СОДЕРЖАНИЕ

Практическая работа № 17. Разработка интерактивных программ на языке Джава с использованием паттерна MVC

Цель: введение в разработку программ с с использованием событийного программирования на языке программирования Джава с использованием паттерна MVC.

Теоретические сведения

Шаблон проектирования в программной инженерии — это метод решения часто возникающей проблемы при разработке программного обеспечения. Проектирование по модели указывает, какой тип архитектуры вы используете для решения проблемы или разработки модели.

Существует два *типа моделей проектирования*:

- *архитектура модели 1*
- *архитектура модели 2 (MVC).*

Архитектура MVC в Джава приложениях

Проекты моделей, основанные на архитектуре MVC (model-view-controller), следуют шаблону проектирования MVC и отделяют логику приложения от пользовательского интерфейса при разработке программного обеспечения.

Как следует из названия, шаблон MVC имеет три уровня:

- Модель — представляет бизнес-уровень приложения (model).
- Вид — определяет представление приложения (view).
- Контроллер — управляет потоком приложения (controller).

На рис. 17.1 представлена схема работы архитектуры MVC

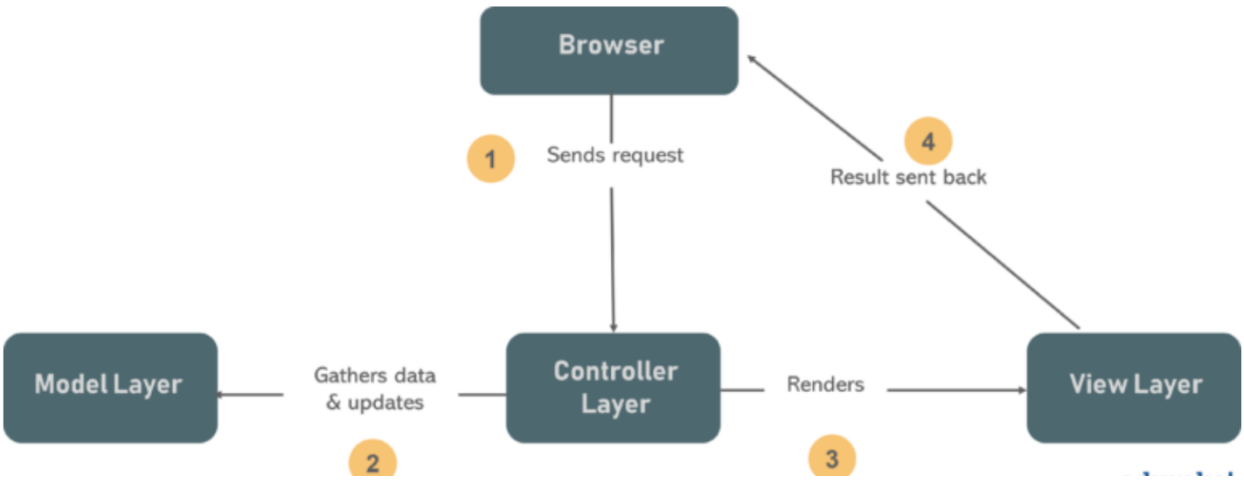

Рисунок 17.1. Архитектура MVC

В контексте программирования модель на Java состоит из простых классов Java, уровень представление отображает данные – то что видит пользователь, а контроллер состоит из сервлетов. Это разделение приводит к тому, что запросы пользователей обрабатываются следующим образом:

1. Браузер на клиенте отправляет запрос на страницу контроллеру, присутствующему на сервере.
2. Контроллер выполняет действие по вызову модели, тем самым извлекая необходимые ему данные в ответ на запрос.
3. Затем контроллер передает полученные данные в представление.
4. Представление визуализируется и отправляется обратно клиенту для отображения в браузере.

Разделение программного приложения на эти три отдельных компонента является хорошей идей по ряду причин.

Преимущества архитектуры MVC в Джава

Архитектура MVC предлагает множество преимуществ для программиста при разработке приложений, в том числе:

- Несколько разработчиков могут одновременно работать с тремя уровнями (модель, представление и контроллер).
- Предлагает улучшенную *масштабируемость* , которая дополняет возможность роста приложения.
- Поскольку компоненты мало зависят друг от друга, их легко поддерживать.
- Модель может быть повторно использована несколькими представлениями, что обеспечивает возможность повторного использования кода.
- Применение MVC делает приложение более выразительным и простым для понимания.
- Расширение и тестирование приложения становится проще

Шаблон MVC является самым популярным шаблоном проектирования для веб разработки.

Пример реализации паттерна MVC с использованием языка Джава

Чтобы реализовать веб-приложение на основе шаблона

проектирования MVC, мы создадим следующие классы:

- Класс Course, который действует как слой модели
- Класс CourseView, определяющий уровень представления (слой представления)
- Класс CourseContoller, который действует как контроллер

Уровень слоя модели

В шаблоне проектирования *модель* MVC представляет собой уровень данных, который определяет бизнес-логику системы, а также представляет состояние приложения. Объекты модели извлекают и сохраняют состояние модели в базе данных. На этом уровне мы применяем правила к данным, которые в конечном итоге представляют концепции, которыми управляет наше приложение. Теперь давайте создадим модель с помощью класса Course см листинг 17.1

Листинг 17.1 – Класс Course

```
package myPackage;

public class Course {
   private String CourseName;
   private String CourseId;
   private String CourseCategory;

   public String getId() {
      return CourseId;
   }

   public void setId(String id) {
      this.CourseId = id;
   }

   public String getName() {
      return CourseName;
   }

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

   public String getCategory() {
      return CourseCategory;
```

```
    }

    public void setCategory(String category) {
        this.CourseCategory = category;
    }

}
```

Код прост для понимания и не требует пояснений. Он состоит фактттически из методов геттеров и сеттеров для получения/установки сведений о курсе.

Уровень слоя представления

Этот уровень шаблона проектирования MVC представляет выходные данные приложения или пользовательского интерфейса. Он отображает данные, полученные контроллером из уровня модели, и предоставляет данные пользователю по запросу. Он получает всю необходимую информацию от контроллера, и ему не нужно напрямую взаимодействовать с бизнес-уровнем. Давайте создадим представление с помощью класса CourseView см листинг 17.2.

Листинг 17.2 – Класс *CourseView.*

```
package   myPackage;

public class CourseView {
    public void printCourseDetails(String CourseName, String CourseId,
String CourseCategory){
        System.out.println("Course Details: ");
        System.out.println("Name: " + CourseName);
        System.out.println("Course ID: " + CourseId);
        System.out.println("Course Category: " + CourseCategory);
    }
}
```

Этот код просто выводит значения на консоль. Далее нам предстоит создать контроллер для отслеживания событий на уровне представления данных и изменения модели данных. Этот слой фактттически отвечает за бизнес логику приложения.

Уровень слоя контроллера

Контроллер похож на интерфейс между моделью и

представлением. Он получает пользовательские запросы от уровня представления и обрабатывает их, включая необходимые проверки. Затем запросы отправляются в модель для обработки данных. После обработки данные снова отправляются обратно в контроллер, а затем отображаются в представлении. Давайте создадим класс CourseContoller, который действует как контроллер см листинг 17.3.

Листинг 17.3 – Класс CourseContoller

```
package    myPackage;

public class CourseController {
   private Course model;
   private CourseView view;

   public CourseController(Course model, CourseView view){
      this.model = model;
      this.view = view;
   }

   public void setCourseName(String name){
      model.setName(name);
   }

   public String getCourseName(){
      return model.getName();
   }

   public void setCourseId(String id){
      model.setId(id);
   }

   public String getCourseId(){
      return model.getId();
   }

   public void setCourseCategory(String category){
      model.setCategory(category);
   }

   public String getCourseCategory(){
      return model.getCategory();
   }
```

```
   public void updateView(){
      view.printCourseDetails(model.getName(), model.getId(),
model.getCategory());
   }
}
```

Из кода на листинге 17.3 видно на, что этот класс контроллера просто отвечает за вызов модели для получения/установки данных и обновления представления на основе этой информации. Теперь соединим все вместе, для этого напишем тестовый класс, назовем его MVCPatternDemo.java.

Листинг 17.4 – Класс MVCPatternDemo

```
package myPackage;

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

      //fetch student record based on his roll no from the database
      Course model  = retriveCourseFromDatabase();

      //Create a view : to write course details on console
      CourseView view = new CourseView();

      CourseController controller = new CourseController(model, view);

      controller.updateView();

      //update model data
      controller.setCourseName("Python");
      System.out.println("nAfter updating, Course Details are as follows");

      controller.updateView();
   }

   private static Course retriveCourseFromDatabase(){
      Course course = new Course();
      course.setName("Java");
      course.setId("01");
      course.setCategory("Programming");
      return course;
   }
}
```

}

Приведенный на листинге класс извлекает данные курса из функции с помощью которой пользователь вводит набор значений. Затем он помещает эти значения в модель курса. Затем он инициализирует новое представление, которое мы создали ранее. Кроме того, он также вызывает класс CourseController и связывает его с классом Course и классом CourseView, метод updateView() , который является частью контроллера, затем обновляет сведения о курсе на консоли.

Результат работы программы

```
Course Details:
Name: Java
Course ID: 01
Course Category: Programming

After updating, Course Details are as follows
Course Details:
Name: Python
Course ID: 01
Course Category: Programming
```

Вывод. Архитектура MVC обеспечивает совершенно новый уровень модульности вашего кода, что делает его более читабельным и удобным в сопровождении.

Задания на практическую работу № 1

1. Напишите реализацию программного кода по UML диаграмме, представленной на рис.17.2 . Прграмма должна демонстрировать использование паттерна MVC.

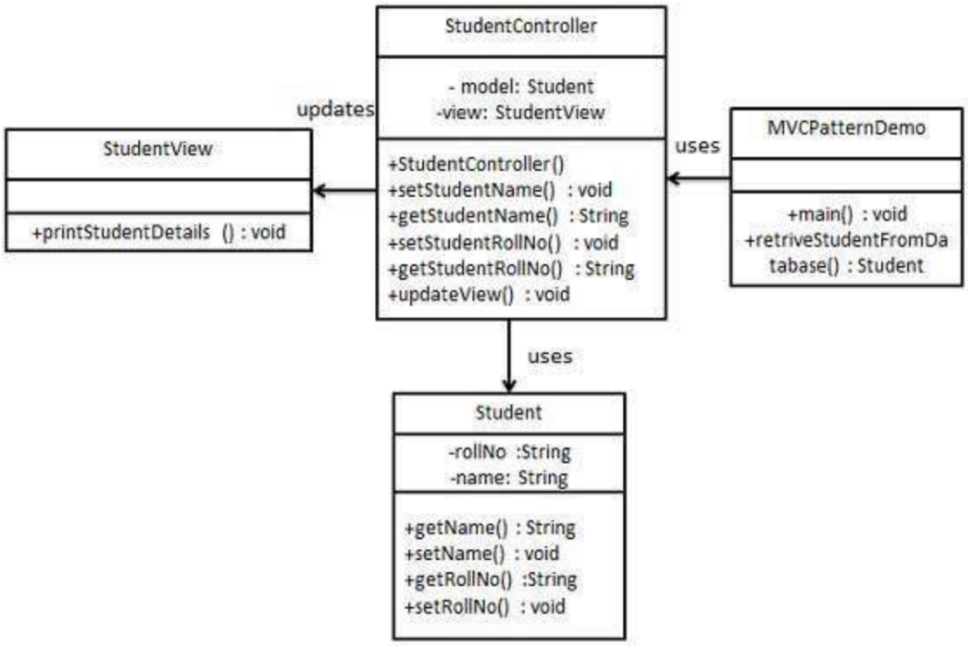

Рисунок 17.2. UML диаграмма классов проекта, реализующего MVC

2. Напишите реализацию программного кода, с использованием паттерна MVC для расчета заработной платы сотрудника предприятия. Предлагается использовать следующие классы.

- Класс Employee – сотрудник будет выступать в качестве слоя модели
- Класс EmployeeView будет действовать как слой представления.
- Класс EmployeeContoller будет действовать как уровень контроллера.

3. Вы можете написать программную реализацию, используя собственную идею, реализуя паттерн MVC. Выполнение задания предполагаеьт создание GUI.

Практическая работа № 18. Исключения и работа с ними в Джава

Цель данной практической работы являются получение практических навыков разработки программ, изучение синтаксиса языка Java, освоение основных конструкций языка Java (циклы, условия, создание переменных и массивов, создание методов, вызов методов), а также научиться осуществлять стандартный ввод/вывод данных.

Ключевые слова: try, catch, finally, throw, throws

Теоретические сведения

Механизм исключительных ситуаций в Java поддерживается пятью ключевыми словами:

- try
- catch
- finally
- throw
- throws

В языке Джава Java всего около 50 ключевых слов, и пять из них связано как раз с исключениями, это- try, catch, finally, throw, throws.

Из них catch, throw и throws применяются к экземплярам класса, причём работают они только с Throwable и его наследниками.

На рис. 18.1 представлена иерархия классов исключений, используемая в Java

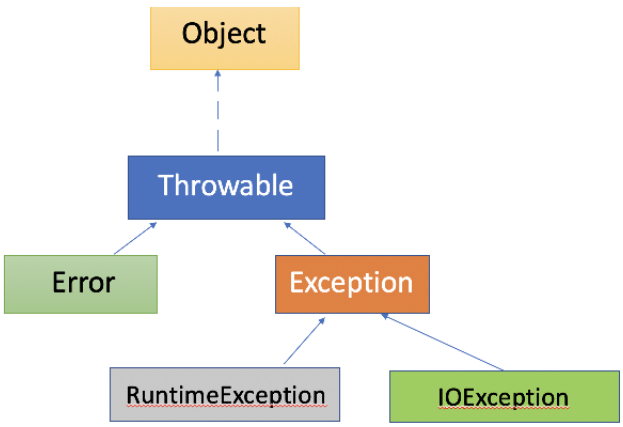

Рисунок 18.1. Иерархия классов исключений в Джава

Наиболее популярные исключений в Java представлены в

таблице 1. Таблица 1. Классы исключений в Java

№ пп	Класс исключения	Класс предок/тип
1.	ArithmeticException	RuntimeException
2.	NegativeArraySizeException	RuntimeException
3.	ArrayIndexOutOfBoundsException	RuntimeException
4.	NoSuchElementException	RuntimeException
5.	ArrayStoreException	RuntimeException
6.	NotSerializableException	Exception
7.	AssertionError	Error
8.	NullPointerException	RuntimeException
9.	ClassCastException	RuntimeException
10.	NumberFormatException	RuntimeException
11.	ClassNotFoundException	Exception
12.	OutOfMemoryError	Error
13.	CloneNotSupportedException	Exception
14.	SecurityException	RuntimeException
15.	ConcurrentModificationException	RuntimeException
16.	StackOverflowError	Error
17.	EOFException	Exception
18.	StringIndexOutOfBoundsException	RuntimeException
19.	FileNotFoundException	Exception
20.	ThreadDeath	Error
21.	IllegalArgumentException	RuntimeException
22.	UnsupportedEncodingException	Exception
23.	InterruptedException	Exception
24.	UnsupportedOperationException	RuntimeException

То, что исключения являются объектами важно по двум причинам:

1) они образуют иерархию с корнем java.lang.Throwable (java.lang.Object — предок java.lang.Throwable, но Object —это не исключение!)
2) они могут иметь поля и методы

По первому пункту: catch — полиморфная конструкция, т.е. catch по типу класса родителя перехватывает исключения для экземпляров объектов как родительского класса, так или его наследников(т.е. экземпляры непосредственно самого родительского класса или любого его потомка).

Листинг 18.1 – Пример обработки исключения

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

```
            try {
               System.err.print(" 0");
               if (true) {throw new RuntimeException();}
               System.err.print(" 1");
            } catch (Exception e) {  // catch по Exception ПЕРЕХВАТЫВАЕТ
RuntimeException
                  System.err.print(" 2");
               }
            System.err.println(" 3");
         }// end main
      }
```

Результат работы программы: представленной на листинге 18.1:
>> 0 2 3

Задания на практическую работу № 18 (Основы Try-Catch- Finally)

Задание № 1

Шаг 1. Выполните следующую программу и посмотрите, что происходит:

Листинг 18.2 – Пример обработки деления на ноль

```
      public class Exception1 {
         public void exceptionDemo() {
           System.out.println( 2 / 0 );
         }
      }
```

Описание работы

Вам необходимо инстанцировать класс и выполнить exceptionDemo(). Что произойдет?

Ответ: программа даст сбой, и вы получите следующее сообщение

java.lang.ArithmeticException: / by zero at Exception1.exceptionDemo(Exception1.java:12)

Это говорит нам о том, что программа пытается выполнить деление на ноль, который он не в состоянии выполнить.

Объясните поведение программы.

Шаг 2. Измените программу следующим образом.

Замените 2/0 на 2,0 / 0,0 и повторно вызовите метод. Что произойдет?

Теперь измените код в классе Exception1 и включите блок try-catch следующим образом:

Листинг 18.3 – Пример обработки исключения

```
public class Exception1 {
   public void exceptionDemo() {
     try{
        System.out.println( 2/0 );
        } catch ( ArithmeticException e )
{ System.out.println("Attempted division by zero");
        }
    }
}
```

Шаг 3. Запустите программу и обратите внимание на новое поведение.

Объясните поведение программы.

Задание № 2

Шаг 1.Измените код в листинге 18.3 на следующий:

Листинг 18.4 – Пример программы

```
public class Exception2 {
public void exceptionDemo() {
Scanner myScanner = new Scanner( System.in);
System.out.print( "Enter an integer ");
String intString = myScanner.next();
int i = Integer.parseInt(intString);
System.out.println( 2/i );
     }
}
```

Шаг 2. Запустите эту программу со следующими выводами: Qwerty 0 1.2 1. Посмотрите на вывод.

Объясните какие исключения выбрасываются?

Шаг 3. Измените код, добавив блоки try – catch, чтобы иметь дело с определяемыми исключениями.

Объясните поведение программы

Задание № 3

С помощью перехватывания исключений можно оказывать влияние на поведение программы. В вашем решении в предыдущем упражнении вы можете добавить новый пункт - catch в начале списка пунктов catch.

Шаг 1. Выполите это действие, чтобы поймать общее исключение класса Exception.

Шаг 2. Перезапустите программу с приведенными выше данными и обратите внимание на ее поведение.

Объясните новое поведение программы

Задание № 4

Шаг 1. Добавьте блок finally к решению Задания №2.

Шаг 2. Повторно запустите программу, чтобы проверить ее поведение. Объясните новое поведение программы

Генерация собственных исключений

На предыдущем шаге при выполнении заданий мы рассмотрели, как Java работает с предопределенными исключениями, теперь перейдем к тому, как генерируется исключение.

Все исключения в рассмотренных ранее примерах и заданиях были определены заранее. В этом разделе практичнской работы вы будете создавать и пробрасывать свои собственные исключения (exceptions).

Задание № 5

Самый простой способ генерации исключения показан в следующем примере кода:

Листинг 18.5. Класс ThrowsDemo

```
public class ThrowsDemo {
    public void getDetails(String key) {
        if(key == null) {
throw newNullPointerException("null key in getDetails" );
         }
      // do something with the key
     }
}
```

Шаг 1. Когда определяется условие ошибки, то мы выбрасываем исключение с определенным именем. Сообщение может быть связано с исключением. Откомпилируйте этот класс, создайте его экземпляр и выполните метод getDetails() с нулем в качестве значения параметра.

Вы можете получить следующийвывод:

java.lang.NullPointerException: null key in getDetails at ThrowsDemo.getDetails(ThrowsDemo.java:13)

Шаг 2. Добавьте блок try-catch таким образом, чтобы перехватить исключение и рассмотреть его внутри метода.

Подумайте, является ли этот способ подходящим, чтобы иметь

дело с этим исключением?

Объясните поведение программы.

Ответ. Причиной ошибки, может является, например неправильное значение для параметра. Может было бы лучше, если бы метод вызывал getDetails() и там решалась бы эта проблема.

Обратите внимание на следующее:

Листинг 18.6 Пример видоизмененной программы ThrowsDemo

```
public class ThrowsDemo {
    public void printMessage(String key) {
      String message = getDetails(key); System.out.println( message );
    }
public String getDetails(String key) {
if(key == null) {
throw new NullPointerException( “null key in getDetails” );
   }
    return “data for” + key; }
}
```

Задание № 6

Шаг 1. Откомпилируйте и запустите эту программу с правильным значением для ключа и с нулем в качестве значения. При выполнении с нулевым значением вы увидите некоторый вывод.

Шаг 2. Обобщите все вышесказанное и выполните эту программу с правильным значением для ключа и с нулем в ключе.

java.lang.NullPointerException: null key in getDetails

at ThrowsDemo.getDetails(ThrowsDemo.java:21) at ThrowsDemo.printMessage(ThrowsDemo.java:13)

Шаг 3. Теперь добавьте блоки try-catch, чтобы использовать для вывода сообщений метод printMessage(), таким образом, чтобы исключения обрабатывались и программа не “ломалась”.

Объясните ее поведение.

Задание № 7

Теперь мы расширим наш пример для демонстрации прохождения исключения через цепочку вызовов.

Листинг 18.7

```
public class ThrowsDemo {
public void getKey() {
    Scanner myScanner = new Scanner( System.in );
```

```
    String key = myScanner.next();
    printDetails( key );
  }
  public void printDetails(String key) {
    try {  String message = getDetails(key);
           System.out.println( message );
          }catch ( Exception e){
            throw e;
          }
     }
  private String getDetails(String key) {
    if(key == "") {
           throw new Exception( "Key set to empty string" );
    }
  return "data for " + key; }
}
```

Шаг 1. Создайте следующий класс (листинг 18.7) и попытайтесь его скомпилировать.

При попытке компиляции вы получите следующий синтаксис ошибки:

Исключение Unreported java.lang.Exception

В результате успешного пробрасывания исключение должен быть поймано или объявлено. Объясните причину.

Ответ. Причиной полученной ошибки является выражение ***throw e.***

Пояснение. В данном случае метод printDetails () решил, что он не может иметь дело с исключением и проходит все дерево его вызовов. Поскольку метод getKey() не имеет блока try-catch для обработки исключений, то Java становится перед выбором, что в таком случае делать.

Проблему можно решить несколькими способами:

1) Добавьте соответствующие try-catch блоки таким образом, чтобы в конечном итоге один из них обрабатывал исключение;
2) Удалите блоки try-catch для всех методов, кроме одного, который обрабатывает исключение. Добавьте throws, котрый бросает исключение методу, который проходит исключение без обработки.

Задание №8

Шаг 1. Измените код из предыдущего примера следующим

образом:

1. Удалите throws Exception из метода getKey().
2. Измените метод getKey(), добавив try-catch блок для обработки исключений.
3. Добавьте цикл к getKey() таким образом, чтобы пользователь получил еще один шанс на ввод значение ключа

Замечания: Инструкция throw очень аналогична инструкции return – она прекращает выполнение метода, дальше мы не идем. Если мы нигде не ставим catch, то у нас выброс Exception очень похож на System.exit() – система завершает процесс. Но мы в любом месте можем поставить catch и, таким образом, предотвратить поломку кода.

Выводы*:*

Фактически при работе с исключениями весь материал делится на две части: синтаксис (ответ на вопрос, что компилятор пропустит, а что нет) и семантика (вопрос, как лучше делать) исключений. В отличие от вариантов с for, while, switch, использование исключений – более сложный механизм. Но он сложен не синтаксически, а семантически, по своему подходу. То есть при генерации исключений нужно думать о том - не как правильно его использовать, и с каким умыслом его использовать. То есть вопрос стоит так, в каких ситуациях стоит ли ломать систему и где, а в каких ситуациях ее восстанавливать.

В хорошей инженерной системе каждый любой модуль всегда проверяет все входные данные.

Можно еще сказать что существует иерархия различных способов прекращения выполнения некоего действия (или ряда действий) в виде участка кода и эта иерархия классифицирует возможные действия по мощности используемого оператора: continue, break, return, throw.

- **continue** прекращает выполнение данной итерации в цикле;
- **break** прекращает выполнение данного блока (например цикла);
- **return** – это инструкция выхода из данного метода (например прекращение выполнения функции);
- **throw** – еще более сильная инструкция, она прекращает выполнение данного метода и метода, который его вызвал. Так-как исключения вообще-то позволяют сломать весь стек работы программы.

Практическая работа № 19. Создание пользовательских исключений

Цель данной практической работы – научиться создавать собственные исключения.

Теоретические сведения

Язык Java предоставляет исчерпывающий набор классов исключений, но иногда при разработке программ вам потребуется создавать новые – свои собственные исключения, которые являются специфическими для потребностей именно вашего приложения. В этой практической работе вы научитесь создавать свои собственные пользовательские классы исключений. Как вы уже знаете, в Java есть два вида исключений- проверяемы и непроверяемые. Для начала рассмотрим создание пользовательских проверяемых исключений.

Создание проверяемых пользовательских исключений

Проверяемые исключения — это исключения, которые необходимо обрабатывать явно. Рассмотрим пример кода:

Листинг 19.1 – Пример обработки исключения

```
try (Scanner file = new Scanner(new File(fileName))){
  if (file.hasNextLine()) return file.nextLine();
} catch(FileNotFoundException e) {
  // Logging, etc
}
```

Приведенный на листинге 19.1 код является классическим способом обработки проверяемых исключений на Java. Хотя код выдает исключение FileNotFoundException, но в целом неясно, какова точная причина ошибки – не такого файла нет или же имя файла является недопустимым.

Чтобы создать собственное пользовательское исключение, мы будем наследоваться от класса java.lang.Exception. Давайте рассмотрим пример как это реализуется на практике и создадим собственный класс для проверяемого исключения с именем BadFileNameException:

Листинг 19.2 – Пример класса исключения

```
public class BadFileNameException extends Exception {
   public BadFileNameException(String errorMessage){
```

```
        super(errorMessage);
    }
}
```

Обратите внимание, что мы также должны написать конструктор в нашем классе, который принимает параметр типа String в качестве сообщения об ошибке, в котором вызывается конструктор родительского класса. Фактически это все, что нам нужно сделать, чтобы определить свое собственное пользовательское исключение.

Далее, давайте посмотрим, как мы можем использовать пользовательское исключение в программе

Листинг 19.3 – Пример генерации исключения из за неправильного имени файла

```
try (Scanner file = new Scanner(new File(fileName))){
    if (file.hasNextLine())
        return file.nextLine();
} catch (FileNotFoundException e) {
    if (!isCorrectFileName(fileName)) {
        throw new BadFileNameException("Bad filename : " + fileName );
    }
    //...
}
```

Мы создали и использовали свое собственное пользовательское исключение, теперь в случае ошибки, можно понять, что произошло, и какое именно исключение сработало. Как вы думаете, этого достаточно? Если ваш ответ да, то мы не узнаем основную причину, по которой сработало исключения. Как исправить программу. Для этого мы также можем добавить параметр java.lang.Throwable в конструктор. Таким образом, мы можем передать родительское исключение во время вызова метода:

Листинг 19.4 – Пример использования исключения

```
public BadFileNameException(String errorMessage, Throwable err) {
    super(errorMessage, err);
}
```

Теперь мы связали BadFileNameException с основной причиной возникновения данного исключения, например:

Листинг 19.5 – Пример обработки исключительной ситуации

некорректного имени файла

```
try (Scanner file = new Scanner(new File(fileName))){
    if (file.hasNextLine()) {
        return file.nextLine();
    }
} catch (FileNotFoundException err) {
    if (!isCorrectFileName(fileName)) {
        throw new BadFileNameException(
          "Bad filename: " + fileName, err);
    }
    // ...
}
```

Мы рассмотрели, как мы можем использовать пользовательские исключения в программах, учитывая их связь с причинами по которым они могут возникать.

Создание непроверяемых пользовательских исключений

В том же примере, который мы рассматривали выше предположим, что нам нужно такое пользовательское исключение, в котором обрабатывается ошибка, если файла не содержит расширения.

В этом случае нам как раз понадобится создать пользовательское непроверяемое исключение, похожее на предыдущее, потому что данная ошибка будет обнаружена только во время выполнения участка кода. Чтобы создать собственное непроверяемое исключение, нам нужно наследоваться от класса java.lang.RuntimeException:

Листинг 19.6 – Пример создания пользовательского класса исключения

```
public class BadFileExtensionException
  extends RuntimeException {
    public BadFileExtensionException(String errorMessage, Throwable
err) {
        super(errorMessage, err);
    }
}
```

Теперь, мы можем использовать это нестандартное исключение в рассматриваемом нами выше примере:

Листинг 19.7 – Пример использования нестандартного исключения

```
try (Scanner file = new Scanner(new File(fileName))){
    if (file.hasNextLine()) {
        return file.nextLine();
    } else {
        throw new IllegalArgumentException("Non readable file");
    }
} catch (FileNotFoundException err) {
    if (!isCorrectFileName(fileName)) {
        throw new BadFileNameException(
          "Bad filename: " + fileName , err);
    }

    //...
} catch(IllegalArgumentException err) {
    if(!containsExtension(fileName)) {
        throw new BadFileExtensionException(
          "Filename does not contain extension: " + fileName, err);
    }

    //...
}
```

Заключение

В приведенных выше примерах мы рассмотрели основные особенности обработки исключений.

Задания на практическую работу № 19

1. Клиент совершает покупку онлайн. При оформлении заказа у пользователя запрашивается фио и номер ИНН. В программе проверяется, действителен ли номер ИНН для такого клиента. Исключение будет выдано в том случае, если введен недействительный ИНН.

2. Предлагается модернизировать задачу из предыдущей практической работы (см. методические указания по выполнению практических работ №1-8) – задача сортировки студентов по среднему баллу. Необходимо разработать пользовательский

интерфейс для задачи поиска и сортировки (использовать массив интерфейсных ссылок- пример в лекции 5). Дополнить ее поиском студента по фио – в случае отсутствия такого студента необходимо выдавать собственное исключение. Схема классов прогоаммы приведена на Рис.19.1.

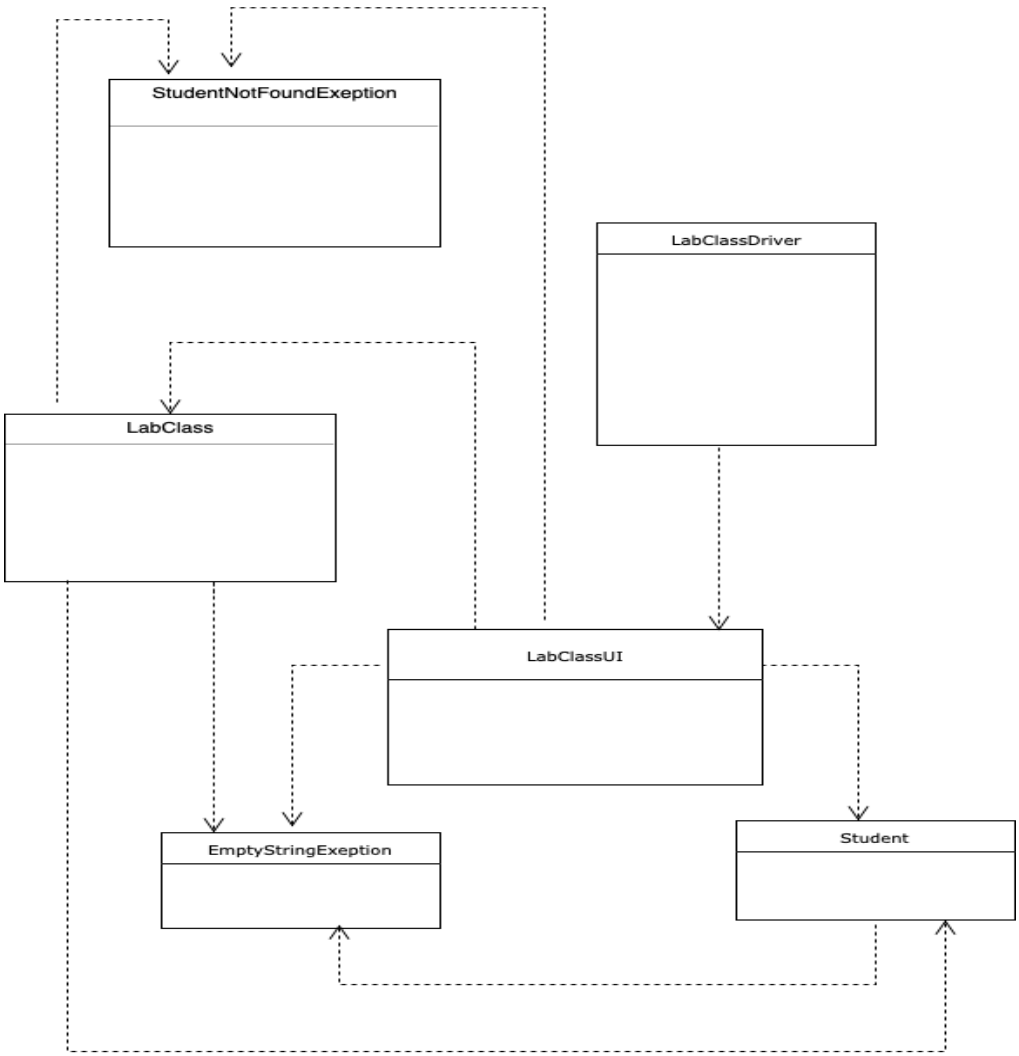

Рисунок 19.1. - UML диаграмма проекта LabClass с обработкой исключений

Ссылки на источники

1. http://www.embeddedsystemonline.com/programming-languages/java/10-java-exceptions

2. http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html
3. https://habrahabr.ru/company/golovachcourses/blog/223821/
4. http://kostin.ws/java/java-exceptions.html

Практическая работа № 20. Работа с дженериками.

Цель данной практической работы – научиться работать с обобщенными типами в Java и применять их в программах.

Теоретические сведения

Понятие дженериков.

Введение в Дженерики. В JDK представлены дженерики (перевод с англ. generics), которые поддерживают абстрагирование по типам (или параметризованным типам). В объявлении классов дженерики представлены обобщенными типами, в то время как пользователи классов могут быть конкретными типами, например во время создания объекта или вызова метода.

Вы, конечно, знакомы с передачей аргументов при вызове методов в языке C++. Когда вы передаете аргументы внутри круглых скобок () в момент вызова метода, то аргументы подставляются вместо формальных параметров с которыми объявлен и описан метод. Схожим образом в generics вместо передаваемых аргументов мы передаем информацию только о типах аргументов внутри угловых скобок <> (так называемая diamond notation или алмазная запись).

Основное назначение использования дженериков — это абстракция работы над типами при работе с коллекциями («Java Collection Framework»).

Например, класс ArrayList можно представить следующим образом для получения типа дженериков <E> следующим образом:

Листинг 20.1 – Класс ArrayList с параметризованным типом E

```
public class ArrayList<E> implements List<E> .... {
  //конструктор
  public ArrayList() { ..... }

  // методы public
  public boolean add(E e) { ..... }
  public void add(int index, E element) { ..... }
  public boolean addAll(int index, Collection<? extends E> c)
  public abstract E get(int index) { ..... }
  public E remove(int index)
  .....
}
```

Чтобы создать экземпляр ArrayList, пользователям необходимо предоставить фактический тип для <E> для данного конкретного экземпляра. Фактический тип будет заменять все ссылки на E внутри класса. Например:

Листинг 20.2 – Пример создания экземпляра класса ArrayList с типом Integer

```
ArrayList<Integer> lst1 = new ArrayList<Integer>();
// E подстановка Integer
lst1.add(0, new Integer(88));
lst1.get(0);

ArrayList<String> lst2 = new ArrayList<String>();
// E подстановка String
lst2.add(0, "Hello");
lst2.get(0);
```

В приведенном выше примере показано, что при проектировании или определении классов, они могут быть типизированными по общему типу; в то время как пользователи классов предоставляют конкретную фактическую информацию о типе во время создания экземпляра объекта типа класс. Информация о типе передается внутри угловых скобок <>, так же как аргументы метода передаются внутри круглой скобки ().

В этом плане общие коллекции не являются безопасными типами!

Если вы знакомы с классами- коллекциями, например, такими как ArrayList, то вы знаете, что они предназначены для хранения объектов типа java.lang.Object. Используя основной принцип ООП-полиморфизм, любой подкласс класса Object может быть заменен на Object. Поскольку Object является общим корневым классом всех классов Java, то коллекция, предназначенная для хранения Object, может содержать любые классы Java. Однако есть одна проблема. Предположим, например, что вы хотите определить ArrayList из объектов класса String. То при выполнении операций, например операции add(Object) объект класса String будет неявным образом преобразовываться в Object компилятором. Тем не менее, во время поиска ответственность программиста заключается в том, чтобы явно отказаться от Object обратно в строку. Если вы непреднамеренно добавили объект не-String в коллекцию, то компилятор не сможет обнаружить ошибку, а

понижающее приведение типов (от родителя к дочернему) завершится неудачно во время выполнения (будет сгенерировано ClassCastException throw).

Ниже приведен пример:

Листинг 20.3 – Пример использования интерфейсной ссылки List для инициализации объектом класса ArrayList

```
import java.util.*;
public class ArrayListWithoutGenericsTest {
  public static void main(String[] args) {
    List strLst = new ArrayList();
// List и ArrayList содержит тип Objects
    strLst.add("alpha");
// Неявное преобразование String в Object
    strLst.add("beta");
    strLst.add("charlie");
    Iterator iter = strLst.iterator();
    while (iter.hasNext()) {
      String str = (String)iter.next();
// необходимо выполнить понижающее преобразование типов Object обратно в String
      System.out.println(str);
    }
    strLst.add(new Integer(1234));
// на этапе Compile/runtime невозможно определить ошибку
    String str = (String)strLst.get(3);
// компиляция ok, но будет runtime ClassCastException
  }
}
```

Замечание. Мы могли бы использовать оператор создания объектов типа класс для проверки правильного типа перед добавлением. Но опять же, при создании объектов (инстанцировании) проблема обнаруживается во время выполнения. Как насчет проверки типа во время компиляции?

Использование дженериков в программах

Давайте напишем наш собственный «безопасный тип» ArrayList

Мы проиллюстрируем использование дженериков путем

написания нашего собственного типа изменяемого размера массива для хранения определенного типа объектов (аналогично ArrayList).

Начнем с версии MyArrayList без дженериков:

Листинг 20.4 – Пример создания динамически размещаемого массива

```
/* Динамически размещаемый массив, котрый содержит большая часть коллекции java.lang.Object — без дженериков.
*/
public class MyArrayList {
  private int size;
// количество элементов – размер коллекции
  private Object[] elements;

  public MyArrayList() {
//конструктор
    elements = new Object[10];
// начальная инициализация емкостью 10 элементов
    size = 0;
  }

  public void add(Object o) {
    if (size < elements.length) {
      elements[size] = o;
    } else {
//выделить массив большего размера и добавить элемент,
    }
   ++size;
  }

  public Object get(int index) {
    if (index >= size)
      throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
    return elements[index];
  }
  public int size() { return size; }
}
```

В данном примере класс MyArrayList не является безопасным типом. Например, если мы создаем MyArrayList, который предназначен для хранения только String, но допустим в процессе работы с ним добавляется Integer. Что произойдет? Компилятор не сможет обнаружить ошибку. Это связано с тем, что MyArrayList предназначен для хранения объектов Object, и любые классы Java являются производными от Object.

Листинг 20.5 – Пример использования класса MyArrayList

```
public class MyArrayListTest {
  public static void main(String[] args) {
/*Предусмотрено для хранения списка строк, но не является типобезопасным*/
    MyArrayList strLst = new MyArrayList();
/*добавление элементов строк (типа String) –это повышающее или расширяющее преобразование (upcasting) к типу Object*/

    strLst.add("alpha");
    strLst.add("beta");
/*при получении – необходимо явное понижающее преобразование (downcasting) назад к String*/
    for (int i = 0; i < strLst.size(); ++i) {
      String str = (String)strLst.get(i);
      System.out.println(str);
    }

/* случайно добавленный не-String объект, произойдет вызов во время выполнения ClassCastException. Компилятор не может отловить ошибку*/
    strLst.add(new Integer(1234));
//компиляция/выполнение - не можем обнаружить эту ошибку
    for (int i = 0; i < strLst.size(); ++i) {
      String str = (String)strLst.get(i);
/*компиляция ok,  при выполнении (runtime) ClassCastException*/
      System.out.println(str);
    }
  }
}
```

Если вы намереваетесь создать список объектов String, но непреднамеренно добавленный в этот список не-String-объекты будут преобразованы к типу Object. Компилятор не сможет проверить, является ли понижающее преобразование типов действительным во время компиляции (это известно как позднее связывание или динамическое связывание). Неправильное понижающее преобразование типов будет выявлено только во время выполнения программы, в виде исключения ClassCastException, а это слишком поздно, для того чтобы внести изменения в код и исправить работу программы. Компилятор не сможет поймать эту ошибку в момент компиляции. Вопрос в том, как заставить компилятор поймать эту ошибку и обеспечить безопасность использования типа во время выполнения.

Классы дженерики или параметризованные классы

В JDK введены так называемые обобщенные или параметризованные типы – generics или по-другому обобщенные типы для решения вышеописанной проблемы. Параметризованных (generic) классы и методы, позволяют использовать более гибкую и в то же время достаточно строгую типизацию, что особенно важно при работе с коллекциями. Использование параметризации позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр.

Дженерики или обобщенные типы позволяют вам абстрагироваться от использования конкретных типов. Вы можете создать класс с таким общим типом и предоставить информацию об определенном типе во время создания экземпляра объекта типа класс. А компилятор сможет выполнить необходимую проверку типов во время компиляции. Таким образом, вы сможете убедиться, что во время выполнения программы не возникнет ошибка выбора типа еще на этапе компиляции, что как раз является безопасностью для используемого типа.

Рассмотрим пример:

Листинг 20.6 – Объявление интерфейса java.util.List <E>:

```
public interface List<E> extends Collection<E> {
  boolean add(E o);
  void add(int index, E element);
  boolean addAll(Collection<? extends E> c);
```

```
    boolean containsAll(Collection<?> c);
    ......
}
```

Такая запись - <E> называется формальным параметром типа, который может использоваться для передачи параметров «типа» во время создания фактического экземпляра типа. Механизм похож на вызов метода Напомним, что в определении метода мы всегда объявляем формальные параметры для передачи данных в метод (при описании метода используются формальные параметры, а при вызове на их место подставляются аргументы). Например как представлено ниже:

```
// Определение метода
public static int max(int a, int b) {
// где int a, int b это формальные параметры
  return (a > b) ? a : b;
}
```

А во время вызова метода формальные параметры заменяются фактическими параметрами (аргументами). Например так:

```
//Вызов: формальные параметры, замененные фактическими параметрами
int maximum = max(55, 66);
// 55 и 66 теперь фактические параметры
int a = 77, b = 88;
maximum = max(a, b);        // a и b теперь фактические параметры
```

Параметры формального типа, используемые в объявлении класса, имеют ту же цель, что и формальные параметры, используемые в объявлении метода. Класс может использовать формальные параметры типа для получения информации о типе, когда экземпляр создается для этого класса. Фактические типы, используемые во время создания, называются фактическими типами параметров.

Вернемся к java. util.List <E>, итак в действительности, когда тип определен, например List <Integer>, все вхождения параметра формального типа <E> заменяются актуальным или фактическим параметром типа <Integer>. Используя эту дополнительную информацию о типе, компилятор может выполнить проверку типа во время компиляции и убедиться, что во время выполнения не будет ошибки при использовании типов.

Конвенция кода Java об именах для формальных типов

Мы должны помнить, что написания “чистого кода” необходимо руководствоваться конвенцией кода на Java. Поэтому, для создания имен формальных типов используйте один и тот же символ в верхнем регистре. Например,

- <E> для элемента коллекции;
- <T> для обобщенного типа;
- <K, V> ключ и значение.
- <N> для чисел
- S,U,V, и т.д. для второго, третьего, четвертого типа параметра

Рассмотрим пример параметризованного или обобщенного типа как класса. В нашем примере класс GenericBox принимает общий параметр типа E, содержит содержимое типа E. Конструктор, геттер и сеттер работают с параметризованным типом E. Метод нашего класса toString() демонстрирует фактический тип содержимого.

Листинг 20.7 – Пример параметризованного класса

```
public class GenericBox<E> {
  // Private переменная класса
  private E content;

  // конструктор
  public GenericBox(E content) {
    this.content = content;
  }

  public E getContent() {
    return content;
  }

  public void setContent(E content) {
    this.content = content;
  }

  public String toString() {
   return content + " (" + content.getClass() + ")";
  }
}
```

В следующем примере мы создаем GenericBoxes с различными типами (String, Integer и Double). Обратите внимание, что при преобразовании типов происходит автоматическая автоупаковка и распаковка для преобразования между примитивами и объектами-оболочками (мы с вами это изучали ранее).

Листинг 20.8 – Пример использования параметризованного класса

```
public class TestGenericBox {
  public static void main(String[] args) {
    GenericBox<String> box1 = new GenericBox<String>("Hello");
    String str = box1.getContent(); // явного понижающего преобразования (downcasting) не требуется
    System.out.println(box1);
    GenericBox<Integer> box2 = new GenericBox<Integer>(123);
//автоупаковка типа int в тип Integer
    int i = box2.getContent();      // (downcast) понижающее преобр. к типу Integer, автоупаковка в тип int
    System.out.println(box2);
    GenericBox<Double> box3 = new GenericBox<Double>(55.66);
////автоупаковка типа double в тип  Double
    double d = box3.getContent();      // (downcast) понижающее преобр.  к типу Double,
//автоупаковка в тип double
    System.out.println(box3);
  }
}
```

Вывод программы:

```
Hello (class java.lang.String)
123 (class java.lang.Integer)
55.66 (class java.lang.Double)
```

Задания на практическую работу №20

1. Создать обобщенный класс с тремя параметрами (T, V, K).
2. Класс содержит три переменные типа (T, V, K), конструктор, принимающий на вход параметры типа (T, V, K), методы возвращающие значения трех переменных. Создать метод, выводящий на консоль имена классов для трех переменных класса.

3. Наложить ограничения на параметры типа: T должен реализовать интерфейс Comparable (классы оболочки, String), V должен реализовать интерфейс Serializable и расширять класс Animal, K

4. Написать обобщенный класс MinMax, который содержит методы для нахождения минимального и максимального элемента массива. Массив является переменной класса. Массив должен передаваться в класс через конструктор. Написать класс Калькулятор (необобщенный), который содержит обобщенные статические методы - sum, multiply, divide, subtraction. Параметры этих методов - два числа разного типа, над которыми должна быть произведена операция.

5. Написать класс Matrix, на основе обобщенного типа, реализовать операции с матрицами

ССЫЛКИ ДЛЯ ЧТЕНИЯ

1) http://java-s-bubnom.blogspot.com/2015/07/generic.html
2) http://www.quizful.net/post/java-generics-tutorial
3) http://crypto.pp.ua/2010/06/parametrizovannye-klassy-java/
4) https://habr.com/post/207360/
5) https://habr.com/ru/company/sberbank/blog/416413/
6) https://www.geeksforgeeks.org/wildcards-in-java/
7) https://docs.oracle.com/javase/tutorial/java/generics/QandE/generics-questions.html
8) https://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html

Практическая работа № 21. Стирание типов в Джава

Цель данной практической работы – научиться работать с обобщенными типами в Java и применять прием стирание типов разработке программ на Джава

Теоретические сведения. Стирание типов

Из предыдущего примера может создаться видимость того, что компилятор заменяет параметризованный тип <E> актуальным или фактическим типом (таким как String, Integer) во время создания экземпляра объекта типа класс. Если это так, то компилятору необходимо будет создавать новый класс для каждого актуального или фактического типа (аналогично шаблону C ++).

На самом же деле происходит следующее - компилятор заменяет всю ссылку на параметризованный тип E на ссылку на Object, выполняет проверку типа и вставляет требуемые операторы, обеспечивающие понижающее приведения типов. Например, GenericBox компилируется следующим образом (который совместим с кодами без дженериков):

Листинг 21.1 – Пример класса GenericBox с полем Object

```
public class GenericBox {
  // Private переменная
  private Object content;

  // Конструктор
  public GenericBox(Object content) {
    this.content = content;
  }

  public Object getContent() {
    return content;
  }

  public void setContent(Object content) {
    this.content = content;
  }

  public String toString() {
```

```
        return content + " (" + content.getClass() + ")";
      }
    }
```

Компилятор сам вставляет требуемый оператор понижения типа (downcasting) в код:

```
GenericBox box1 = new GenericBox("Hello");  // upcast безопасно для типов
String str = (String)box1.getContent();      // компилятор вставляет операцию понижения типа (downcast)
System.out.println(box1);
```

Вывод. Таким образом, для всех типов используется одно и то же определение класса. Самое главное, что байт-код всегда совместим с теми классами, у которых нет дженериков. Этот процесс называется *стиранием типа.*

Рассмотрим операцию “стирания типов” с помощью нашего «безопасного типа» ArrayList, который мы рассматривали ранее в паримере.

Вернемся, к примеру MyArrayList. С использованием дженериков мы можем переписать нашу программу как показано в листинге ниже:

Листинг 21.2 – Пример параметризованного класса GenericBox

```
//Динамически выделяемая память для массива с дженериками
public class MyGenericArrayList<E> {
  private int size;
// количество элементов- емкость списка
  private Object[] elements;

  public MyGenericArrayList() {  //конструктор
    elements = new Object[10];
// выделяем память сразу для 10 элементов списка при его создании
    size = 0;
  }

  public void add(E e) {
    if (size < elements.length) {
      elements[size] = e;
    } else {
// добавляем элемент к списку и увеличиваем счетчик количества
```

элементов

```
            }
            ++size;
        }

        public E get(int index) {
            if (index >= size)
                throw new IndexOutOfBoundsException("Index: " + index + ",
Size: " + size);
            return (E)elements[index];
        }

        public int size() { return size; }
    }
```

В объявлении MyGenericArrayList <E> объявляется класс-дженерик с формальным параметром типа <E>. Во время фактического создания объектов типа класс, например, MyGenericArrayList <String>, определенного типа <String> или параметра фактического типа, подставляется вместо параметра формального типа <E>.

Помните! Что Дженерики реализуются компилятором Java в качестве интерфейсного преобразования, называемого стиранием, которое переводит или перезаписывает код, который использует дженерики в не обобщенный код (для обеспечения обратной совместимости). Это преобразование стирает всю информацию об общем типа. Например, ArrayList <Integer> станет ArrayList. Параметр формального типа, такой как <E>, заменяется объектом по умолчанию (или верхней границей типа). Когда результирующий код не корректен, компилятор вставляет оператор преобразования типа.

Следовательно, код, транслированный компилятором, т о есть переведенный код выглядит следующим образом:

Листинг 21.3 – Пример кода транслированногоо компилятором

```
public class  MyGenericArrayList {
    private int size;     // количество элементов
    private Object[] elements;
```

```
    public MyGenericArrayList() {  // конструктор
      elements = new Object[10];  // выделяем память для первых 10
объектов
      size = 0;
    }

    // Компилятор заменяет параметризованный тип E на Object, но
проверяет, что параметр e имеет тип E, когда //он используется для
обеспечения безопасности типа

    public void add(Object e) {
      if (size < elements.length) {
        elements[size] = e;
      } else {
        // allocate a larger array and add the element, omitted
      }
      ++size;
    }

    // Компилятор заменяет E на Object и вводит оператор
понижающего преобразования типов (E <E>) для //типа
возвращаемого значения при вызове метода

    public Object get(int index) {
      if (index >= size)
        throw new IndexOutOfBoundsException("Index: " + index + ",
Size: " + size);
      return (Object)elements[index];
    }

    public int size() {
      return size;
    }
  }
```

Когда класс создается с использованием актуального или фактического параметра типа, например. MyGenericArrayList <String>,

компилятор гарантирует, что add(E e) работает только с типом String. Он также вставляет соответствующий оператор понижающее преобразование типов в соответствие с типом возвращаемого значения E для метода get(). Например,

Листинг 21.4 – Пример использования класса дженерика

```
public class MyGenericArrayListTest {
  public static void main(String[] args) {
    // безопасный тип для хранения списка объектов Strings (строк)
    MyGenericArrayList<String>  strLst = new MyGenericArrayList <String>();

    strLst.add("alpha");   // здесь компилятор проверяет, является ли аргумент типом String
    strLst.add("beta");

    for (int i = 0; i < strLst.size(); ++i) {
      String str = strLst.get(i); //компилятор вставляет оператор
  //понижающего преобразования operator (String)
      System.out.println(str);
    }
    strLst.add(new Integer(1234));  // компилятор обнаруживает аргумент,
  //который не является объектом String, происходит ошибка компиляции
  }
  }
```

Выводы: с помощью дженериков компилятор может выполнять проверку типов во время компиляции и обеспечивать безопасность использования типов во время выполнения.

Запомните. В отличие от «шаблона» в C ++, который создает новый тип для каждого определенного параметризованного типа, в Java класс generics компилируется только один раз, и есть только один файл класса, который используется для создания экземпляров для всех конкретных типов.

Обобщенные методы (параметризованные методы)

Методы также могут быть определены с помощью общих типов (аналогично родовому классу). Например,

Листинг 21.5 – Пример параметризованного метода

```
public static <E> void ArrayToArrayList(E[] a, ArrayList<E> lst) {
  for (E e : a) lst.add(e);
}
```

При объявлении обобщенного метода или метода-дженерика могут объявляться параметры формального типа (например, такие как <E>, <K, V>) перед возвращаемым типом (в примере выше это static <E> void). Параметры формального типа могут затем использоваться в качестве заполнителей для типа возвращаемого значения, параметров метода и локальных переменных в общем методе для правильной проверки типов компилятором.

Подобно классам-дженерикам, когда компилятор переводит общий метод, он заменяет параметры формального типа, используя операцию стирания типа. Все типы заменяются типом Object по умолчанию (или верхней границей типа – типом классом, стоящим на самой вершине иерархии наследования). Переведенная на язык компилятора версия программы выглядит следующим образом:

Листинг 21.6 – Пример трансляции кода компилятором

```
public static void ArrayToArrayList(Object[] a, ArrayList lst) {
// компилятор проверяет, есть ли тип E[],
// lst имеет тип ArrayList<E>
 ArrayList<E> for (Object e : a) lst.add(e);
// компилятор проверяет является ли e типом E
}
```

Компилятор всегда проверяет, что a имеет тип E[], lst имеет тип ArrayList <E>, а e имеет тип E во время вызова для обеспечения безопасности типа. Например,

Листинг 21.7 – Пример класса с использованием параметризованного метода

```
import java.util.*;
public class TestGenericMethod {
  public static <E> void ArrayToArrayList(E[] a, ArrayList<E> lst) {
    for (E e : a) lst.add(e);
  }
```

```
    public static void main(String[] args) {
    ArrayList<Integer> lst = new ArrayList<Integer>();
        Integer[] intArray = {55, 66};  // автоупаковка
        ArrayToArrayList(intArray, lst);
        for (Integer i : lst) System.out.println(i);

        String[] strArray = {"one", "two", "three"};
        //ArrayToArrayList(strArray, lst);
    //ошибка компиляции ниже
      }
TestGenericMethod.java:16:<E>ArrayToArrayList(E[],java.util.ArrayList<E>)
in TestGenericMethod cannot be applied to
(java.lang.String[],java.util.ArrayList<java.lang.Integer>)
ArrayToArrayList(strArray, lst);
    }
```

У дженериков есть необязательный синтаксис для указания типа для общего метода. Вы можете поместить фактический тип в угловые скобки <> между оператором точки и именем метода.

```
TestGenericMethod.<Integer>ArrayToArrayList(intArray, lst);
```

Обратите внимание на точку после имени метода

Подстановочный синтаксис в Java (WILD CARD)

Одним из наиболее сложных аспектов generic-типов (обобщенных типов) в языке Java являются wildcards (подстановочные символы, в данном случае – «?»), и особенно – толкование и разбор запутанных сообщений об ошибках, происходящих при wildcard capture (подстановке вычисляемого компилятором типа вместо wildcard). В своем труде «Теория и практика Java» (Java theory and practice) старейший Java-разработчик Брайен Гетц расшифровывает некоторые из наиболее загадочно выглядящих сообщений об ошибках, выдаваемых компилятором «javac», и предлагает решения и варианты обхода, которые помогут упростить использование generic-типов.

Рассмотрим следующую строку кода:

```
ArrayList<Object> lst = new ArrayList<String>();
```

Компиляция вызовет ошибку - «несовместимые типы», поскольку ArrayList <String> не является ArrayList <Object>. Эта ошибка противоречит нашей интуиции в отношении полиморфизма, поскольку

мы часто присваиваем экземпляр подкласса ссылке на суперкласс.

Рассмотрим эти два утверждения:

```
List<String> strLst = new ArrayList<String>();
// строчка 1
List<Object> objLst = strList;
// строчка 2 – ошибка компиляции
```

Выполнение строки 2 генерирует ошибку компиляции. Но если строка 2 выполняется, то некоторые объекты добавляются в objLst, а strLst будут «повреждены» и больше не будет содержать только строки. (так-как переменные objLst и strLst содержат одинаковую ссылку или ссылаются на одну и ту-же область памяти).

Учитывая вышеизложенное, предположим, что мы хотим написать метод printList (List <.>) Для печати элементов списка. Если мы определяем метод как printList (List <Object> lst), он может принимать только аргумент List <object>, но не List <String> или List <Integer>. Например,

Листинг 21.8– Пример параметризованного метода для печати элементов списка

```
import java.util.*;
public class TestGenericWildcard {

  public static void printList(List<Object> lst) {  // принимает только список объектов,не список подклассов объектов
    for (Object o : lst) System.out.println(o);
  }

  public static void main(String[] args) {
    List<Object> objLst = new ArrayList<Object>();
    objLst.add(new Integer(55));
    printList(objLst);   // соответствие

    List<String> strLst = new ArrayList<String>();
    strLst.add("one");
    printList(strLst);  // ошибка компиляции
  }
}
```

Использование подстановочного знака без ограничений в описании типа <?>

Чтобы разрешить проблему, описанную выше, необходимо использовать подстановочный знак (?), он используется в дженериках

для обозначения любого неизвестного типа. Например, мы можем переписать наш printList () следующим образом, чтобы можно было передавать список любого неизвестного типа.

```
public static void printList(List<?> lst) {
  for (Object o : lst) System.out.println(o);
}
```

Использование подстановочного знака в начале записи типа <? extends тип>

Подстановочный знак <? extends type> обозначает тип и его подтип. Например,

```
public static void printList(List<? extends Number> lst) {
  for (Object o : lst) System.out.println(o);
}
```

List<? extends Number> принимает список Number и любой подтипNumber, например, List <Integer> и List <Double>. Понятно, что обозначение типа <?> можно интерпретировать как <? extends Object>, который применим ко всем классам Java.

Другой пример,

```
//List<Number> lst = new ArrayList<Integer>();     // ошибка компиляции
List<? extends Number> lst = new ArrayList<Integer>();
```

На самом деле стирание типов обеспечивает совместимость вашего кода со старыми версиями java, которые могут вообще не содержать дженериков.

Задания на практическую работу №21

1. Написать метод для конвертации массива строк/чисел в список.
2. Написать класс, который умеет хранить в себе массив любых типов данных (int, long etc.).
3. Реализовать метод, который возвращает любой элемент массива по индексу.
4. Написать функцию, которая сохранит содержимое каталога в список и выведет первые 5 элементов на экран.
5. *Реализуйте вспомогательные методы в классе Solution, которые должны создавать соответствующую коллекцию и помещать туда переданные объекты. Методы newArrayList, newHashSet параметризируйте общим типом T. Метод newHashMap параметризируйте парой <K, V>, то есть типами K- ключ и V-значение. Аргументы метода newHashMap должны принимать. Класс содержит

три переменные типа (T, V, K), конструктор, принимающий на вход

Практическая работа № 22. Абстрактные типы данных. Стек

Цель данной практической работы – научиться разрабатывать программы с абстрактными типами данных на языке Джава и применять паттерн MVC при разработке программ

Теоретические сведения

Стек — это линейная структура данных, которая следует принципу LIFO (Last In First Out) . Стек имеет один конец, куда мы можем добавлять элементя и извлекать их оттуда, в отличие от очереди котрая имеет два конца (спереди и сзади).Стек содержит только один указатель top(верхушка стека), указывающий на самый верхний элемент стека. Всякий раз, когда элемент добавляется в стек, он добавляется на вершину стека, и этот элемент может быть удален только из стека только сверху. Другими словами, стек можно определить как контейнер, в котором вставка и удаление элементов могут выполняться с одного конца, известного как вершина стека.

Примеры стеков – пирамида, стопка тарелок или книг, магазин в пистолете

Стеку присущи следующие характеристики:

- Стек — это абстрактный тип данных с заранее определенной емкостью, что означает, что эта структура данных имеет ограниченный размер, то есть может хранить количество элементов, опредееленное размерностью стека.
- Это структура данных, в которой строго определен порядок вставки и удаления элементов, и этот порядок может быть LIFO или FILO.

Порядок работы со стеком

Рассмотрим пример, допустим стек работает по схеме LIFO. Как видно на рис. ниже, в стеке пять блоков памяти; поэтому размер стека равен 5.

Предположим, мы хотим хранить элементы в стеке, и предположим, что в начале стек пуст. Мы приняли размер стека равным 5, как показано на рис.22.1 ниже, в который мы будем помещать элементы один за другим, пока стек не заполнится.

Поскольку наш стек заполнен, то количество элементов в нем равно 5. В приведенных выше случаях мы можем наблюдать, что он

заполняется элементами снизу вверх, при каждом добавлении нового элемента в стек. Стек растет снизу вверх.

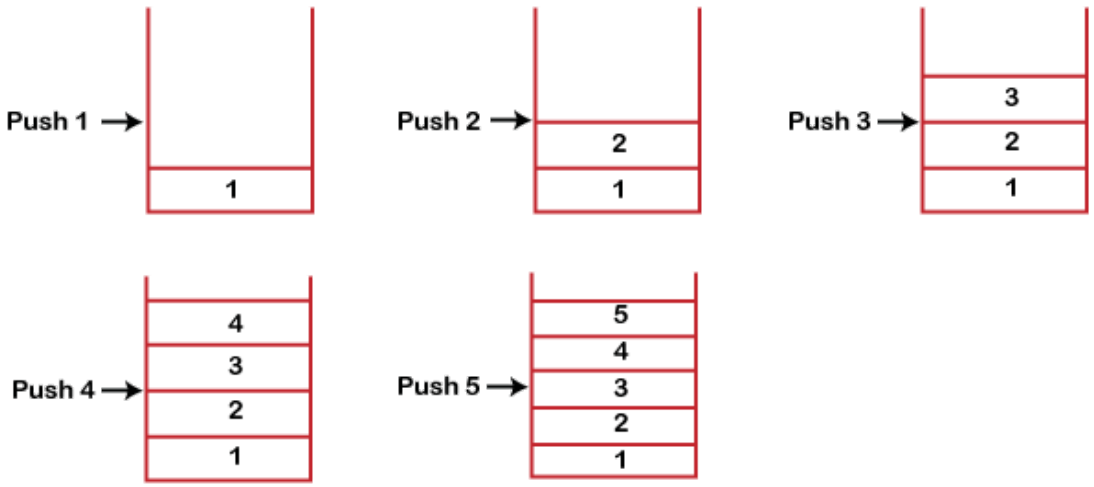

Рисунок 22.1. – Пример работы стека из пяти элементов

В приведенном выше случае значение 5 вводится первым, поэтому оно будет удалено только после удаления всех остальных элементов.

Стандартные операции со стеком

Ниже приведены некоторые общие операции, реализованные в стеке:

- push(): когда мы добавляем элемент в стек, эта операция называется push. Если стек заполнен, возникает состояние переполнения.
- pop(): Когда мы удаляем элемент из стека, эта операция называется pop. Если стек пуст, это означает, что в стеке нет элементов, это состояние известно как состояние потери значимости.
- isEmpty(): определяет, пуст стек в настоящий момент или нетили нет.
- isFull(): определяет, заполнен стек или нет.
- peek(): возвращает элемент в заданной позиции.
- count(): возвращает общее количество элементов, доступных в стеке.
- change(): изменяет элемент в заданной позиции.
- display(): печатает все элементы, доступные в стеке.

Опрация push()

Рассмотрим шаги, связанные с выполнением этой операции:

1) Прежде чем вставить элемент в стек, мы проверяем, заполнен ли стек.
2) Если мы пытаемся вставить элемент в стек, а стек уже полон, то возникает условие *переполнения* .

Когда мы инициализируем стек, мы устанавливаем значение верхушки стека как top как -1, чтобы убедиться, что стек пуст.

Когда новый элемент помещается в стек, сначала увеличивается значение вершины, т. е . **top=top+1,** и элемент будет помещен в новую позицию верхушки стека (top). Элементы будут вставляться до тех пор, пока мы не достигнем *максимального* размера стека.

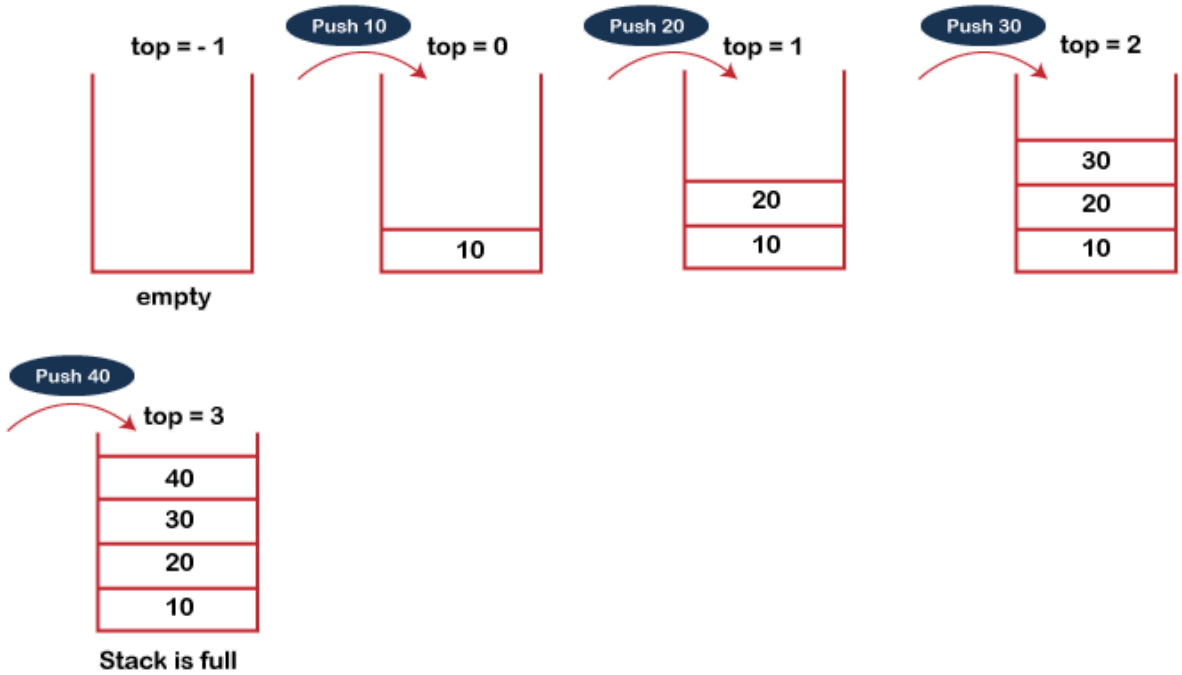

Рисунок 22.2. – Пример выполнения операции push() на стеке из пяти элементов

Операция pop()

Рассмотрим шаги, связанные с выполнением этой операции:

1) Перед удалением элемента из стека мы проверяем, не пуст ли стек.
2) Если мы попытаемся удалить элемент из пустого стека, то возникнет состояние потери значимости.
3) Если стек не пуст, мы сначала обращаемся к элементу, на который указывает вершина.
4) После выполнения операции извлечения значение top уменьшается на 1, т . е. top=top-1.

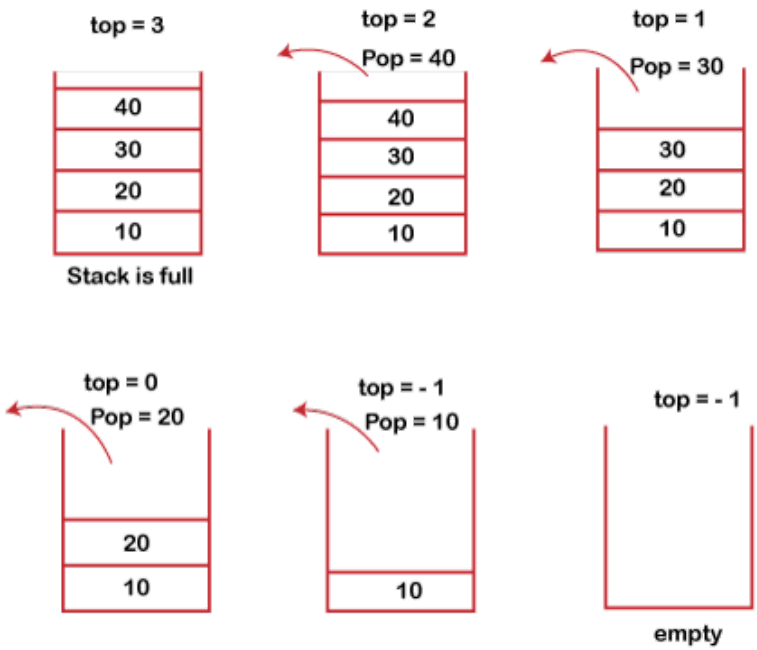

Рисунок 22.3. – Пример выполнения операции pop() на стеке из пяти элементов

Пример реализации класса стека представлен на листинге 22.1

Листинг 22.1 – Реализация стека на массиве на языке Джава

```
class Stack {
  // store elements of stack
  private int arr[];
  // represent top of stack
  private int top;
  // total capacity of the stack
  private int capacity;

  // Creating a stack
  Stack(int size) {
    // initialize the array
    // initialize the stack variables
    arr = new int[size];
    capacity = size;
    top = -1;
  }

  // push elements to the top of stack
```

```
public void push(int x) {
  if (isFull()) {
    System.out.println("Stack OverFlow");

    // terminates the program
    System.exit(1);
  }

  // insert element on top of stack
  System.out.println("Inserting " + x);
  arr[++top] = x;
}

// pop elements from top of stack
public int pop() {

  // if stack is empty
  // no element to pop
  if (isEmpty()) {
    System.out.println("STACK EMPTY");
    // terminates the program
    System.exit(1);
  }

  // pop element from top of stack
  return arr[top--];
}

// return size of the stack
public int getSize() {
  return top + 1;
}

// check if the stack is empty
public Boolean isEmpty() {
  return top == -1;
}
```

```
  // check if the stack is full
  public Boolean isFull() {
    return top == capacity - 1;
  }

  // display elements of stack
  public void printStack() {
    for (int i = 0; i <= top; i++) {
      System.out.print(arr[i] + ", ");
    }
  }

  public static void main(String[] args) {
    Stack stack = new Stack(5);

    stack.push(1);
    stack.push(2);
    stack.push(3);

    System.out.print("Stack: ");
    stack.printStack();

    // remove element from stack
    stack.pop();
    System.out.println("\nAfter popping out");
    stack.printStack();
```

Пример реализации стека приведен в приложении А

Язык Джава предоставляет встроенный Stack, который можно использовать для реализации стека.

Листинг 22.2 – Реализация стека с помощью Stack

```
import java.util.Stack;

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

```
    // create an object of Stack class
    Stack<String> animals= new Stack<>();

    // push elements to top of stack
    animals.push("Dog");
    animals.push("Horse");
    animals.push("Cat");
    System.out.println("Stack: " + animals);

    // pop element from top of stack
    animals.pop();
    System.out.println("Stack after pop: " + animals);
    }
}
```

В листинге 22.2 означает:

- animals.push() - вставить элементы на вершину стека
- animals.pop() - удалить элемент из вершины стека

Обратите внимание, мы при создании стека так называемая алмазная запись - угловые скобки <String>. Это означает, что стек имеет универсальный тип данных.

Задания на практическую работу №22

Общее заданиие Написать калькулятор для чисел с использованием RPN (Reverse Polish Notation в пер. на русск. яз. - обратной польской записи)

Необходимые сведения об алгоритме

Алгоритм Обратной польской нотации (ОПН) — форма записи математических выражений, в которой операнды расположены перед знаками операций. Также именуется как обратная польская запись, обратная бесскобочная запись (ОБЗ).

Рассмотрим запись арифметических выражений, в которых сначала следуют два операнда арифметической операции, а затем знак операции. Например:

Обратная польская нотация	Обычная нотация
2 3 +	2 + 3

2 3 * 4 5 * +	(2 * 3) + (4 * 5)
2 3 4 5 6 * + - /	2 / (3 - (4 + (5 * 6)))

Нотация записи выражений, представленная в левом столбце таблицы называется обратной польской нотацией (записью) (Reverse Polish Notation, RPN). В теории языков программирования эта нотация называется *постфиксной нотацией*. Обычная нотация называется алгебраической или *инфиксной нотацией* («ин» от англ. *inside*, то есть между операндами).

Есть также префиксная нотация, активно используемая в языке Си (сначала имя функции, а затем её аргументы), а также в языке LISP.

Заметьте, что скобки в обратной польской нотации не нужны. В частности, если во втором примере мы опустим скобки, выражение по прежнему будет интерпретироваться однозначно.

Транслятор RPN-выражений основан на стэке. Каждое следующее число помещается в стэк. Если встречается знак операции (обозначим его *), то два числа из стека извлекаются (a = pop(), b = pop()), для них вычисляется значение соответствующей бинарной арифметической операции, и результат помещается в стек (push(a * b)).

Задание 1. Напишите программу-калькулятор арифметических выражений записанных в обратной польской нотации (RPN-калькулятор).

Задание 2. Напишите графический интерфейс для калькулятора, используя знания полученные ранее при програмиировании GUI с использованием SWING и AWT. Используйте паттерн проектирования MVC. Интерфейс может выглядеть как на рис. 22.1 или как на рис. 22.2

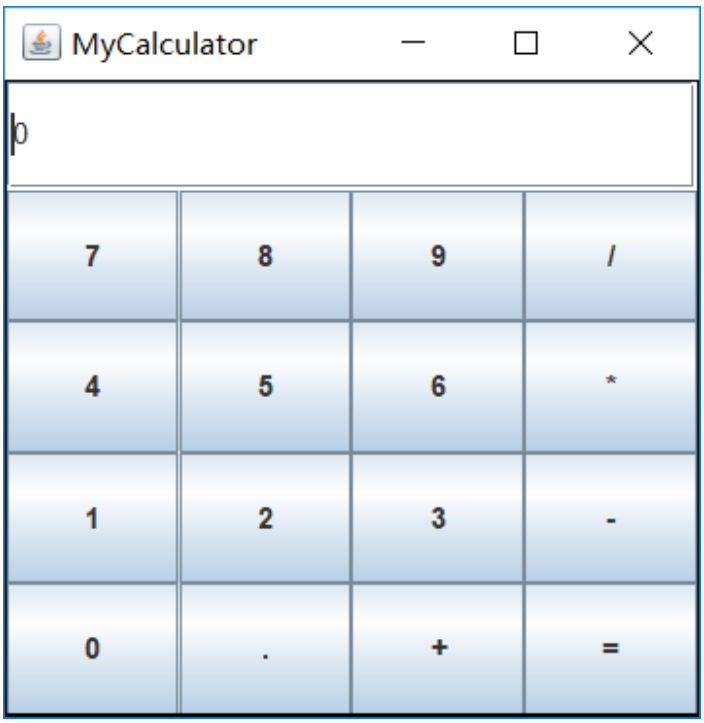

Рисунок 22.4. – Общий вид графического интерфейса для программы калькулятора

Рисунок 22.5. – Общий вид графического интерфейса для проораммы калькулятора

Задание 2. Постройте систему тестов и проверьте, что ваш калькулятор успешно проходит все тесты и «защищён от дурака» (как дурака-пользователя программы, так и дурака-программиста, использующего ваш стек и калькулятор). Например, если вводится выражение, в котором число операций

превосходит число помещенных в стек элементов (например 1 2 + *), то программа не допустит уменьшения переменной sp до отрицательных значений, а выдаст предупреждение «Невозможно выполнить POP для пустого стека».

Запмечание: вы можете выполнить работу можно в двух вариантах (первый вариант проще, второй- труднее).

Практическая работа № 23. Абстрактные типы данных. Очередь

Цель: цель данной практической работы – – научиться разрабатывать программы с абстрактными типами данных на языке Джава

Теоретические сведения. Очередь

1. Очередь можно пределить как упорядоченный список, который позволяет выполнять операции вставки на одном конце, называемом REAR , и операции удаления, которые выполняются на другом конце, называемом FRONT

2. Очередь называется работает по дисциплине обслуживания «первый пришел — первый обслужен» (FCFS, first come first served)

3. Например, люди, стоящие в кассу магазина образуют оччередь оплаты покупок.

Пример очереди можно увидеть на рис. 23.1. Операция dequeue означает удаление элемента из начала очереди, а операция enqueue добавление элемента в конец очереди.

Рисунок 23.1. – Общая схема работы очереди

Использование очередей в разработке программ.

Очереди широко используются

- в качестве списков ожидания для одного общего ресурса, такого как принтер, диск, ЦП;
- при асинхронной передаче данных (когда данные не передаются с одинаковой скоростью между двумя процессами), например. трубы, файловый ввод-вывод, сокеты;

- в качестве буферов в большинстве приложений, таких как медиаплеер MP3, проигрыватель компакт-дисков и т. д.;
- для ведения списка воспроизведения в медиаплеерах, чтобы добавлять и удалять песни из списка воспроизведения;
- в операционных системах для обработки прерыванийи и при реализации работы алгоритмов планирования и диспетчиризации.

Таблица 23.1 Сравнение временной сложности для различных операций над очередью

	Временная сложность							
	Среднее				Худшее			
	Доступ	Поиск	Вставка	Удаление	Доступ	Поиск	Вставка	Удаление
Очередь	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)

Классификация очередей

Очередь — это структура данных, похожая на очередь в реальном мире.

Очередь — это структура данных, в которой элемент, котрый приходит в очередь первым, удаляется первым, то есть работа очереди строго сосотвествует политике FIFO (First-In-First-Out).

Очередь также можно определить как список или коллекцию, в которой вставка выполняется с одного конца, известного как конец очереди, тогда как удаление выполняется с другого конца, известного как начало очереди.

Реальным примером очереди является очередь за билетами возле кинозала, где человек, входящий в очередь первым, получает билет первым, а последний человек, входящий в очередь, получает билет последним. Аналогичный подход используется в очереди как в структуре данных.

На рис.23.2 представлена структура очереди из четырех элементов.

Рисунок 23.2. – Очередь из четырех элементов

Виды очередей:

- Простая очередь или линейная очередь
- Циклическая очередь
- Очередь с приоритетами
- Двусторонняя очередь или Дек (англ. Deque)

Простая или линейная очередь

В линейной очереди или Queue вставка элмента происходит с одного конца, а удаление — с другого. Конец, на котором происходит вставка, называется задняя часть очереди, а конец, на котором происходит удаление, называется передняя часть очереди. Работа линейной очереди организуется по правилу FIFO – первый пришел-первый ушел.

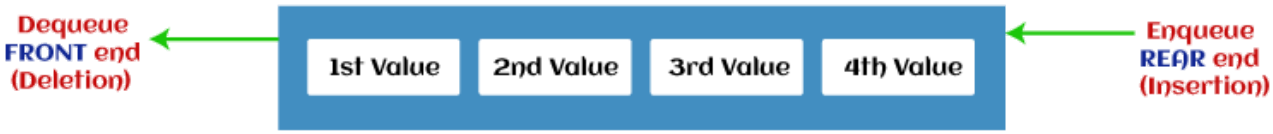

Рисунок 23.3. – Очередь из четрых элементов

Основным недостатком использования линейной очереди является то, что вставка выполняется только с заднего конца. Если первые три элемента будут удалены из очереди, мы не сможем вставить больше элементов, даже если в линейной очереди есть свободное место. В этом случае линейная очередь показывает состояние переполнения, поскольку задняя часть указывает на последний элемент очереди.

Циклическая очередь

В циклической очереди все узлы представлены как элементы цепочки. После последнего элемента очереди сразу идет первый или начальный элемент. Эта очередь похожа на линейную очередь, за исключением того, что последний элемент очереди соединяется с первым элементом. Эта очередь получила название кольцевого

буфера, так как все ее концы соединены друг с другом. Схематично эта очередь представлена на рис. 23.4.

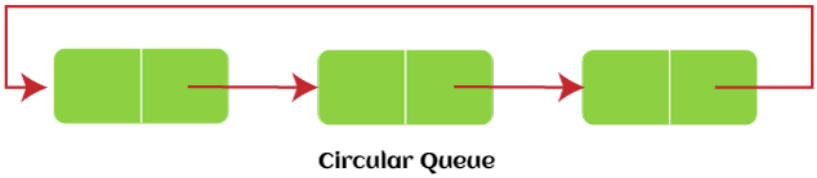

Рисунок 23.4. – Кольцевая очередь

Недостаток линейной очереди преодолевается при использовании круговой очереди. Если в циклической очереди есть пустое место, новый элемент можно добавить в пустое место, просто увеличив значение Rear.

Основным преимуществом использования циклической очереди является лучшее использование памяти.

Очередь с приоритетами

Это особый тип очереди, в которой элементы располагаются в зависимости от их приоритета. Это особый тип структуры данных очереди, в которой каждый элемент такой очереди имеет связанный с ним приоритет. Допустим, какие-то элементы встречаются с одинаковым приоритетом, тогда они будут располагаться по принципу FIFO. Представление очереди с приоритетами показано на рис.23.5.

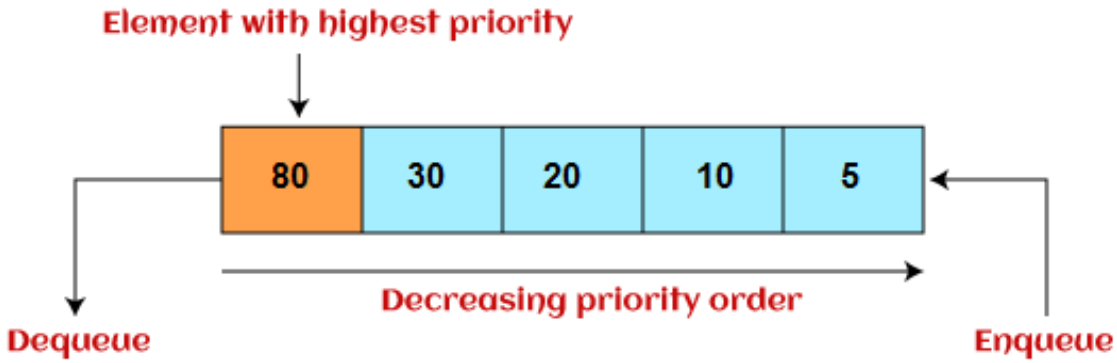

Рисунок 23.5. – Очередь с приоритетами

Вставка в такую очередь происходит на основе поступления элемента в соотвествии с его приоритетом, а удаление в данной очереди происходит на основе приоритета. Очередь с приоритетом в основном используется для реализации алгоритмов планирования ЦП

в операционных системах.

Типы очередей с приоритетами

Существует два типа очереди, которые обсуждаются следующим образом:

1) Очередь с возрастающим приоритетом. В очереди с возрастающим приоритетом элементы могут быть вставлены в произвольном порядке, но только самые маленькие могут быть удалены первыми. Предположим массив с элементами 7, 5 и 3 в одном порядке, поэтому вставка может быть выполнена с той же последовательностью, но порядок удаления элементов 3, 5, 7.
2) Очередь с убывающим приоритетом. В очереди с убывающим приоритетом элементы могут быть вставлены в произвольном порядке, но первым может быть удален только самый большой элемент, то есть с самым высоким приооритетом. Предположим, массив с элементами 7, 3 и 5 вставлены в одном порядке, поэтому вставка может быть выполнена с той же последовательностью, но порядок удаления элементов 7, 5, 3.

Deque (или двойная очередь)

В Deque или Double Ended Queue вставка и удаление могут выполняться с обоих концов очереди либо спереди, либо сзади. Это означает, что мы можем вставлять и удалять элементы как с переднего, так и с заднего конца очереди.

Deque можно использовать как средство проверки палиндрома, что означает, что если мы читаем строку с обоих концов, то строка будет одинаковой.

Deque можно использовать и как стек, и как очередь, поскольку он позволяет выполнять операции вставки и удаления на обоих концах. Deque можно рассматривать как стек, потому что стек следует принципу LIFO (Last In First Out), в котором вставка и удаление могут выполняться только с одного конца. А в структуре данных Дек можно выполнять и вставку, и удаление с одного конца, и нужно отметить ччто работа дека не следует правилу FIFO. Схематично представление

дека представлено на рис. 23.6.

Рисунок 23.6. – Очередь Дек

Существует два типа дека:

- Очередь ограниченным вводом
- Очередь с ограниченным выводом

Очередь с ограниченным вводом. Как следует из названия, в очереди с ограниченным вводом операция вставки может выполняться только на одном конце, а удаление может выполняться на обоих концах.

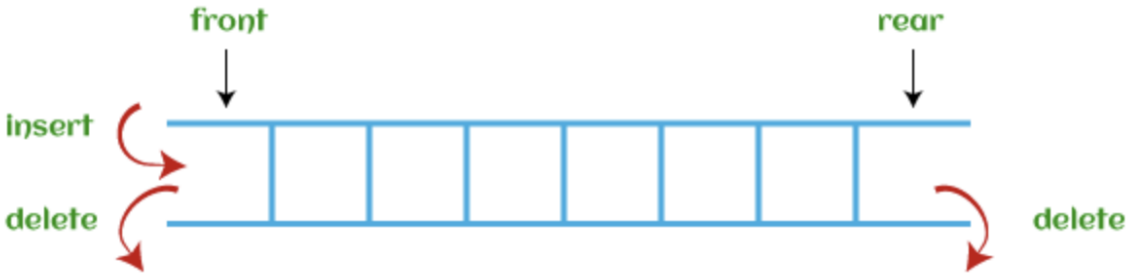

Рисунок 23.7. – Очередь с ограниченным вводом

Очередь с ограничениями нва вывод. Ограниченная очередь вывода —как следует из названия, в очереди вывода с ограниченным доступом операция удаления может выполняться только на одном конце, а вставка — на обоих концах.

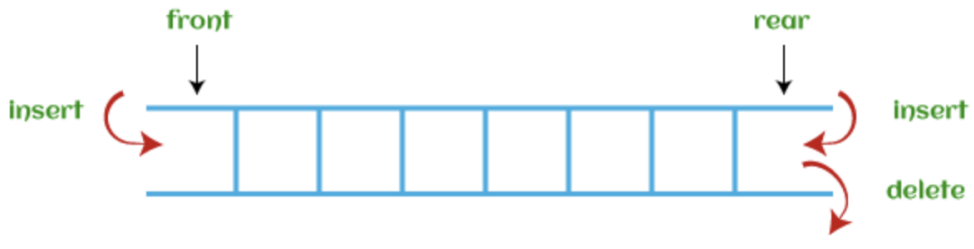

Рисунок 23.8. – Очередь с ограниченным выводом

Операции, выполняемые над очередью

Основные операции, которые можно выполнять в очереди,

перечислены ниже:

- **Enqueue():** эта операция используется для вставки элемента в конец очереди. Возвращает пустоту.
- **Dequeue():** Операция выполняет удаление из внешнего интерфейса очереди. Он также возвращает элемент, который был удален из внешнего интерфейса. Он возвращает целочисленное значение.
- **Peek()** Просмотр очереди: это третья операция, которая возвращает элемент, на который указывает передний указатель в очереди, но не удаляет его.
- **isfFull() (Queue overflow):** Проверка переполнения очереди (заполнено): показывает состояние переполнения, когда очередь полностью заполнена.
- **isEmpty() (Queue underflow):** проверка очереди на пустоту Показывает состояние потери значимости, когда очередь пуста, т. е. в очереди нет элементов.

Способы реализации очереди

Существует два способа реализации очереди:

- Реализация с использованием массива: последовательное размещение в очереди может быть реализовано с использованием массива.
- Реализация с использованием связанного списка: размещение связанного списка в очереди может быть реализовано с использованием связанного списка.

Листинг 23.1 – Реализация очереди на языке Джава

```
public class Queue {
  int SIZE = 5;
  int items[] = new int[SIZE];
  int front, rear;

  Queue() {
    front = -1;
    rear = -1;
  }

  // check if the queue is full
  boolean isFull() {
    if (front == 0 && rear == SIZE - 1) {
```

```
        return true;
    }
    return false;
}

// check if the queue is empty
boolean isEmpty() {
    if (front == -1)
        return true;
    else
        return false;
}

// insert elements to the queue
void enQueue(int element) {

    // if queue is full
    if (isFull()) {
        System.out.println("Queue is full");
    }
    else {
        if (front == -1) {
            // mark front denote first element of queue
            front = 0;
        }

        rear++;
        // insert element at the rear
        items[rear] = element;
        System.out.println("Insert " + element);
    }
}

// delete element from the queue
int deQueue() {
    int element;

    // if queue is empty
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return (-1);
    }
    else {
        // remove element from the front of queue
```

```
        element = items[front];

        // if the queue has only one element
        if (front >= rear) {
            front = -1;
            rear = -1;
        }
        else {
            // mark next element as the front
            front++;
        }
        System.out.println( element + " Deleted");
        return (element);
    }
}

// display element of the queue
void display() {
    int i;
    if (isEmpty()) {
        System.out.println("Empty Queue");
    }
    else {
        // display the front of the queue
        System.out.println("\nFront index-> " + front);

        // display element of the queue
        System.out.println("Items -> ");
        for (i = front; i <= rear; i++)
            System.out.print(items[i] + "  ");

        // display the rear of the queue
        System.out.println("\nRear index-> " + rear);
    }
}

public static void main(String[] args) {

    // create an object of Queue class
    Queue q = new Queue();

    // try to delete element from the queue
    // currently queue is empty
    // so deletion is not possible
```

```
        q.deQueue();

        // insert elements to the queue
        for(int i = 1; i < 6; i ++) {
            q.enQueue(i);
        }

        // 6th element can't be added to queue because queue is full
        q.enQueue(6);

        q.display();

        // deQueue removes element entered first i.e. 1
        q.deQueue();

        // Now we have just 4 elements
        q.display();

    }
}
```

Результат выполнения программы на листинге 23.1

```
Queue is empty
Insert 1
Insert 2
Insert 3
Insert 4
Insert 5
Queue is full

Front index-> 0
Items ->
1  2  3  4  5
Rear index-> 4
1 Deleted

Front index-> 1
Items ->
2  3  4  5
Rear index-> 4
```

Вам не обязательно самим писать АТД Очередь. Ведь язык Java

предоставляет встроенный интерфейс Queue, который можно использовать для реализации очереди. Такая реализация представлена на листинге ниже.

Листинг 23.2 – Реализация очереди с помощью интерфейса Queue

```
import java.util.Queue;
import java.util.LinkedList;

class Main {

  public static void main(String[] args) {
    // Creating Queue using the LinkedList class
    Queue<Integer> numbers = new LinkedList<>();

    // enqueue
    // insert element at the rear of the queue
    numbers.offer(1);
    numbers.offer(2);
    numbers.offer(3);
    System.out.println("Queue: " + numbers);

    // dequeue
    // delete element from the front of the queue
    int removedNumber = numbers.poll();
    System.out.println("Removed Element: " + removedNumber);

    System.out.println("Queue after deletion: " + numbers);
  }
}
```

Результат выполнения программы на листинге 23.2

```
Queue: [1, 2, 3]
Removed Element: 1
Queue after deletion: [2, 3]
```

Понятие инварианта в объектно-ориентированном программировании относится к некоторому набору условий или утверждений, которые должны выполняться на протяжении всей жизни объекта класса. Эти утверждения должны выполняться с момента вызова конструктора для создания объекта, в конце вызова каждого метода, изменяющего состояние объекта (сеттера) до конца жизни объекта. Эти условия подтверждают, что поведение объекта неизменно в течение всей его жизни и что объект поддерживает свое

четко определенное состояние, как и предполагалось при его проектировании. Однако инвариант не обязательно должен оставаться истинным во время выполнения метода, изменяющего состояние (сеттера), но должен оставаться истинным в конце его выполнения.

Инвариант класса - это просто свойство, которое выполняется для всех экземпляров класса, всегда, независимо от того, что делает другой код.

Задания на практическую работу №23

Замечания

Для выполнения данной практической работы вам необходимы следующие знания:

1 Классы в Джава

- ✔ Инвариант класса
- ✔ Задачи инкапсуляции

2 Интерфейсы в Джава

- ✔ Интерфейс как синтаксический контракт
- ✔ Интерфейс как семантический контракт

3 Абстрактные базовые классы и наследование

- ✔ Устранение дублирования
- ✔ Вынос изменяемой логики в наследников

Задание 1. Реализовать очередь на массиве

- Найдите инвариант структуры данных «очередь». Определите функции, которые необходимы для реализации очереди. Найдите их пред- и постусловия.
- Реализуйте классы, представляющие циклическую очередь с применением массива.
  - ✔ Класс ArrayQueueModule должен реализовывать один экземпляр очереди с использованием переменных класса.
  - ✔ Класс ArrayQueueADT должен реализовывать очередь в виде абстрактного типа данных (с явной передачей ссылки на экземпляр очереди).
  - ✔ Класс ArrayQueue должен реализовывать очередь в виде

класса (с неявной передачей ссылки на экземпляр очереди).

- ✔ Должны быть реализованы следующие функции(процедуры)/методы:
  - enqueue – добавить элемент в очередь;
  - element – первый элемент в очереди;
  - dequeue – удалить и вернуть первый элемент в очереди;
  - size – текущий размер очереди;
  - isEmpty – является ли очередь пустой;
  - clear – удалить все элементы из очереди.

- Инвариант, пред- и постусловия записываются в исходном коде в виде комментариев.
- Обратите внимание на инкапсуляцию данных и кода во всех трех реализациях.
- Напишите тесты реализованным классам.

Задание 2. Очередь на связанном списке

4 Определите интерфейс очереди Queue и опишите его контракт.

5 Реализуйте класс LinkedQueue — очередь на связном списке.

6 Выделите общие части классов LinkedQueue и ArrayQueue в базовый класс AbstractQueue.

Дополнительные задания

Задание 3. Вычисление выражений

1. Разработайте классы Const, Variable, Add, Subtract, Multiply, Divide для вычисления выражений с одной переменной.
2. Классы должны позволять составлять выражения вида

new Subtract(new Multiply(new Const(2), new Variable("x")), new Const(3)).evaluate(5)

Замечание. При вычислении такого выражения вместо каждой переменной подставляется значение, переданное в качестве параметра методу evaluate (на данном этапе имена переменных игнорируются). Таким образом, результатом вычисления приведенного примера должно стать число 7.

3. Для тестирования программы должен быть создан класс Main, который вычисляет значение выражения x^2-2x+1, для x, заданного в командной строке.
4. При выполнение задания следует обратить внимание на:

- Выделение общего интерфейса создаваемых классов.
- Выделение абстрактного базового класса для бинарных операций.

Задание 4

1. Доработайте предыдущее задание, так что бы выражение строилось по записи вида x * (y - 2)*z + 1
2. Для этого реализуйте класс ExpressionParser с методом TripleExpression parse(String).
3. В записи выражения могут встречаться: умножение *, деление /, сложение +, вычитание -, унарный минус -, целочисленные константы (в десятичной системе счисления, которые помещаются в 32-битный знаковый целочисленный тип), круглые скобки, переменные (x, y и z) и произвольное число пробельных символов в любом месте (но не внутри констант).
4. Приоритет операторов, начиная с наивысшего
    - унарный минус;
    - умножение и деление;
    - сложение и вычитание.
5. Для выражения 1000000*x*x*x*x*x/(x-1) вывод программы должен иметь следующий вид:

```
          x      f
0         0
1         division by zero
2         32000000
3         121500000
4         341333333
5         overflow
6         overflow
7         overflow
8         overflow
9         overflow
10        overflow
```

Ограничения

1. Результат division by zero (overflow) означает, что в процессе вычисления произошло деление на ноль (переполнение).
2. Разбор выражений рекомендуется производить методом рекурсивного спуска. Алгоритм должен работать за линейное время.
3. При выполнении задания следует обратить внимание на дизайн и обработку исключений.

4. Человеко-читаемые сообщения об ошибках должны выводится на консоль.
5. Программа не должна «вылетать» с исключениями (как стандартными, так и добавленными)

Практическая работа № 24. Паттерны проектирования. порождающие паттерны: абстрактная фабрика, фабричный метод

Цель: научиться применять порождающие паттерны при разработке программ на Java. В данной практической работе рекомендуется использовать следующие паттерны: Абстрактная фабрика и фабричный метод.

Теоретические сведения. Понятие паттерна.

Паттерны (или шаблоны) проектирования описывают типичные способы решения часто встречающихся проблем при проектировании программ.

Некоторые из преимуществ использования шаблонов проектирования:

1) Шаблоны проектирования уже заранее определены и обеспечивают стандартный отраслевой подход к решению повторяющихся в программном коде проблем, вследствие этого разумное применение шаблона проектирования экономит время на разработку.
2) Использование шаблонов проектирования способствует реализации одного из преимуществ ООП – повторного использования кода, что приводит к более надежному и удобному в сопровождении коду. Это помогает снизить общую стоимость владения программного продукта.
3) Поскольку шаблоны проектирования заранее определены то это упрощает понимание и отладку нашего кода. Это приводит к более быстрому развитию проектов, так как новые новые члены команды понимают код.

Шаблоны проектирования Джава делятся на три категории: порождающие, структурные и поведенческие шаблоны проектирования. Так же есть еще шаблоны проектировани, например MVC – model-view-controller.

Шаблон проектирования Фабрика (factory), его также называют фабричный метод используется, когда у нас есть суперкласс с несколькими подклассами, и на основе ввода нам нужно вернуть один из подклассов. Этот шаблон снимает с себя ответственность за создание экземпляра класса из клиентской программы в класс

фабрики. Давайте сначала узнаем, как реализовать фабричный шаблон проектирования в java, а затем мы рассмотрим преимущества фабричного шаблона. Мы увидим некоторые примеры использования фабричного шаблона проектирования в JDK. Обратите внимание, что этот шаблон также известен как шаблон проектирования фабричный метод.

Суперкласс в шаблоне проектирования Фабрика может быть интерфейсом, абстрактным классом или обычным классом Java. Для нашего примера шаблона проектирования фабрики у нас есть абстрактный суперкласс с переопределенным toString() методом для целей тестирования см. листинг 24.1

Листинг 24.1 – Пример абстрактного класса Computer.java

```
package ru.mirea.it;

public abstract class Computer {

    public abstract String getRAM();
    public abstract String getHDD();
    public abstract String getCPU();

    @Override
    public String toString(){
        return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+",
CPU="+this.getCPU();
    }
}
```

Создадим производные классы шаблона фабрики. Допустим, у нас есть два подкласса ПК и сервер с реализацией ниже см листинг 24.2.

Листинг 24.2 –Дочерний класс PC фабричного шаблона.

```
package ru.mirea.it;

public class PC extends Computer {

    private String ram;
    private String hdd;
    private String cpu;

    public PC(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
```

```
        this.cpu=cpu;
    }
    @Override
    public String getRAM() {
        return this.ram;
    }

    @Override
    public String getHDD() {
        return this.hdd;
    }

    @Override
    public String getCPU() {
        return this.cpu;
    }

}
```

На листинге 24.3 представлен еще один дочерний класс шаблона фабрика.

Листинг 24.3 –Дочерний класс Server фабричного шаблона.

```
package ru.mirea.it
public class Server extends Computer {

    private String ram;
    private String hdd;
    private String cpu;

    public Server(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public String getRAM() {
        return this.ram;
    }

    @Override
    public String getHDD() {
```

```
        return this.hdd;
    }

    @Override
    public String getCPU() {
        return this.cpu;
    }

}
```

Теперь, когда у нас есть готовые суперклассы и подклассы, мы можем написать наш фабричный класс. На листинге 24.4 представлена его базовая реализация.

Листинг 24.4 – Базовая реализация фабричного класса

```
package ru.mirea.it;

import ru.mirea.it.Computer;
import ru.mirea.it.PC;
import ru.mirea.it.Server;

public class ComputerFactory {

    public static Computer getComputer(String type, String ram, String hdd,
String cpu){
        if("PC".equalsIgnoreCase(type)) return new PC(ram, hdd, cpu);
        else if("Server".equalsIgnoreCase(type)) return new Server(ram, hdd,
cpu);

        return null;
    }
}
```

Отметим некоторые важные моменты метода Factory Design Pattern;

Мы можем оставить класс Factory Singleton или оставить метод, возвращающий подкласс, как статический .

Обратите внимание, что на основе входного параметра создается и возвращается другой подкласс. getComputer() является заводским методом.

Простая тестовая клиентская программа, в которой используется приведенная выше реализация шаблона проектирования factory представлена на листинге 24.5.

Листинг 24.5 – Класс Тестер

```
package ru.mirea.it.abstractfactory;
import ru.mirea.it.ComputerFactory;
import ru.mirea.it.Computer;

public class TestFactory {

   public static void main(String[] args) {
      Computer pc = ComputerFactory.getComputer("pc","2 GB","500
GB","2.4 GHz");
      Computer server = ComputerFactory.getComputer("server","16 GB","1
TB","2.9 GHz");
      System.out.println("Factory PC Config::"+pc);
      System.out.println("Factory Server Config::"+server);
   }

}
```

Результат работы тестовой программы:

Factory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz

Factory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz

Преимущества шаблона Фабрика

Шаблон проектирования Factory обеспечивает подход к коду для интерфейса, а не для реализации.

Фабричный шаблон удаляет экземпляры реальных классов реализации из клиентского кода. Фабричный шаблон делает наш код более надежным, менее связанным и легко расширяемым. Например, мы можем легко изменить реализацию класса ПК (PC), потому что клиентская программа не знает об этом.

Основное преимущество, котрое мы получаем - Фабричный шаблон обеспечивает абстракцию между реализацией и клиентскими классами посредством наследования.

Примеры шаблонов проектирования Factory в JDK

Методы java.util.Calendar, ResourceBundle и NumberFormat getInstance() используют шаблон Factory. Методы valueOf()метод в классах-оболочках, таких как Boolean, Integer и т. д. также используют шаблон Factory

Паттерн Abstract Factory похож на паттерн Factory и представляет собой фабрику фабрик. Если вы знакомы с шаблоном проектирования factory в java, вы заметите, что у нас есть один класс Factory, который возвращает различные подклассы на основе

предоставленных входных данных, и для достижения этого класс factory использует операторы if-else или switch. В шаблоне абстрактной фабрики мы избавляемся от блока if-else и создаем класс фабрики для каждого подкласса, а затем класс абстрактной фабрики, который возвращает подкласс на основе входного фабричного класса.

Мы рассмотрели на листингах 24.1, 24.2, 24.3 классы, теперь прежде всего нам нужно создать интерфейс Abstract Factory или абстрактный класс .ComputerAbstractFactory.java см. листинг 24.6

Листинг 24.6 – Абстрактный класс .ComputerAbstractFactory.java

```
package ru.mirea.it.abstractfactory;

import ru.mirea.it.Computer;

public interface ComputerAbstractFactory {

    public Computer createComputer();

}
```

братите внимание, что метод createComputer() возвращает экземпляр суперкласса Computer. Теперь наши фабричные классы будут реализовывать этот интерфейс и возвращать соответствующий подкласс PCFactory.java см.листинг 24.7

Листинг 24.7 – Фабричный класс PCFactory.java

```
package ru.mirea.it.abstractfactory;

import ru.mirea.it.Computer;
import ru.mirea.it.PC;

public class PCFactory implements ComputerAbstractFactory {

    private String ram;
    private String hdd;
    private String cpu;

    public PCFactory(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public Computer createComputer() {
```

```
        return new PC(ram,hdd,cpu);
    }

}
```

очно так же у нас будет фабричный класс для Server подкласса ServerFactory.java см листинг 24.8

Листинг 24.8 – Фабричный класс ServerFactory.java

```
package ru.mirea.it.abstractfactory;

import ru.mirea.it.Computer;
import ru.mirea.it.Server;

public class ServerFactory implements ComputerAbstractFactory {

    private String ram;
    private String hdd;
    private String cpu;

    public ServerFactory(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }

    @Override
    public Computer createComputer() {
        return new Server(ram,hdd,cpu);
    }

}
```

Теперь мы создадим клиентский класс, который предоставит клиентским классам точку входа для создания подклассов ComputerFactory.java см листинг 24.9

Листинг 24.9 – Класс ComputerFactory

```
package ru.mirea.it.abstractfactory;

import ru.mirea.it.Computer;

public class ComputerFactory {

    public static Computer getComputer(ComputerAbstractFactory factory){
        return factory.createComputer();
```

```
    }
}
```

Обратите внимание, что это простой класс и метод getComputer, принимающий аргумент ComputerAbstractFactory и возвращающий объект Computer. На этом этапе реализацияуже должна быть понятной. Давайте напишем простой тестовый метод и посмотрим, как использовать абстрактную фабрику для получения экземпляров подклассов TestDesignPatterns.java см листинг 24.10.

Листинг 24.10 – Тестовый класс TestDesignPatterns.java

```
Package  ru.mirea.it.design.test;

import ru.mirea.it.abstractfactory.PCFactory;
import ru.mirea.it.design.abstractfactory.ServerFactory;
import ru.mirea.it..design.factory.ComputerFactory;
import ru.mirea.it.design.model.Computer;

public class TestDesignPatterns {

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

    private static void testAbstractFactory() {
        Computer pc =
com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new
PCFactory("2 GB","500 GB","2.4 GHz"));
        Computer server =
com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new
ServerFactory("16 GB","1 TB","2.9 GHz"));
        System.out.println("AbstractFactory PC Config::"+pc);
        System.out.println("AbstractFactory Server Config::"+server);
    }
}
```

Рузультат работы тестовой программы

AbstractFactory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz

AbstractFactory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz

Ниже представлена UML диаграмма классов с использованием шаблона Фабрика см рис 21.1

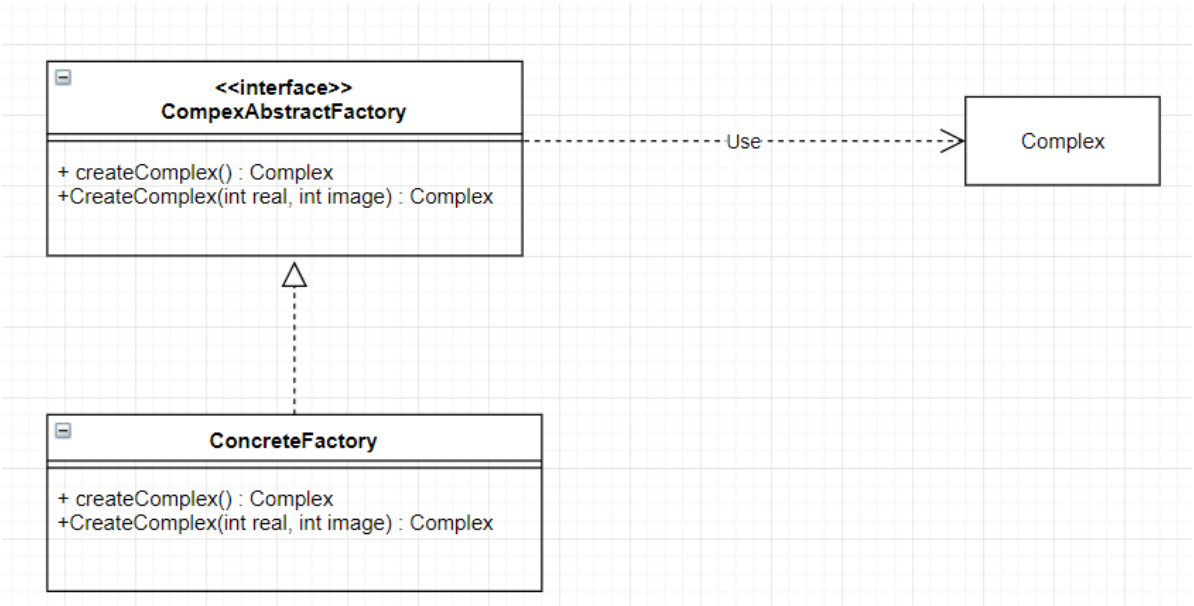

Рисунок 24.1. – UML диаграмма проекта ComplexNumber с обработкой исключений

Задания на практическую работу № 21

Задание 1.

Разработать программную реализацию по UML диаграмме, представленной на рис.24.1с использованием изучаемых паттернов.

Задание 2.

Реализовать класс Абстрактная фабрика для различных типов стульев: Викторианский стул, Многофункциональный стул, Магический стул, а также интерфейс Стул, от которого наследуются все классы стульев, и класс Клиент, который использует интерфейс стул в своем методе Sit (Chair chair).

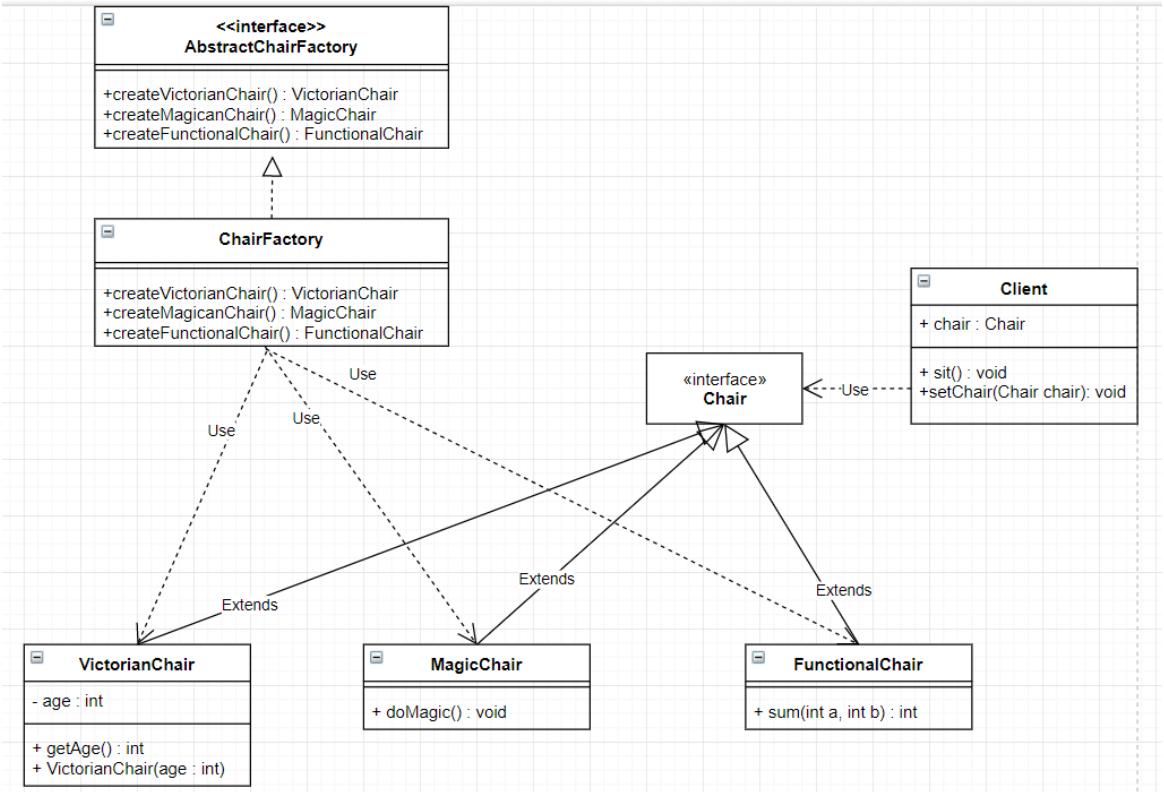

Рисунок 21.2. - UML диаграмма проекта для компании XXX

Задание 3.

Легенда “Компании XXX нужно написать редактор текста, редактор изображений и редактор музыки. В трёх приложениях будет очень много общего: главное окно, панель инструментов, команды меню будут весьма схожими. Чтобы не писать повторяющуюся основу трижды, вам поручили разработать основу (каркас) приложения, которую можно использовать во всех трёх редакторах.”

На совещании в компании была принята следующая архитектура проекта:

Исходные данные. Есть некий базовый интерфейс IDocument, представляющий документ неопределённого рода. От него впоследствии будут унаследованы конкретные документы: TextDocument, ImageDocument, MusicDocument и т.п. Интерфейс IDocument перечисляет общие свойства и операции для всех документов.

- При нажатии пунктов меню File -> New и File -> Open требуется создать новый экземпляр документа (конкретного подкласса). Однако каркас не должен быть привязан ни к какому конкретному виду документов.
- Нужно создать фабричный интерфейс ICreateDocument. Этот интерфейс содержит два абстрактных фабричных метода:

CreateNew и CreateOpen, оба возвращают экземпляр IDocument

- Каркас оперирует одним экземпляром IDocument и одним экземпляром ICreateDocument. Какие конкретные классы будут подставлены сюда, определяется во время запуска приложения.

Требуется:

1. создать перечисленные классы. Создать каркас приложения — окно редактора с меню File. В меню File реализовать пункты New, Open, Save, Exit.

продемонстрировать работу каркаса на примере текстового редактора. Потребуется создать конкретный унаследованный класс TextDocument и фабрику для него — CreateTextDocument.

Список литературы

1. Зорина Н.В. Программирование на языке Джава. Ч.1 [Электронный ресурс]: учебное пособие / Н. В. Зорина. — М.: РТУ МИРЭА, 2021. — Электрон. опт. диск (ISO) https://library.mirea.ru/share/4174
2. Зорина Н.В. Программирование на языке Джава. Ч.1 [Электронный ресурс]: практикум / Н. В. Зорина. — М.: РТУ МИРЭА, 2021. — Электрон. опт. диск (ISO) https://library.mirea.ru/share/4182
3. The Java Tutorials, http://docs.oracle.com/javase/tutorial/index.html
4. История версий Java Edition [Электронный ресурс]. – Режим доступа: URL: https://minecraft.fandom.com/ru/wiki/%D0%98%D1%81%D1%82%D0%BE%D1%80%D0%B8%D1%8F_%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B9_(Java_Edition) дата доступа: 15.08.2021.
5. Зорина Н.В. Объектно-ориентированный анализ и программирование [Электронный ресурс]: учебное пособие / Н. В. Зорина. – М.: РТУ МИРЭА, 2019 — 111 с.
6. Хорстманн, К. Java 2. Библиотека профессионала. Том 1. Основы [Текст] / Кей Хорстманн 11 издание. – М.: Издательский дом Вильямс, 2019 — 864 с., с ил.; ISBN 978-5-907114-79-1, 978-0-13-516630-7
7. Хорстманн Кей С. Java SE 9. Базовый курс. - Издательство Альфа-книга 2019 — 576 с., с ил.; ISBN: 978-0-13-469472-6
8. Блох Джошуа, Java: эффективное программирование 3-е издание. – М. : Издательский дом Вильямс, 2019 — 464 с., с ил.; ISBN 978-5-6041394-4-8, 978-0-13-468599-1
9. Брюс Эккель, Философия Java. Серия: Классика computer science, 4-е издание. Питер: 2019 — 1168 с., с ил.; ISBN: 978-5-4461-1107-7
10. Java Platform, Standard Edition Oracle JDK Migration Guide Release 13 F19399-01 September 2019 [Электронный ресурс]. – Режим доступа: https://docs.oracle.com/en/java/javase/13/migrate/migration-guide.pdf, дата доступа: 12.05.2021.
11. JavaSE APIs & Documentation [Электронный ресурс]. – Режим доступа: https://docs.oracle.com/en/java/javase/index.html, дата доступа: 21.06.2021.

12. Спецификация языка Java 8 [Электронный ресурс]. – Режим доступа: URL: https://docs.oracle.com/javase/specs/jls/se8/html/index.html, дата доступа: 11.12.2021

13. Зорина Н.В. Объектно-ориентированное программирование [Электронный ресурс]: конспект лекций/ Н. В. Зорина. – М.: РТУ МИРЭА, 2019 — 119 с.

14. Зорина Н.В. Объектно-ориентированный анализ и программирование [Электронный ресурс]: учебное пособие / Н. В. Зорина. — М.: РТУ МИРЭА, 2019. — Электрон. опт. диск (ISO) https://library.mirea.ru/share/3240

15. Зорина Н.В. Объектно-ориентированное программирование на Java [Электронный ресурс]: метод. рекомендации / Н. В. Зорина [и др.]. — М.: РТУ МИРЭА, 2019. — Электрон. опт. диск (ISO) https://library.mirea.ru/share/2995

16. Объектно-ориентированное программирование на Java [Электронный ресурс]: практикум / Н. В. Зорина. — М.: РТУ МИРЭА, 2019. — Электрон. опт. https://library.mirea.ru/share/3195

17. Кишори, Ш. Java 9. Полный обзор нововведений. Для быстрого ознакомления и миграции / Ш. Кишори; перевод с английского А.А. Слинкин. — Москва: ДМК Пресс, 2019. — 544 с. — ISBN 978-5-97060-575-2. — Текст: электронный // Электронно-библиотечная система «Лань» : [сайт]. — URL: https://e.lanbook.com/book/108130 (дата обращения: 19.11.2019). — Режим доступа: для авториз. пользователей.

18. Васильев Алексей Николаевич Java. Объектно-ориентированное программирование: Базовый курс по объектно-ориентированному программированию: для магистров и бакалавров / А. Н. Васильев. — СПб.: Питер, 2014. — 397 с.: ил. — (Учебное пособие). — Библиогр.: с. 377

19. https://javarush.ru/groups/posts/458-key-khorstmann-i-ego-1500-slov-o-tom-kak-statjh-luchshim-java-programmistom--

ПРИЛОЖЕНИЕ А. Листинг программы реализация стека на Джава

```
import java.util.EmptyStackException;
import java.lang.Exception;

   public class MyStack{
      Node head;
      Node tail;

      private class Node{
         Node next;
         String data;

         public Node(String data){
            this.data = data;
         }

         public String toString(){
            return data;
         }
      }

//return null if unsuccessful (assuming there is no String null stored in the stack)
//or return a special value that can never represent a legal piece of data
//(eg: return -1 for a stack containing only non-negative numbers)
//In C, return error code and pointer to the data to return
//In Java, handle exception

      public void push(String data){
         Node newHead = new Node(data);
         newHead.next = this.head;
         this.head = newHead;
      }
//return null if no more
      public String pop() throws EmptyStackException{
         Node toReturn = this.head;
         if(toReturn != null){
            this.head = toReturn.next;
            return toReturn.data;
         }else{
            throw new EmptyStackException();
```

```
        }
    }

    public String toString(){
        Node current = this.head;
        String s = new String();
        while(current != null){
            s += current.data + " ";
            current = current.next;
        }
        return s;
    }
//Maintain Linked List Tail Pointer P35
    public void remove(String data){
//check data cannot be null
        if (data == null) {
// throw new Exception("data cannot be null");
        }

        Node current = this.head;
        if (current == null) {
// throw new Exception("Empty Stack");
            return;
//will jumoed out automatically by exception
        }
        if (current.data == data) {
            this.head = current.next;
            //if the only one is deleted
            if (this.head == null) {
                this.tail = null;
            }
//return current;
            return;
        }
        while(current.next != null){
            if(current.next.data == data){
                Node toDelte = current.next;
//if changes the last item, update the tail
                if (toDelte.next == null) {
                    this.tail = current;
                }
                current.next = toDelte.next;
                //return toDelete;
                return;
```

```
            }
            current = current.next;
        }
//      throw new Exception("No such object");
    }
//after null means insert at the begining
    public void insertAfter(String data, String after){
        if (data == null) {
//throw new Exception("data cannot be null");
        }
        Node toInsert = new Node(data);
        Node current = this.head;
        //if after is null, add at the most front
        if (after == null) {
            toInsert.next = current;
            this.head = toInsert;
            return;
        }
        while(current != null){
            if(current.data == after){
                toInsert.next = current.next;
                //if add at the end, update the tail
                if (current.next == null) {
                    this.tail = toInsert;
                }
                current.next = toInsert;
                return;
            }
            current = current.next;
        }
//the object "after" do not exist, throw an exception
//throw new Exception("after do not exist !!");
    }

//when call pop, try... catch...

}

public class Main {
    public static void main(String[] args){
        MyStack stack = new MyStack();
        stack.push("1");
        System.out.println(stack);
        System.out.println(stack.pop());
```

```
        System.out.println(stack);
        stack.push("1");
        System.out.println(stack);
        stack.push("2");
        System.out.println(stack);


        stack.push("3");
        stack.push("4");
        System.out.println(stack);
        stack.remove("3");
        System.out.println(stack);
        stack.remove("5");

        stack.insertAfter("8","1");
        System.out.println(stack);
        stack.insertAfter("3",null);
        System.out.println(stack);
        stack.insertAfter("9","2");
        System.out.println(stack);
        System.out.println(stack.head);
        stack.pop();
        System.out.println(stack.head);
        System.out.println(stack.tail);
    }
}
```