new LinkedList<String>();
list.add("element 1");// Добавление элементов в список
list.addLast("element 3"); // Добавление элемента в конец списка
```

```
    list.addFirst("element 0"); // Добавление элемента в начало
списка
    String element = list.get(1); // Получение элемента из списка
    list.remove("element 1"); // Удаление элемента из списка
    int size = list.size(); // Определение размера списка
    boolean contains = list.contains("element 2"); // Проверка
наличия элемента
```

Перебор элементов списка:

```
for (String str : list) {
     System.out.println(str);
}
```

Далее будет рассмотрен *HashSet*. Он применяется для хранения уникальных элементов. При добавлении элемента в *HashSet* автоматически создается хеш-код (*hash code*) для каждого элемента и используется для определения адреса внутри [6]. Пример использования коллекции:

```
import java.util.HashSet;
public class Example {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<String>()
        // Добавление элементов в HashSet
        set.add("apple");
        set.add("banana");
        set.add("cherry");
        set.add("apple"); // Попытка добавить дубликат
        // Выведение размера HashSet
        System.out.println("Size of HashSet: " + set.size())
        // Проверка, содержит ли HashSet элемент "banana"
        if (set.contains("banana")) {
            System.out.println("HashSet contains banana");
        }
        // Удаление элемента "cherry"
        set.remove("cherry");
        // Выведение всех элементов HashSet
        System.out.println("Elements of HashSet: " + set);
    }
}
```

После выполнения программы выведется следующий результат:

```
Size of HashSet: 3
HashSet contains banana
Elements of HashSet: [banana, apple]
```

Как можно увидеть, *HashSet* не позволяет хранить дубликаты (два элемента с одинаковым значением), поэтому после добавления элемента *apple* второй раз *HashSet* оставил только один элемент с таким значением. Также обратите внимание, что порядок элементов в *HashSet* не является заданным, он зависит от внутреннего алгоритма хеширования.

TreeSet использует форму дерева для хранения упорядоченных элементов. Он реализует интерфейс *SortedSet*, который позволяет получать элементы в отсортированном порядке. *TreeSet* достаточно эффективен в работе с большими наборами данных, т. к. внутреннее представление *TreeSet* реализовано в виде красно-черного дерева [7].

Кроме того, *TreeSet* предоставляет множество методов для работы с элементами, таких как *add()*, *remove()*, *contains()*, *ceiling()*, *floor()* и многие другие. Подробнее о последних двух методах:

– *ceiling()* возвращает наименьший элемент в наборе, равный или больший, чем данный элемент, или *null* при отсутствии подобных элементов;

– *floor()* возвращает наибольший элемент в наборе, который меньше или равен указанному элементу, или *null* при остутствии подобных элементов.

Пример использования *TreeSet* для хранения строк в отсортированном порядке:

```
import java.util.TreeSet;
public class TreeSetExample {
   public static void main(String[] args) {
      TreeSet<String> treeSet = new TreeSet<>();

      treeSet.add("apple");
      treeSet.add("banana");
      treeSet.add("orange");

      System.out.println("Tree set elements: " + treeSet);
   }
}
```

Результат:

```
Tree set elements: [apple, banana, orange]
```

HashMap позволяет хранить пары «ключ – значение». Она используется для быстрого поиска и доступа к элементам по ключу.

Основные методы, которые предоставляются *HashMap*:

– *put(K key, V value)* добавляет в *HashMap* новую пару «ключ – значение»;

– *get(Object key)* возвращает значение, соответствующее ключу;

– *remove(Object key)* удаляет элемент, соответствующий ключу;

– *containsKey(Object key)* проверяет, содержится ли ключ в *HashMap*;

– *size()* возвращает количество элементов в *HashMap*.

Пример использования *HashMap*:

```
HashMap<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);
System.out.println(map.get("two")); // Выведет 2
```

Здесь создается *HashMap*, которая будет хранить пары «ключ – значение» типа *String-Integer*. Затем в нее добавляются три элемента. Для доступа к значению по ключу *two* используется метод *get()*.

Treemap представляет собой структуру данных, хранящую пары «ключ – значение». Она представляет отображение, где каждому ключу соответствует некоторое значение.

В отличие от *HashMap TreeMap* содержит элементы, упорядоченные по значению и расположенные в дереве. Значения в *TreeMap* могут быть отсортированы по возрастанию или убыванию ключей. Как и *TreeSet*, дерево является бинарным красно-черным.

Пример использования *TreeMap* в *Java*:

```
import java.util.TreeMap;
public class TreeMapExample {
   public static void main(String[] args) {
      TreeMap<Integer, String> treeMap = new TreeMap<Integer,
String>();
      treeMap.put(1, "One");
      treeMap.put(2, "Two");
      treeMap.put(4, "Four");
      treeMap.put(3, "Three");
      // Вывод элементов в TreeMap
      System.out.println("TreeMap: " + treeMap);
      // Получение значения по ключу
      System.out.println("Value for key 2: " + treeMap.get(2));
```

```
            // Удаление элемента по ключу
            treeMap.remove(4);
            // Проверка наличия ключа
            System.out.println("Contains key 4:" +treeMap.contains-
Key(4));
            // Вывод размера TreeMap
            System.out.println("Size of TreeMap: " + treeMap.size());
            // Очистка TreeMap
            treeMap.clear();
        }
    }
```

Вывод программы:

```
TreeMap: {1=One, 2=Two, 3=Three, 4=Four}
Value for key 2: Two
Contains key 4: false
Size of TreeMap: 3
```

Также стоит упомянуть метод *hashCode()*, который возвращает хеш-код, используемый для быстрого доступа к элементам в больших коллекциях. Пример кода для перебора коллекции приведен ниже:

```
public boolean containsByHashCode(Set<?> set, int hashCode) {
    for (Object obj : set) {
        if (obj.hashCode() == hashCode) {
            return true;
        }
    }
    return false;
}
```

Метод проходит по всем элементам коллекции *set*, вызывает для каждого элемента метод *hashCode()* и сравнивает полученный результат со значением параметра *hashCode*. Если хеш-коды совпадают, то метод возвращает *true*. Если такой элемент не найден, то метод возвращает *false*. Важно отметить, что поиск элемента по хеш-коду может быть неэффективным и занимать много времени, например при большом количестве элементов с одинаковыми хеш-кодами (при так называемой коллизии). Поэтому перед использованием данного метода следует внимательно оценить его эффективность в конкретной ситуации.

3.3 Практическая часть

1 Изучить описание лабораторной работы, ознакомиться с примерами реализации представленных коллекций.

2 Преобразовать методику хранения данных программы, написанной для первой лабораторной работы, с использованием коллекции *ArrayList, TreeSet, HashSet* или *HashMap.*

3 Добавить сортировку данных по ключевому параметру.

3.4 Содержание отчета

1 Титульный лист.

2 Цель работы.

3 Краткие теоретические сведения.

4 Иллюстрация решения задачи.

5 Выводы.

6 Список использованных источников.

3.5 Контрольные вопросы

1 Перечислите основные методы интерфейса *Collection.*

2 Перечислите и опишите основные методы коллекции *ArrayList*.

3 В чем состоят отличия коллекций *HashMap* и *TreeMap*?

4 Для чего используется итератор при работе с коллекциями?

5 В чем отличие обычного списка от двусвязного?

6 Перечислите основные методы *HashMap*.

7 Для чего может быть использован хеш-код?

3.6 Список использованных источников

1 Эккель, Б. Философия Java / Б. Эккель. – 4-е изд. – СПб. : Питер, 2015. – 1168 с.

2 Типы коллекций. Интерфейс Collection [Электронный ресурс]. – Режим доступа: https://metanit.com/java/tutorial/5.1.php.

3 Виды коллекций [Электронный ресурс]. – Режим доступа: https://javarush.com/groups/posts/1937-klass-collections.

4 Java: Iterator и ListIterator [Электронный ресурс]. – Режим доступа: https://proglang.su/java/iterator-and-listiterator.

5 Двусвязный список в Java [Электронный ресурс]. – Режим доступа: https://www.delftstack.com/ru/howto/java/doubly-linked-list-in-java.

6 Множества: Set, HashSet, LinkedHashSet, TreeSet [Электронный ресурс]. – Режим доступа: https://developer.alexanderklimov.ru/android/java/set.php.

7 Красно-черные деревья и реализация множеств на их основе [Электронный ресурс]. – Режим доступа: http://mech.math.msu.su/~vvb/2course/Borisenko/lecTree.html.

8 Реализация HashMap в Java Collections Framework [Электронный ресурс]. – Режим доступа: https://hr-vector.com/java/hashmap.

9 Руководство по Java Core. Коллекции. TreeMap [Электронный ресурс]. – Режим доступа: https://hr-vector.com/java/treemap.

ЛАБОРАТОРНАЯ РАБОТА № 4.
ПОТОКИ ВВОДА-ВЫВОДА. СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ

4.1 Цель работы

Изучить механизмы передачи данных с использованием потоков ввода-вывода информации. Научиться разрабатывать программы с использованием механизмов сериализации и десериализации.

4.2 Теоретическая часть

4.2.1 Потоки ввода-вывода

Одна из главных задач, ставящихся при программировании, – осуществление возможности передачи данных между различными файлами и пакетами. В языке программирования *Java* подобная возможность реализуется с помощью таких инструментов, как потоки ввода-вывода информации.

Поток (*stream*) ввода-вывода информации – программная среда (объект), которая позволяет считывать (вводить) и записывать (выводить) поступающую информацию. Если рассматривать потоки ввода-вывода на примере из повседневной жизни, то представим, что при подаче горячей воды из водопроводного крана (потока ввода-вывода информации) осуществляется запись абстрактной информации, а при подаче холодной – ее считывание. По сути кран – это инструмент, посредством которого мы можем подключаться к водопроводной системе всего дома и осуществлять подачу воды.

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-вывод в консоли мы будем решать в *Java* с помощью потоков.

В языке программирования *Java* инструменты для работы с потоками ввода-вывода информации находятся пакете *java.io* (*input-output*).

Пакет *java.io* содержит почти каждый класс, который может потребоваться для совершения ввода и вывода в *Java*. Все названные потоки представлены потоком ввода и адресом вывода. Поток в пакете *java.io* осуществляет поддержку различных данных, таких как примитивы, объекты, локализованные символы и т. д.

В основе всех классов для работы с потоками ввода-вывода находятся два абстрактных класса: *InputStream* (для взаимодействия с потоками ввода) и *OutputStream* (для взаимодействия с потоками вывода).

Для облегчения работы с потоками символьных данных в *java.io* были добавлены абстрактные классы *Reader* (для чтения потоков символов) и *Writer* (для записи потоков символов).

Иерархия всех классов, предназначенных для работы с потоками ввода-вывода, представлена в таблице 4.1.

Таблица 4.1. – Иерархия классов для работы с потоками ввода-вывода

Абстрактный класс	***InputStream***	***OutputStream***	***Reader***	***Writer***
Наследники абстрактного класса	*FileInputStream,* *BufferedInputStream,* *ByteArrayInputStream,* *FilterInputStream,* *DataInputStream,* *ObjectInputStream*	*FileOutputStream,* *BufferedOutputStream,* *ByteArrayOutputStream,* *FilterOutputStream,* *DataOutputStream,* *ObjectOutputStream*	*FileReader,* *BufferedReader,* *CharArrayReader,* *FilterReader*	*FileWriter,* *BufferedWriter,* *CharArrayWriter,* *FilterWriter*

Класс *InputStream* является базовым для всех классов, управляющих байтовыми потоками ввода. Основные методы данного класса рассмотрены в таблице 4.2.

Таблица 4.2 – Основные методы класса *InputStream*

Тип возвращаемых данных	**Синтаксическое описание метода**	**Задача, выполняемая методом**
int	*available()*	Метод возвращает количество байтов, которые доступны для чтения в потоке ввода
void	*close()*	Метод закрывает поток ввода
int	*read()*	Метод считывает следующий байт данных из потока ввода. При отсутствии в потоке байтов, доступных для чтения, метод возвращает -1
int	*read(byte[] buffer)*	Метод считывает байты из потока ввода 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)*	Метод считывает количество байтов, равное заданному значению *length*, из потока ввода в массив *buffer*. При этом считанные байты помещаются в массиве, начиная со смещения *offset*, т. е. с элемента *buffer[offset]*. Метод возвращает число успешно прочитанных байтов
long	*skip(long number)*	Метод пропускает в потоке *number* байт при считывании

Класс *OutputStream* является базовым для всех классов, которые работают с байтовыми потоками вывода. Основные методы класса представлены в таблице 4.3.

Таблица 4.3 – Основные методы класса *OutputStream*

Тип возвращаемых данных	**Синтаксическое описание метода**	**Задача, выполняемая методом**
void	*close()*	Метод закрывает поток вывода
void	*flush()*	Метод очищает буфер, сохраняя все его содержимое
void	*write(int b)*	Метод записывает в поток вывода один байт, представленный целочисленным параметром *b*
void	*write(byte[] buffer)*	Метод записывает в поток вывода массив байтов *buffer*
void	*write(byte[] buffer, int offset, int length)*	Метод записывает в поток вывода количество байтов *length* из массива *buffer* начиная со смещения *offset*, т. е. с элемента *buffer[offset]*

Класс *Reader* – абстрактный класс, определяющий символьный потоковый ввод. В случае ошибок все методы класса передают исключение *IOException*. Абстрактный класс *Reader* необходим для чтения текстовой информации.

Основные методы класса *Reader* представлены в таблице 4.4.

Таблица 4.4 – Основные методы класса *Reader*

Синтаксическое описание метода	**Задача, выполняемая методом**
abstract void close()	Метод закрывает поток ввода
int read()	Метод возвращает следующий символ в потоке в виде целого числа. Если таких символов нет, и достигнут конец файла, то возвращается число -1
int read(char[] buffer)	Метод считывает из потока символы в массив *buffer*, количество которых равно длине массива *buffer*. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1
int read(CharBuffer buffer)	Метод считывает из потока символы в объект *CharBuffer*. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1
abstract int read(char[] buffer, int offset, int count)	Метод считывает в массив *buffer* начиная со смещения *offset* символы из потока, количество которых равно *count*
long skip(long count)	Метод пропускает *count* символов и возвращает число успешно пропущенных символов

Класс *Writer* – абстрактный класс, определяющий символьный потоковый вывод. В случае ошибок все методы класса передают исключение *IOException*.

Класс *Writer* предназначен для потоков вывода, работающих с данными в виде символов. Его основные методы представлены в таблице 4.5.

Таблица 4.5 – Основные методы класса *Writer*

Синтаксическое описание метода	**Задача, выполняемая методом**
Writer append(char c)	Метод добавляет в конец выходного потока символ *c*. Возвращает объект класса *Writer*
Writer append(CharSequence chars)	Метод помещает последовательность символов *chars* в конец выходного потока. Возвращает объект *Writer*
abstract void close()	Метод закрывает поток вывода
abstract void flush()	Метод очищает буферы потока вывода
void write(int c)	Метод записывает в поток один символ, который имеет целочисленное представление
void write(char[] buffer)	Метод записывает в поток массив символов
absract void write(char[] buffer, int off, int len)	Метод записывает в поток из массива *buffer* количество *len* символов начиная с индекса *off*
void write(String str)	Метод записывает в поток строку *str*
void write(String str, int off, int len)	Метод записывает в поток из строки количество *len* символов начиная с индекса *off*

4.2.2 Сериализация

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

Сериализации подлежат только объекты, реализующие интерфейс *Serializable*. Для сериализации объектов в поток используется класс *ObjectOutputStream*. Основные методы для записи данных в *ObjectOutputStream* представлены в таблице 4.6.

Таблица 4.6 – Основные методы класса *ObjectOutputStream*

Синтаксическое описание метода	**Задача, выполняемая методом**
void close()	Метод закрывает поток
void flush()	Метод очищает буфер и сбрасывает его содержимое в выходной поток
void write(byte[] buf)	Метод записывает в поток массив байтов
void write(int val)	Метод записывает в поток один младший байт из *val*
void writeBoolean(boolean val)	Метод записывает в поток значение *boolean*
void writeByte(int val)	Метод записывает в поток один младший байт из *val*
void writeChar(int val)	Метод записывает в поток значение типа *char*, представленное целочисленным значением
void writeDouble(double val)	Метод записывает в поток значение типа данных *double*
void writeFloat(float val)	Метод записывает в поток значение типа данных *float*
void writeInt(int val)	Метод записывает целочисленное значение типа данных *int*
void writeLong(long val)	Метод записывает значение типа данных *long*
void writeShort(int val)	Метод записывает значение типа данных *short*
void writeUTF(String str)	Метод записывает в поток строку в кодировке *UTF*-8
void writeObject(Object obj)	Метод записывает в поток отдельный объект

Методы из таблицы 4.6 охватывают весь спектр данных, доступных для сериализации.

Сохранение в файл объекта класса *Student*:

```
import java.io.*;

     public class MainProg {
         public static void main(String[] args) {

             try(ObjectOutputStream objOutStr = new Objec-
tOutputStream(new FileOutputStream("Student.dat")))
             {
                 Student stud = new Student("Paul", 711801,
9.6);
                 objOutStr.writeObject(stud);
             }
             catch(Exception ex){

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

```
class Student implements Serializable{

        private String name;
        private int group;
        private double rating;

        Student(String n, int g, double r){

            name=n;
            group=g;
            rating=r;
        }
        String getName() {return name;}
        int getGroup(){ return group;}
        double getRating(){return rating;}
    }
```

Класс *ObjectInputStream* ответственен за чтение ранее сериализованных данных из потока. В конструкторе он принимает ссылку на поток ввода.

Функционал *ObjectInputStream* сосредоточен на чтении различных типов данных. Рассмотрим основные методы этого класса в таблице 4.7.

Таблица 4.7 – Основные методы класса *ObjectInputStream*

Синтаксическое описание метода	**Задача, выполняемая методом**
void close()	Метод закрывает поток
int skipBytes(int len)	Метод пропускает при чтении несколько байтов, количество которых равно *len*
int available()	Метод возвращает количество байтов, доступных для чтения
int read()	Метод считывает из потока один байт и возвращает его целочисленное представление
boolean readBoolean()	Метод считывает из потока одно значение *boolean*
byte readByte()	Метод считывает из потока один байт
char readChar()	Метод считывает из потока один символ типа данных *char*
double readDouble()	Метод считывает значение типа данных *double*
float readFloat()	Метод считывает из потока значение типа данных *float*
int readInt()	Метод считывает целочисленное значение типа данных *int*
long readLong()	Метод считывает значение типа данных *long*
short readShort()	Метод считывает значение типа данных *short*
String readUTF()	Метод считывает строку в кодировке *UTF*-8
Object readObject()	Метод считывает из потока объект

Для оптимизации операций ввода-вывода используются буферизуемые потоки *BufferedInputStream* и *BufferedOutputStream*. Эти потоки добавляют к стандартным специальный буфер в памяти, с помощью которого повышается производительность при чтении и записи потоков.

Извлечение объекта класса *Student* из файла:

```
import java.io.*;

public class MainProg {

    public static void main(String[] args) {

        try(ObjectInputStream objInpStr =
new ObjectInputStream(new FileInputStream("Student.dat")))
        {
            Student stud =(Student) objInpStr.readObject();
            System.out.printf("Name: %s \t Group: %d \n",
stud.getName(), stud.getGroup());
        }
        catch(Exception ex){

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

4.3 Практическая часть

1 Изучить описание к лабораторной работе.

2 Организовать чтение / запись в файл основных типов данных выстроенной иерархии классов из лабораторной работы № 1. Чтение и запись в файл осуществить через механизмы сериализации/десериализации объектов. Для записи и чтения объектов в поток / из потока необходимо использовать классы *ObjectOutputStream* и *ObjectInputStream* соответственно.

4.4 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Иллюстрация решения задачи.
5 Выводы.
6 Список использованных источников.

4.5 Контрольные вопросы

1 Что такое поток данных? Какие потоки данных существуют в *Java*?
2 Какие классы байтовых потоков ввода-вывода существуют?
3 Какие классы символьных потоков ввода-вывода существуют?

4 Для чего используются классы *BufferedInputStream*, *BufferedOutputStream*, *BufferedReader*, *BufferedWriter*?

5 Для чего используются классы *FilterInputStream*, *FilterOutputStream*, *FilterReader*, *FilterWriter*?

6 Для чего применяются классы *InputStreamReader* и *OutputStreamWriter*?

7 Что такое сериализация, для чего нужна, когда применяется? Правила сериализации объектов.

8 Что такое десериализация? Правила десериализации объектов.

9 Будет ли повторно сериализоваться уже сериализованный объект?

10 Как сериализовать и десериализовать объект? Какие классы и интерфейсы для этого необходимо использовать?

4.6 Список использованных источников

1 Хорстманн, К. С. Java. Библиотека профессионала. В 2 т. Т. 1 : Основы / К. С. Хорстманн. – 11-е изд. – СПб. : Диалектика, 2019. – 864 с.

2 Сайт о программировании Metanit [Электронный ресурс]. – Режим доступа: https://metanit.com. – Дата доступа: 14.03.2023.

3 Блинов, И. Н. Java from EPAM : учеб.-метод. пособие / И. Н. Блинов, В. С. Романчик. – 2-е изд. – Минск : Четыре четверти, 2021. – 560 с.

ЛАБОРАТОРНАЯ РАБОТА № 5.
ЛЯМБДА-ВЫРАЖЕНИЯ

5.1 Цель работы

Ознакомиться с теоретическими сведениями, изучить особенности применения лямбда-выражений. Освоить создание и применение функциональных интерфейсов, рассмотреть разновидности встроенных функциональных интерфейсов. Применить использование лямбда-выражений на практике.

5.2 Теоретическая часть

Лямбда-выражения появились в версии *JDK* 8 для усовершенствования языка *Java*. Введение лямбда-выражений позволило повысить выразительность языка *Java*, упрощая реализацию некоторых общепринятых конструкций. Также введение выражений позволило расширить возможности библиотек программного интерфейса (*API*): упрощение параллельной обработки, улучшение работы с потоками ввода-вывода в программном интерфейсе.

Сама по себе лямбда является набором инструкций, которые можно выделить в отдельную переменную, после чего многократно вызывать в различных местах программы. Главной частью лямбда-выражения является лямбда-оператор (–>). Использование этого оператора разделяет лямбда-выражение на две части: левая, содержащая параметры лямбда выражения, и правая, содержащая тело лямбда-выражения. Тело (код) лямбда-выражения может формироваться одним из двух способов:

– содержать одиночное выражение, лямбда выражение будет называться одиночным;

– содержать фигурные скобки: {}. Данный случай используется при необходимости указания нескольких операторов. Такие лямбда-выражения называются блочными.

Пример создания лямбда-выражения представлен ниже:

```
(Параметр1, Параметр2) -> {
     // Реализация метода
     // …
     // …
     return результат;
}
```

Также возможна реализация, когда лямбда-выражение вовсе не принимает никаких параметров. В этом случае общая форма создания блочного лямбда-выражения выглядит так:

```
( ) -> {
     // Реализация метода
     // …
 }
```

Однако стоит учитывать, что лямбда-выражение не выполняется самостоятельно, а представляет из себя исключительно реализацию метода, который был предопределен в функциональном интерфейсе. При этом одним из основных условий является то, что функциональный интерфейс будет состоять только из одного метода без реализации.

Необходимо более подробно рассмотреть понятие и особенности создания функционального интерфейса. Функциональный интерфейс, необходимый для работы лямбда-выражения, может содержать один и только один абстрактный метод. Этот единственный метод определяет назначение интерфейса. Объявление функционального интерфейса может выглядеть следующим образом:

```
interface Имя_Интерфейса {
     Тип_Возвращаемого_Объекта (список_параметров)
}
```

С появлением лямбда-выражений были добавлены так называемые встроенные функциональные интерфейсы, которые довольно часто применяются в *Stream API* (инструментарий, созданный для упрощения работы с данными). Встроенные функциональные интерфейсы существуют следующих видов:

– функциональный интерфейс *Predicate <T>*, соблюдающий условие. При соблюдении возвращает значение *true*:

```
public interface Predicate <T> {
    bool Название_метода (T Параметр);
}
```

– функциональный интерфейс *BinaryOperator <T>*, выполняющий бинарную операцию над двумя объектами типа *T*, результат возвращается в виде объекта *T*:

```
public interface BinaryOperator <T> {
    T apply (T Параметр1, T Параметр2);
}
```

– функциональный интерфейс *UnaryOperator <T>*, выполняющий операцию над объектом типа *T*, результат возвращается в виде объекта *T*:

```
public interface UnaryOperator <T> {

    T apply (T Параметр);
}
```

– функциональный интерфейс *Function <T, R>*, выполняющий переход от объекта типа *T* к объекту типа *R*:

```
public interface Function <T, R> {
    R apply (T Параметр);}
```

– функциональный интерфейс *Consumer <T>*, выполняющий действие над объектом *T*, не возвращая никакого значения:

```
public interface Consumer <T> {
    void accept(T Параметр);
}
```

– функциональный интерфейс *Supplier <T>*, не принимающий никаких аргументов, но возвращающий значение в виде объекта *T*:

```
public interface Supplier <T> {
    T get();
}
```

Вышеперечисленные встроенные функциональные интерфейсы используются в различных ситуациях и в различных *API*.

Как ранее было сказано, одним из преимуществ использования лямбда-выражений является отложенное выполнение. То есть определение в одном месте программы лямбда-выражения, а затем его вызов при необходимости неопределенное количество раз в различных частях программы. Отложенное выполнение может потребоваться, к примеру, в следующих случаях:

– выполнение кода отдельным потоком;

– 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 действительно необходим.

Резюмируем, определив, какая именно последовательность шагов используется при построении лямбда-выражений. Для того, чтобы в программе использовать лямбда-выражения, необходимо выполнить следующие шаги:

1 Объявить функциональный интерфейс, содержащий только один абстрактный метод. Сигнатура параметров метода и тип возвращаемого результата должны соответствовать решаемой задаче.

2 Объявить ссылку на функциональный интерфейс в методе, где нужно использовать лямбда-выражение.

3 Сформировать код лямбда-выражения согласно синтаксису и присвоить этот код ссылке на функциональный интерфейс. Количество параметров в лямбда-выражении должно совпадать с количеством параметров, которые объявлены в методе функционального интерфейса.

4 Используя ссылку, вызвать метод.

5.3 Практическая часть

1 Изучить описание лабораторной работы, ознакомиться с примерами реализации представленных лямбда-выражений.

2 В соответствии с выбранной в лабораторной работе № 1 предметной областью и по согласованию с преподавателем реализовать применение лямбда-выражений для преобразования методов / вызова методов.

5.4 Содержание отчета

1 Титульный лист.

2 Цель работы.

3 Краткие теоретические сведения.

4 Иллюстрация решения задачи.

5 Выводы.

6 Список использованных источников.

5.5 Контрольные вопросы

1 Что такое лямбда-выражение?

2 Какие условия необходимо выполнить для создания и использования лямбда-выражения?

3 Какие лямбда-выражения называют блочными?

4 Как передать параметры в лямбда-выражения? Можно ли передать локальные переменные?

5 Что такое функциональный интерфейс? Что прописывается в функциональном интерфейсе?

5.6 Список использованных источников

1 Блинов, И. Н. Java from EPAM : учеб.-метод. пособие / И. Н. Блинов, В. С. Романчик. – 2-е изд. – Минск : Четыре четверти, 2021. – 560 с.

2 Java. Лямбда-выражения. Основные понятия. Функциональный интерфейс. Обобщенные функциональные интерфейсы и лямбда-выражения. Примеры // BestProg [Электронный русурс]. – Режим доступа: https://www.bestprog.net/ru/2020/12/08/java-lambda-expressions-baxis-consepts-functional-interface-generalized-functional-interfaces-and-lambda-expressions-examples-ru. – Дата доступа: 03.03.2023.

3 Лямбда-выражения // Metanit.com [Электронный русурс]. – Режим доступа: https://metanit.com/java/tutorial/9.1.php. – Дата доступа: 03.03.2023.

ЛАБОРАТОРНАЯ РАБОТА № 6.
STREAM API JAVA

6.1 Цель работы

Ознакомиться с принципами работы со структурами данных с применением *Stream API Java*. Научиться разрабатывать программы с использованием потоков данных и применять к ним операции фильтрации, сортировки, отображения и т. д.

6.2 Теоретическая часть

Stream API появился в языке *Java* начиная с *JDK* 8. Он позволяет использовать функциональный стиль при работе с разными структурами данных. Для начала стриму нужен источник, из которого он будет получать объекты, чаще всего это коллекции. Вся основная функциональность данного *API* сосредоточена в пакете *java.util.stream*. Ключевым понятием в *Stream API* является поток данных.

Данные в стриме обрабатываются на промежуточных операциях. Например: мы можем отфильтровать данные, пропустить несколько элементов, ограничить выборку, выполнить сортировку. Затем выполняется терминальная операция. Она поглощает данные и выдает результат.

Основным преимуществом является помощь пользователям в ускорении и упрощении обработки данных. Сам по себе *API* предоставляет инструмент, который позволяет обрабатывать объекты.

Рассмотрим пример: найти в массиве количество всех чисел, которые больше 0, до *Stream API*.

```
int[] numbers = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
int count=0;
for(int i:numbers)
{
 if(i > 0) count++;
}
System.out.println(count);
```

А теперь применим *Stream API*:

```
import java.util.stream.*;
//.....................
long count = IntStream.of(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4,
5).filter(w -> w > 0).count();
System.out.println(count);
```

Вместо цикла и множества условных конструкций можно записать цепочку методов, которые будут выполнять те же действия.

6.2.1 Терминальные и промежуточные потоки

При работе со *Stream API* важно понимать, что все операции с потоками бывают либо терминальными (*terminal*), либо промежуточными (*intermediate*). Промежуточные операции возвращают трансформированный поток.

Так, в примере выше метод *filter()* принимал поток чисел и возвращал уже преобразованный поток, в котором только числа больше 0. К возвращенному потоку также можно применить ряд промежуточных операций.

Конечные или терминальные операции возвращают конкретный результат. Например, в примере выше метод *count()* представляет терминальную операцию и возвращает число. После этого никаких промежуточных операций применять нельзя.

Все потоки производят вычисления, в том числе в промежуточных операциях, только тогда, когда к ним применяется терминальная операция. То есть в данном случае применяется отложенное выполнение.

В основе *Stream API* лежит интерфейс *BaseStream*:

```
interface BaseStream <T , S extends BaseStream <T , S>>
```

Параметр *T* означает тип данных в потоке, а *S* – тип потока, который наследуется от интерфейса *BaseStream*.

BaseStream определяет базовый функционал для работы с потоками, которые реализуется через его методы:

– *void close()* закрывает поток;

– *boolean isParallel()* возвращает *true*, если поток является параллельным;

– *Iterator<T> iterator()* возвращает ссылку на итератор потока;

– *Spliterator<T> spliterator()* возвращает ссылку на сплитератор потока;

– *S parallel()* возвращает параллельный поток (параллельные потоки могут задействовать несколько ядер процессора в многоядерных архитектурах);

– *S sequential()* возвращает последовательный поток;

– *S unordered()* возвращает неупорядоченный поток.

От интерфейса *BaseStream* наследуется ряд интерфейсов, предназначенных для создания конкретных потоков:

– *Stream<T>* используется для потоков данных, представляющих любой ссылочный тип;

– *IntStream* используется для потоков с типом данных *int*;

– *DoubleStream* используется для потоков с типом данных *double*;

– *LongStream* используется для потоков с типом данных *long*.

При работе с потоками, которые представляют определенный примитивный тип – *double, int, long,* проще использовать интерфейсы *DoubleStream, IntStream, LongStream*. Но в большинстве случаев, как правило, работа происходит с более сложными данными, для которых предназначен интерфейс *Stream<T>*. Рассмотрим некоторые его методы:

– *boolean allMatch(Predicate<? super T> predicate)* возвращает *true*, если все элементы потока удовлетворяют условию в предикате. Терминальная операция;

– *boolean anyMatch(Predicate<? super T> predicate)* возвращает *true*, если хоть один элемент потока удовлетворяют условию в предикате. Терминальная операция;

– *<R,A> R collect(Collector<? super T,A,R> collector)* добавляет элементы в неизменяемый контейнер с типом *R*. *T* представляет тип данных из вызывающего потока, а *A* – тип данных в контейнере. Терминальная операция;

– *long count()* возвращает количество элементов в потоке. Терминальная операция;

– *Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)* объединяет два потока. Промежуточная операция;

– *Stream<T> distinct()* возвращает поток, в котором имеются только уникальные данные с типом *T*. Промежуточная операция;

– *Stream<T> dropWhile(Predicate<? super T> predicate)* пропускает элементы, которые соответствуют условию в *predicate*, пока не попадется элемент, который не соответствует условию. Выбранные элементы возвращаются в виде потока. Промежуточная операция;

– *Stream<T> filter(Predicate<? super T> predicate)* фильтрует элементы в соответствии с условием в предикате. Промежуточная операция;

– *Optional<T> findFirst()* возвращает первый элемент из потока. Терминальная операция;

– *Optional<T> findAny()* возвращает первый попавшийся элемент из потока. Терминальная операция;

– *void forEach(Consumer<? super T> action)* – для каждого элемента выполняется действие *action*. Терминальная операция;

– *Stream<T> limit(long maxSize)* оставляет в потоке только *maxSize* элементов. Промежуточная операция;

– *Optional<T> max(Comparator<? super T> comparator)* возвращает максимальный элемент из потока. Для сравнения элементов применяется компаратор *comparator*. Терминальная операция;

– *Optional<T> min(Comparator<? super T> comparator)* возвращает минимальный элемент из потока. Для сравнения элементов применяется компаратор *comparator*. Терминальная операция;

– *<R> Stream<R> map(Function<? super T,? extends R> mapper)* преобразует элементы типа *T* в элементы типа *R* и возвращает поток с элементами *R*. Промежуточная операция;

– *<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)* позволяет преобразовать элемент типа *T* в несколько элементов типа *R* и возвращает поток с элементами *R*. Промежуточная операция;

– *boolean noneMatch(Predicate<? super T> predicate)* возвращает *true*, если ни один из элементов в потоке не удовлетворяет условию в предикате. Терминальная операция;

– *Stream<T> skip(long n)* возвращает поток, в котором отсутствуют первые *n* элементов. Промежуточная операция;

– *Stream<T> sorted()* возвращает отсортированный поток;

– *Stream<T> sorted(Comparator<? super T> comparator)* возвращает отсортированный в соответствии с компаратором поток. Промежуточная операция;

– *Stream<T> takeWhile(Predicate<? super T> predicate)* выбирает из потока элементы, пока они соответствуют условию в *predicate*. Выбранные элементы возвращаются в виде потока. Промежуточная операция;

– *Object[] toArray()* возвращает массив из элементов потока. Терминальная операция.

Несмотря на то, что все эти операции позволяют взаимодействовать с потоком как неким набором данных наподобие коллекции, важно понимать отличие коллекций от потоков:

1 Потоки не хранят элементы. Элементы, используемые в потоках, могут храниться в коллекции либо при необходимости могут быть напрямую сгенерированы.

2 Операции с потоками не изменяют источника данных. Операции с потоками лишь возвращают новый поток с результатами этих операций.

3 Для потоков характерно отложенное выполнение. Выполнение всех операций с потоком происходит лишь тогда, когда выполняется терминальная операция и возвращается конкретный результат, а не новый поток.

Для создания потока данных можно применять различные методы. В качестве источника потока используются коллекции. В частности, в интерфейс *Collection*, который реализуется всеми классами коллекций, были добавлены два метода для работы с потоками:

– *default Stream<E> stream*: возвращается поток данных из коллекции;

– *default Stream<E> parallelStream*: возвращается параллельный поток данных из коллекции.

Рассмотрим на примере с *ArrayList*:

```
import java.util.stream.Stream;
import java.util.*;
public class Program {
    public static void main(String[] args) {

        ArrayList<String> cities = new ArrayList<String>();
        Collections.addAll(cities, "Париж", "Лондон",
"Мадрид");
        cities.stream() // Получение потока
            .filter(s->s.length()==6) // Применение фильтрации
по длине строки
            .forEach(s->System.out.println(s)); // Выведение
отфильтрованных строк на консоль
    }}
```

С помощью вызова *cities.stream()* получаем поток, который использует данные из списка *cities*. С помощью каждой промежуточной операции, которая применяется к потоку, мы также можем получить поток с учетом модификаций. Мы можем изменить предыдущий пример:

```
ArrayList<String> cities = new ArrayList<String>();
Collections.addAll(cities, "Париж", "Лондон", "Мадрид");
Stream<String> citiesStream = cities.stream(); // Получение потока
citiesStream = citiesStream.filter(s->s.length()==6); // Применение фильтрации по длине строки
citiesStream.forEach(s->System.out.println(s)); // Выведение отфильтрованных строк на консоль
```

Важно, что после использования терминальных операций другие терминальные или промежуточные операции к этому же потоку не могут быть применены, поток уже употреблен. В следующем случае мы получим ошибку:

```
citiesStream.forEach(s->System.out.println(s)); // Терминальная операция употребляет поток
long number = citiesStream.count(); // Здесь ошибка, т. к. поток уже употреблен
System.out.println(number);
citiesStream = citiesStream.filter(s->s.length()>5); // Тоже нельзя, т. к. поток уже употреблен
```

Жизненный цикл потока проходит следующие три стадии:

1 Создание потока.

2 Применение к потоку ряда промежуточных операций.

3 Применение к потоку терминальной операции и получение результата.

Кроме вышерассмотренных методов мы можем использовать еще ряд способов для создания потока данных. Один из таких способов представляет метод *Arrays.stream(T[] array)*, который создает поток данных из массива:

```
Stream<String> citiesStream = Arrays.stream(new String[]{"Париж", "Лондон", "Мадрид"}) ;
citiesStream.forEach(s->System.out.println(s)); // Выведение всех элементов массива
```

Для создания потоков *IntStream*, *DoubleStream*, *LongStream* можно использовать соответствующие перегруженные версии этого метода:

```
IntStream intStream = Arrays.stream(new int[]{1,2,4,5,7});
intStream.forEach(i->System.out.println(i));
LongStream longStream = Arrays.stream(new long[]{100,250,400,5843787,237});
longStream.forEach(l->System.out.println(l));
DoubleStream doubleStream = Arrays.stream(new double[] {3.4, 6.7, 9.5, 8.2345, 121});
doubleStream.forEach(d->System.out.println(d));
```

6.2.2 Сортировка строк в Stream API Java

Сортировка строк в *Stream API Java* может быть выполнена с помощью метода *sorted()*.

Пример, который демонстрирует, как отсортировать список строк в порядке возрастания:

```
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie",
"Dave");
    List<String> sortedNames = names.stream()
                                    .sorted()
                                    .collect(Collec-
    tors.toList());
    System.out.println(sortedNames); // [Alice, Bob, Charlie, Dave]
```

Метод *sorted()* по умолчанию сортирует элементы в порядке возрастания, но мы также можем определить свой порядок сортировки, передавая *Comparator* в метод *sorted()*. Например, вот как можно отсортировать список строк в порядке убывания:

```
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie",
"Dave");
    List<String> reverseSortedNames = names.stream()
                                           .sorted(Compara-
    tor.reverseOrder())
                                           .collect(Collec-
    tors.toList());
    System.out.println(reverseSortedNames);  //  [Dave,  Charlie,
Bob, Alice]
```

В этом примере мы используем метод *sorted()* с параметром *Comparator.reverseOrder()* для того, чтобы отсортировать элементы в порядке убывания.

Также можно использовать метод *sorted()* для сортировки объектов класса, определенного пользователем. Для этого класс должен реализовать интерфейс *Comparable* или нужно передать *Comparator* в метод *sorted()*.

Например, предположим, что у нас есть список объектов класса *Person*, который содержит имена и возраст каждого человека. Вот как мы можем отсортировать список по возрасту:

```
    List<Person> people = Arrays.asList(new Person("Alice", 25),
new Person("Bob", 30), new Person("Charlie", 20));
    List<Person> sortedPeople = people.stream()
                                      .sorted(Comparator.com-
paringInt(Person::getAge))
                                      .collect(Collec-
tors.toList());
    System.out.println(sortedPeople); // [Charlie (20), Alice (25),
Bob (30)]
```

В этом примере мы используем метод *sorted()* с параметром *Comparator.comparingInt()*, чтобы отсортировать элементы по возрасту. Метод *comparingInt()* получает функцию, которая принимает объект класса *Person* и возвращает его возраст.

6.2.3 Фильтрация в Stream API Java

Фильтрация является одной из основных операций в *Stream API Java*, которая позволяет отфильтровать элементы потока на основе заданного условия.

Для фильтрации элементов в потоке используется метод *filter()*. Этот метод принимает на вход предикат, который определяет, должен ли элемент остаться в потоке или быть отфильтрованным.

Например, предположим, что у нас есть список имен и мы хотим получить список имен, начинающихся с буквы «*A*». Вот как мы можем это сделать с помощью метода *filter()*:

```
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie",
"Dave");
    List<String> filteredNames = names.stream()
                                          .filter(name ->
name.startsWith("A"))
                                          .collect(Collec-
tors.toList());
    System.out.println(filteredNames); // [Alice]
```

В этом примере мы использовали метод *filter()* для фильтрации имен, начинающихся с буквы «*A*». Мы передали лямбда-выражение, которое проверяет, начинается ли имя с «*A*». Затем мы вызываем метод *collect()* с параметром *Collectors.toList()*, чтобы сохранить отфильтрованные имена в список.

Метод *filter()* может быть использован с различными предикатами, которые проверяют условия на основе разных критериев, таких как длина строки, наличие определенного символа или атрибуты объектов.

Кроме того, метод *filter()* можно использовать вместе с другими методами *Stream API Java*, такими как *map(), flatMap(), sorted(), distinct()* и т. д. В этом случае элементы потока сначала будут отфильтрованы, а затем пройдут через другие операции.

В целом метод *filter()* является мощным инструментом для работы с потоками данных в *Stream API Java*, который позволяет выбрать только те элементы, которые соответствуют заданным условиям.

6.2.4 Отображение в Stream API Java

Отображение (*mapping*) – это еще одна основная операция в *Stream API Java*, которая позволяет преобразовать элементы потока в другие элементы на основе заданного правила.

Для отображения элементов в потоке используется метод *map()*. Этот метод принимает на вход функцию, которая принимает один элемент и возвращает новый элемент после преобразования.

Например, предположим, что у нас есть список чисел, и мы хотим создать новый список, в котором каждое число будет возведено в квадрат. Вот как мы можем это сделать с помощью метода *map()*:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squaredNumbers = numbers.stream()
               .map(number -> number * number)
               .collect(Collectors.toList());
System.out.println(squaredNumbers); // [1, 4, 9, 16, 25]
```

В этом примере мы использовали метод *map()* для возведения каждого числа в квадрат. Мы передали лямбда-выражение, которое принимает число и возвращает квадрат этого числа. Затем мы вызываем метод *collect()* с параметром *Collectors.toList()*, чтобы сохранить новые числа в список.

Метод *map()* может быть использован с различными функциями, которые преобразуют элементы в потоке на основе разных критериев, таких как изменение типа объекта, вычисление значения на основе атрибутов объектов и т. д.

Кроме того, метод *map()* можно использовать вместе с другими методами *Stream API Java*, такими как *filter(), flatMap(), sorted(), distinct()* и т. д. В этом случае элементы потока сначала будут преобразованы, а затем пройдут через другие операции.

В целом метод *map()* является мощным инструментом для работы с потоками данных в *Stream API Java*, который позволяет преобразовать элементы потока в другие элементы на основе заданных правил.

6.3 Практическая часть

1 Изучить описание к лабораторной работе.

2 Исходя из предметной области лабораторной работы № 1 написать программу, в которой будет создаваться поток данных на базе коллекций, реализованных в лабораторной работе № 3, после чего применить к потоку данных операции фильтрации, сортировки, группировки и др.

6.4 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Иллюстрация решения задачи.
5 Выводы.
6 Список использованных источников.

6.5 Контрольные вопросы

1 Что такое *Stream API Java* и для чего он используется?
2 Как создать поток данных в *Stream API Java*?
3 Какие операции доступны в *Stream API Java*?
4 Что такое промежуточные и терминальные операции в *Stream API Java*?
5 Какие ограничения есть при использовании *Stream API Java*?

6.6 Список использованных источников

1 Дашнер, С. И. Изучаем Java. Современное программирование / С. И. Дашнер. – СПб. : Питер, 2018. – 384 с.

2 Васильев, А. Н. Java. Объектно-ориентированное программирование / А. Н. Васильев. – СПб. : Питер, 2020. – 400 с.

ЛАБОРАТОРНАЯ РАБОТА № 7.
МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ

7.1 Цель работы

Изучить основные методы и подходы, использующиеся при реализации многопоточных программ в *Java*. Ознакомиться с основными программными абстракциями многопоточности и способами их применения при разработке прикладных программ.

7.2 Теоретическая часть

Введение

Многопоточные программы расширяют принцип многозадачности, перенося его на один уровень ниже, чтобы отдельные программы могли выполнять многие задачи одновременно. Каждая задача обычно называется потоком исполнения или потоком управления. Программы, способные одновременно выполнять больше одного потока исполнения, называются многопоточными [1].

Так в чем же отличие между процессами и потокам исполнения? Оно состоит в следующем: если у каждого процесса имеется собственный набор переменных, то потоки могут разделять одни и те же общие данные. Разделяемые переменные обеспечивают более высокую эффективность взаимодействия потоков и облегчают реализацию связи между ними. Кроме того, в некоторых операционных системах потоки исполнения являются более «легковесными», чем процессы – они требуют меньших издержек на создание и уничтожение по сравнению с процессами [1].

Потоки выполнения являются необходимой частью функциональности языка *Java*, поскольку они могут упростить разработку систем, превращая сложный асинхронный код в более простой и прямолинейный. Кроме того, потоки – это лучший способ задействовать сразу несколько вычислительных мощностей многопроцессорных систем. И по мере увеличения числа процессоров эффективность применение потоков будет расти [2].

7.2.1 Реализация многопоточности в Java

Для реализации многопоточности в *Java* следует воспользоваться классом *java.lang.Thread*. В этом классе определены методы, которые позволяют создавать потоки, управлять их состоянием и осуществлять их синхронизацию.

Существует два способа использования класса *Thread*.

1 Создать дочерний класс на базе класса *Thread*. При этом необходимо переопределить метод *run()*. Код этого метода будет выполняться в отдельном потоке после вызова метода *start()* у объекта дочернего класса.

2 Создать класс, который реализует интерфейс *Runnable*. При этом в рамках данного класса необходимо переопределить метод *run()*, код внутри которого будет выполнятся в отдельном потоке после вызова метода *start()* у объекта

класса *Thread*, в конструктор которого следует передать объект вышеупомянутого класса.

Второй способ считается более приемлемым, особенно когда пользовательский класс должен также быть унаследован от какого-либо другого класса, помимо класса *Thread*, и при этом он должен являться классом-реализацией потока. Так как в языке программирования *Java* отсутствует множественное наследование, невозможно создать класс, для которого в качестве родительского будет выступать и класс *Thread*, и любой другой класс. В этом случае использование интерфейса *Runnable* является единственным способом решения данной задачи.

7.2.2 Класс Thread

В классе *Thread* определены три поля, несколько конструкторов и множество методов, предназначенных для работы с потоками [3].

С помощью конструкторов можно создавать потоки различными способами, указывая при необходимости для них имя и группу. Имя предназначено для идентификации потока и является необязательным атрибутом. Что же касается групп, то они предназначены для организации защиты потоков друг от друга в рамках одной программы.

Методы класса *Thread* предоставляют все необходимые возможности для управления потоками, в том числе для их синхронизации. Базовыми методами данного класса являются: *public void run()* и *public void start()*, которые используются для создания и запуска потока исполнения.

Класс, реализующий поток исполнения:

```
class PrimeThread extends Thread {
          long minPrime;
          PrimeThread(long minPrime) {
              this.minPrime = minPrime;
          }

          public void run() {
              // Вычисление простых чисел, больших чем minPrime
               . . .
          }
      }
```

Код метода *main*:

```
 PrimeThread p = new PrimeThread(143);
     p.start();
```

7.2.3 Интерфейс Runnable

Описанный выше способ создания потоков как объектов класса *Thread* или унаследованных от него классов кажется достаточно естественным. Однако этот способ не единственный. Если необходимо, чтобы класс, который реализует поток, в будущем мог также унаследовать какой-либо другой класс, то лучше выбрать второй способ с использованием интерфейса *Runnable*.

Класс, реализующий поток исполнения:

```
class PrimeRun implements Runnable {
         long minPrime;
         PrimeRun(long minPrime) {
             this.minPrime = minPrime;
         }

         public void run() {
             // Вычисление простых чисел, больших чем minPrime
              . . .
         }
     }
```

Код метода *main*:

```
PrimeRun p = new PrimeRun(143);
   new Thread(p).start();
```

Внутри класса необходимо определить метод *run()*, который будет выполняться в рамках отдельного потока. Для создания потока используется оператор *new* и конструктор. Запускается поток после вызова метода *start()*.

При этом, когда поток запустится, управление получит метод *run()*, определенный в классе *PrimeRun*.

7.2.4 Завершение и остановка потоков

Поток исполнения завершается по одной из следующих причин:

– прекращает свое существование естественным образом при нормальном завершении метода *run()*;

– прекращает свое существование внезапно, поскольку неперехваченное исключение прерывает выполнение метода *run()*.

В частности поток исполнения можно уничтожить, вызвав его метод *stop()*. Этот метод генерирует объект ошибки типа *ThreadDeath*, который уничтожает поток исполнения. Но использование метода *stop()* не рекомендовано, поэтому следует избегать его применения в прикладном коде [1].

7.2.5 Приоритеты потоков

Если процесс создал несколько потоков, то все они выполняются параллельно, причем время центрального процессора распределяется между этими потоками.

Распределением времени центрального процессора занимается специальный модуль операционной системы – планировщик. Планировщик по очереди передает управление отдельным потокам, так что даже в однопроцессорной системе создается полная иллюзия параллельной работы запущенных потоков.

Распределение времени выполняется по прерываниям системного таймера. Поэтому каждому потоку дается определенный интервал времени, в течение которого он находится в активном состоянии.

Для оптимизации параллельной работы потоков в *Java* имеется возможность устанавливать приоритеты потоков.

Каждый поток исполнения имеет свой приоритет. По умолчанию поток исполнения наследует приоритет того потока, который его создал. Повысить или понизить приоритет любого потока исполнения можно, вызвав метод *setPriority()*. А установить приоритет потока исполнения можно, указав любое значение в пределах от *MIN_PRIORITY* (определено в классе *Thread* равным единице) до *MAX_PRIORITY* (определено в классе *Thread* равным десяти). Обычному приоритету соответствует значение *NORM_PRIORITY*, равное пяти.

Всякий раз, когда планировщик потоков выбирает новый поток для исполнения, он предпочитает потоки с более высоким приоритетом. Но приоритеты потоков исполнения сильно зависят от системы. Когда виртуальная машина полагается на реализацию потоков средствами главной платформы, на которой она выполняется, приоритеты потоков *Java* привязываются к уровням приоритетов этой платформы, где может быть больше или меньше уровней приоритетов [1].

7.2.6 Средства синхронизации потоков в Java

Всякий раз, когда более чем один поток обращается к переменной в общей памяти и один из потоков, возможно, записывает в нее данные, все потоки должны координировать свой доступ к переменной с помощью синхронизации.

Синхронизация необходима как для взаимоисключения потоков, так и для надежного взаимодействия между ними.

Синхронизацию в *Java* обеспечивают ключевое слово *synchronized*, дающее эксклюзивную блокировку, а также *volatile* и атомарные переменные [2].

Ключевое слово *synchronized* гарантирует, что в любой момент времени метод или блок кода будет выполняться только одним потоком [4].

Для синхронизации необходимо определить некоторый ресурс, который может быть заблокирован. Таким ресурсом может быть любой объект (но не данные элементарного типа). Далее определяются критические участки программы. При входе в такой участок ресурс блокируется и становится недоступным для всех других потоков. При выходе из критического участка выполняется разблокировка ресурса.

В качестве критического участка программы в *Java* может быть определен некоторый нестатический метод или блок. При вызове метода блокируется объект, для которого вызван данный метод (*this*). Для блока блокируемый объект указывается явно.

Синтаксис определения критических участков следующий.

Для метода мы просто при его описании указываем *synchronized*, например:

```
public void synchronized f() {
   ...
}
```

Для блока мы непосредственно перед блоком помещаем конструкцию *synchronized* (объект), где объект – это ссылка на блокируемый объект. Например:

```
synchronized(ref) {
   ...
}
```

Здесь в качестве критического участка определяется блок, а в качестве блокируемого объекта – объект, на который ссылается *ref*.

Язык *Java* также предоставляет альтернативную, более слабую форму синхронизации – использование *volatile*-переменных, обновления которых распространяются предсказуемо всеми потоками. Переменная с модификатором *volatile* для компилятора и рабочей среды является совместной, т. е. операции над ней не будут переупорядочены с другими операциями в памяти. *Volatile*-переменные не кешируются в регистрах или кешах, где данные скрыты от других процессоров, поэтому их чтение всегда возвращает самый последний результат операций записи [2].

В версии *Java SE* 5.0 появился класс *ReentrantLock* – еще один способ синхронизации потоков.

Защита блока кода средствами класса *ReentrantLock* выглядит следующим образом:

```
myLock.lock () ; // Объект типа ReentrantLock
try
{
     // Критический раздел кода
}
finally
{
myLock.unlock(); // Непременно снять блокировку,
     // даже если генерируется исключение
}
```

7.2.7 Пулы потоков

Создание нового потока исполнения – довольно дорогостоящая операция с точки зрения потребляемых ресурсов, поскольку она включает в себя взаимодействие с операционной системой. Если в программе создается большое количество кратковременных потоков исполнения, то имеет смысл использовать пул потоков. В пуле потоков содержится целый ряд простаивающих потоков, готовых к запуску. Так, объект типа *Runnable* размещается в пуле, а один из потоков вызывает его метод *run()*. Когда метод *run()* завершается, его поток не уничтожается, но остается в пуле готовым обслужить новый запрос.

Другая причина для использования пула потоков заключается в необходимости ограничить количество параллельно исполняемых потоков. Создание огромного числа потоков исполнения может отрицательно сказаться на производительности и даже привести к полному отказу виртуальной машины. Поэтому, если применяется алгоритм, создающий большое количество потоков, следует установить фиксированный пул потоков, чтобы ограничить общее количество параллельно исполняемых потоков. В состав класса *Executors* входит целый ряд статических фабричных методов для построения пулов потоков [1].

Пример использования одного из таких методов *newSingleThreadExecutor()*:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class MyRunnable implements Runnable {
    private final String task;

    MyRunnable(String task) {
        this.task = task;
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
           System.out.println("Executing "+ task +" with
"+Thread.currentThread().getName());
        }
        System.out.println();
    }
}

public class Exec {

    public static void main(String[] args) {
ExecutorService executor =
Executors.newSingleThreadExecutor();
        for (int i = 1; i <= 5; i++) {
            Runnable worker = new MyRunnable("Task" + i);
            executor.execute(worker);
        }
```

```
            executor.shutdown();
        /* После этого исполнитель перестанет принимать какие-либо
новые потоки и завершит все существующие в очереди */
            }
        }
```

В примере выше на исполнение отправляется пять задач. Но по причине применения метода *newSingleThreadExecutor()* будет создан только один новый поток и одновременно будет выполняться только одна задача. Остальные четыре задачи находятся в очереди ожидания. Как только задача выполнится потоком, этот поток тут же выберет и выполнит следующую. Метод *shutdown()* ожидает завершения выполнения задач, в настоящий момент переданных исполнителю, чтобы завершить его работу. Однако, если необходимо завершить работу исполнителя без ожидания, следует использовать метод *shutdownNow()*.

7.3 Практическая часть

1 Изучить описание к лабораторной работе, ознакомиться с примерами реализации основных программных абстракций многопоточности.

2 Преобразовать реализацию операций чтения / записи в файл основных типов данных выстроенной иерархии классов из четвертой лабораторной работы с использованием абстракций многопоточности.

7.4 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Иллюстрация решения задачи.
5 Выводы.
6 Список использованных источников.

7.5 Контрольные вопросы

1 Дайте определение понятию «поток».
2 Дайте определение понятию «процесс».
3 Дайте определение понятию «синхронизация потоков».
4 В каких случаях целесообразно создавать несколько потоков?
5 Что может произойти, если два потока будут выполнять один и тот же код в программе?
6 Какие существуют способы создания и запуска потоков?
7 Какой метод запускает поток на выполнение?
8 Какой метод описывает действие потока во время выполнения?
9 Когда поток завершает свое выполнение?

7.6 Список использованных источников

1 Хорстманн, К. С. Java. Библиотека профессионала. В 2 т. Т. 1 : Основы / К. С. Хорстманн. – 11-е изд. – СПб. : Диалектика, 2019. – 864 с.

2 Java Concurrency на практике / Б. Гетц [и др.]. – СПб. : Питер, 2020. – 464 с.

3 Java Platform, Standard Edition & Java Development Kit Version 19 API Specification [Электронный ресурс]. – Режим доступа: https://docs.oracle.com/en/java/javase/19/docs.

4 Блох, Д. Java: эффективное программирование / Д. Блох. – 3-е изд. – СПб. : Диалектика, 2019. – 464 с.

ЛАБОРАТОРНАЯ РАБОТА № 8.
ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

8.1 Цель работы

Ознакомиться с паттернами проектирования и их назначением. Научиться разрабатывать программы с использованием таких паттернов, как абстрактная фабрика, фабричный метод, декоратор.

8.2 Теоретическая часть

Введение

Паттерн проектирования – это часто встречающаяся архитектурная конструкция, предлагающая решение общей проблемы проектирования в рамках конкретной предметной области. Паттерн представляет собой не какой-то конкретный код, который может быть встроен в программу подобно готовым библиотекам, а лишь общую концепцию решения той или иной проблемы, которую нужно будет подстроить под нужды программы. Шаблоны проектирования показывают отношения и взаимодействия между классами или объектами без определения того, какие конечные классы или объекты приложения будут использоваться.

Существуют различные классификации паттернов проектирования, наиболее распространенной является классификация по назначению. Данная классификация выделяет порождающие, структурные и поведенческие шаблоны проектирования.

Порождающие паттерны предназначены для организации удобного и безопасного создания объектов.

К порождающим паттернам относятся:

– абстрактная фабрика (*abstract factory*);
– одиночка (*singleton*);
– прототип (*prototype*);
– строитель (*builder*);
– фабричный метод (*factory method*).

Структурные паттерны отвечают за построение удобных в поддержке иерархий классов.

К структурным паттернам относятся:

– адаптер (*adapter*);
– декоратор (*decorator*);
– заместитель (*proxy*);
– компоновщик (*composite*);
– легковес (*flyweight*);
– мост (*bridge*);
– фасад (*facade*).

Поведенческие паттерны предназначены для обеспечения эффективных способов взаимодействия между объектами программы.

К поведенческим паттернам относятся:
– итератор (*iterator*);
– команда (*command*);
– наблюдатель (*observer*);
– посетитель (*visitor*);
– посредник (*mediator*);
– снимок (*memento*);
– состояние (*state*);
– стратегия (*strategy*);
– цепочка обязанностей (*chain of responsibility*);
– шаблонный метод (*template method*).

8.2.1 Паттерн «абстрактная фабрика»

Абстрактная фабрика – порождающий паттерн проектирования, позволяющий создавать семейства связанных объектов без привязки к конкретным классам создаваемых объектов.

Данный шаблон целесообразно применять, когда:

– система не должна зависеть от способа создания и компоновки новых объектов;

– создаваемые объекты являются взаимосвязанными и должны использоваться вместе.

В общем виде абстрактная фабрика состоит из следующих элементов:

1 Абстрактные продукты – интерфейсы для классов, объекты которых будут создаваться в программе.

2 Конкретные продукты – большой набор классов, относящихся к различным абстрактным продуктам, имеющих одни и те же вариации.

3 Абстрактная фабрика – абстрактный класс, объявляющий методы создания различных абстрактных продуктов.

4 Конкретные фабрики относятся каждая к своей вариации продуктов и реализуют методы абстрактной фабрики, позволяя создавать все продукты определенной вариации.

5 Класс клиента использует класс фабрики для создания объектов. При этом он использует абстрактную фабрику и абстрактные классы продуктов и никак не зависит от их конкретных реализаций. Через абстрактные интерфейсы клиент сможет работать с любыми вариациями продуктов.

В качестве примера использования паттерна рассматривается производство семейств кроссплатформенных элементов пользовательского интерфейса. В роли двух семейств продуктов выступают кнопки и чекбоксы. Оба семейства продуктов имеют одинаковые вариации: для работы в *MacOS* и *Windows*. Абстрактная фабрика задает интерфейс создания продуктов всех семейств. Конкретные фабрики создают различные продукты одной вариации (*MacOS* или *Windows*). Клиенты фабрики работают как с фабрикой, так и с продуктами только через абстрактные интерфейсы.

Иерархия продуктов-кнопок:

```
package abstract_factory.buttons;
// Интерфейс Button
public interface Button {
  void paint();
}

// Класс MacOSButton
package abstract_factory.buttons;

public class MacOSButton implements Button {

    @Override
    public void paint() {
        // ... код ...
    }
}

// Класс WindowsButton
package abstract_factory.buttons;
public class WindowsButton implements Button {

    @Override
    public void paint() {
        // ... код ...
    }
}
```

Иерархия продуктов-чекбоксов:

```
// Интерфейс Checkbox
package abstract_factory.checkboxes;

public interface Checkbox {
    void paint();
}

// Класс MacOSCheckbox
package abstract_factory.checkboxes;

public class MacOSCheckbox implements Checkbox {

    @Override
    public void paint() {
        // ... код ...
    }
}

// Класс WindowsCheckbox
package abstract_factory.checkboxes;
```

```
public class WindowsCheckbox implements Checkbox {

    @Override
    public void paint() {
        // ... код ...
    }
}
```

Иерархия фабрик:

```
// Интерфейс абстрактной фабрики
package.abstract_factory.factories;

import abstract_factory.buttons.Button;
import abstract_factory.checkboxes.Checkbox;

public interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

// Конкретная фабрика MacOSFactory
package abstract_factory. factories;

import abstract_factory.buttons.Button;
import.abstract_factory.buttons.MacOSButton;
import abstract_factory.checkboxes.Checkbox;
import abstract_factory.checkboxes.MacOSCheckbox;

public class MacOSFactory implements GUIFactory {

    @Override
    public Button createButton() {
        return new MacOSButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new MacOSCheckbox();
    }
}

// Конкретная фабрика WindowsFactory
package abstract_factory.factories;

import abstract_factory.buttons.Button;
import abstract_factory.buttons.WindowsButton;
import abstract_factory.checkboxes.Checkbox;
import abstract_factory.checkboxes.WindowsCheckbox;
```

```
public class WindowsFactory implements GUIFactory {

    @Override
    public Button createButton() {
        return new WindowsButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}
```

Клиентский код:

```
package abstract_factory.app;

import abstract_factory.buttons.Button;
import abstract_factory.checkboxes.Checkbox;
import abstract_factory.factories.GUIFactory;

public class Application {
    private Button button;
    private Checkbox checkbox;

    public Application(GUIFactory factory) {
        button = factory.createButton();
        checkbox = factory.createCheckbox();
    }

    public void paint() {
        button.paint();
        checkbox.paint();
    }
}
```

Работа приложения:

```
package abstract_factory;

import abstract_factory.app.Application;
import abstract_factory.factories.GUIFactory;
import abstract_factory.factories.MacOSFactory;
import abstract_factory.factories.WindowsFactory;

public class Main {

    private static Application configureApplication() {
        Application app;
        GUIFactory factory;
```

```
            String osName = System.getProperty("os.name").toLow-
erCase();
            if (osName.contains("mac")) {
                factory = new MacOSFactory();
            } else {
                factory = new WindowsFactory();
            }
            app = new Application(factory);
            return app;
        }

        public static void main(String[] args) {
            Application app = configureApplication();
            app.paint();
        }
    }
```

Достоинствами абстрактной фабрики можно считать:

– сочетаемость создаваемых продуктов;

– избавление клиентского код от привязки к конкретным классам продуктов;

– выделение кода производства продуктов в одно место, что упрощает поддержку кода.

К недостаткам данного паттерна можно отнести то, что его использование усложняет код программы из-за введения множества дополнительных классов. Также данный подход вынуждает иметь в наличии все типы продуктов в каждой вариации.

8.2.2 Паттерн «фабричный метод»

Фабричный метод – порождающий паттерн проектирования, определяющий общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов. Паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

Варианты применения паттерна:

– когда заранее неизвестно, объекты каких типов необходимо создавать;

– когда система должна иметь возможность расширяться и быть независимой от процесса создания новых объектов: в нее можно легко вводить новые классы, объекты которых будет создавать система;

– когда создание новых объектов необходимо делегировать из базового класса классам наследника.

Фабричный метод состоит из следующих элементов:

1 Продукт – общий интерфейс объектов, которые может произвести создатель и его подклассы.

2 Конкретные продукты – классы, содержащие код различных продуктов. Продукты будут отличаться реализацией, но следуют общему интерфейсу.

3 Создатель – абстрактный класс, в котором объявлен фабричный метод, возвращающий новые объекты продуктов. Тип результата должен совпадать с

общим интерфейсом продуктов. Фабричный метод можно сделать абстрактным, чтобы все подклассы реализовали его по-своему. Но он может возвращать и некий стандартный продукт. Помимо создания продуктов, класс создателя может содержать и другой код для работы с продуктом.

4 Конкретные классы создателей – наследники абстрактного создателя – по-своему реализуют фабричный метод, производя те или иные конкретные продукты. Для каждого конкретного класса продукта определяется свой конкретный класс создателя.

Но фабричный метод не обязан все время создавать новые объекты. Его можно реализовать таким образом, чтобы возвращать уже существующие объекты из какого-то хранилища или кеша.

В качестве примера можно рассмотреть создание кроссплатформенных элементов пользовательского интерфейса. В качестве продуктов будут выступать кнопки, а в качестве создателя – диалог. Разным типам диалогов должны соответствовать свои собственные типы элементов. Поэтому для каждого типа диалога необходимо создать свой подкласс и переопределить в нем фабричный метод. Каждый конкретный диалог будет создавать подходящие ему кнопки. Базовый код диалогов будет работать с продуктами только через их общий интерфейс.

Общий интерфейс кнопок:

```
package factory_method.buttons;

public interface Button {
    void render();
    void onClick();
}
```

Конкретные классы кнопок:

```
// Класс HtmlButton
package factory_method.buttons;

public class HtmlButton implements Button {

    public void render() {
        // ... код ...
        onClick();
    }

    public void onClick() {
       // some code …
    }
}
```

```
// Класс WindowsButton
package factory_method.buttons;

import javax.swing.*;
import java.awt.*;

public class WindowsButton implements Button {
    JPanel panel = new JPanel();
    JFrame frame = new JFrame();
    JButton button;

    public void render() {
        // ... код ...

        onClick();
    }

    public void onClick() {
        // ... код ...
    }
}
```

Базовый диалог – базовый класс фабрики:

```
package.factory_method.factory;

import refactoring_guru.factory_method.example.buttons.But-
ton;

public abstract class Dialog {

    public void renderWindow() {
        // ... остальной код диалога ...

        Button okButton = createButton();
        okButton.render();
    }
    public abstract Button createButton();
}
```

Конкретные классы диалогов:

```
// Класс HtmlDialog
package factory_method.factory;

import factory_method.buttons.Button;
import factory_method.buttons.HtmlButton;

public class HtmlDialog extends Dialog {

    @Override
    public Button createButton() {
```

```
            return new HtmlButton();
        }
    }

    // Класс WindowsDialog
    package factory_method.factory;

    import factory_method.buttons.Button;
    import factory_method.buttons.WindowsButton;

    public class WindowsDialog extends Dialog {

        @Override
        public Button createButton() {
            return new WindowsButton();
        }
    }
```

Клиентский код:

```
    package factory_method;

    import factory_method.Dialog;
    import factory_method.HtmlDialog;
    import factory_method.WindowsDialog;

    public class Main {
        private static Dialog dialog;

        public static void main(String[] args) {
            configure();
            runBusinessLogic();
        }


    // Приложение создает фабрику в зависимости от конфигурации
        static void configure() {
            if (System.getProperty("os.name").equals("Windows
10")) {
                dialog = new WindowsDialog();
            } else {
                dialog = new HtmlDialog();
            }
        }

    /* Остальной клиентский код работает с фабрикой и продуктами
    * только через общий интерфейс, поэтому для него не важно, какая
    * фабрика была создана
    */

      static void runBusinessLogic() {
            dialog.renderWindow();    }}
```

К преимуществам данного паттерна можно отнести то, что он:

– позволяет сделать код создания объектов более универсальным, не привязываясь к конкретным классам, а оперируя лишь общим интерфейсом;

– дает возможность установить связь между параллельными иерархиями продуктов;

– упрощает добавление новых продуктов в программу.

К недостаткам шаблона можно отнести то, что его использование может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

8.2.3 Паттерн «декоратор»

Декоратор – структурный паттерн проектирования, позволяющий динамически добавлять объекту дополнительную функциональность.

Для определения нового функционала в классах часто используется наследование. Декоратор же предлагает более гибкую альтернативу наследованию, что позволяет динамически в процессе выполнения программы определять новые возможности у объектов.

Варианты применения паттерна:

1 Когда надо динамически добавлять к объекту новые функциональные возможности. Объекты помещают в обертки, имеющие дополнительный функционал. Когда и обертки, и объекты следуют общему интерфейсу, клиенту не важно с каким объектом работать – с обычным или с обернутым.

2 Когда применение наследования сильно усложняет код. Например, нужно определить множество функциональностей, но если для каждой функциональности наследовать отдельный класс, то структура классов может очень сильно разрастись, особенно если необходимо эти функциональности сочетать друг с другом.

3 Когда нельзя расширить функциональность объекта с помощью наследования. Например, классы с модификатором *final* можно расширить только с помощью декоратора.

Паттерн «декоратор» состоит из следующих элементов:

1 Компонент – общий интерфейс для компонентов и декораторов.

2 Конкретный компонент – класс оборачиваемых объектов. Он содержит базовую функциональность, которую потом дополняют декораторы.

3 Базовый декоратор хранит ссылку на вложенный объект-компонент, в качестве которого может выступать как конкретный компонент, так и один из конкретных декораторов. Базовый декоратор делегирует свои функции вложенному объекту. Дополнительное поведение располагается в конкретных декораторах.

4 Конкретные декораторы – различные вариации декораторов, содержащие дополнительно поведение. Оно может выполняться как до, так и после вызова аналогичного поведения обернутого объекта.

5 Клиент может оборачивать простые компоненты и декораторы в другие декораторы, работая со всеми объектами через общий интерфейс компонентов.

В качестве примера можно рассмотреть добавление новой функциональности объекту без изменения его класса. Основной класс бизнес-логики считывает и записывает чистые данные напрямую из файлов и в них. При помощи патерна «декоратор» создается класс-обертка, дополняющий функционал основного класса, что позволяет осуществлять шифрование данных.

Интерфейс, задающий базовые операции чтения и записи данных:

```
public interface DataSource {

  void writeData(String data);

  String readData();
}
```

Основной класс, реализующий чтение и запись данных:

```
import java.io.*;

public class FileDataSource implements DataSource {
  private String name;

  public FileDataSource(String name) {
        this.name = name;
  }

  @Override
  public void writeData(String data) {

        // ... код записи данных...
  }

  @Override
  public String readData() {

        // ... код чтения данных...
  }
}
```

Базовый декоратор:

```
public class DataSourceDecorator implements DataSource {
  private DataSource wrappee;
  DataSourceDecorator(DataSource source) {
        this.wrappee = source;
  }

  @Override
  public void writeData(String data) {
        wrappee.writeData(data);
  }
```

```
    @Override
    public String readData() {
        return wrappee.readData();
    }
}
```

Декоратор шифрования данных:

```
public class EncryptionDecorator extends DataSourceDecorator
{

    public EncryptionDecorator(DataSource source) {
        super(source);
    }

    @Override
    public void writeData(String data) {
        super.writeData(encode(data));
    }

    @Override
    public String readData() {
        return decode(super.readData());
    }

    private String encode(String data) {
        // ... код...
    }

    private String decode(String data) {
        // ... код...
    }
}

public class Main {
    public static void main(String[] args) {
        String stringRecords = "text";
        DataSourceDecorator encoded = new EncryptionDecora-
tor(
                        new FileDataSource("out/Out-
putDemo.txt"));
        encoded.writeData(stringRecords);
        DataSource plain = new FileDataSource("out/Out-
putDemo.txt");
    System.out.println(stringRecords);
    System.out.println(plain.readData());
    System.out.println(encoded.readData());
    }
}
```

К преимуществам данного подхода можно отнести то, что данный подход обладает большей гибкостью, чем применение наследования. Также применение декораторов позволяет добавлять различную функциональность во время выполнения программы. К недостаткам можно отнести сложность конфигурирования многократно обернутых объектов.

8.3 Практическая часть

1 Изучить описание к лабораторной работе, ознакомиться с примерами реализации представленных паттернов проектирования.

2 Преобразовать реализацию предметной области, выбранной для лабораторной работы № 1, с использованием паттернов «абстрактная фабрика» / «фабричный метод» и «декоратор».

8.4 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Иллюстрация решения задачи.
5 Выводы.
6 Список использованных источников.

8.5 Контрольные вопросы

1 Что такое паттерн проектирования?
2 Какие паттерны проектирования бывают?
3 Для чего применяется паттерн «абстрактная фабрика»?
4 Как реализуется паттерн «декоратор»?
5 Что общего и в чем отличие паттернов «фабричный метод» и «абстрактная фабрика»?

8.6 Список использованных источников

1 Head First. Паттерны проектирования. Обновленное юбилейное издание / Э. Фримен [и др.]. – СПб. : Питер, 2018. – 656 с.: ил.

2 Паттерны объектно-ориентированного проектирования / Э. Гамма [и др.]. – СПб. : Питер, 2020. – 448 с. : ил.

ЛАБОРАТОРНАЯ РАБОТА № 9.
ГРАФИЧЕСКИЙ ИНТЕРФЕЙС. БИБЛИОТЕКА JAVAFX

9.1 Цель работы

Ознакомиться с принципами построения графического интерфейса программ. Научиться разрабатывать программы с использованием библиотеки *JavaFX*.

9.2 Теоретическая часть

Введение

JavaFX – это библиотека для создания на языке *Java*, является кроссплатформенной. Поэтому мы можем создавать оконные приложения под *Windows*, *Linux* и *Mac*. До версии *Java* 8 библиотека *JavaFX* была доступна отдельно. В версиях *Java* с 8 по 10 включительно библиотека *JavaFX* входит в состав стандартной библиотеки, поэтому ее дополнительно устанавливать не нужно. Начиная с *Java* 11 *JavaFX* поставляется отдельно в виде набора модулей. Программный интерфейс *JavaFX API* версии 2.0 содержит пакеты (рисунок 9.1).

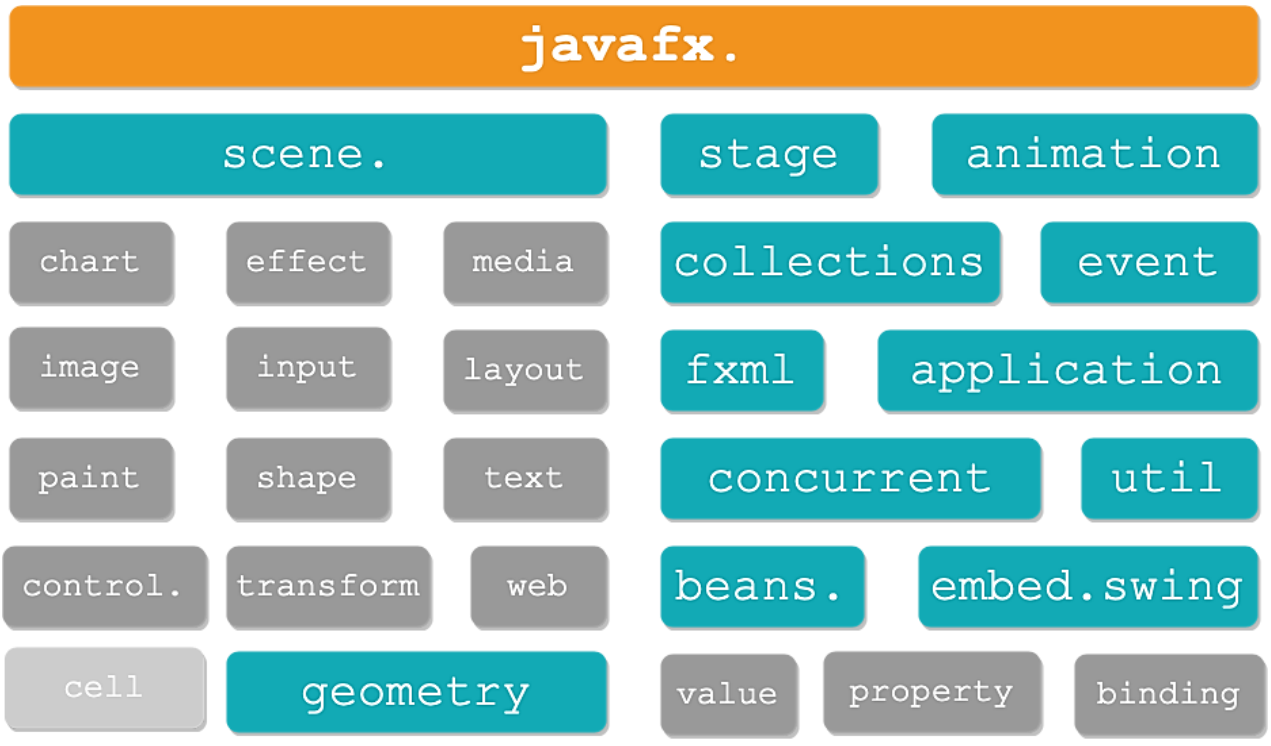

Рисунок 9.1 – Пакеты библиотеки *JavaFX*

JavaFX включает в себя набор классов и интерфейсов, который определяет современный графический интерфейс пользователя (*GUI*). *JavaFX* предоставляет разнообразный набор элементов управления, таких как кнопки, переключатели, метки, флажки, гиперссылки, списки, таблицы, панели, деревья, меню, окна, текстовые поля, индикаторы, разделители и ползунки, которые можно настроить практически для любого приложения. Кроме того, для повышения визуальной

привлекательности элементов управления можно использовать эффекты, преобразования и анимацию.

Приложение *JavaFX* создает экземпляр класса, унаследованного от класса из пакета *javafx.application.Application*. Объект этого класса проходит ряд этапов жизненного цикла (рисунок 9.2). Все эти этапы представлены методами класса *Application*, которые вызываются автоматически средой *JavaFX*.

Рисунок 9.2 – Жизненный цикл *JavaFX*-приложения

Стоит отметить, что метод *init()* не должен использоваться для создания графического интерфейса или отдельных его частей.

В итоге весь процесс работы приложения выглядит следующим образом:

1 Запускается исполняемая среда *JavaFX* (если она не запущена).

2 Вызывается конструктор класса, который расширяет класс *Application*, тем самым создавая экземпляр данного класса.

3 *JavaFX* вызывает метод *init()*.

4 Вызывается метод *start(javafx.stage.Stage)*, в который передается созданный объект *Stage*. Таким образом, приложение начинает работать.

5 Далее среда ожидает, пока либо в приложении не будет вызван программным способом метод *Platform.exit()*, либо не будет закрыто окно программы.

6 После завершения работы приложения *JavaFX* вызывает метод *stop()*.

При создании проекта на *JavaFX* среда разработки приложений (*IDE*) автоматически создает класс *Main* с методами *main()*, *launch()* и *start()*, включающими загрузку файла описания графического интерфейса сцены из текущего доступного ресурса, установку названия сцены и запуск цикла, получение сообщений от пользователя. Основная часть реализации графического интерфейса находится в методе *start()*.

Графическая сцена *JavaFX* – это древовидная структура данных, которая упорядочивает (группирует) графические объекты для облегчения логического представления. Это также позволяет графическому движку отображать объекты наиболее эффективным образом, полностью или частично пропуская объекты, которые не будут видны на конечном изображении. На рисунке 9.3 показан пример архитектуры сцены *JavaFX*.

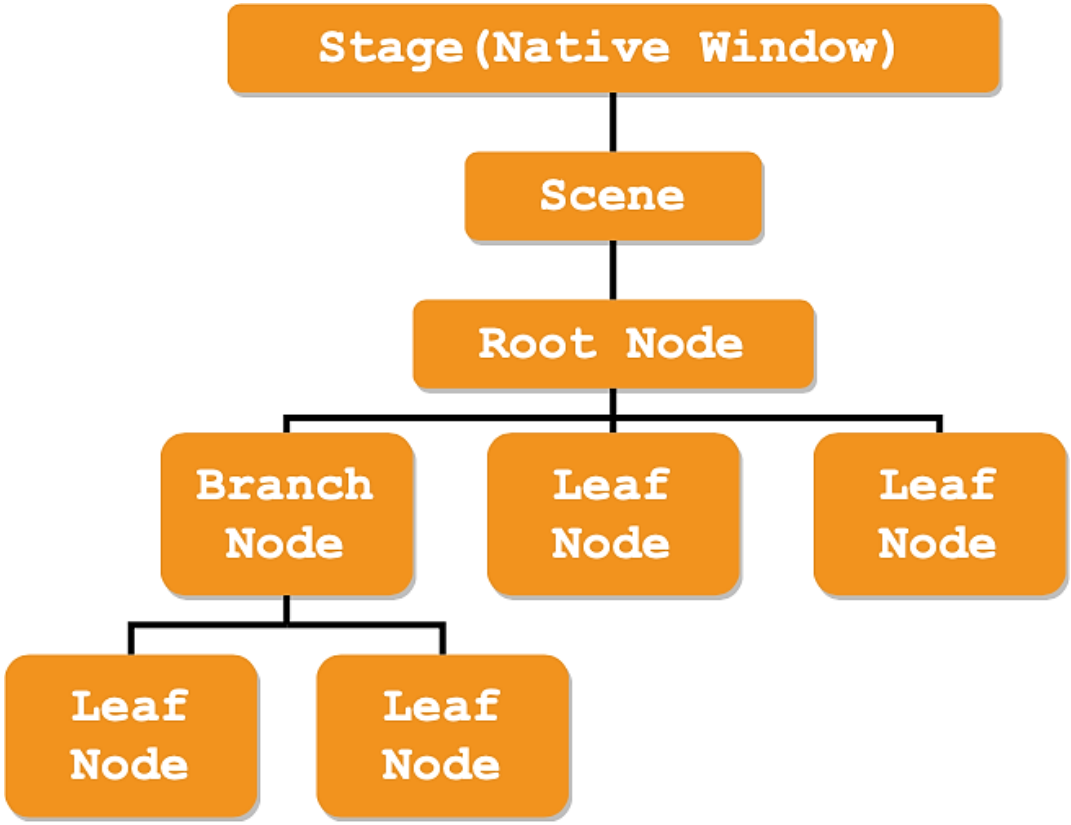

Рисунок 9.3 – Архитектура сцены *JavaFX*

На самом верху архитектуры находится контейнер (*Stage*). *Stage* – это *JavaFX*-представление собственного окна со стандартными кнопками: закрыть, свернуть, развернуть, операционная система. В любой момент времени к *Stage* может быть прикреплена одна *Scene*.

Все элементы в *JavaFX* представлены как объекты узлов (*Node*). Существует три типа узлов: корень (*Root*), ветвь (*Branch*) и лист (*Leaf*). Корневой узел (*Root*) – это единственный узел, который не имеет родителя и непосредственно содержится в сцене, что видно на рисунке выше. Разница между ветвью и листом заключается в том, что у листового узла нет потомков.

В графе сцены многие свойства родительского узла являются общими для дочерних узлов. Например, преобразование или событие, примененное к родительскому узлу, также будет рекурсивно применено к его дочерним узлам. Таким образом, сложную иерархию узлов можно рассматривать как единый узел для упрощения модели программирования.

На рисунке 9.4 представлена графическая сцена для *Hello World*.

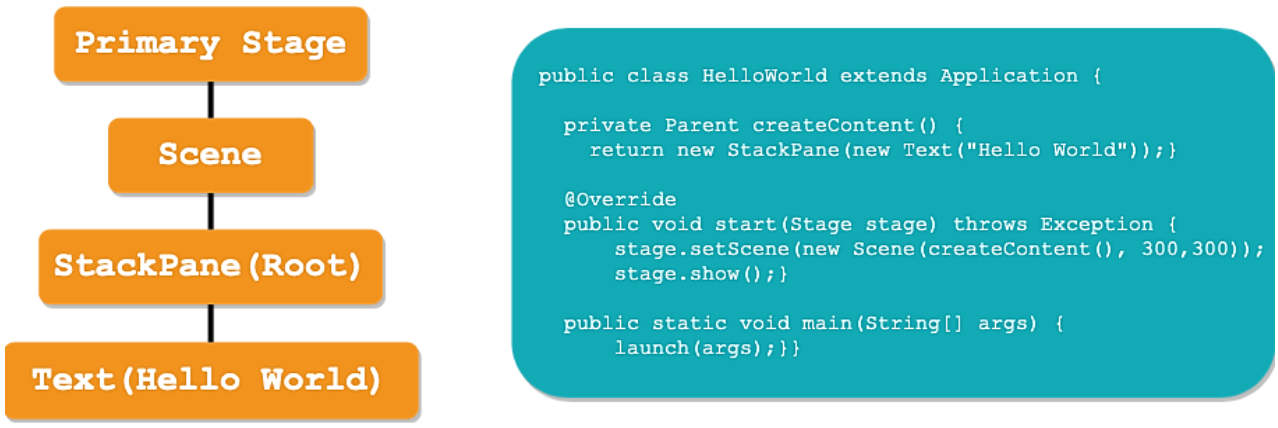

Рисунок 9.4 – Пример графической сцены для *Hello World*

Одна из возможных реализаций, которая создаст график сцены, соответствующий приведенному выше рисунку 9.4. Результат запуска кода показан на рисунке 9.5.

Рисунок 9.5 – Графическое окно *Hello World*

9.2.1 Обработка событий

При взаимодействии пользователя с окном происходят события. В ответ на события генерируются определенные сигналы. События являются важнейшей составляющей приложения с графическим интерфейсом, поэтому необходимо знать, как назначить обработчик события, как удалить обработчик, а также уметь правильно обработать событие.

9.2.2 *Класс Event*

Класс *Event* является базовым классом для всех классов событий в *JavaFX*. Инструкция импорта класса:

```
// Модуль javafx.base
import javafx.event.Event;
```

Иерархия наследования представлена на рисунке 9.6.

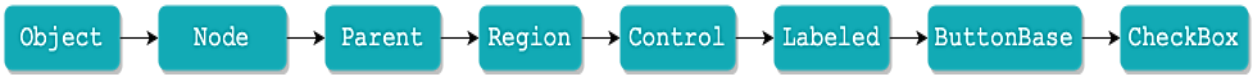

Рисунок 9.6 – Иерархия наследования *Event*

Конструкторы класса:

```
Event(EventType<? extends Event> eventType)
Event(Object source, EventTarget target,
     EventType‹ ? extends Event> eventType)
```

Приведем основные методы:

1 *getEventType()* – возвращает тип события. Пример метода:

```
public EventType<? extends Event> getEventType()
```

2 *getSource()* – возвращает ссылку на объект, который обрабатывает событие. Например, если обработчик нажатия кнопки зарегистрирован для объекта контейнера, то при нажатии кнопки внутри обработчика мы получим ссылку на контейнер, а не на кнопку. Метод наследуется из класса *EventObject*. Пример метода:

```
public Object getSource()
```

3 *getTarget()* – возвращает ссылку на цель события. Пример метода:

```
public EventTarget getTarget()
```

4 *fireEvent()* – позволяет генерировать событие. Пример метода:

```
public static void fireEvent(EventTarget eventTarget,
                             Event event)
```

5 *consume()* – устанавливает флаг прерывания распространения события. Пример метода:

```
public void consume()
```

6 *isConsumed()* – возвращает статус флага прерывания распространения события. Пример метода:

```
public boolean isConsumed()
```

7 *clone()* – создает копию объекта события. Пример метода:

```
public Object clone()
```

Класс *Event* содержит также следующие статические константы:
1 *NULL_SOURCE_TARGET* – обозначает неизвестную цель события. Пример:

```
public static final EventTarget NULL_SOURCE_TARGET
```

2 *ANY* – базовый тип для всех типов событий. Пример:

```
public static final EventType<Event> ANY
```

Каждый класс события, помимо основных типов, содержит тип *ANY*, который используется для перехвата любого события. Если мы хотим перехватывать все события ввода: события клавиатуры, мыши, сенсорного экрана и другие, то при назначении обработчика можно указать базовый класс *InputEvent* и его тип *ANY*. Пример обработки всех событий ввода:

```
scene.addEventFilter(InputEvent.ANY, event -> {
  System.out.println("Событие " + event.getEventType()});
```

9.2.3 Назначение обработчиков событий

Назначить обработчик события можно несколькими способами:
1 С помощью методов, названия которых начинаются с префикса *setOn*. Например, с помощью метода *setOnAction()*:

```
Button button = new Button("Кнопка");
button.setOnAction(event -> {
     System.out.println("Нажата кнопка");});
```

Такие методы изменяют значение соответствующего *javafx*-свойства. Например, метод *setOnAction()* задает значение свойства *onAction*. Получить текущее значение можно с помощью метода, название которого начинается с приставки *getOn*, например *getOnAction()*. Если обработчик не был назначен, то метод вернет значение *null*. Получить ссылку на свойство *onAction* позволяет метод *onActionProperty()*.
2 С помощью методов *addEventFilter()* и *addEventHandler()*. Обработчик, назначенный методом *addEventFilter()*, вызывается перед обработчиком, назначенным методом *addEventHandler()*. Обратите внимание: эти методы позволяют зарегистрировать несколько обработчиков. Примеры:

```
public final <T extends Event> void addEventFilter(
EventType<T> eventType, EventHandler<? super T>
    eventFilter)
public final <T extends Event> void addEventHandler(
EventType<T> eventType, EventHandler<? super T>
    eventHandler)
```

Пример назначения обработчика нажатия кнопки с помощью метода *addEventHandler()*:

```
button.addEventHandler(ActionEvent.ACTION, event -> { Sys-
tem.out.println ("Нажата кнопка")});
```

Если внутри обработчика, назначенного с помощью метода *addEventFilter()*, вызвать метод *consume()* через объект события, то обработчик, назначенный с помощью метода *addEventHandler()*, вызван не будет. Пример:

```
button.addEventFilter(ActionEvent.ACTION, event -> {
   System.out.println("Нажата кнопка (Filter)");
   event.consume();}); // Прерывание всплывания события
button.addEventHandler(ActionEvent.ACTION, event -> {
   // Обработчик вызван не будет
   System.out.println("Нажата кнопка (Handler)");});
```

3 Изменения различных значений можно отслеживать с помощью *javafx*-свойств. В этом случае для назначения обработчика используется метод *addlistener()*. Пример:

```
public void addlistener(ChangeListener<? super T> listener)
```

Интерфейс *ChangeListener<T>* содержит объявление метода *changed()*:

```
public void changed(ObservableValue<? extends T>
       observable,   T oldValue, T newValue)
```

Например, отследить изменение положения окна по оси *x* можно следующим образом:

```
stage.xProperty().addListener((obj, oldValue, newValue) ->
   System.out.println("Изменение X " + newValue);});
```

В метод регистрации обработчика мы можем передать:

1 Объект класса, реализующего функциональный интерфейс *EventHandler<T extends Event>*. Интерфейс содержит объявление метода *handle()*:

```
public void handle(T event)
```

2 Анонимный вложенный класс:

```
button1.setOnAction(new EventHandler<ActionEvent>(){
    @Override
    public void handle(ActionEvent event) {
      System.out.println ("Нажата кнопка 1");}};
```

3 Ссылку на метод:

```
button.setOnAction(this::onClick);
```

4 Лямбда-выражение:

```
button.setOnAction(event -> {
System.out.println("Нажата кнопка")};
```

Пример назначения обработчиков нажатия кнопки разными способами приведен ниже.

```
package application;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javax.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class Main extends Application {

   public static void main(String[] args) {
     Application.launch(args);)
@Override
public void start(Stage stage) throws Exception {
  VBox root = new VBox(20. 0);
  root.setAlignment(Pos.CENTER) ;
  Button button1 = new Button("Кнопка 1");
  Button button2 = new Button("Кнопка 2");
  Button button3 = new Button("Кнопка 3");
  Button button4 = new Button("Кнопка 4");
  Button buttons = new Button("Кнопка 5");
  root.getChildren().addAll(button1, button2, button3,
                            button4, button5);
  button1.setOnAction(new EventHandler<ActionEvent>(){
   @Override
   public void handle(ActionEvent event) {
     System.out.println("Нажата кнопка 1");}});
  button2.setOnAction(this::onClick);
  button3.setOnAction(event -> {
     System.out.println("Нажата кнопка 3");
  button4.addEventFilter(ActionEvent.ACTION, event -> {
    System.out.println("Нажата кнопка 4 (Filter)"); });
  button4.addEventHandler(ActionEvent.ACTION, event -> {
     System.out.println("Нажата кнопка 4 (Handler)");});
  button5.addEventFilter(ActionEvent.ACTION, event -> {
     System.out.println(Нажата кнопка 5 (Filter)");
     event.consume();// Прерывание всплывания события
  button5.addEventHandler(ActionEvent.ACTION, event -> {
     // Обработчик вызван не будет
     System.out.println("Нажата кнопка 5 (Handler)");});
  Scene scene = new Scene(root, 500.0, 500.0);
  stage.setTitle("Назначение обработчиков");
```

```
  stage.setScene(scene);
  stage.xProperty().addListener((obj,oldValue,newValue)->{
     System.out.println("Изменение X " + newValue);});
  stage.show();}
private void onClick(ActionEvent event) {
System.out.println("Нажата кнопка 2");}}
```

9.2.4 Блокировка и удаление обработчика

Если необходимо временно блокировать обработчики событий для узла, то достаточно сделать узел неактивным с помощью метода *setDisable()* из класса *Node*. Пример:

```
button.setDisable(true);
```

Если обработчик назначался с помощью метода, название которого начинается с приставки *setOn*, то для удаления обработчика можно вызвать тот же метод, но вместо ссылки передать ему значение *null*. Пример:

```
button.setOnAction(null);
```

Если обработчик назначался с помощью метода *addEventHandler()*, то для удаления обработчика нужно воспользоваться методом *removeEventHandler()*. Пример метода:

```
public final <T extends Event> void removeEventHandler(
    EventType<T> eventType, EventHandler<? super T>
          eventHandler)
```

В первом параметре метод принимает тип события, а во втором – ссылку на обработчик. Чтобы правильно удалить обработчик, необходимо указать ссылку на тот же обработчик, который был и при назначении. Лучше сохранить ссылку на обработчик в какой-либо переменной, а затем использовать ее и при назначении обработчика, и при удалении его. Пример сохранения ссылок в полях класса:

```
private EventHandler<ActionEvent> handler1 = this::onClick;
private EventHandler<ActionEvent> handler2 = event -> {
System.out.println ("Нажата кнопка 3");};
```

Пример назначения и удаления обработчика:

```
button2.addEventHandler(ActionEvent.ACTION, handler1);
button2.removeEventHandler(ActionEvent.ACTION, handler1);
```

Если обработчик назначался с помощью метода *addEventFilter()*, то для удаления обработчика нужно воспользоваться методом *removeEventFilter()*. Пример:

```
public final <T extends Event> void removeEventFilter(
EventType<T> eventType,EventHandler<? super T> event Filter)
```

Пример назначения и удаления обработчика:

```
button.addEventFilter(ActionEvent.ACTION, handler);
button.removeEventFilter(ActionEvent.ACTION, handler);
```

Если обработчик назначался с помощью метода *addListener()*, то для удаления обработчика нужно воспользоваться методом *removeListener()*. Пример:

```
public void removeListener(ChangeListener <? super T>
  listener)
```

Не забудьте предварительно сохранить ссылку на обработчик, например так:

```
private ChangeListener<? super Number>
  listener = (obj, oldValue, newValue) -> {
    System.out.println("Изменение X " + newValue);};
```

Пример назначения и удаления обработчика:

```
stage.xProperty().addListener(listener);
stage.xProperty().removelistener(listener);
```

9.2.5 Последовательность вызова обработчиков

Последовательность вызова при нажатии кнопки будет выглядеть так:

```
button - Filter
button - Handler
button - setOnAction
```

Итак, вначале выполняется обработчик, назначенный с помощью метода *addEventFilter()*. Внутри этого обработчика мы можем фильтровать событие. Если необходимо прервать распространение события, то внутри обработчика через объект события нужно вызвать метод *consume()*. Пример:

```
button.addEventFilter(ActionEvent.ACTION, event-> {
System.out.println("button - Filter");
event.consume();}
```

После вызова метода *consume()* последовательность будет такой:

```
button - Filter
```

Другие обработчики вызваны не будут. Если внутри обработчика метод *consume()* не вызвать, то распространение события продолжится.

Методы *addEventFilter()* и *addEventHandler()* позволяют зарегистрировать несколько обработчиков. Последовательность вызова обработчиков в этом случае будет такой:

```
button - Filter
button - Filter2
button - Handler
```

```
button - Handler2
button - setOnAction
```

Методы *addEventFilter()* и *addEventHandler()* имеют все узлы, а также объекты сцены и окна. Если назначить обработчик нажатия кнопки для объекта контейнера, то можно перехватить нажатие любых кнопок внутри контейнера. Если назначить обработчик для объектов сцены или окна, то внутри обработчика будут доступны события нажатий вообще всех кнопок. Предположим, что мы назначили обработчики сразу для кнопки, контейнера, сцены и окна. Последовательность событий в этом случае будет такой:

```
stage  - Filter
scene  - Filter
root   - Filter
button - Filter
button - Handler
button - setOnAction
root   - Handler
scene  - Handler
stage  - Handler
```

Итак, событие вначале движется от объекта окна до источника, вызвавшего событие, а затем обратно к объекту окна. Таким образом событие проходит по всей древовидной иерархии от объекта окна до узла и обратно. Если на какой-либо стадии внутри обработчика вызвать метод *consume()* через объект события, то распространение события будет остановлено.

9.2.6 Генерация события из программы

В некоторых случаях необходимо вызвать событие из программы. Например, при заполнении последнего текстового поля и нажатии клавиши <*Enter*> можно имитировать нажатие кнопки и тем самым запустить обработчик этого события. Выполнить генерацию события из программы позволяет статический метод *fireEvent()* из класса *Event*. Пример метода:

```
import javafx.event.Event;
public static void fireEvent(EventTarget eventTarget,
                             Event event)
```

В первом параметре указывается экземпляр класса, реализующего интерфейс *EventTarget*. Этот интерфейс реализуют все классы контейнеров и компонентов, а также классы сцены и окна, поэтому любой узел можно указать в качестве цели. Во втором параметре задается объект события. По нажатии одной кнопки генерируем событие для другой кнопки:

```
button2.setOnAction(event -> {
  System.out.println("button2");
  Event.fireEvent(button1, event);}
```

Для генерации события можно также воспользоваться методом *fireEvent()* из классов *Node* и *Window*. Пример метода:

```
public final void fireEvent (Event event)
```

Пример:

```
button.setOnAction(event -> {
  System.out.println("button");
  button.fireEvent (event);}
```

9.2.7 Элементы графического интерфейса пользователя

Элементы графического интерфейса пользователя платформы *JavaFX* представлены такими пакетами *JavaFX API*, как *javafx.scene.control, javafx.scene.chart, javafx.scene.image, javafx.scene.layout, javafx.scene.media, javafx.scene.shape, javafx.scene.text, javafx.scene.web* и *javafx.stage*.

Как писали выше, все элементы *GUI*-интерфейса являются объектами *Node* узлов графа сцены и характеризуются идентификатором, *CSS*-стилем, границами, визуальными эффектами, прозрачностью, трансформациями, обработчиками событий, состоянием, режимом наложения и участием в анимации.

Самыми основными пакетами в *JavaFX* являются *javafx.scene*, который содержит группу (*Group*) и сцену (*Scene*), и *javafx.stage*, который предоставляет *GUI*-элементы окон *Stage, Popup* и *FileChooser*.

Пакет *javafx.scene.control* предоставляет *GUI*-элементы, которые показаны на рисунке 9.7.

Также в *JavaFX* есть такие пакеты, как *javafx.scene.chart* (обеспечивает создание диаграмм), *javafx.scene.image* (для отображения изображения), *javafx.scene.layout* (предоставляет панели компоновки), *javafx.scene.media* (содержит *GUI*-компоненты медиаконтента), *javafx.scene.shape* (обеспечивает рисование геометрических форм), *javafx.scene.text* (для отображения текста), *javafx.scene.web (*обеспечивает отображение и редактирование *HTML*-контента). Элементы данных пакетов можно посмотреть на рисунке 9.8.

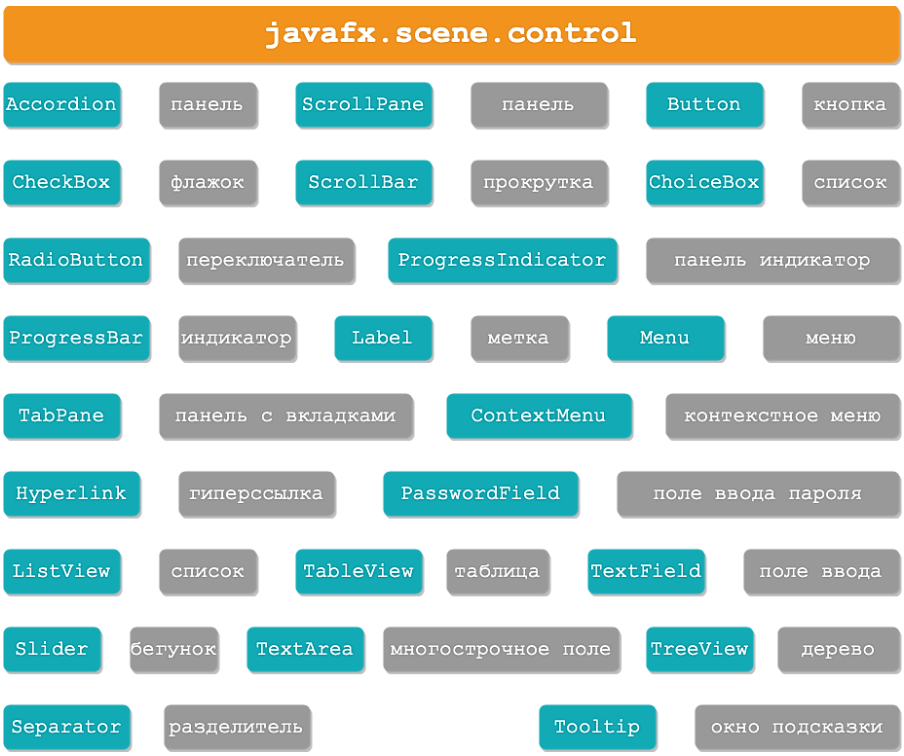

Рисунок 9.7 – Элементы пакета *javafx.scene.control*

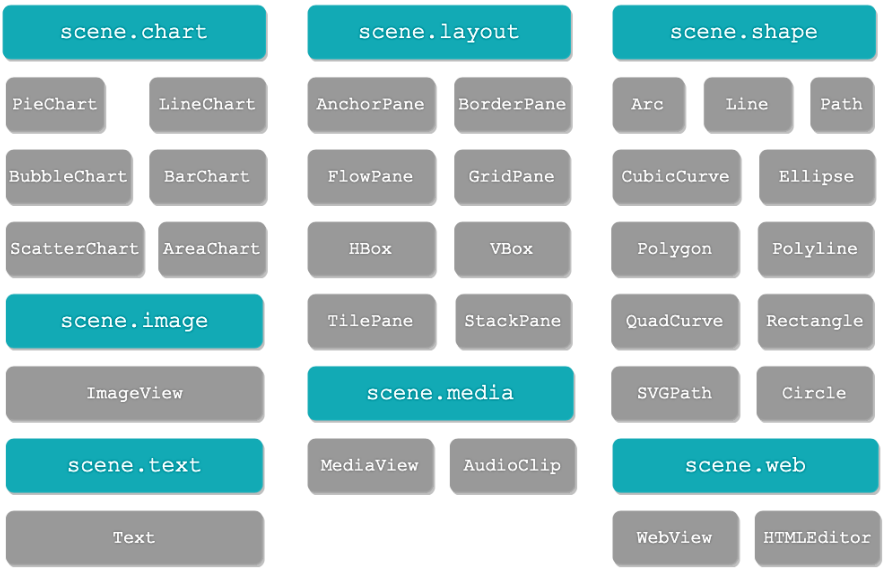

Рисунок 9.8 – Элементы пакета *javafx.scene.control*

Абстрактный класс *Labeled* является базовым для всех классов, компоненты которых содержат текстовую надпись. *ButtonBase* является базовым классом для кнопок, переключателей, флажков и гиперссылок.

9.2.8 Класс Button (кнопка)

Button является наиболее часто используемым компонентом. При нажатии кнопки внутри обработчика обычно выполняется какая-либо операция. Кнопка реализуется с помощью класса *Button*. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Button;
```

Иерархия наследования *Button* представлена на рисунке 9.9.

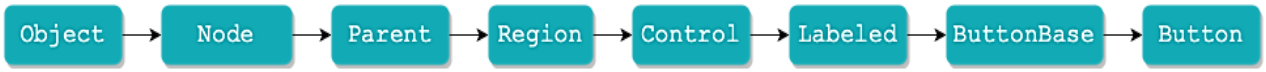

Рисунок 9.9 – Иерархия наследования *Button*

Конструкторы класса:

```
Button()
Button(String text)
Button(String text, Node graphic)
```

Пример обработки нажатия кнопки:

```
Button button = new Button("Нажми меня");
button.setOnAction(event -> {
    System.out.println("Нажата кнопка");});
```

Класс *Button* содержит следующие свойства:

1 *defaultButton* – если свойство содержит значение *true*, то кнопка будет обозначена как кнопка по умолчанию. Нажать кнопку по умолчанию можно с помощью клавиши <*Enter*>. Чтобы нажать обычную кнопку, находящуюся в фокусе ввода, нужно нажать клавишу <Пробел>. Методы доступа:

```
public final void setDefaultButton(boolean value)
public final boolean isDefaultButton()
public final BooleanProperty defaultButtonProperty()
```

2 *cancelButton* – если свойство содержит значение *true*, то кнопка будет обозначена как кнопка отмены. Нажать кнопку отмены можно с помощью клавиши <*Esc*>. Методы доступа:

```
public final void setCancelButton(boolean value)
public final boolean isCancelButton()
public final BooleanProperty cancelButtonProperty()
```

Обычно после нажатия кнопки приходится выполнять какую-либо длительную операцию. В этом случае, чтобы пользователь не нажимал кнопку повторно несколько раз, думая, что программа «зависла», нужно сделать кнопку

недоступной. Если узел недоступен, то он не принимает событий, а следовательно, при попытке нажать недоступную кнопку обработчик вызван не будет. Пример:

```
Button.button = new Button("Кнопка");
button.setOnAction(event -> {
  System.out.println("Нажата кнопка");
  button.setDisable(true);
  PauseTransition tr = new PauseTransition(
                    Duration.seconds(2.0));
  tr.setOnFinished(e -> {I
    System.out.println("Операция завершена");
    button1.setDisable(false);});
    tr.play();});
```

В этом примере мы воспользовались классом *PauseTransition* для имитации выполнения длительного процесса. После нажатия кнопки мы делаем ее недоступной, затем выполняем операцию, а после завершения выполнения операции выводим результат и делаем кнопку доступной.

9.2.9 Класс CheckBox (флажок)

CheckBox предназначен для включения или выключения каких-либо опций настроек программы пользователем и может иметь несколько состояний: установлен, сброшен и неопределенность. Флажок реализуется с помощью класса *CheckBox*. Инструкция импорта:

```
// Модуль javafx.controls
import javax.scene.control.CheckBox;
```

Иерархия наследования *CheckBox* представлена на рисунке 9.10.

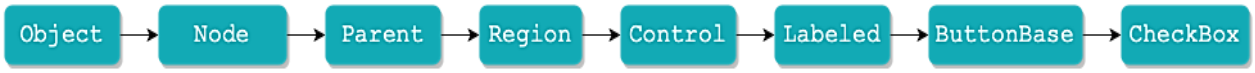

Рисунок 9.10 – Иерархия наследования *CheckBox*

Конструкторы класса:

```
CheckBox()
CheckBox(String text)
```

Управлять компонентом позволяют следующие свойства:

1 *selected* – содержит значение *true*, если флажок установлен. Методы доступа:

```
public final void setSelected(boolean value)
public final boolean isSelected()
public final BooleanProperty selectedProperty()
```

2 *indeterminate* – содержит значение *true*, если флажок находится в неопределенном состоянии, или значение *false* – в противном случае. В неопределенном состоянии внутри рамки отображается символ <<->>. Методы доступа:

```
public final void setIndeterminate(boolean value)
public final boolean isIndeterminate()
public final Boolean?property indeterminateProperty()
```

3 *allowIndeterminate* – если свойство имеет значение *true*, то пользователь может выбрать одно из трех состояний флажка (установлен, сброшен или не определен), а если *false* (значение по умолчанию), то только одно из двух (установлен или сброшен). Методы доступа:

```
public final void setAllowIndeterminate(boolean value)
public final boolean isAllowIndeterminate()
public final BooleanProperty allowIndeterminateProperty()
```

Пример использования *CheckBox* с тремя состояниями:

```
CheckBox checkBox = new CheckBox("Текст рядом с флажком");
checkBox.setAllowIndeterminate(true);
checkBox.setIndeterminate(true);
checkBox.selectedProperty().addListener((obj, oldValue,
        newValue) -> {
System.out.println("checkBox selected = + newValue); ));
checkBox.indeterminateProperty().addListener(obj,oldValue,
        newValue) -> {
System.out.println("checkBox indeterminate = " +
        newValue);})})
```

9.2.10 Класс ToolBar (панель инструментов)

Панели инструментов предназначены для горизонтального или вертикального выравнивания кнопок с часто используемыми командами, а также любых других узлов. Очень часто кнопки на панели инструментов дублируют пункты главного меню приложения. Создать панель инструментов позволяет класс *ToolBar*. Если компоненты не помещаются на панели, то отобразится кнопка, с помощью которой можно выбрать скрытые компоненты. Инструкция импорта:

```
// Модуль javafx.controls
import javax.scene.control.ToolBar;
```

Иерархия наследования *ToolBar* представлена на рисунке 9.11.

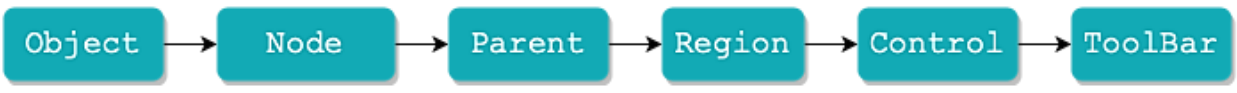

Рисунок 9.11 – Иерархия наследования ToolBar

Конструкторы класса:

```
ToolBar()
ToolBar(Node … items)
```

Первый конструктор создает пустую панель инструментов, а второй – позволяет сразу добавить компоненты на панель. Чтобы добавить компоненты после создания объекта, следует воспользоваться методом *getItems()*. Пример метода:

```
public final ObservableList<Node> getItems()
```

Пример создания панели инструментов с горизонтальной ориентацией и добавления десяти кнопок на панель:

```
ToolBar toolBar = new ToolBar();
toolBar.setOrientation(Orientation.HORIZONTAL);
for (int i= l; i< 11; i++) {
    Button = new Button("Кнопка" + i);
    toolBar.getItems().add(b);}
```

Для прикрепления панели инструментов к какой-либо боковой части окна удобно воспользоваться контейнером *BorderPane*:

```
BorderPane pane = new BorderPane();
pane.setTop(toolBar);
```

По умолчанию создается панель с горизонтальной ориентацией. Чтобы задать вертикальную ориентацию, следует присвоить константу *VERTICAL* из перечисления *Orientation* свойству *orientation*. Методы доступа:

```
import javafx.geometry.Orientation;
public final void setOrientation(Orientation value)
public final Orientation getOrientation()
public final ObjectProperty<Orientation> orientationProperty()
```

Пример:

```
toolBar.setOrientation(Orientation.VERTICAL);
```

9.2.11 Класс TextInputControl

Абстрактный класс *TextInputControl* является базовым классом для текстовых полей. Инструкция импорта:

```
// Модуль javafx.controls
import javax.scene.control.TextInputControl;
```

Иерархия наследования *TextInputControl* представлена на рисунке 9.12.

Рисунок 9.12 – Иерархия наследования *TextInputControl*

Основные свойства и методы. Приведем основные свойства класса *TextInputControl*.

1 *text* – содержит текст, отображаемый в текстовом поле. Методы доступа:

```
public final void setText(String value)
public final String getText()
public final StringProperty textProperty()
```

Пример вставки текста в поле из программы:

```
TextField text = new TextField();
text.setText("Java");
```

2 *promptText* – задает текст подсказки пользователю, который будет выводиться в поле, когда оно не содержит значения и находится вне фокуса ввода. Методы доступа:

```
public final void setPromptText(String value)
public final String getPromptText()
public final StringProperty promptText Property()
```

Пример:

```
text.setPromptText("Введите фамилию");
```

3 *font* – задает характеристики шрифта. Методы доступа:

```
public final void setFont(Font value)
public final Font getFont()
public final ObjectProperty<Font> fontProperty()
```

Пример:

```
text.setFont(Font.font("Verdana",18.0));
```

4 *length* – содержит количество символов в тексте. Методы доступа:

```
public final int getLength()
public final ReadOnlyIntegerProperty lengthProperty()
```

5 *editable* – если свойство имеет значение *false*, то поле будет доступно только для чтения. Значение по умолчанию: *true* (пользователь может редактировать содержимое поля).

Методы доступа:

```
public final void setEditable(boolean value)
public final boolean isEditable()
public final BooleanProperty editableProperty()
```

Приведем основные методы класса *TextInputControl*:

1 *appendText()* – добавляет текст в конец существующего значения. Пример метода:

```
public void appendText(String text)
```

2 *insertText()* – вставляет текст в указанную позицию. Пример метода:

```
public void insertText(int index, String text)
```

Вставим текст в начало поля:

```
text.insertText(0, "javafx");
```

3 *clear()* – очищает текстовое поле. Пример метода:

```
public void clear()
```

4 *deletePreviousChar()* – удаляет символ, расположенный слева от текстового курсора. Пример метода:

```
public boolean deletePreviousChar()
```

5 *deleteNextChar()* – удаляет символ, расположенный справа от текстового курсора. Пример метода:

```
public boolean deleteNextChar()
```

6 *commitValue()* – применяет ввод и отправляет данные на преобразование типов. Эквивалентно нажатию пользователем клавиши <*Enter*>. Пример метода:

```
public final void commitValue ()
```

7 *cancelEdit()* – отменяет ввод и вставляет в поле предыдущее значение. Эквивалентно нажатию пользователем клавиши <*Esc*>. Пример метода:

```
public final void cancelEdit()
```

Текстовые поля по умолчанию реализуют функционал стандартных клавиш – например: <*Delete*> (удалить следующий символ), <*Backspace*> (удалить предыдущий символ) и др.

9.2.12 Класс TextField (однострочное текстовое поле)

Класс *TextField* реализует однострочное текстовое поле, предназначенное для ввода и редактирования текста небольшого объема. Поле по умолчанию поддерживает стандартные комбинации клавиш быстрого доступа, работу с буфером обмена и многое другое. Инструкция импорта:

```
// Модуль javax.controls
import javax.scene.control.TextField;
```

Иерархия наследования *TextField* представлена на рисунке 9.13.

Рисунок 9.13 – Иерархия наследования *TextField*

Конструкторы класса:

```
TextField()
TextField(String text)
```

Первый конструктор создает пустое поле, а второй конструктор позволяет указать текст, который будет отображаться в поле. Создадим текстовое поле, назначим текст подсказки и клавишу быстрого доступа:

```
TextField text = new TextField();
text.setPromptText("Введите слово");
Label label = new Label("Имя");
label.setLabelfor(text);
label.setMnemonicParsing(true);
```

Обратите внимание: так как текстовое поле не имеет надписи, назначение клавиши быстрого доступа осуществляется с помощью отдельного компонента *Label*. Связь между надписью и текстовым полем задается с помощью метода *setLabelFor()*.

Получим значение поля после нажатия кнопки:

```
Button button = new Button("Получить значение");
button.setonAction(event -> {
  System.out.println(text.getText());});
```

Для получения значения поля можно также воспользоваться методом *getCharacters()* из класса *TextField*. Пример метода:

```
public Charsequence getCharacters()
```

Пример:

```
System.out.println(text.getCharacters());
```

Класс *TextField* наследует все методы из класса *TextInputControl*, а также из других базовых классов и добавляет следующие свойства (не забывайте смотреть на иерархию наследования):

1 *alignment* – задает выравнивание текста внутри свободной области поля. В качестве значения указываются следующие константы из перечисления *javafx*. Значение по умолчанию: *CENTER_LEFT*. Методы доступа:

```
public final void setAlignment(Pos value)
public final Pos getAlignment()
public final ObjectProperty<Pos> alignmentProperty()
```

Пример выравнивания текста по правому краю поля:

```
TextField text = new TextField("Текст");
text.setAlignment(Pos.CENTER_RIGHT);
```

2 *prefColumnCount* – значение этого свойства используется для вычисления предпочтительной ширины поля. Методы доступа:

```
public final void setPrefColumnCount(int value)
public final int getPrefColumnCount()
public final IntegerProperty prefColumnCountProperty()
```

Значение по умолчанию хранится в константе *DEFAULT_PREF_COLUMN_COUNT*. Пример:

```
public static final int DEFAULT_PREF_COLUMN_COUNT
```

Выведем значение константы:

```
System.out.println(TextField.DEFAULT_PREF_COLUMN_COUNT);
```

3 *onAction* – позволяет назначить обработчик события нажатия пользователем клавиши <*Enter*>. Методы доступа:

```
public final void setOnAction(EventHandler<ActionEvent> value)
public final EventHandler<ActionEvent> getOnAction()
public final ObjectProperty<EventHandler<ActionEvent>>
             onActionProperty()
```

9.2.13 Класс PasswordField (поле для ввода пароля)

Класс *PasswordField* расширяет класс *TextField* и реализует однострочное текстовое поле, предназначенное для ввода пароля. При вводе пароля в поле отображаются черные кружки, а не текст. Инструкция импорта:

```
// Модуль javafx. controls
import javafx.scene.control.PasswordField;
```

Иерархия наследования *PasswordField* представлена на рисунке 9.14.

Рисунок 9.14 – Иерархия наследования *PasswordField*

Конструктор класса:

```
PasswordField()
```

Пример создания поля и назначения клавиши быстрого доступа:

```
PasswordField passwordField = new PasswordField();
passwordField.setPromptText("Введите пароль");
Label label = new Label("Пароль");
label.setMnemonicParsing(true);
label.setLabelFor(passwordField);
```

Пример получения значения после нажатия кнопки:

```
Button button = new Button("Получить значения");
pane.getChildren().add(button);
button.setOnAction(event -> {
   System.out.println(passwordField.getText());});
```

Класс *PasswordField* наследует все методы из базовых классов и не добавляет никаких собственных методов. Однако он переопределяет методы *cut()* и *copy()*. Эти методы в классе *PasswordField* ничего не делают.

9.2.14 Класс Dialog<R> (диалоговое окно)

Диалоговые окна предназначены для информирования пользователя, а также для получения данных от пользователя. В большинстве случаев диалоговые окна являются модальными (блокирующими все окна приложения или только родительское окно) и отображаются на непродолжительный промежуток времени.

Класс *Dialog<R>* позволяет создать пользовательское диалоговое окно. По умолчанию окно выводится с рамкой и заголовком, в котором расположены меню и кнопка <Закрыть>. Инструкция импорта:

```
// Модуль javafx.controls
import javax.scene.control.Dialog;
```

Создание и отображение диалогового окна. Конструктор класса *Dialog<R>*:

```
Dialog()
```

Создадим модальное диалоговое окно с сообщением и кнопкой (<*OK*>) внутри обработчика нажатия кнопки.

```
Button button = new Button ("Открыть окно");
button.setOnAction(event -> {
  Dialog<ButtonType> dialog = new Dialog<ButtonType>();
  dialog.setTitle("Заголовок окна");
  dialog.setHeaderText("Текст заголовка сообщения");
  dialog.setGraphic(new ImageView(createImage (20, 20,
                                  Color.BLUE)));
  dialog.setContentText("Текст сообщения");
  dialog.getDialogPane().getButtonTypes().
                                 add(ButtonType.OK);
  dialog.show();});
```

Рассмотрим основные свойства из класса *Dialog<R>*:

1 *title* – задает текст, выводимый в заголовке диалогового окна. Методы доступа:

```
public final void setTitle(String title)
public final String getTitle()
public final StringProperty titleProperty()
```

2 *headerText* – задает текст заголовка сообщения. Если свойство имеет значение *nul1*, то заголовок сообщения не отображается. Методы доступа:

```
public final void setHeaderText(String headerText)
public final String getHeaderText()
public final StringProperty headerTextProperty()
```

3 *contentText* – задает текст сообщения. Методы доступа:

```
public final void setContentText(String contentText)
public final String getContenText()
public final StringProperty contentTextProperty()
```

4 *graphic* – позволяет отобразить узел справа от текста заголовка сообщения или слева от текста сообщения, если заголовок сообщения не отображается. Методы доступа:

```
public final void setGraphic(Node graphic)
public final Node getGraphic()
public final ObjectProperty<Node> graphicProperty()
```

Основные методы класса *Dialog<R>*:

1 *show()* – отображает диалоговое окно на экране без ожидания действия пользователя (неблокирующий диалог). Используется, когда не нужно определять, какая кнопка нажата пользователем, или не надо получать данные из диалогового окна. Пример метода:

```
public final void show()
```

2 *showAndWait()* – отображает диалоговое окно и ожидает действия пользователя (блокирующий диалог). Используется, когда нужно получить данные из диалогового окна. Пример метода:

```
import java.util.Optional;
public final Optional<R> showAndWait()
```

3 *hide()* и *close()* – скрывают диалоговое окно. Примеры методов:

```
public final void hide()
public final void close()
```

4 *initModality()* – указывает, является ли диалоговое окно модальным (блокирующим другие окна). Пример метода:

```
import javafx.stage.Modality;
public final void initModality(Modality modality)
```

В качестве параметра могут быть указаны следующие константы из перечисления *Modality*:

– *NONE* – окно не является модальным;

– *WINDOW_MODAL* – окно блокирует только родительские окна в пределах иерархии. При использовании этого значения необходимо указать ссылку на родительское окно с помощью метода *initOwner()*;

– *APPLICATION_MODAL* – окно блокирует все окна приложения (значение по умолчанию).

5 *getModality()* – позволяет узнать, является ли окно модальным. Пример метода:

```
public final Modality getModality()
```

6 *initOwner()* – позволяет указать ссылку на родительское окно. Пример метода:

```
public final void initOwner(Window window)
```

7 *getOwner()* – возвращает ссылку на родительское окно или значение *null*. Пример метода:

```
public final Window getOwner()
```

8 *initStyle()* – задает стиль окна. Пример метода:

```
public final void initStyle(StageStyle style)
```

Обработка событий: назначить обработчики событий можно с помощью описанных ниже свойств.

1 *showing* – содержит значение *true*, если окно отображается на экране, и *false* – если скрыто. Методы доступа:

```
public final boolean isShowing()
public final ReadOnlyBooleanProperty showingProperty()
```

Пример назначения обработчика:

```
dialog.showingProperty().addListener
                        (obj, oldValue, newValue) -> {
   System.out.printIn("showing" + newValue);});
```

2 *onShown* – задает обработчик события отображения окна. Методы доступа:

```
public final void setOnShown(EventHandler<DialogEvent> value)
public final EventHandler<DialogEvent> getOnShown()
public final ObjectProperty<EventHandler<DialogEvent>>
             onShow Property()
```

3 *onHiding* – задает обработчик события, возникающего перед сокрытием окна. Методы доступа:

```
public final void setOnhiding(EventHandler<DialogEvent>
value)
public final EventHandler<DialogEvent> getOnHiding()
public final ObjectProperty<EventHandler<DialogEvent>>
             onHidingProperty()
```

4 *onHidden* – задает обработчик события закрытия окна. Методы доступа:

```
public final void setOnHidden(EventHandler<DialogEvent>
value)
public final EventHandler<DialogEvent> getOnHidden()
public final ObjectProperty<EventHandler<DialogEvent>>
             onHiddenProperty()
```

Пользователь может закрыть диалоговое окно с помощью нажатия кнопки <Закрыть> в заголовке окна. Если необходимо перехватить и обработать закрытие окна этим способом, то внутри программы следует назначить обработчик события с помощью свойства *onCloseRequest*. Методы доступа:

```
public final void setOnCloseRequest(EventHandler<DialogEvent>
  value)
public final EventHandler<DialogEvent> getOnCloseRequest()
public final ObjectProperty<EventHandler<DialogEvent>>
               onCloseRequestProperty()
```

Если внутри обработчика через объект события вызвать метод *consume()*, то окно закрыто не будет. Пример:

```
dialog.setOnCloseRequest(e -> {
  Alert alert = new Alert(AlertType.CONFIRMATION,
                      "Вы хотите закрыть окно?");
```

```
alert.setTitle("Закрытие окна");
alert.setHeaderText(null);
Optional<ButtonType> result = alert.showAndWait();
if (result.isPresent() && result.get() == ButtonType.OK) {
   System.out.println("Окно будет закрыто");
else {
   System.out.println("Окно не будет закрыто");
   e.consume();}});
```

9.2.15 Класс Button Type (кнопки внутри диалогового окна)

Для добавления кнопок в диалоговое окно следует воспользоваться методом *getButtonTypes()* из класса *DialogPane*. Пример метода:

```
public final ObservableList<ButtonType> getButtonTypes()
```

Пример добавления кнопок <*OK*> и <*Cancel*>:

```
dialog.getDialogPane().getButtonTypes().addAl1(
                       ButtonType.OK, ButtonType.CANCEL);
```

Если требуется изменить свойства кнопки, то нужно получить ссылку на нее с помощью метода *lookupButton()* из класса *DialogPane*. Пример метода:

```
public final Node lookupButton(ButtonType buttonType)
```

Сделаем кнопку <*OK*> неактивной:

```
Node node = dialog.getDialogPane().lookupButton(Button-
Type.OK);
if (node != null) node.setDisable(true);
```

Создать кнопку для диалогового окна позволяет класс *ButtonType*. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ButtonType;
```

Конструкторы класса:

```
ButtonType(String text)
ButtonType(String text, ButtonData buttonData)
```

Параметр *text* задает текст, отображаемый на кнопке, а параметр *buttonData* – предназначение кнопки. В первом конструкторе параметр *buttonData* имеет значение *ButtonData.OTHER*. Пример создания и добавления кнопок:

```
dialog.getDialogPane().getButtonTypes().addAl1(
          new ButtonType("OK", ButtonData.OK_DONE),
          new ButtonType("Отмена", ButtonData.CANCEL_CLOSE));
```

Получить текст, отображаемый на кнопке, и предназначение кнопки позволяют методы *getText()* и *getButtonData()* соответственно. Примеры методов:

```
public final String getText()
public final ButtonData getButtonData()
```

Для создания стандартных кнопок можно также воспользоваться следующими статическими константами из класса *ButtonType*:

```
public static final ButtonType OK
public static final ButtonType CANCEL
public static final ButtonType CLOSE
public static final ButtonType YES
public static final ButtonType NO
public static final ButtonType APPLY
public static final ButtonType FINISH
public static final ButtonType PREVIOUS
public static final ButtonType NEXT
```

Пример добавления кнопки <*OK*>:

```
dialog.getDialogPane().getButtonTypes().add(ButtonType.OK);
```

Получение результата происходит следующим образом. После закрытия диалогового окна часто нужно определить, какую кнопку нажал пользователь, и получить данные из диалогового окна. При вызове метода *showAndWait()* диалоговое окно отображается, и после нажатия кнопки пользователем метод возвращает объект класса *Optional*<*R*>, внутри которого содержится результат. Пример метода:

```
import java.util.Optional;
public final Optional<R> showAndWait()
```

Давайте создадим диалоговое окно с кнопками <*OK*> и <Отмена> и определим, какую кнопку нажал пользователь.

```
Button button = new Button("Открыть окно с кнопками OK и Отмена") ;
button.setOnAction (event -> {
   Dialog<ButtonType> dialog = new Dialog<ButtonType>();
        dialog.setTitle("Заголовок окна");
   dialog.setHeaderText("Текст заголовка сообщения");
   dialog.setContentText("Текст сообщения");
   dialog.getDialogPane().getButtonTypes().addAll(
             new ButtonType("OK", ButtonData.OK_ DONE),
             new ButtonType("Отмена", ButtonData.CANCEL_CLOSE));
   Optional<ButtonType> result = dialog.showAndWait();
   if (result.isPresent()) {
    if (result.orElseThrow().getButtonData() = ButtonData.OK_DONE){
       System.out.println("Нажата кнопка OK");
    else if (result.orElseThrow().getButtonData() ==
                                                ButtonData.CAN-
CEL_CLOSE){
```

```
        System.out.println("Нажата кнопка Отмена или кнопка За-
крыть");}}});
```

Для получения результата можно воспользоваться следующими свойствами из класса *Dialog<R>*:

1 *result* – содержит результат после обработки. Методы доступа:

```
public final void setResult R(value)
public final R getResult()
public final ObjectProperty<R> resultProperty()
```

Пример назначения обработчика:

```
dialog.resultProperty().addlistener(
                        (obj, old Value, new Value) -> {
       System.out.println("result " + newValue);});
```

2 *resultConverter* – позволяет назначить конвертер для предварительной обработки результата. Методы доступа:

```
public final void setResultConverter(Callback<ButtonType, R> value)
public final Callback<ButtonType, R> getResultConverter()
public final ObjectProperty<Callback<ButtonType, R>>
       resultConverterProperty()
```

9.2.16 Класс Alert (окно с сообщением)

Класс *Alert* наследует класс *Dialog<R>* и реализует модальное диалоговое окно с сообщениями различных типов. Инструкция импорта:

```
// Модуль javafx.controls
import javax.scene.control.Alert;
```

Конструкторы класса:

```
Alert(Alert.AlertType alertType)
Alert(Alert.AlertType alertType, String contentText,
       ButtonType... buttons)
```

В качестве параметра могут быть указаны следующие константы:
– *INFORMATION* – окно с информационным сообщением;
– *WARNING* – окно с предупреждающим сообщением;
– *ERROR* – окно с сообщением об ошибке;
– *CONFIRMATION* – окно с запросом подтверждения;
– *NONE* – произвольное окно.

Управлять типом окна позволяет свойство *alertType*. Методы доступа:

```
public final void setAlertType(Alert.AlertType alertType)
public final Alert.AlertType getAlertType()
public final ObjectProperty<Alert.AlertType> alertTypeProp-
erty()
```

Добавить кнопки можно с помощью метода *getButtonTypes()*. Пример метода:

```
public final ObservableList‹ButtonType> getButtonTypes()
```

9.2.17 Класс TextInputDialog (окно с текстовым полем)

Класс *TextInputDialog* наследует класс *Dialog<R>* и реализует модальное диалоговое окно с текстовым полем. Инструкция импорта:

```
// Модуль javafx.controls
import javax.scene.control.TextInputDialog;
```

Конструкторы класса:

```
TextInputDialog()
TextInputDialog(String defaultValue)
```

В параметре *defaultValue* можно указать текст, отображаемый в текстовом поле по умолчанию. Получить значение по умолчанию позволяет метод *getDefaultValue()*. Пример метода:

```
public final String getDefaultValue()
```

Для получения ссылки на текстовое поле предназначен метод *getEditor()*. Пример метода:

```
public final TextField getEditor()
```

Пример создания диалогового окна и получения результата.

```
Button button = new Button("Открыть окно");
button.setOnAction(event -> {
TextInputDialog dialog = new TextInputDialog("10");
dialog.setTitle("Заголовок окна");
dialog.setHeaderText("Введите значение");
dialog.getEditor().setPrefWidth(200.0);
Optional<String> result = dialog.showAndWait();
if (result.isPresent()) {
   System.out.println(result.get());
else System.out.println("Нажата кнопка Cancel");});
```

9.3 Практическая часть

1 Изучить описание к лабораторной работе, ознакомиться с примерами реализации *JavaFX*.

2 Для предметной области, выбранной в лабораторной работе № 1, и исходного кода по предыдущим работам необходимо разработать графический интерфейс с использованием библиотеки *JavaFX*. При запуске приложения должно выводиться окно входа с полями для ввода логина и пароля. Если в приложение

будет вход под учетной записью администратора, то выводятся графические элементы, которые необходимы для администрирования приложения, т. е. добавление новой информации, изменение и удаление имеющейся информации, блокировка/разблокировка, редактирование и удаление пользователей и т. д. Если в приложение будет вход под учетной записью пользователя, то выводятся графические элементы для выбора, назначения, поиска, сортировки, фильтрации, подсчета и т. д. Для всех форм ввода должна быть реализована валидация. Сообщения для пользователя должны выводиться в диалоговых окнах.

9.4 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Иллюстрация решения задачи.
5 Выводы.
6 Список использованных источников.

9.5 Контрольные вопросы

1 Что из себя представляет *JavaFX*? Какие пакеты содержит в себе *JavaFX*?
2 Архитектура сцены и жизненный цикл в *JavaFX*.
3 Обработка событий в *JavaFX*.
4 Какие методы имеет класс *Event*?
5 Какими способами можно назначить обработчик события?
6 Как можно блокировать и удалить обработчик события?
7 Какими методами можно сделать генерацию события из программы?
8 Какими пакетами представлены элементы графического интерфейса пользователя платформы *JavaFX*?
9 Какой класс является базовым для всех классов, компоненты которых содержат текстовую надпись?
10 Какой класс является базовым для кнопок, переключателей, флажков и гиперссылок?
11 Какие свойства содержит класс *Button*?
12 Какие свойства содержит класс *CheckBox*?
13 Какие свойства и методы содержит класс класс *Dialog<R>*?

9.6 Список использованных источников

1 Прохоренок, Н. А. JavaFX / Н. А. Прохоренок. – СПб. : БХВ-Петербург, 2020. – 768 с. : ил.
2 Машнин, Т. С. JavaFX 2.0 / Т. С. Машнин. – СПб. : БХВ-Петербург, 2012. – 320 с. : ил.
3 JavaFX [Электронный ресурс]. – Режим доступа: https://www.oracle.com/nl/java/technologies/javase/javafx-overview.html.

ПРИЛОЖЕНИЕ А

Варианты заданий

1 Цветочница. Определить иерархию цветов. Создать несколько объектов-цветов. Собрать букет (используя аксессуары) с определением его стоимости. Провести сортировку цветов в букете на основе уровня свежести. Найти цветок в букете, соответствующий заданному диапазону длин стеблей.

2 Новогодний подарок. Определить иерархию конфет и прочих сладостей. Создать несколько объектов-конфет. Собрать детский подарок с определением его веса. Провести сортировку конфет в подарке на основе одного из параметров. Найти конфету в подарке, соответствующую заданному диапазону содержания сахара.

3 Домашние электроприборы. Определить иерархию электроприборов. Включить некоторые в розетку. Подсчитать потребляемую мощность. Провести сортировку приборов в квартире на основе мощности. Найти прибор в квартире, соответствующий заданному диапазону параметров.

4 Шеф-повар. Определить иерархию овощей. Сделать салат. Подсчитать калорийность. Провести сортировку овощей для салата на основе одного из параметров. Найти овощи в салате, соответствующие заданному диапазону калорийности.

5 Звукозапись. Определить иерархию музыкальных композиций. Записать на диск сборку. Подсчитать продолжительность. Провести перестановку композиций диска на основе принадлежности к стилю. Найти композицию, соответствующую заданному диапазону длины треков.

6 Камни. Определить иерархию драгоценных и полудрагоценных камней. Отобрать камни для ожерелья. Подсчитать общий вес (в каратах) и стоимость. Провести сортировку камней ожерелья на основе ценности. Найти камни в ожерелье, соответствующие заданному диапазону параметров прозрачности.

7 Мотоциклист. Определить иерархию амуниции. Экипировать мотоциклиста. Подсчитать стоимость. Провести сортировку амуниции на основе веса. Найти элементы амуниции, соответствующие заданному диапазону параметров цены.

8 Транспорт. Определить иерархию подвижного состава железнодорожного транспорта. Создать пассажирский поезд. Подсчитать общую численность пассажиров и багажа. Провести сортировку вагонов поезда на основе уровня комфортности. Найти в поезде вагоны, соответствующие заданному диапазону параметров числа пассажиров.

9 Авиакомпания. Определить иерархию самолетов. Создать авиакомпанию. Посчитать общую вместимость и грузоподъемность. Провести сортировку

самолетов компании по дальности полета. Найти самолет в компании, соответствующий заданному диапазону параметров потребления горючего.

10 Таксопарк. Определить иерархию легковых автомобилей. Создать таксопарк. Подсчитать стоимость автопарка. Провести сортировку автомобилей парка по расходу топлива. Найти автомобиль в компании, соответствующий заданному диапазону параметров скорости.

11 Страхование. Определить иерархию страховых обязательств. Собрать из обязательств дериватив. Подсчитать стоимость. Провести сортировку обязательств в деривативе на основе уменьшения степени риска. Найти обязательство в деривативе, соответствующее заданному диапазону параметров.

12 Мобильная связь. Определить иерархию тарифов мобильной компании. Создать список тарифов компании. Подсчитать общую численность клиентов. Провести сортировку тарифов на основе размера абонентской платы. Найти тариф в компании, соответствующий заданному диапазону параметров.

13 Фургон кофе. Загрузить фургон определенного объема грузом на определенную сумму из различных сортов кофе, находящихся к тому же в разных физических состояниях (зерно, молотый, растворимый в банках и пакетиках). Учитывать объем кофе вместе с упаковкой. Провести сортировку товаров на основе соотношения цены и веса. Найти в фургоне товар, соответствующий заданному диапазону параметров качества.

14 Игровая комната. Подготовить игровую комнату для детей разных возрастных групп. Игрушек должно быть фиксированное количество в пределах выделенной суммы денег. Должны встречаться игрушки родственных групп: маленькие, средние и большие машины, куклы, мячи, кубики. Провести сортировку игрушек в комнате по одному из параметров. Найти игрушки в комнате, соответствующие заданному диапазону параметров.

15 Налоги. Определить множество и сумму налоговых выплат физического лица за год с учетом доходов с основного и дополнительного мест работы, авторских вознаграждений, продажи имущества, получения в подарок денежных сумм и имущества, переводов из-за границы, льгот на детей и материальной помощи. Провести сортировку налогов по сумме.

16 Счета. Клиент может иметь несколько счетов в банке. Учитывать возможность блокировки/разблокировки счета. Реализовать поиск и сортировку счетов. Вычисление общей суммы по счетам. Вычисления суммы по всем счетам, имеющим положительный и отрицательный балансы отдельно.

17 Туристические путевки. Сформировать набор предложений клиенту по выбору туристической путевки различного типа (отдых, экскурсии, лечение, шопинг, круиз и т. д.). Учитывать возможность выбора транспорта, питания и числа дней. Реализовать выбор и сортировку путевок.

18 Кредиты. Сформировать набор предложений клиенту по целевым кредитам различных банков для оптимального выбора. Учитывать возможность досрочного погашения кредита и/или увеличения кредитной линии. Реализовать выбор и поиск кредита.

19 Факультатив. Преподаватель объявляет запись на курс. Студент записывается на курс, обучается, и по окончании преподаватель выставляет оценку, которая сохраняется в архиве. Студентов, преподавателей и курсов может быть несколько.

20 Платежи. Клиент имеет счет в банке и кредитную карту. Клиент может оплатить заказ, сделать платеж на другой счет, заблокировать кредитную карту и аннулировать счет. Администратор может заблокировать кредитную карту за превышение суммы кредита.

21 Больница. Пациенту назначается лечащий врач. Врач может сделать назначение пациенту (процедуры, лекарства, операции). Медсестра или другой врач выполняют назначение. Пациент может быть выписан из больницы по окончании лечения, при нарушении режима или при иных обстоятельствах.

22 Вступительные экзамены. Абитуриент регистрируется на факультет, сдает экзамены. Преподаватель выставляет оценку. Система подсчитывает средний балл и определяет абитуриентов, зачисленных в учебное заведение.

23 Библиотека. Читатель оформляет заказ на книгу. Система осуществляет поиск в каталоге. Библиотекарь выдает читателю книгу на абонемент или в читальный зал. При невозвращении книги читателем он может быть занесен администратором в черный список.

24 Конструкторское бюро. Заказчик представляет техническое задание на проектирование многоэтажного дома. Конструктор регистрирует техническое задание, определяет стоимость проектирования и строительства, выставляет заказчику счет за проектирование и создает бригаду конструкторов для выполнения проекта.

25 Телефонная станция. Абонент оплачивает счет за разговоры и услуги, может попросить администратора сменить номер и отказаться от услуг. Администратор изменяет номер, услуги и временно отключает абонента за неуплату.

26 Автобаза. Диспетчер распределяет заявки на рейсы между водителями и назначает для этого автомобиль. Водитель может сделать заявку на ремонт. Диспетчер может отстранить водителя от работы. Водитель делает отметку о выполнении рейса и состоянии автомобиля.

27 Интернет-магазин. Администратор добавляет информацию о товаре. Клиент делает и оплачивает заказ на товары. Администратор регистрирует продажу и может занести неплательщиков в черный список.

28 Железнодорожная касса. Пассажир делает заявку на станцию назначения, время и дату поездки. Система регистрирует заявку и осуществляет поиск подходящего поезда. Пассажир делает выбор поезда и получает счет на оплату.

Администратор вводит номера поездов, промежуточные и конечные станции, цены.

29 Городской транспорт. На маршрут назначаются автобус, троллейбус или трамвай. Транспортные средства должны двигаться с определенным для каждого маршрута интервалом. При поломке на маршрут должен выходить резервный транспорт или увеличиваться интервал движения.

30 Аэрофлот. Администратор формирует летную бригаду (пилоты, штурман, радист, стюардессы) на рейс. Каждый рейс выполняется самолетом с определенной вместимостью и дальностью полета. Рейс может быть отменен из-за погодных условий в аэропорту отлета или назначения. Аэропорт назначения может быть изменен в полете из-за технических неисправностей, о которых сообщил командир.

31 Периодические издания. Читатель может сделать заявку, предварительно выбрав периодические издания из списка. Система подсчитывает сумму для оплаты. Читатель оплачивает заявку. Администратор добавляет заявку в черный список, если клиент не оплачивает ее в определенный срок.

32 Заказ гостиницы. Клиент оставляет заявку на номер, указав количество мест в номере, класс апартаментов и время пребывания. Администратор рассматривает заявку, подтверждает или отклоняет ее. Результат просматривает клиент. В случае подтверждения заявки клиент оплачивает услуги.

33 Жилищно-коммунальные услуги. Квартиросъемщик отправляет заявку, в которой указывает род работ, масштаб и желаемое время выполнения. Диспетчер формирует соответствующую бригаду и регистрирует ее в плане работ. Диспетчер может отклонить заявку в случае занятости всех бригад.

34 Прокат автомобилей. Клиент выбирает автомобиль из списка доступных, заполняет форму заказа, указывая паспортные данные, срок аренды. Администратор может отклонить заявку, указав причины отказа. При подтверждении заявки клиент оплачивает заказ. В случае повреждения автомобиля клиентом администратор вносит соответствующие пометки.

35 Олимпиада. Учитель объявляет запись на подготовку к олимпиаде. Ученик записывается на подготовку, обучается и по окончании проходит олимпиаду, получая какое-то количество баллов, которые сохраняются в архиве. Учеников, учителей и олимпиад при обучении может быть несколько.

36 Новогодние украшения. Определить иерархию украшений. Подобрать комплект украшений для дома. Подсчитать цену. Провести сортировку украшений для комплекта на основе одного из параметров. Найти украшения в комплекте, соответствующие заданному диапазону цены.

37 Лесхоз. Определить иерархию деревьев. Создать несколько объектов-деревьев. Составить список на вырубку из различных видов, подсчитать выручку. Провести сортировку деревьев в вырубке на основе их высоты. Найти дерево в вырубке, соответствующее заданному диапазону лет.

38 Аквариум. Определить иерархию аквариумных рыбок. Составить список рыбок для аквариума. Подсчитать цену за всех рыбок. Провести сортировку рыбок на основе их продолжительности жизни. Найти рыбок, соответствующих заданному диапазону размеров.

39 Магазин лампочек. Определить иерархию лампочек. Подобрать лампочки для квартиры. Подсчитать цену. Провести сортировку лампочек в соответствии с их яркостью. Найти лампочки, соответствующие заданному диапазону цены.

40 Прокат велосипедов, самокатов. Клиент выбирает велосипед или самокат из списка доступных, заполняет форму заказа, указывая паспортные данные, срок аренды. Администратор может отклонить заявку, указав причины отказа. При подтверждении заявки клиент оплачивает заказ. В случае повреждения велосипеда или самоката клиентом администратор вносит соответствующие пометки.

41 Одежда на прокат. Клиент выбирает одежду из списка доступной, заполняет форму заказа, указывая свои параметры, срок аренды. Администратор может отклонить заявку, указав причины отказа. При подтверждении заявки клиент оплачивает заказ. В случае повреждения одежды клиентом администратор вносит соответствующие пометки.

42 Экскурсия. Клиент делает заявку на посещение определенного места, время и дату посещения. Система регистрирует заявку и осуществляет поиск подходящего гида. Клиент делает выбор гида и получает счет на оплату. Администратор вводит контакты гида, дополнительные услуги и маршрутный лист, цены.

43 Кондитерская. Определить иерархию десертов. Создать витрину. Подсчитать стоимость витрины. Провести сортировку десертов витрины по цене. Найти десерт, соответствующий заданному диапазону калорий.

44 Магазин фруктов. Определить иерархию фруктов. Создать несколько объектов-фруктов. Собрать подарочную корзинку с фруктами (используя аксессуары) с определением его стоимости. Провести сортировку фруктов в корзинке на основе уровня свежести. Найти фрукт в корзине, соответствующий заданному диапазону веса.

45 Студия керамики. Определить иерархию керамических изделий. Создать несколько объектов-изделий. Собрать подарочный комплект с определением его стоимости. Провести сортировку изделий в комплекте на основе времени его изготовления. Найти изделие в комплекте, соответствующее заданному диапазону хрупкости (шкала от 1 до 10).

46 Азиатский ресторан. Определить иерархию блюд. Создать несколько объектов-блюд. Собрать сет из блюд с определением его стоимости. Провести

сортировку блюд в сете на основе уровня свежести рыбы (замороженная / охлажденная / только выловленная). Найти блюдо в сете, соответствующее заданному диапазону веса.

47 Фургон мороженого. Загрузить фургон определенного объема грузом на определенную сумму из различных сортов мороженого, содержащих к тому же разные начинки. Учитывать вес мороженого вместе с упаковкой. Провести сортировку товаров на основе соотношения цены и веса. Найти в фургоне товар, соответствующий заданному диапазону веса.

48 Чайная. Закупить в заведение определенную сумму различных сортов чая, находящихся к тому же в разных фракциях (листовой, ломаный, крошка и пыль). Учитывать вес чая вместе с упаковкой. Провести сортировку товаров на основе соотношения цены и веса. Найти в заведении товар, соответствующий заданному диапазону веса.

49 Магазин игрушек. Определить иерархию игрушек. Создать несколько объектов-игрушек. В зависимости от размера и вида игрушек стоимость изменяется. Покупатель добавляет понравившиеся ему игрушки в корзину. Провести сортировку игрушек по цене. Найти игрушку в магазине, соответствующую заданному размеру.

50 Платные курсы. Реализовать две роли – преподаватель и ученик, у различных преподавателей свой набор курсов и учеников. Реализовать для ученика возможность выбора срока, времени проведения (утром, днем или вечером), формы обучения (очно или дистанционно) и преподавателя. Вычислить общую стоимость за обучение.

51 Аптека. Определить иерархию лекарств. Создать несколько объектов-лекарств. В зависимости от формы и вида стоимость изменяется. Покупатель добавляет нужные ему лекарства в корзину. Провести сортировку лекарств по цене. Найти лекарство в аптеке, соответствующее заданной форме и виду.

52 Платные медицинские услуги. Реализовать две роли – врач и пациент, у различных врачей свой набор услуг и пациентов. Реализовать для пациента возможность выбора нескольких услуг у различных врачей со своим временем проведения (утром, днем или вечером). Вычислить общую стоимость услуг.

53 Услуги по ремонту ноутбуков. Реализовать две роли – мастер и клиент, у различных мастеров свой набор услуг и клиентов. Реализовать для клиента возможность выбора нескольких услуг, срочность, доставку курьером или самовывоз. В зависимости от срочности и варианта получения ноутбука стоимость изменяется. Вычислить общую стоимость услуг.

54 Магазин книг. Определить иерархию книг. Создать несколько объектов-книг. В зависимости от вида переплета, издательства и качества бумаги стоимость изменяется. Покупатель добавляет понравившиеся ему книги в корзину. Провести сортировку книг по цене. Найти книгу в магазине, соответствующую заданному виду переплета и качеству бумаги.

55 Услуги по ремонту домов. Реализовать две роли – мастер и клиент, у различных мастеров свой набор услуг и клиентов. Реализовать для клиента возможность выбора нескольких услуг, выбор материалов (дорогие или дешевые), скорость выполнения работы. В зависимости от материалов и скорости выполнения стоимость изменяется. Вычислить общую стоимость услуг.

56 Автосалон. Определить иерархию автомобилей. Создать несколько объектов-автомобилей. В зависимости от срока гарантии и дополнительных возможностей автомобиля стоимость изменяется. Покупатель выбирает автомобиль и оформляет заказ. Провести сортировку автомобилей по цене. Найти автомобиль в автосалоне, соответствующий заданному расходу топлива.

57 Склад. Создать склад со определенным объемом. Клиенты заполняют заявку на добавление предмета на склад, указывая длину, ширину и высоту предмета, система вычисляет объем предмета и если предмет помещается на складе, то заявка будет одобрена, а клиенту выдается чек с присвоенным его предмету идентификационным номером. Клиент может добавлять несколько предметов на склад, а также просматривать одобренные и отклоненные заявки. Также клиент может забрать предмет, указав идентификационный номер.

58 СТО. Реализовать две роли – мастер и клиент, у различных мастеров свой набор услуг и клиентов. Реализовать для клиента возможность выбора нескольких услуг, выбор материалов (дорогие или дешевые), сроков выполнения. В зависимости от материалов и сроков выполнения стоимость изменяется. Вычислить общую стоимость услуг.

59 Лизинг сельхозтехники. Клиент выбирает тип сельхозтехники, срок лизинга и дополнительные инструменты, заполняет форму заказа, указывая паспортные данные. Администратор может отклонить заявку, указав причины отказа. При подтверждении заявки клиент оплачивает первый взнос. У клиента есть возможность выкупа техники.

60 Почта. Клиенты заполняют заявку на доставку посылки, указывая длину, ширину, высоту и вес посылки, а также срочность и дальность доставки, система, исходя из всех параметров, вычисляет стоимость доставки, клиенту выдается чек с присвоенным его посылке идентификационным номером. Клиент может отправлять несколько посылок, а также просматривать уже отправленные посылки и следить за их статусом. Администратор изменяет статус посылки.

61 Зоопарк. Клиент делает заявку на посещение определенной секции, время и дату посещения. Система регистрирует заявку и осуществляет поиск подходящего зоолога. Клиент делает выбор зоолога и получает счет на оплату. Администратор вводит контакты зоолога, дополнительные услуги, маршрутный лист и цены.

62 Компьютерные игры на прокат. Клиент выбирает игры из списка доступных, заполняет форму заказа, указывая срок аренды. Администратор может отклонить заявку, указав причины отказа. При подтверждении заявки клиент

оплачивает заказ. В случае повреждения диска клиентом администратор вносит соответствующие пометки.

63 Настольные игры на прокат. Клиент выбирает игры из списка доступных, заполняет форму заказа, указывая срок аренды. Администратор может отклонить заявку, указав причины отказа. При подтверждении заявки клиент оплачивает заказ. В случае повреждения игры клиентом администратор вносит соответствующие пометки.

64 Парикмахерская. Клиент делает заявку на определенную прическу, время и дату посещения. Система регистрирует заявку и осуществляет поиск подходящего парикмахера. Клиент делает выбор парикмахера и получает счет на оплату. Администратор вводит контакты парикмахера, дополнительные услуги и цены.

65 Сборка компьютера. Определить иерархию комплектующих. Собрать компьютер. Подсчитать стоимость. Провести сортировку комплектующих на основе цены. Найти комплектующие, соответствующие заданному диапазону параметров цены.

66 Прокат лодок. Клиент выбирает лодку из списка доступных, заполняет форму заказа, указывая паспортные данные, срок аренды. Администратор может отклонить заявку, указав причины отказа. При подтверждении заявки клиент оплачивает заказ. В случае повреждения лодки клиентом администратор вносит соответствующие пометки.

67 Прокат удочек. Клиент выбирает удочку из списка доступных, заполняет форму заказа, указывая паспортные данные, срок аренды. Администратор может отклонить заявку, указав причины отказа. При подтверждении заявки клиент оплачивает заказ. В случае повреждения удочки клиентом администратор вносит соответствующие пометки.

68 Магазин бытовой техники. Определить иерархию бытовой техники. Создать несколько объектов бытовой техники. Собрать комплект техники для одной комнаты с определением его стоимости. Провести сортировку техники в комплекте на основе мощности прибора. Найти технику в комплекте, соответствующую заданному диапазону цены.

69 Магазин ювелирных изделий. Определить иерархию ювелирных изделий. Создать несколько объектов ювелирных изделий. Собрать комплект определенной серии, состоящей из трех разных украшений одной серии, с определением его стоимости. Провести сортировку украшений в комплекте на основе веса. Найти украшение в комплекте, соответствующее заданному диапазону цены.

70 Пиццерия. Клиент делает заказ на доставку определенной пиццы, указывая время и дату. Система регистрирует заказ и осуществляет поиск подходящего курьера. Клиент выбирает курьера и получает счет на оплату и номер телефона курьера. Администратор вводит контакты курьера, добавляет пиццы.

71 Доставка роллов. Клиент делает заказ на доставку определенного сета, указывая время и дату. Система регистрирует заказ и осуществляет поиск подходящего курьера. Клиент выбирает курьера и получает счет на оплату и номер телефона курьера. Администратор вводит контакты курьера, добавляет сеты роллов.

72 Кинотеатр. Клиент делает заказ на определенный фильм, время и дату посещения. Система регистрирует заказ и осуществляет поиск подходящего кинотеатра. Клиент выбирает кинотеатр и получает счет на оплату. Администратор вводит фильмы, кинотеатры и время.

73 Биржа криптовалют. Клиент может иметь несколько криптовалют в кошельке. Реализовать возможность покупки/продажи криптовалюты. Администратор может блокировать/разблокировать клиентов. Реализовать сортировку криптовалют. Вычисление общей суммы кошелька.

74 Обувной магазин. Определить иерархию обуви. Создать несколько объектов моделей обуви. Собрать комплект одного размера обуви на все сезоны с определением его стоимости. Провести сортировку обуви в комплекте на основе температуры, оптимальной для их носки. Найти обувь в комплекте, соответствующую заданному диапазону цены.

75 Оптика. Определить иерархию очков и линз. Создать несколько объектов – очков и линз. Собрать комплект очков и линз определенных диоптрий с определением его стоимости. Провести сортировку линз по сроку ношения. Найти очки, соответствующие заданному диапазону цены.

Св. план 2023, поз. 12

Учебное издание

Хорошко Виталий Викторович
Горбач Антон Петрович
Бруй Никита Михайлович и др.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ И ПРОГРАММИРОВАНИЕ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

ПОСОБИЕ

Редактор *А. Ю. Шурко*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *В. А. Долгая*

Подписано в печать 30.07.2025. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 6,98. Уч.-изд. л. 7,0. Тираж 50 экз. Заказ 95.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск