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

Факультет компьютерного проектирования

Кафедра проектирования информационно-компьютерных систем

Е. Н. Шнейдеров, А. Ю. Писарчик, В. О. Казючиц

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ЯЗЫКЕ JAVA. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальности 1-39 03 02 «Программируемые мобильные системы»

Минск БГУИР 2023

УДК 004.438(076.5)
ББК 32.973.2я73
Ш76

Р е ц е н з е н т ы:

кафедра автоматизированных систем управления производством учреждения образования «Белорусский государственный аграрный технический университет»
(протокол №9 от 22.03.2019);

профессор кафедры информационных систем и технологий учреждения образования «Белорусский государственный технологический университет»
доктор технических наук, профессор П. П. Урбанович

Шнейдеров, Е. Н.

Ш76 Разработка приложений на языке Java. Лабораторный практикум : пособие / Е. Н. Шнейдеров, А. Ю. Писарчик, В. О. Казючиц. – Минск : БГУИР, 2023. – 92 с. : ил.

ISBN 978-985-543-561-8.

Приведены принципы разработки и тестирования кросс-платформенных приложений на примере объектно-ориентированного языка Java.

УДК 004.438(076.5)
ББК 32.973.2я73

ISBN 978-985-543-561-8

СОДЕРЖАНИЕ

Введение

Разработку всех современных компьютерных приложений нельзя представить себе без использования высокоуровневых языков программирования, позволяющих существенно сократить время разработки данных приложений и способствующих повышению их гибкости и масштабируемости. Java – кросс-платформенный высокоуровневый объектно-ориентированный язык программирования, который предоставляет возможность разрабатывать оконные, консольные и веб-приложения, а также мобильные приложения для операционной системы Android.

В рейтинге TIOBE за 2019 год Java занимает лидирующую позицию среди языков программирования, что свидетельствует о его популярности и широкой используемости при разработке корпоративных приложений. Изучение Java позволит обучающимся приобрести необходимые знания для написания компьютерных программ, а также постоянно совершенствовать их по мере изучения новых принципов и подходов в программировании, так как для языка Java реализовано множество библиотек и фреймворков, большинство из которых поддерживается и постоянно оптимизируется с учетом современных веяний в информационных технологиях.

Основная цель лабораторных работ, выполняемых на занятиях по дисциплине «Разработка приложений на языке Java», – изучение принципов разработки объектно-ориентированных приложений с использованием языка программирования Java.

Объектно-ориентированный подход подразумевает под собой оперирование классами и объектами при написании программ.

Таким образом, изучение взаимодействия между объектами и описывающими их классами является первостепенной задачей. Далее необходимо изучить основные программные конструкции и структуры данных языка Java. Изучение потоков выполнения позволит перейти к реализации прикладных программ, таких как клиент-серверное приложение, приложение по взаимодействию с базой данных.

Заключительным этапом в освоении данного пособия станет разработка веб-приложения для клиентской и серверной стороны посредством использования таких технологий, как Servlets, JSP и JSTL.

ЛАБОРАТОРНАЯ РАБОТА №1. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Цель: ознакомиться с основными понятиями и особенностями разработки объектно-ориентированных приложений.

1.1 Теоретическая часть

Объектно-ориентированное программирование (ООП) в настоящее время является доминирующей методикой программирования, вытеснившей «структурные» процедурно-ориентированные подходы, разработанные в 1970-х годах. Java представляет собой полностью объектно-ориентированный язык, и для продуктивной работы на этом языке вам необходимо ознакомиться с основными принципами ООП [1].

Объектно-ориентированная программа состоит из объектов. Каждый объект обладает определенными функциональными возможностями, предоставляемыми в распоряжение пользователей, а также скрытой реализацией. Одни объекты для своих программ вы можете взять в готовом виде из библиотеки, другие вам придется спроектировать самостоятельно. Строить ли свои объекты или приобретать готовые – зависит от вашего бюджета и времени. Но, как правило, до тех пор, пока объекты удовлетворяют вашим требованиям, вам не нужно особенно беспокоиться о том, как реализованы их функциональные возможности [1].

Для дальнейшей работы нужно усвоить основные понятия и терминологию ООП. Наиболее важным понятием является класс. Класс – это шаблон или образец, по которому будет создан объект [1].

Объект – обладающий именем набор данных (полей и свойств объекта), физически находящихся в памяти компьютера, и методов, имеющих доступ к ним. Имя используется для работы с полями и методами объекта.

Любой объект относится к определенному классу. В классе дается обобщенное описание некоторого набора родственных объектов. Объект – конкретный экземпляр класса [2].

Объектно-ориентированное программирование основано на принципах:

- инкапсуляции;
- наследования;
- полиморфизма, в частности, «позднего связывания».

Инкапсуляция (encapsulation) – принцип, объединяющий данные и код, манипулирующий этими данными, а также защищающий данные от прямого внешнего доступа и неправильного использования. Другими словами, доступ к данным класса возможен только посредством методов этого же класса [2].

Наследование (inheritance) – процесс, посредством которого один класс может наследовать свойства другого класса и добавлять к ним свойства и методы, характерные только для него.

Наследование бывает двух видов:

- одиночное наследование – когда подкласс (производный класс) имеет один и только один суперкласс (предок);

- множественное наследование – когда класс может иметь любое количество предков (что в Java запрещено) [2].

Полиморфизм (polymorphism) – механизм, использующий одно и то же имя метода для решения похожих, но несколько отличающихся задач в различных объектах при наследовании из одного суперкласса. Целью полиморфизма является использование одного имени при выполнении общих для суперкласса и подклассов действий [2].

Введение в разработку объектно-ориентированных приложений

Java является объектно-ориентированным языком программирования. Если сравнивать в этом смысле Java и C++, то между ними есть существенные различия. В Java нет средств, позволяющих писать необъектно-ориентированные программы, из чего сразу следует вывод: нельзя научиться программировать на Java, не овладев основами объектно-ориентированного подхода.

Отметим основные его принципы:

1 Все является объектом. Все данные программы хранятся в объектах. Каждый объект создается (есть средства для создания объектов), существует какое-то время, потом уничтожается.

2 Программа – группа объектов, общающихся друг с другом. Кроме того, что объект хранит какие-то данные, он умеет выполнять различные операции над своими данными и возвращать результаты этих операций. Теоретически эти операции выполняются как реакция на получение некоторого сообщения данным объектом. Практически это происходит при вызове метода данного объекта.

3 Каждый объект имеет свою память, состоящую из других объектов и/или элементарных данных. Объект хранит некоторые данные. Эти данные – это другие объекты, входящие в состав данного объекта и/или данные элементарных типов, такие как целое, вещественное, символ и т. п.

4 Каждый объект имеет свой тип (класс). То есть в объектно-ориентированном подходе не рассматривается возможность создания произвольного объекта, состоящего из того, например, что мы укажем в момент его создания. Все объекты строго типизированы. Мы должны сначала описать (создать) тип (класс) объекта, указав в этом описании, из каких элементов (полей) будут состоять объекты данного типа. После этого мы можем создавать объекты этого типа. Все они будут состоять из одних и тех же элементов (полей).

5 Все объекты одного и того же типа могут получать одни и те же сообщения. Кроме описания структуры данных, входящих в объекты данного типа, описание типа содержит описание всех сообщений, которые могут получать объекты данного типа (всех методов данного класса). Более того, в описании типа мы должны задать не только перечень и сигнатуру сообщений данного типа, но и алгоритмы их обработки.

Базовые типы данных

В языке Java определено восемь базовых типов данных. Для каждого базового типа данных отводится конкретный размер памяти. Этот размер не зависит от платформы, на которой выполняется приложение Java.

Все базовые типы данных по умолчанию инициализируются, поэтому программисту не нужно об этом беспокоиться. Вы можете также инициализировать переменные базовых типов в программе или при их определении.

Операции (operators) в языке Java

Большинство операций Java просты и интуитивно понятны. Это такие операции, как +, –, *, /, <, > и др. Операции имеют свой порядок выполнения и приоритеты. В выражении сначала выполняется умножение, а потом сложение, поскольку операция умножения перед операцией сложения имеет приоритет. Но операции Java имеют и свои особенности. Не вдаваясь в детальное описание простейших операций, остановимся на особенностях.

Начнем с присваивания. В отличие от ряда других языков программирования в Java присваивание – это не оператор, а операция. Операция присваивания обозначается символом «=». Она вычисляет значение своего правого операнда и присваивает его левому операнду, а также выдает в качестве результата присвоенное значение. Это значение может быть использовано другими операциями. Последовательность из нескольких операций присваивания выполняется справа налево.

В простейшем случае все выглядит как обычно:

```
x = a + b;
```

Здесь происходит именно то, что мы интуитивно подразумеваем, – вычисляется сумма a и b, результат заносится в x. Но вот два других примера:

```
a = b = 1;    a + b;
```

В первом сначала 1 заносится в b – результатом операции является 1. Потом этот результат заносится в a. Во втором примере вычисляется сумма a и b – и результат теряется. Это бессмысленно, но синтаксически допустимо.

Литералы (константы)

Для указания типа константы применяются суффиксы: арифметические (l (или L) – long, f (или F) – float, d (или D) – double); логические (true (истина) и false (ложь)); строковые (записываются в двойных кавычках); символьные (записываются в апострофах, например 'F', 'ш').

Массивы в Java

В Java есть как одномерные, так и многомерные массивы.

Но реализация массивов в Java имеет свои особенности. Во-первых, массив в Java – это объект, состоящий из размера массива (поле length) и собственно массива.

Рассмотрим сначала простейшие одномерные массивы базовых типов – int intAry[] или int[] intAry. Это описание переменной (или поля) intAry – ссылки на массив. Соответственно, в этом описании размер массива не указывается и сам массив не создается. Как и любой другой объект, массив должен быть создан операцией new – intAry = new int[10].

Для массивов допустима инициализация списком значений:

```
int intAry[] = {1, 2, 3, 4}
```

Здесь описан массив из четырех элементов и сразу определены их начальные значения. Элементы массивов в Java нумеруются от нуля. При обращении к элементу массивы его индексы задаются в квадратных скобках. Java жестко контролирует выход за пределы массива. При попытке обратиться к несуществующему элементу массива возникает IndexOutOfBoundException.

Для двумерных массивов ставится не одна пара скобок, а две, для трехмерных – три и т. д. Например, s = sAray[i][0]; tAray[i][j][k] = 10. Двумерный массив – это массив ссылок на объекты-массивы. Трехмерный массив – это массив ссылок на массивы, которые, в свою очередь, являются массивами ссылок на массивы. Как уже указывалось, массив является объектом, который, в частности, хранит поле length – размер массива. Это позволяет задавать обработку массивов произвольно. Они строятся по принципу «массив массивов».

Возможные способы инициализации массивов:

1) явное создание:

```
int ary[][] = new int[3][3];
```

2) использование списка инициализации:

```
int ary[][] = new int[][] {
                        {1, 1, 1},
                        {2, 2, 2},
                        {1, 2, 3}, };
```

3) массивы в языке Java являются объектами некоторого встроенного класса. Для этого класса существует возможность определить размер массива, обратившись к элементу данных класса с именем length, например:

```
int[] nAnotherNumbers;
nAnotherNumbers = new int[15];
```

```
for(int i = 0; i < nAnotherNumbers.length; i++)
{
  nAnotherNumbers[i] = nInitialValue;
}
```

В языке Java используются однострочные и блочные **комментарии** // и /**/, аналогичные комментариям, применяемым в C++. Введен также новый вид комментария /** */, который может содержать дескрипторы следующего вида:

@author – задает сведения об авторе;

@exception – задает имя класса исключения;

@param – описывает параметры, передаваемые методу;

@return – описывает тип, возвращаемый методом;

@throws – описывает исключение, генерируемое методом.

Из java-файла, содержащего такие комментарии, соответствующая утилита javadoc.exe может извлекать информацию для документирования классов и сохранения ее в виде HTML-документа.

Абстрактные классы

Класс может представлять собой как бы заготовку, в которой часть методов реализована, а часть – нет. В этом случае в описании класса перед словом class должен стоять описатель abstact и при описании нереализованных методов тоже должен использоваться этот описатель.

Пример:

```
public abstract class D {
  ...
  int g1(int s) {
     ...    }

   public abstract g2(String str);
   ...}
```

В классе D метод g1 – это обычный метод; g2 – абстрактный, содержащий только заголовок, но не содержащий реализации. Как видно из примера, тело абстрактного метода отсутствует, сразу после заголовка метода стоит точка с запятой.

Абстрактный класс не может использоваться непосредственно для порождения объектов. Для этого необходимо, используя этот класс как базовый, породить другой класс, в котором нужно определить все абстрактные методы. Тогда можно будет создавать объекты.

С другой стороны, не запрещено описывать переменные абстрактного класса. Просто им нужно присваивать ссылки на объекты неабстрактных классов.

Интерфейсы

Понятие интерфейса похоже на абстрактный класс. Интерфейс – это полностью абстрактный класс, не содержащий никаких полей, кроме констант (static final – поля).

Существует, однако, серьезное отличие интерфейсов от классов вообще и от абстрактных классов в частности. Интерфейсы допускают множественное наследование. То есть один класс может удовлетворять нескольким интерфейсам сразу.

Это связано с тем, что интерфейсы не порождают проблем с множественным наследованием, поскольку они не содержат полей.

Синтаксис:

```
public interface Stack{
 int size = 100;
    public void push (int x);
    public   int pop();
    int f(String s);
}
```

Это описание интерфейса Stack. Внутри скобок могут находиться только описания методов (без реализации) и описания констант (static final – полей). В данном случае интерфейс Stack содержит, в частности, метод f(...):

```
public class R implements Serializable, Stack {
……………
}
```

Класс R реализует интерфейсы Serializable и Stack.

Внутри класса, реализующего некоторый интерфейс, должны быть реализованы все методы, описанные в этом интерфейсе. Поскольку Stack имеет метод f(...), то в классе R он должен быть реализован следующим образом:

```
public class R implements Serializable, Stack {
 ...
 public int f(String s)  {
 ...}
...}
```

В интерфейсе f(...) описан без описателя public, в классе R – с описателем public. Дело в том, что все методы интерфейса по умолчанию считаются public, так что этот описатель там можно опустить. А в классе R необходимо его использовать явно. Еще одним общим моментом интерфейсов и абстрактных классов является то, что хотя и нельзя создавать объекты интерфейсов, но можно описывать переменные типа интерфейсов.

Интерфейсы широко используются при написании различных стандартов. Стандарт определяет, в частности, набор каких-то интерфейсов. И, кроме того, он содержит вербальное описание семантики этих интерфейсов. После этого прикладные разработчики могут писать программы, используя интерфейсы стандарта, а фирмы-разработчики могут разрабатывать конкретные реализации этих стандартов. При внедрении (deployment) прикладного программного обеспечения можно взять продукт любой фирмы-разработчика, реализующий данный стандарт (на практике, конечно, все несколько сложнее).

1.2 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Результаты выполнения практического задания.
5 Выводы.

1.3 Задания к лабораторной работе №1

1 Изучить теоретическую часть.

2 Создать консольное приложение, удовлетворяющее следующим требованиям:

а) использовать возможности ООП: классы, абстрактные классы, интерфейсы, наследование, полиморфизм, инкапсуляция;

б) определить иерархию классов, соответствующую вашей предметной области;

в) каждый класс должен иметь отражающее смысл название и информативный состав;

г) определить конструкторы и методы setТип(), getТип(), toString(), equals();

д) наследование должно применяться только тогда, когда это имеет смысл.

Тема приложения:

1 Цветочница. Определить иерархию цветов. Создать несколько объектов-цветов. Собрать букет (используя аксессуары) с определением его стоимости. Провести сортировку цветов в букете на основе уровня свежести. Найти цветок в букете, соответствующий заданному диапазону длин стеблей.

2 Новогодний подарок. Определить иерархию конфет и прочих сладостей. Создать несколько объектов-конфет. Собрать детский подарок с определением его веса. Провести сортировку конфет в подарке на основе одного из параметров. Найти конфету в подарке, соответствующую заданному диапазону содержания сахара.

3 Домашние электроприборы. Определить иерархию электроприборов. Включить некоторые в розетку. Подсчитать потребляемую мощность. Провести

сортировку приборов в квартире на основе мощности. Найти в квартире прибор, соответствующий заданному диапазону параметров.

4 Шеф-повар. Определить иерархию овощей. Приготовить салат. Подсчитать калорийность. Провести сортировку овощей для салата на основе одного из параметров. Найти овощи в салате, соответствующие заданному диапазону калорийности.

5 Звукозапись. Определить иерархию музыкальных композиций. Записать на диск сборку. Подсчитать продолжительность. Провести перестановку композиций диска на основе принадлежности к стилю. Найти композицию, соответствующую заданному диапазону длины треков.

6 Камни. Определить иерархию драгоценных и полудрагоценных камней. Отобрать камни для ожерелья. Подсчитать общий вес (в каратах) и стоимость. Провести сортировку камней ожерелья на основе ценности. Найти камни в ожерелье, соответствующие заданному диапазону параметров прозрачности.

7 Мотоциклист. Определить иерархию амуниции. Экипировать мотоциклиста. Подсчитать стоимость. Провести сортировку амуниции на основе веса. Найти элементы амуниции, соответствующие заданному диапазону параметров цены.

8 Транспорт. Определить иерархию подвижного состава железнодорожного транспорта. Создать пассажирский поезд. Подсчитать общую численность пассажиров и багажа. Провести сортировку вагонов поезда на основе уровня комфортности. Найти в поезде вагоны, соответствующие заданному диапазону параметров числа пассажиров.

9 Авиакомпания. Определить иерархию самолетов. Создать авиакомпанию. Посчитать общую вместимость и грузоподъемность. Провести сортировку самолетов компании по дальности полета. Найти самолет в компании, соответствующий заданному диапазону параметров потребления горючего.

10 Таксопарк. Определить иерархию легковых автомобилей. Создать таксопарк. Подсчитать стоимость автопарка. Провести сортировку автомобилей парка по расходу топлива. Найти автомобиль в компании, соответствующий заданному диапазону параметров скорости.

11 Страхование. Определить иерархию страховых обязательств. Собрать из обязательств дериватив. Подсчитать стоимость. Провести сортировку обязательств в деривативе на основе уменьшения степени риска. Найти обязательство в деривативе, соответствующее заданному диапазону параметров.

12 Мобильная связь. Определить иерархию тарифов мобильной компании. Создать список тарифов компании. Подсчитать общую численность клиентов. Провести сортировку тарифов на основе размера абонентской платы. Найти тариф в компании, соответствующий заданному диапазону параметров.

13 Фургон кофе. Загрузить фургон определенного объема грузом на определенную сумму из различных сортов кофе, находящихся к тому же в разных физических состояниях (зерно, молотый, растворимый в банках и пакетиках).

Учитывать объем кофе вместе с упаковкой. Провести сортировку товаров на основе соотношения цены и веса. Найти в фургоне товар, соответствующий заданному диапазону параметров качества.

14 Игровая комната. Подготовить игровую комнату для детей разных возрастных групп. Игрушек должно быть фиксированное количество в пределах выделенной суммы денег. Должны встречаться игрушки родственных групп: маленькие, средние и большие машины, куклы, мячи, кубики. Провести сортировку игрушек в комнате по одному из параметров. Найти игрушки в комнате, соответствующие заданному диапазону параметров.

15 Налоги. Определить множество и сумму налоговых выплат физического лица за год с учетом доходов с основного и дополнительного мест работы, авторских вознаграждений, продажи имущества, получения в подарок денежных сумм и имущества, переводов из-за границы, льгот на детей и материальной помощи. Провести сортировку налогов по сумме.

16 Счета. Клиент может иметь несколько счетов в банке. Учитывать возможность блокировки/разблокировки счета. Реализовать поиск и сортировку счетов. Вычисление общей суммы по счетам. Вычисление суммы по всем счетам, имеющим положительный и отрицательный балансы, отдельно.

17 Туристические путевки. Сформировать набор предложений клиенту по выбору туристической путевки различного типа (отдых, экскурсии, лечение, шопинг, круиз и т. д.) для оптимального выбора. Учитывать возможность выбора транспорта, питания и числа дней. Реализовать выбор и сортировку путевок.

18 Кредиты. Сформировать набор предложений клиенту по целевым кредитам различных банков для оптимального выбора. Учитывать возможность досрочного погашения кредита и/или увеличения кредитной линии. Реализовать выбор и поиск кредита.

19 Система *Факультатив*. *Преподаватель* объявляет запись на *Курс*. *Студент* записывается на *Курс*, обучается и по окончании *Преподаватель* выставляет *Оценку*, которая сохраняется в *Архиве*. *Студентов*, *Преподавателей* и *Курсов* при обучении может быть несколько.

20 Система *Платежи*. *Клиент* имеет *Счет* в банке и *Кредитную Карту* (*КК*). *Клиент* может оплатить *Заказ*, сделать платеж на другой *Счет*, заблокировать *КК* и аннулировать *Счет*. *Администратор* может заблокировать *КК* за превышение кредита.

21 Система *Больница*. *Пациенту* назначается лечащий *Врач*. *Врач* может сделать назначение *Пациенту* (процедуры, лекарства, операции). *Медсестра* или другой *Врач* выполняют назначение. *Пациент* может быть выписан из *Больницы* по окончании лечения, при нарушении режима или при иных обстоятельствах.

22 Система *Вступительные экзамены*. *Абитуриент* регистрируется на *Факультет*, сдает *Экзамены*. *Преподаватель* выставляет *Оценку*. *Система* подсчитывает средний балл и определяет *Абитуриентов*, зачисленных в учебное заведение.

23 Система *Библиотека*. *Читатель* оформляет *Заказ* на *Книгу*. Система осуществляет поиск в *Каталоге*. *Библиотекарь* выдает *Читателю Книгу* на абонемент или в читальный зал. При невозвращении *Книги Читателем* он может быть занесен *Администратором* в «черный список».

24 Система *Конструкторское бюро*. Заказчик представляет *Техническое Задание* (*ТЗ*) на проектирование многоэтажного *Дома*. *Конструктор* регистрирует *ТЗ*, определяет стоимость проектирования и строительства, выставляет *Заказчику Счет* за проектирование и создает *Бригаду Конструкторов* для выполнения *Проекта*.

25 Система *Телефонная станция*. *Абонент* оплачивает *Счет* за разговоры и *Услуги*, может попросить *Администратора* сменить номер и отказаться от услуг. *Администратор* изменяет номер, *Услуги* и временно отключает *Абонента* за неуплату.

26 Система *Автобаза*. Диспетчер распределяет заявки на *Рейсы* между *Водителями* и назначает для этого *Автомобиль*. *Водитель* может сделать заявку на ремонт. *Диспетчер* может отстранить *Водителя* от работы. *Водитель* делает отметку о выполнении *Рейса* и состоянии *Автомобиля*.

27 Система *Интернет-магазин*. *Администратор* добавляет информацию о *Товаре*. *Клиент* делает и оплачивает *Заказ* на *Товары*. *Администратор* регистрирует *Продажу* и может занести неплательщиков в «черный список».

28 Система *Железнодорожная касса*. *Пассажир* делает *Заявку* на станцию назначения, время и дату поездки. Система регистрирует *Заявку* и осуществляет поиск подходящего *Поезда*. *Пассажир* делает выбор *Поезда* и получает *Счет* на оплату. *Администратор* вводит номера *Поездов*, промежуточные и конечные станции, цены.

29 Система *Городской транспорт*. На *Маршрут* назначаются *Автобус*, *Троллейбус* или *Трамвай*. Транспортные средства должны двигаться с определенным для каждого *Маршрута* интервалом. При поломке на *Маршрут* должен выходить резервный транспорт или увеличиваться интервал движения.

30 Система *Аэрофлот*. *Администратор* формирует летную *Бригаду* (пилоты, штурман, радист, стюардессы) на *Рейс*. Каждый *Рейс* выполняется *Самолетом* с определенной вместимостью и дальностью полета. *Рейс* может быть отменен из-за погодных условий в *Аэропорту* отлета или назначения. *Аэропорт* назначения может быть изменен в полете из-за технических неисправностей, о которых сообщил командир.

Контрольные вопросы

1 Назовите принципы ООП и расскажите о каждом.
2 Дайте определение понятию «класс».
3 Что такое поле/атрибут класса?
4 Как правильно организовать доступ к полям класса?
5 Дайте определение понятию «конструктор».

6 Чем отличаются конструкторы по умолчанию, конструктор копирования и конструктор с параметрами?

7 Какие модификации уровня доступа вы знаете? Расскажите о каждом из них.

8 Расскажите об особенностях класса с единственным закрытым (private) конструктором.

9 О чем говорят ключевые слова this, super? Где и как их можно использовать?

10 Дайте определение понятию «метод».

ЛАБОРАТОРНАЯ РАБОТА №2. КОЛЛЕКЦИИ

Цель: изучить способы использования коллекций при разработке java-приложений.

2.1 Теоретическая часть

Понятие коллекции

Для хранения большого количества однотипных данных могут использоваться массивы, но они не всегда являются идеальным решением. Во-первых, длина массива задается заранее и в случае, если количество элементов заранее неизвестно, придется либо выделять память «с запасом», либо предпринимать сложные действия по переопределению массива. Во-вторых, элементы массива имеют жестко заданное размещение в его ячейках, поэтому, например, удаление элемента из массива не является простой операцией. В программировании давно и эффективно используются такие структуры данных, как стек, очередь, список, множество и т. д., объединенные общим названием коллекция.

Коллекция – это группа элементов с операциями добавления, извлечения и поиска элемента. Механизм работы операций существенно различается в зависимости от типа коллекции.

Элементы стека упорядочены в последовательность, добавление нового элемента может происходить только в конец этой последовательности, и получить можно только элемент, находящийся в конце (т. е. добавленный последним).

Очередь, напротив, позволяет получить лишь первый элемент (элементы добавляются в один конец последовательности, а «забираются» с другого). Другие коллекции (например, список) позволяют получить элемент из любого места последовательности.

Множество вообще не упорядочивает элементы и позволяет (помимо добавления и удаления) только узнать, содержится ли в нем данный элемент.

Язык Java предоставляет библиотеку стандартных коллекций, которые собраны в пакете java.util, поэтому нет необходимости программировать их самостоятельно.

При работе с коллекциями главное – избегать ошибки начинающих пользоваться наиболее универсальной коллекцией вместо той, которая необходима для решения задачи, например, списком вместо стека. Если логика работы программы такова, что данные должны храниться в стеке (появляться и обрабатываться в обратной последовательности), следует использовать именно стек. В этом случае вы не сможете нарушить логику обработки данных, обратившись напрямую к середине последовательности, а значит, шанс появления трудно обнаруживаемых ошибок резко уменьшается.

Чтобы выбрать коллекцию, которая лучше всего подходит условию задачи, необходимо знать особенности каждой из них. Эти знания являются обязательными для любого программиста, поскольку без применения тех или иных

коллекций редко обходится любая современная задача. Некоторые сведения вы сможете почерпнуть из дальнейшего изложения.

Классы-коллекции

В Java коллекции объектов разбиты на три большие категории:

1) List (список) – это список объектов. Объекты можно добавлять в список (метод add()), заменять в списке (метод set()), удалять из списка (метод remove()), извлекать (метод get()). Существует также возможность организации прохода по списку при помощи итератора;

2) Set (множество) – множество объектов. Те же возможности, что и у List, но объект может входить в множество только один раз. То есть двойное добавление одного и того же объекта в множество не изменяет само множество;

3) Map (отображение) – отображение или ассоциативный массив. В Map мы добавляем не отдельные объекты, а пары объектов (ключ, значение). Соответственно, есть операции поиска значения по ключу. Добавление пары с уже существующим в Map ключом приводит к замене, а не к добавлению. Из отображения (Map) можно получить множество (Set) ключей и список (List) значений.

Начнем изучение коллекций в Java с примера. Одним из широко используемых классов коллекций является ArrayList.

Пример использования этой коллекции:

```
import java.util.*;
import java.io.*;
public class ArrayListTest {
ArrayList lst = new ArrayList();
Random generator = new Random();
void addRandom() {
lst.add(new Integer(generator.nextInt()));
}
public String toString() {
return lst.toString();
}
public static void main(String args[]) {
ArrayListTest tst = new ArrayListTest();
for(int i = 0; i < 100; i++ )
tst.addRandom();
System.out.println("Сто случайных чисел: "+tst.toString());
}
}
```

Рассмотрим данный пример подробнее. Здесь, кроме класса ArrayList, использован еще ряд классов бибилиотеки Java:

1) Random – класс из java.util. Расширяет возможности класса Math по генерации случайных чисел (см. документацию);

2) Integer – так называемый wrapper-класс (класс-обертка) для целых (int). Он использован, потому что в коллекцию нельзя занести данные элементарных типов, а только объекты классов.

Класс ArrayListTest имеет два поля: поле lst класса ArrayList и поле generator класса Random, используемое для генерации случайных чисел. Метод addRandom() генерирует и заносит в коллекцию очередное случайное число. Метод toString() просто обращается к методу toString() класса ArrayList, который обеспечивает формирование представления списка в виде строки.

Метод main(...) создает объект класса ArrayListTest и организует цикл порождения 100 случайных чисел с занесением их в коллекцию, вызвав метод addRandom(). После этого он печатает результат. Запустим данную программу.

Этот пример не демонстрирует особых преимуществ коллекций, а лишь технику их использования. Из него видно, что добавить элемент в коллекцию можно методом add(...) класса ArrayList, и при этом мы нигде не указываем размер коллекции.

Итераторы

В коллекциях широко используются итераторы. В Java итератор – это вспомогательный объект, используемый для прохода по коллекции объектов. Как и сами коллекции, итераторы базируются на интерфейсе. Это интерфейс Iterator, определенный в пакете java.util (см. документацию). То есть любой итератор, как бы он не был устроен, имеет следующие три метода:

1) boolean hasNext() – проверяет, есть ли еще элементы в коллекции;
2) Object next() – выдает очередной элемент коллекции;
3) void remove() – удаляет последний выбранный элемент из коллекции.

Кроме Iterator, есть еще ListIterator – расширенный вариант итератора с дополнительными возможностями и Enumerator – устаревший вариант, оставленный для совместимости с предыдущими версиями. В свою очередь интерфейс Collection имеет метод Iterator iterator().

Это обязывает все классы коллекций создавать поддержку итераторов (обычно реализованы с использованием inner-классов, удовлетворяющих интерфейсу Iterator).

Рассмотрим теперь интерфейс List. В дополнение к методу iterator() он имеет метод ListIterator listIterator(). Соответственно, все коллекции-списки реализуют List-итераторы.

Вернемся к примеру с коллекцией из 100 случайных чисел. В нем плохо то, что строка, изображающая случайные числа, не форматирована и результат не выглядит удобочитаемым. Поставим себе задачу разбить эту строку на подстроки так, чтобы каждая из них содержала не более шести чисел. Для этого нам достаточно переписать метод toString():

```
public String toString() {
String res = "";
Iterator iter = lst.iterator();
for(int i = 0; iter.hasNext(); i++) {
if( i%6 == 0 )
res += "\n";
res += " " + iter.next().toString(); // !!!
}
return res;
}
```

Внесем эти изменения и запустим программу. Теперь числа выводятся в более удобочитаемом виде. Разберем, как реализован метод toString(). Метод iterator() из ArrayList возвращает объект, ссылку на который мы запоминаем в переменной iter. Этот объект не принадлежит интерфейсу Iterator (нельзя построить объект интерфейса). Это объект некоторого класса, определенного внутри ArrayList, удовлетворяющего интерфейсу Iterator. Этот класс имеет свое имя, поля, возможно, какие-то private-методы. Но нас это не интересует – мы просто приводим его к типу Iterator и используем как итератор. Чаще всего подобные классы строятся с использованием механизма вложенных классов.

Этот пример, кроме всего прочего, демонстрирует случай, когда не требуется выполнять приведение типа после извлечения объекта из коллекции.

Рассмотрим строку:

```
res += " " + iter.next().toString();
```

Извлеченный из коллекции методом next() объект мы не приводим к типу Integer, а сразу применяем метод toString(). Здесь используется то свойство, что все классы имеют метод toString(), т. к. он есть в Object.

Для коллекций, элементы которых проиндексированы, определен более функциональный итератор, позволяющий двигаться как в прямом, так и в обратном направлении, а также добавлять в коллекцию элементы. Такой итератор имеет интрефейс ListIterator, унаследованный от интерфейса Iterator и дополняющий его следующими методами:

1) previous() – возвращает предыдущий элемент (и делает его текущим);
2) hasPrevious() – возвращает true, если предыдущий элемент существует (т. е. текущий элемент не является первым элементом для данного итератора);
3) add(Object item) – добавляет новый элемент перед текущим элементом;
4) set(Object item) – заменяет текущий элемент;
5) nextIndex() и previousIndex() – служат для получения индексов следующего и предыдущего элементов соответственно.

В интерфейсе List определен метод listIterator(), возвращающий итератор ListIterator для обхода данного списка.

Интерфейс Collection

Интерфейс Collection содержит набор общих методов, которые используются в большинстве коллекций.

Рассмотрим основные из них:

1) add(Object item) – добавляет в коллекцию новый элемент, если элементы коллекции каким-то образом упорядочены. Новый элемент добавляется в конец коллекции;

2) clear() – удаляет все элементы коллекции;

3) contains(Object obj) – возвращает true, если объект obj содержится в коллекции, и false, если нет;

4) isEmpty() – проверяет, пуста ли коллекция;

5) remove(Object obj) – удаляет из коллекции элемент obj, возвращает false, если такого элемента в коллекции не нашлось;

6) size() – возвращает количество элементов коллекции.

Интерфейс List

Интерфейс List описывает упорядоченный список. Элементы списка пронумерованы, начиная с нуля, и к конкретному элементу можно обратиться по целочисленному индексу. Интерфейс List является наследником интерфейса Collection, поэтому содержит все его методы и добавляет к ним несколько своих:

1) add(int index, Object item) – вставляет элемент item в позицию index, при этом список раздвигается (все элементы, начиная с позиции index, увеличивают свой индекс на 1);

2) get(int index) – возвращает объект, находящийся в позиции index;

3) indexOf(Object obj) – возвращает индекс первого появления элемента obj в списке;

4) lastIndexOf(Object obj) – возвращает индекс последнего появления элемента obj в списке;

5) add(int index, Object item) – заменяет элемент, находящийся в позиции index, объектом item;

6) subList(int from, int to) – возвращает новый список, представляющий собой часть данного (начиная с позиции from до позиции to – 1 включительно).

Интерфейс Set

Интерфейс Set описывает множество. Элементы множества не упорядочены, множество не может содержать двух одинаковых элементов. Интерфейс Set унаследован от интерфейса Collection, но никаких новых методов не добавляет. Изменяется только смысл метода add(Object item) – он не добавляет объект item, если он уже присутствует во множестве.

Интерфейс Queue

Интерфейс Queue описывает очередь. Элементы могут добавляться в очередь только с одного конца, а извлекаться с другого (аналогично очереди в магазине). Интерфейс Queue также унаследован от интерфейса Collection.

Специфические для очереди методы:

1) poll() – возвращает первый элемент и удаляет его из очереди;
2) Методы интерфейса Queue:

- peek() – возвращает первый элемент очереди, не удаляя его;
- offer(Object obj) – добавляет в конец очереди новый элемент и возвращает true, если вставка удалась.

Класс Vector

Vector (вектор) – набор упорядоченных элементов, к каждому из которых можно обратиться по индексу. По сути эта коллекция представляет собой обычный список.

Класс Vector реализует интерфейс List, основные методы которого названы выше. К этим методам добавляется еще несколько. Например, метод firstElement() позволяет обратиться к первому элементу вектора, метод lastElement() – к его последнему элементу. Метод removeElementAt(int pos) удаляет элемент в заданной позиции, а метод removeRange(int begin, int end) удаляет несколько подряд идущих элементов. Все эти операции можно было бы осуществить комбинацией базовых методов интерфейса List, так что функциональность принципиально не меняется.

Класс ArrayList

Класс ArrayList – аналог класса Vector. Он представляет собой список и может использоваться в тех же ситуациях. Основное отличие в том, что он не синхронизирован и одновременная работа нескольких параллельных процессов с объектом этого класса не рекомендуется. В обычных же ситуациях он работает быстрее.

Класс Stack

Stack – коллекция, объединяющая элементы в стек. Стек работает по принципу LIFO (последним пришел – первым ушел). Элементы кладутся в стек «друг на друга», причем взять можно только «верхний» элемент, т. е. тот, который был положен в стек последним.

Для стека характерны операции, реализованные в следующих методах класса Stack:

1) push(Object item) – помещает элемент на вершину стека;

2) pop() – извлекает из стека верхний элемент;

3) peek() – возвращает верхний элемент, не извлекая его из стека;

4) empty() – проверяет, не пуст ли стек;

5) search(Object item) – ищет «глубину» объекта в стеке. Верхний элемент имеет позицию 1, находящийся под ним – 2 и т. д., если объекта в стеке нет, то возвращает позицию 1.

Класс Stack является наследником класса Vector, поэтому имеет все его методы (и, разумеется, реализует интерфейс List). Однако, если в программе нужно моделировать именно стек, рекомендуется использовать только пять вышеперечисленных методов.

Интерфейс Map

Интерфейс Map из пакета java.util описывает коллекцию, состоящую из пар «ключ – значение», которые широко используются для хранения настроек в файлах конфигурации. У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (Map).

Интерфейс Map содержит следующие методы, работающие с ключами и значениями:

1) boolean containsKey (Object key) – проверяет наличие ключа key;

2) boolean containsValue (Object value) – проверяет наличие значения value;

3) Set entrySet() – представляет коллекцию в виде множества, каждый элемент которого – пара из данного отображения, с которой можно работать методами вложенного интерфейса Map.Entry;

4) Object get(Object key) – возвращает значение, отвечающее ключу key; Set keySet() – представляет ключи коллекции в виде множества;

5) Object put (Object key, Object value) – добавляет пару key – value, если такой пары не было, и заменяет значение ключа key, если такой ключ уже есть в коллекции;

6) void putAll (Map m) – добавляет к коллекции все пары из отображения m;

7) collection values () – представляет все значения в виде коллекции.

В интерфейс Map вложен интерфейс Map.Entry, содержащий методы работы с отдельной парой.

Вложенный интерфейс Map.Entry

Этот интерфейс описывает методы работы с парами, полученными методом entrySet() из объекта типа Map. Методы getKey() и getValue() позволяют получить ключ и значение пары, метод setValue (Object value) меняет значение в данной паре.

Сортировка и поиск

Библиотека Java имеет развитые средства для выполнения сортировки и поиска. Эти средства реализованы в пакете java.util при помощи классов Arrays и Collections (не путать с интерфейсом Collection). Класс Arrays обеспечивает сортировку и поиск в массивах, а класс Collections – в коллекциях.

Если обратиться к документации, то можно увидеть, что оба эти класса не имеют public-конструктора, и все их методы статические. Один подобный класс мы уже рассматривали – это класс java.lang.Math. В таких классах мы пользуемся их статическими методами без порождения объекта класса.

Класс Collections

Предназначен для работы с коллекциями, а не массивами. Как и класс Arrays, Collections имеет ряд методов для сортировки и двоичного поиска:

1) public static void sort(List list);
2) public static void sort(List list, Comparator c);
3) public static int binarySearch(List list, Object key);
4) public static int binarySearch(List list, Object key, Comparator c).

Эти методы аналогичны одноименным методам класса Arrays, только в качестве первого параметра принимают List (список).

Вычислить, сколько раз каждая буква встречается в тексте, можно следующим образом:

```
import java.util.HashMap;
import java.util.*;
public class Main {
public static void main(String[] args) {
String txt = " лабораторная работа " ;
HashMap<Character, Integer> map = new HashMap<Character, Integer>(40);
for (int i = 0; i < txt.length(); ++i) {
char c = txt.charAt(i);
// проверяем, является ли символ буквой
if (Character.isLetter(c)) {
if (map.containsKey(c)) {
map.put(c, map.get(c) + 1);
} else {
map.put(c, 1);
}
}
}
// вывод на экран букв с частотой их появления
for (Entry<Character, Integer> entry : map.entrySet()) {
```

```
        System.out.println("буква: "+entry.getKey()+" количество: "+entry.get-
Value());
        }}}
```

2.2 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Результаты выполнения практического задания.
5 Выводы.

2.3 Задания к лабораторной работе №2

1 Ввести строки из файла, записать в список. Вывести строки в файл в обратном порядке.

2 Ввести число, занести его цифры в стек. Вывести число, у которого цифры идут в обратном порядке.

3 Создать в стеке индексный массив для быстрого доступа к записям в бинарном файле.

4 Создать список из элементов каталога и его подкаталогов.

5 Создать стек из номеров записи. Организовать прямой доступ к элементам записи.

6 Занести стихотворения одного автора в список. Провести сортировку по возрастанию длин строк.

7 Задать два стека, поменять информацию местами.

8 Определить множество на основе множества целых чисел. Создать методы для определения пересечения и объединения множеств.

9 Списки (стеки, очереди) I(1...n) и U(1...n) содержат результаты n измерений тока и напряжения на неизвестном сопротивлении R. Найти приближенное число R методом наименьших квадратов.

10 С использованием множества выполнить попарное суммирование произвольного конечного ряда чисел по следующим правилам: на первом этапе суммируются попарно рядом стоящие числа, на втором – результаты первого этапа и т. д. до тех пор, пока не останется одно число.

11 Сложить два многочлена заданной степени, если коэффициенты многочленов хранятся в объекте HashMap.

12 Умножить два многочлена заданной степени, если коэффициенты многочленов хранятся в различных списках.

13 Не используя вспомогательных объектов, переставить отрицательные элементы данного списка в конец, а положительные – в начало списка.

14 Ввести строки из файла, записать в список ArrayList. Выполнить сортировку строк, используя метод sort() из класса Collections.

15 Задана строка, состоящая из символов (,), [,], {, }. Проверить правильность расстановки скобок. Использовать стек.

16 Задан файл с текстом на английском языке. Выделить все различные слова. Слова, отличающиеся только регистром букв, считать одинаковыми. Использовать класс HashSet.

17 Задан файл с текстом на английском языке. Выделить все различные слова. Для каждого слова подсчитать частоту его встречаемости. Слова, отличающиеся регистром букв, считать различными. Использовать класс HashMap.

Контрольные вопросы

1 Дайте определение понятию «коллекция».
2 Назовите преимущества использования коллекций.
3 Какие данные могут хранить коллекции?
4 Какова иерархия коллекций?
5 Что вы знаете о коллекциях типа List?
6 Что вы знаете о коллекциях типа Set?
7 Что вы знаете о коллекциях типа Queue?
8 Что вы знаете о коллекциях типа Map, в чем их принципиальное отличие?
9 Назовите основные реализации List, Set, Map.
10 Какие реализации SortedSet вы знаете и в чем их особенность?
11 В чем отличия/сходства List и Set?

ЛАБОРАТОРНАЯ РАБОТА №3. ПОТОКИ ВЫПОЛНЕНИЯ

Цель: изучить основные способы реализации многопоточности в Java.

3.1 Теоретическая часть

Реализация многопоточности в Java

Для реализации многопоточности мы должны воспользоваться классом java.lang.Thread. В этом классе определены все методы, необходимые для создания потоков, управления их состоянием и синхронизации.

Как пользоваться классом Thread? Есть две возможности.

Во-первых, вы можете создать свой дочерний класс на базе класса Thread. При этом вы должны переопределить метод run. Ваша реализация этого метода будет работать в рамках отдельного потока.

Во-вторых, ваш класс может реализовать интерфейс Runnable. При этом в рамках вашего класса необходимо определить метод run, который будет работать как отдельный поток.

Второй способ особенно удобен в тех случаях, когда ваш класс должен быть унаследован от какого-либо другого класса и при этом вам нужна многопоточность. Так как в языке программирования Java нет множественного наследования, невозможно создать класс, для которого в качестве родительского будут выступать классы Applet и Thread. В этом случае реализация интерфейса Runnable является единственным способом решения задачи.

Методы класса Thread

В классе Thread определены три поля, несколько конструкторов и большое количество методов, предназначенных для работы с потоками. Ниже приведено краткое описание полей, конструкторов и методов.

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

Методы класса Thread предоставляют все необходимые возможности для управления потоками, в том числе для их синхронизации.

Реализация интерфейса Runnable

Описанный выше способ создания потоков как объектов класса Thread или унаследованных от него классов кажется достаточно естественным. Однако этот

способ не единственный. Если вам нужно создать только один поток, работающий одновременно с кодом Applet, то проще выбрать второй способ с использованием интерфейса Runnable.

Идея заключается в том, что основной класс апплета, который является дочерним по отношению к классу Applet, дополнительно реализует интерфейс Runnable, как это показано ниже:

```
public class MultiTask extends
  Applet implements Runnable
{
  Thread m_MultiTask = null;
  . . .
  public void run()
  {
    . . .
  }
  public void start()
  {
    if (m_MultiTask == null)
    {
      m_MultiTask = new Thread(this);
      m_MultiTask.start();
    }
  }
  public void stop()
  {
    if (m_MultiTask != null)
    {
      m_MultiTask.stop();
      m_MultiTask = null;
    }
  }
}
```

Внутри класса необходимо определить метод run, который будет выполняться в рамках отдельного потока. При этом можно считать, что код апплета и код метода run работают одновременно как разные потоки.

Для создания потока используется оператор new. Поток создается как объект класса Thread, причем конструктору передается ссылка на класс апплета:

```
m_MultiTask = new Thread(this);
```

При этом, когда поток запустится, управление получит метод run, определенный в классе Applet. Как запустить поток? Запуск выполняется, как и раньше, методом start. Обычно поток запускается из метода start Applet, когда пользователь отображает страницу сервера Web, содержащую Applet. Остановка потока выполняется методом stop.

Завершение и остановка потоков

Как уже указывалось, нормальное завершение нити происходит при выходе из метода run. Кроме того, иногда в приложении требуется выполнить временную остановку нити, чтобы потом возобновить ее работу.

В классе Thread есть методы stop (завершить), suspend (приостановить) и resume (возобновить), но все они объявлены deprecated (устаревшими) и их использование не рекомендовано. О причинах можно почитать подробнее в документации по Java и их здесь рассматривать не будем.

Как же поступить в случае, если нужно приостановить выполнение процесса? Для этого нужно не останавливать процесс, а обеспечить такой алгоритм, при котором он работает вхолостую.

Приоритеты потоков

Если процессор создал несколько потоков, то все они выполняются параллельно, причем время центрального процессора (или нескольких центральных процессоров в мультипроцессорных системах) распределяется между этими потоками.

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

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

Заметим, что распределение времени выполняется для потоков, а не для процессов. Потоки, созданные разными процессами, конкурируют между собой за получение процессорного времени.

Для оптимизации параллельной работы нитей в Java имеется возможность устанавливать приоритеты нитей. Нити с большим приоритетом имеют преимущество в получении времени процессора перед нитями с более низким приоритетом.

Работа с приоритетами обеспечивается методами класса Thread.

Метод, который устанавливает приоритет потока, – public final void setPriority(int newPriority).

Метод, который позволяет установить приоритет потока, – public final int getPriority().

Значение параметра в методе setPriority не может быть произвольным. Оно должно находиться в пределах от MIN_PRIORITY до MAX_PRIORITY. При своем создании нить имеет приоритет NORM_PRIORITY.

Средства синхронизации потоков в Java

Проанализируем эту проблему с другой точки зрения. У нас есть некоторый ресурс, в данном случае – случайное число (переменная randValue), доступ к которому нужно упорядочить. То есть нужно заблокировать доступ к этому ресурсу для всех нитей, кроме одной, пока эта нить не выполнит над ним необходимые действия.

В Java есть возможности по синхронизации нитей, построенные на этом принципе. То есть определяется некоторый ресурс, который может быть заблокирован. Таким ресурсом может быть любой объект (но не данные элементарного типа). Далее определяются критические участки программы. При входе в такой участок ресурс блокируется и становится недоступным для всех других нитей (с некоторой оговоркой, которую мы рассмотрим позже). При выходе из критического участка выполняется разблокировка ресурса.

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

Синтаксис определения критических участков следующий.

Для метода мы просто при описании метода указываем описатель synchronized, например:

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

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

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

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

3.2 Содержание отета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Результаты выполнения практического задания.
5 Выводы.

3.3 Задания к лабораторной работе №3

1 Создать Applet, используя поток: строка движется горизонтально, отражаясь от границ Applet и меняя при этом случайным образом свой цвет.

2 Создать Applet, используя поток: строка движется по диагонали. При достижении границ Applet все символы строки случайным образом меняют регистр.

3 Организовать сортировку массива методами Шелла, Хоара, пузырька, на основе бинарного дерева в разных потоках.

4 Реализовать сортировку графических объектов, используя алгоритмы из задания 3.4.

5 Создать Applet с точкой, движущейся по окружности с постоянной угловой скоростью. Сворачивание браузера должно приводить к изменению угловой скорости движения точки для следующего запуска потока.

6 Изобразить точку, пересекающую с постоянной скоростью окно слева направо (справа налево) параллельно горизонтальной оси. Как только точка доходит до границы окна, в этот момент от левого (правого) края с вертикальной координатой у, выбранной с помощью датчика случайных чисел, начинает свое движение другая точка и т. д. Цвет точки также можно выбирать с помощью датчика случайных чисел. Для каждой точки создается собственный поток.

7 Изобразить в приложении правильные треугольники, вращающиеся в плоскости экрана вокруг своего центра. Каждому объекту соответствует поток с заданным приоритетом.

8 Условия предыдущих задач изменяются таким образом, что центр вращения перемещается от одного края окна до другого с постоянной скоростью параллельно горизонтальной оси.

9 Создать фрейм с тремя шариками, одновременно летающими в окне. С каждым шариком связан свой поток со своим приоритетом.

10 Два изображения выводятся в окно. Затем они постепенно исчезают с различной скоростью в различных потоках (случайным образом выбираются точки изображения, и их цвет устанавливается в цвет фона).

11 Условие предыдущей задачи изменить на применение эффекта постепенного проявления двух изображений.

12 Создать Applet «Бегущая строка».

13 Реализовать приложение, в котором пользователь имеет возможность указывать маски файлов для поиска и набор путей, по которым эти файлы нужно искать (например, список логических дисков).

14 Написать секундомер – класс Stopwatch – для замера времени в отдельном потоке выполнения. В классе должны быть реализованы следующие методы:

- start – начинает отсчет времени;
- stop – прерывает отсчет времени;
- reset – сбрасывает текущее значение секундомера;
- getTime – возвращает отсчитанное время в миллисекундах.

Для демонстрации работы секундомера написать консольное приложение. Пользователю должны быть доступными следующие команды:

- start N – запустить секундомер и дать ему идентификатор N;
- stop N – остановить секундомер с идентификатором N;
- reset N – сбросить время у секундомера с идентификатором N;
- time N – показать время у секундомера с идентификатором N;
- help – список команд;
- exit – выход.

15 Подсчет простых чисел. Написать Swing-приложение для подсчета простых чисел в параллельных потоках. Каждый поток подсчитывает простые числа из заданного диапазона. Всего должно быть запущено три потока подсчета. Полученные числа выдаются на экран. Можно запускать/останавливать подсчет несколько раз. Пользователю доступны следующие элементы управления:

- кнопка запуска всех потоков;
- кнопка остановки всех потоков;
- поле (JTextArea) для вывода результата в виде

<номер потока>: <число>;

- четыре поля ввода диапазонов чисел для подсчета. Получается три диапазона (между четырьмя числами) для каждого потока;
- три строки с информацией о состоянии каждого потока.

Контрольные вопросы

1 Дайте определение понятию «процесс».

2 Дайте определение понятию «поток».

3 Дайте определение понятию «синхронизация потоков».

4 Как взаимодействуют программы, процессы и потоки?

5 В каких случаях целесообразно создавать несколько потоков?

6 Что может произойти, если два потока будут выполнять один и тот же код в программе?

7 Что вы знаете о главном потоке программы?

8 Какие есть способы создания и запуска потоков?

9 Какой метод запускает поток на выполнение?
10 Какой метод описывает действие потока во время выполнения?
11 Когда поток завершает свое выполнение?

ЛАБОРАТОРНАЯ РАБОТА №4. СЕТЕВЫЕ ПРОГРАММЫ

Цель: изучить теоретические и практические аспекты создания сетевых программ.

4.1 Теоретическая часть

Серверы и клиенты

В контексте работы в сети используются такие термины, как клиент и сервер. Сервер – это все, что имеет некоторый разделяемый (коллективно используемый) ресурс. Существуют вычислительные серверы, которые обеспечивают вычислительную мощность; серверы печати, которые управляют совокупностью принтеров; дисковые серверы, которые предоставляют работающее в сети дисковое пространство; и веб-серверы, которые хранят веб-приложения. Клиент – любой другой объект, который хочет получить доступ к серверу. Сервер – это постоянно доступный ресурс, в то время как клиент может «отключиться» после того, как он был обслужен. Различие между сервером и клиентом существенно, только когда клиент пытается подключиться к серверу. Как только они соединятся, происходит процесс двухстороннего общения, и не важно, что один является сервером, а другой – клиентом.

Работа сервера – слушать соединение, которое выполняется с помощью специального создаваемого серверного объекта (сокета), содержащего IP-адрес и номер порта. Работа клиента – попытаться создать соединение с сервером, которое выполняется с помощью клиентского сокета. Как только соединение установлено, соединение превращается в потоковый объект ввода/вывода. С этого момента можно рассматривать соединение как файл, который можно читать и в который можно записывать данные. Единственная особенность – файл может обладать определенным интеллектом и обрабатывать передаваемые команды.

Эти функции обеспечиваются расширением программы сетевой библиотеки java.net.*;.

Сокеты

Передача данных по сети – сложный процесс, включающий в себя определение пути доставки данных, организацию взаимодействия, алгоритмы синхронизации, обработки сбойных ситуаций и т. п. Программное обслуживание такого процесса сложное. Для упрощения введено понятие сокета (гнезда) как конечной точки коммуникации. Сокет (гнездо, разъем) – это программная абстракция, используемая для представления «терминалов» соединений между двумя машинами.

Каждый из сокетов определяется типом и ассоциированным с ним процессом. Реально для передачи организуются определенные дескрипторы TCP-соединения, так называемые гнезда (socket): гнездо сервера и гнездо клиента, которые в Internet-домене включают в себя IP-адреса сервера и клиента и номера портов, через которые они взаимодействуют. Сервер обычно имеет закрепленный и постоянный во взаимодействии номер порта, а клиенту, обращающемуся по этому номеру для связи к серверу, назначается некоторый другой (эфемерный) номер порта после установления соединения с сервером на сеанс их взаимодействия. Таким образом, основной порт освобождается для установления последующих связей (номер порта выбирается сервером из числа незанятых в диапазоне от 1024 до 65 535). Эта комбинация (IP-адрес и номера портов) однозначно определяет отдельные сетевые процессы в сети Internet (номера портов до 1024, как правило, резервируются для широко известных приложений, например 80 – для связывания с веб-серверами по протоколу HTTP).

Сокеты для работы в сети можно создать двух типов:

1) потоковые для TCP-соединения. TCP могут передавать данные только между двумя приложениями, т. к. они предполагают наличие канала между этими приложениями;

2) дейтаграммные. Для дейтаграмм не нужно создавать канал – данные посылаются приложению с использованием адреса, состоящего из сокета и номера порта (в дейтаграммах не гарантируются доставка и корректность последовательности передачи пакетов). Для передачи дейтаграмм не нужны ни механизмы подтверждения связи, ни механизмы управления потоком данных.

В данной лабораторной работе рассмотрим TCP-соединения.

Для упрощения представления такого соединения представим себе сокет, размещенный на некоторой машине, и виртуальный «кабель», соединяющий две машины, каждый конец которого вставлен в сокет. Для TCP-соединений в Java используется два класса сокетов: ServerSocket – класс, используемый сервером, чтобы «слушать» входящие соединения, и Socket, используемый клиентом для инициирования соединения.

Сокеты TCP/IP серверов

Как было указано выше, для создания сокетов серверов используется класс ServerSocket. Указанный класс используется для создания серверов, которые прослушивают либо локальные, либо удаленные программы клиента, чтобы соединяться с ними на опубликованных портах.

Конструкторы класса ServerSocket:

- ServerSocket(int port) – создает сокет сервера на указанном порте с длиной очереди по умолчанию 50;
- ServerSocket(int port, int maxQueue) – создает сокет сервера на указанном порте с максимальной длиной очереди maxQueue;

- ServerSocket(int port, int maxQueue, InetAddress localAddress) – создает сокет сервера на указанном порте с максимальной длиной очереди maxQueue.

Класс ServerSocket имеет метод accept(), который является блокирующим вызовом: сервер будет ждать клиента, чтобы инициализировать связь, и затем вернет нормальный Socket-объект, который будет использоваться для связи с клиентом.

Как только клиент создает соединение по сокету, ServerSocket возвращает с помощью метода accept() соответствующий клиенту объект Socket на сервере, по которому будет происходить связь со стороны сервера. Начиная с этого момента, появляется соединение сокет – сокет, и можно считать эти соединения одинаковыми, т. к. они действительно одинаковые:

```
ServerSocket httpServer=new ServerSocket(port);// создание сокета
// сервера
Socket reg=httpServer.accept(); // прослушивание (ожидание
// запроса на соединения)
// прием содержания соединения клиента
….
// отправка клиенту сообщения
…
// разрыв соединения
```

В представленном коде после установки соединения с клиентом метод accept() возвращает объект класса Socket (в данном случае reg), с помощью которого можно создавать и использовать байтовые и символьные потоки для обмена данными с клиентами. Для этого с гнездом связываются входной и выходной потоки, которые реализуются с помощью классов InputStream и OutputStream:

```
// получение входного и выходного потоков
InputStream inputstream = reg.getInputStream();
OutputStream outputstream = reg.getOutputStream();
```

Получив объекты, реализующие потоки, можно воспользоваться предоставляемыми ими методами, чтобы организовать взаимодействие по сети, например, организовать чтение байта из входного потока можно при помощи метода read, а запись байта в выходной поток – с использованием метода write:

```
int c= inputstream.read(); // чтение байта из входного потока
outputstream. write(c); // запись байта в выходной поток
```

Сокеты TCP/IP клиентов

Для создания сокета клиента используется конструктор Socket(String hostname, int port), который создает сокет, соединяющий локальную хост-машину с именованной хост-машиной и портом. Данный конструктор может выбрасывать исключение UnknownHostException или IOException.Socket(InetAddress ipAddress, int port) – создает сокет, аналогичный предыдущему, но используется уже существующий объект класса InetAddres и порт; может выбрасывать исключение IOException.

Сокет может в любое время просматривать связанную с ним адресную и портовую информацию при помощи методов, представленных в таблице 4.1.

Таблица 4.1 – Методы просмотра адресной и портовой информации

Метод	Описание
InetAddress getInetAddress()	Возвращает InetAddress-объект, связанный с Socket-объектом
int getPort()	Возвращает удаленный порт, с которым соединен данный Socket-объект
int getLocalPort()	Возвращает локальный порт, с которым соединен данный Socket-объект

После создания Socket-объекта его можно применять для получения доступа к связанным с ним потокам ввода/вывода.

Рассмотрим пример, в котором необходимо создать приложение клиент – сервер. Клиент считывает строку с клавиатуры, отображает ее на экране, передает серверу. Сервер отображает ее на экране, переводит в верхний регистр и передает клиенту, который, в свою очередь, снова отображает ее на экране.

Программа сервера:

```
import java.io.*;// импорт пакета, содержащего классы для ввода/вывода
import java.net.*;// импорт пакета, содержащего классы для работы в
// сети Internet
public class server
{public static void main(String[] arg)
{// объявление объекта класса ServerSocket
ServerSocket serverSocket = null;
Socket clientAccepted = null;// объявление объекта класса Socket
ObjectInputStream sois = null;// объявление байтового потока ввода
ObjectOutputStream soos = null;// объявление байтового потока вывода
try {
System.out.println("server starting....");
serverSocket = new ServerSocket(2525);// создание сокета сервера для
```

```
// заданного порта
clientAccepted = serverSocket.accept();// выполнение метода, который
// обеспечивает реальное подключение сервера к клиенту
System.out.println("connection established....");
// создание потока ввода soos = new
sois = new ObjectInputStream(clientAccepted.getInputStream());
ObjectOutputStream(clientAccepted.getOutputStream());// создание потока
// вывода
String clientMessageRecieved = (String)sois.readObject();// объявление
// строки и присваивание ей данных потока ввода, представленных
// в виде строки (передано клиентом)
while(!clientMessageRecieved.equals("quite"))// выполнение цикла: пока
// строка не будет равна «quite»
{
System.out.println("message recieved: '"+clientMessageRecieved+"'");
clientMessageRecieved = clientMessageRecieved.toUpperCase();// приведение
// символов строки к
// верхнему регистру
soos.writeObject(clientMessageRecieved);// потоку вывода
// присваивается значение строковой переменной (передается клиенту)
clientMessageRecieved = (String)sois.readObject();// строке
// присваиваются данные потока ввода, представленные в виде строки
// (передано клиентом)
} }catch(Exception e) {
} finally {
try {
sois.close();// закрытие потока ввода
soos.close();// закрытие потока вывода
clientAccepted.close();// закрытие сокета, выделенного для клиента
serverSocket.close();// закрытие сокета сервера
} catch(Exception e) {
e.printStackTrace();// вызывается метод исключения e
}
}
}
}
```

Программа клиента:

```
import java.io.*;// импорт пакета, содержащего классы для
// ввода/вывода
import java.net.*;// импорт пакета, содержащего классы для
// работы в сети
```

```
public class client {
public static void main(String[] arg) {
try {
System.out.println("server connecting....");
Socket clientSocket = new Socket("127.0.0.1",2525);// установление
// соединения между локальной машиной и указанным портом узла сети
System.out.println("connection established....");
BufferedReader stdin =
new BufferedReader(new InputStreamReader(System.in));// создание
// буферизированного символьного потока ввода
ObjectOutputStream coos =
new ObjectOutputStream(clientSocket.getOutputStream());// создание
// потока вывода
ObjectInputStream cois =
new ObjectInputStream(clientSocket.getInputStream());// создание
// потока ввода
System.out.println("Enter any string to send to server \n\t('quite' − programme
terminate)");
String clientMessage = stdin.readLine();
System.out.println("you've entered: "+clientMessage);
while(!clientMessage.equals("quite")) {// выполнение цикла, пока строка
// не будет равна «quite»
coos.writeObject(clientMessage);// потоку вывода присваивается
// значение строковой переменной (передается серверу)
System.out.println("~server~: "+cois.readObject());// выводится на
// экран содержимое потока ввода (переданное сервером)
System.out.println("--------------------------");
clientMessage = stdin.readLine();// ввод текста с клавиатуры
System.out.println("you've entered: "+clientMessage);// вывод в
// консоль строки и значения строковой переменной
}
coos.close();// закрытие потока вывода
cois.close();// закрытие потока ввода
clientSocket.close();// закрытие сокета
}catch(Exception e) {
e.printStackTrace();// выполнение метода исключения
}
}
}
```

Для запуска приведенного кода сервера и клиента сначала необходимо запустить приложение сервера, потом – приложение клиента. После запуска сервера на экране появится консольное окно сервера со строкой

server starting ….

После запуска клиента и установления его соединения с сервером на экране появится консоль с текстом:

server connecting….
connection established ….
Enter any string to send to server
<”quite”– program will terminate>

После установления соединения на сервере появляется еще одна строка, свидетельствующая об установлении соединения:

connection established….

Теперь введем в окне клиента какую-либо строку, например Hello, BSUIR. Пошлем ее серверу. В окне сервера появятся следующие строки:

message recieved: 'Hello, BSUIR'

В окне клиента появятся дополнительные строки:

you've entered: Hello, BSUIR
~server~: HELLO, BSUIR

Теперь введем слово «quite» на сервере клиента. Приложения завершат свою работу. Сначала закрывается окно клиента, затем – окно сервера.

4.2 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Результаты выполнения практического задания.
5 Выводы.

4.3 Задания к лабораторной работе №4

Разработать приложение на основе TCP-соединения, позволяющее осуществлять взаимодействие клиента и сервера по совместному решению задач обработки информации. Приложение должно располагать возможностью передачи и модифицирования получаемых (передаваемых) данных.

1 Разработать приложение – калькулятор для совершения простейших арифметических операций. Исходные параметры и тип операции (+, –, /, ×) вводятся на клиентской части и передаются серверу. Сервер возвращает клиенту результат операции.

2 Разработать приложение – чат. На сервере и клиентской части отображаются передаваемые сообщения и время их отправления.

3 Разработать приложение – генератор случайных чисел. На клиентской части вводится целое положительное число N и передается серверу, а тот в свою очередь возвращает клиенту массив случайных чисел от 1 до N.

4 Разработать приложение – поисковик слов. На сервере хранится определенный текст. На клиентской части вводится слово для поиска и передается серверу, а тот в свою очередь осуществляет поиск этого слова в тексте и возвращает клиенту все предложения, в которых встречается это слово.

5 Разработать приложение – счетчик букв. На клиентской части вводится строка и передается серверу, а тот в свою очередь осуществляет подсчет гласных и согласных букв и возвращает этот результат клиенту.

6 Разработать приложение – определитель матрицы. На клиентской части вводится исходная матрица произвольного порядка и передается серверу, а тот в свою очередь вычисляет определитель этой матрицы и возвращает результат клиенту.

7 Разработать приложение для нахождения обратной матрицы размером 3×3. Исходная матрица вводится на клиентской части и передается серверу, а тот в свою очередь возвращает клиенту обратную матрицу.

8 Разработать приложение для определения победителя лотереи. На сервере хранятся номера билетов. На каждом билете имеются 10 случайных чисел от 1 до 100. На клиентской части вводятся 10 чисел от 1 до 100, и сервер должен определить номер билета, в котором имеется больше всего совпадений с введенными числами.

9 Разработать приложение для определения призовых мест на соревнования по прыжкам в длину. На сервере хранятся фамилии участников соревнований, их идентификационные номера. На клиентской части вводятся результаты прыжков по каждому идентификационному номеру, а сервер возвращает фамилии спортсменов, занявших первое, второе и третье места.

10 Разработать приложение для определения суммы подоходного налога. На клиентской части вводятся заработные платы сотрудников предприятия и передаются серверу, а тот в свою очередь возвращает суммы налога. Причем для

заработной платы меньше 100 000 руб. применяется ставка налога 5 %, для заработной платы от 100 000 руб. до 500 000 руб. – ставка 10 %, для заработной платы больше 500 000 руб. – ставка 15 %.

11 Разработать приложение по поиску квартиры для покупки. Стоимости квартир и их адреса хранятся на сервере. На клиентской части вводится предельная сумма для покупки квартиры, а сервер возвращает клиенту адреса всех квартир с такой или меньшей стоимостью.

12 Разработать приложение, серверная часть которого в матрице произвольного порядка определяла бы индекс строки с минимальным элементом и индекс столбца с максимальным элементом этой матрицы и возвращала этот результат клиенту.

13 Разработать приложение, серверная часть которого в матрице произвольного порядка определяла бы отношение среднего значения элементов, расположенных на главной диагонали, к среднему значению элементов, расположенных на побочной диагонали этой матрицы, и возвращала результат клиенту.

14 Разработать приложение, в котором серверная часть хранит информацию о расписании занятий студентов. Клиентская часть имеет возможность просматривать, редактировать и удалять необходимую информацию.

15 Разработать приложение, в котором серверная часть осуществляет расчет себестоимости продукции. При этом пользователь на клиентской части вводит необходимую информацию, например: основная заработная плата, дополнительная заработная плата, материалы, прочие затраты и т. д., посылает ее на сервер. Сервер производит расчет и высылает назад клиенту рассчитанную полную себестоимость.

Контрольные вопросы

1 За что отвечает пакет java.net?

2 Дайте определение понятию «сокет».

3 Охарактеризуйте протокол TCP/IP.

4 Почему необходимо использовать многопоточность при разработке сетевых программ?

5 С помощью какого типа создаются сокеты?

6 Каким образом происходит взаимодействие серверного и клиентского приложений?

7 Что выполняет метод accept() типа Socket?

ЛАБОРАТОРНАЯ РАБОТА №5. JDBC

Цель: изучить интерфейс JDBC; научиться создавать приложения с доступом к БД (базе данных).

5.1 Теоретические сведения

JDBC (Java DataBase Connectivity) – стандартный прикладной интерфейс языка Java для организации взаимодействия между приложением и системой управления базами данных (СУБД). Взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов [1].

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы только к реляционным БД, для которых существуют драйверы, предоставляющие способ общения с реальным сервером базы данных [1, 2].

Далее приводится последовательность действий для выполнения первого запроса.

Подключение библиотеки с классом-драйвером БД

Дополнительно требуется подключить к проекту библиотеку, содержащую драйвер, предварительно поместив ее в папку /lib приложения:

mysql-connector-java-[номер версии]-**bin.jar**.

Установка соединения с БД

Для установки соединения с БД вызывается статический метод getConnection() класса java.sql.DriverManager. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Загрузка класса драйвера базы данных при отсутствии ссылки на экземпляр этого класса в JDBC 4.0 происходит автоматически при установке соединения экземпляром DriverManager. Метод возвращает объект Connection. URL базы данных, состоящий из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов:

```
Connection cn = DriverManager.getConnection("jdbc:mysql://localhost:3306/
testphones", "root", "pass");
```

В результате возвращен объект Connection и имеет место одно установленное соединение с БД с именем testphones. Класс DriverManager предоставляет средства для управления набором драйверов баз данных. С помощью метода

getDrivers() можно получить список всех доступных драйверов. До появления JDBC 4.0 объект драйвера СУБД нужно было создавать явно с помощью вызова

```
Class.forName("com.mysql.jdbc.Driver");
```

или регистрировать драйвер

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
```

В большинстве случаев в этом нет необходимости, т. к. экземпляр драйвера загружается автоматически при попытке получения соединения с БД [2].

Создание объекта для передачи запросов

После создания объекта Connection и установки соединения можно начинать работу с БД с помощью операторов SQL. Для выполнения запросов применяется объект Statement, создаваемый вызовом метода createStatement() класса Connection:

```
Statement st = cn.createStatement();
```

Объект класса Statement используется для выполнения SQL-запроса без его предварительной подготовки. Могут применяться также объекты классов PreparedStatement и CallableStatement для выполнения подготовленных запросов и хранимых процедур [2, 3].

Выполнение запроса

Созданные объекты можно использовать для выполнения запроса SQL, передавая его в один из методов: execute(String sql), executeBatch(), executeQuery(String sql) или executeUpdate(String sql). Результаты выполнения запроса помещаются в объект ResultSet:

```
/* выборка всех данных таблицы phonebook */
ResultSet rs = st.executeQuery("SELECT * FROM phonebook");
```

Для добавления, удаления или изменения информации в таблице строка запроса помещается в метод executeUpdate().

Обработка результатов выполнения запроса производится методами интерфейса ResultSet, самыми распространенными из которых являются next(), first(), previous(), last(), группа методов по доступу к информации вида getString(int pos), а также аналогичные методы, начинающиеся с getТип(int

pos)(getInt(int pos), getFloat(int pos) и др.) и updateТип(). Среди них следует выделить методы getClob(int pos) и getBlob(int pos), позволяющие извлекать из полей таблицы специфические объекты (Character Large Object, Binary Large Object), которые могут быть, например, графическими или архивными файлами. Эффективным способом извлечения значения поля из таблицы ответа является обращение к этому полю по его имени в строке результатов методами типа int getInt(String columnLabel), String getString(String columnLabel), Object getObject(String columnLabel) и подобными им. Интерфейс располагает большим числом методов по доступу к таблице результатов, поэтому рекомендуется изучить его достаточно тщательно. При первом вызове метода next() указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит значение false [2, 4].

Закрытие соединения statemen:

```
st.close(); // закрывает также и ResultSet
cn.close();
```

После того как база больше не нужна, соединение закрывается. Для того чтобы правильно пользоваться приведенными методами, программисту требуется знать типы полей БД. В распределенных системах это знание предполагается изначально. В Java 7 для объектов-ресурсов, требующих закрытия, реализована технология try with resources [4].

Рассмотрим пример использование СУБД MySQL. СУБД MySQL совместима с JDBC и будет применяться для создания учебных баз данных. Версия СУБД может быть загружена с сайта www.mysql.com. Для корректной установки необходимо следовать инструкциям мастера. В процессе установки следует создать администратора СУБД с именем root и паролем, например, pass. Если планируется разворачивать реально работающее приложение, необходимо исключить тривиальных пользователей сервера БД (иначе злоумышленники могут получить полный доступ к БД). Для запуска следует использовать команду из папки /mysql/bin:

```
mysqld-nt–standalone
```

Если не появится сообщение об ошибке, то СУБД MySQL запущена. Для создания базы данных и ее таблиц используются команды языка SQL [2, 4].

Создание простого соединения и простого запроса

Воспользуемся всеми предыдущими инструкциями и создадим пользовательскую БД с именем testphones и одной таблицей PHONEBOOK. Таблица

должна содержать три поля: числовое (первичный ключ) IDPHONEBOOK, символьное LASTNAME, числовое PHONE и несколько занесенных записей (таблица 5.1).

Таблица 5.1 – Пример таблицы из БД

IDPHONEBOOK	LASTNAME	PHONE
1	Иванов	9379992
2	Петров	8415141

При создании таблицы следует задавать кодировку UTF-8, поддерживающую хранение символов кириллицы. Приложение, осуществляющее простейший запрос на выбор всей информации из таблицы, выглядит следующим образом:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Properties;
import data.subject.Abonent;
public class SimpleJDBCRunner {
public static void main(String[ ] args) {
        String url = "jdbc:mysql://localhost:3306/testphones";
        Properties prop = new Properties();
        prop.put("user", "root");
        prop.put("password", "pass");
        prop.put("autoReconnect", "true");
        prop.put("characterEncoding", "UTF-8");
        prop.put("useUnicode", "true");
        Connection cn = null;
        DriverManager.registerDriver(new com.mysql.jdbc.Driver());
        try { // 1 блок
                cn = DriverManager.getConnection(url, prop);
                Statement st = null;
                try { // 2 блок
                        st = cn.createStatement();
                        ResultSet rs = null;
                        try { // 3 блок
                                rs = st.executeQuery("SELECT * FROM phone-
                                book");
                                ArrayList lst = new ArrayList<>();
                                while (rs.next()) {
```

```
                        int id = rs.getInt(1);
                        int phone = rs.getInt(3);
                        String name = rs.getString(2);
                        lst.add(new Abonent(id, phone, name));
                        }
                        if (lst.size() > 0) {
                        System.out.println(lst);
                        } else {
                        System.out.println("Not found");
                        }
                } finally { // для 3-го блока try
                        /*
                        * закрыть ResultSet, если он был открыт
                        * или ошибка произошла во время
                        * чтения из него данных
                        */
                        if (rs != null) { // был ли создан ResultSet
                                rs.close();
                                } else {
                                System.err.println(
                                "ошибка во время чтения из БД");
                                }
                }
                } finally {
                /*
                * закрыть Statement, если он был открыт или если
                *  произошла ошибка во время создания Statement
                */
                if (st != null) { // для 2-го блока try
                        st.close();
                        } else {
                        System.err.println("Statement не создан");
                        }
        }
        } catch (SQLException e) { // для 1-го блока try
        System.err.println("DB connection error: " + e);
                        /*
                        * вывод сообщения о всех SQLException
                        */
} finally {
/*
* закрыть Connection, если он был открыт
*/
```

```
            if (cn != null) {
                try {
                    cn.close();
                    } catch (SQLException e) {
                        System.err.println("Connection close error: " + e);
                    }
                }
            }
        }
}
```

В несложном приложении достаточно контролировать закрытие соединения, т. к. незакрытое или «провисшее» соединение снижает быстродействие системы. Класс Abonent, используемый приложением для хранения информации, извлеченной из БД, выглядит очень просто:

```
        import java.io.Serializable;
        public abstract class Entity implements Serializable, Cloneable {
        private int id;
        public Entity() {
        }
        public Entity(int id) {
            this.id = id;
            }
        public int getId() {
            return id;
        }
        public void setId(int id) {
            this.id = id;
        }
}

        public class Abonent extends Entity {
        private int phone;
        private String lastname;
        public Abonent() {
        }
        public Abonent(int id, int phone, String lastname) {
            super(id);
            this.phone = phone;
            this.lastname = lastname;
        }
        public int getPhone() {
```

```
        return phone;
    }
    public void setPhone(int phone) {
        this.phone = phone;
    }
    public String getLastname() {
        return lastname;
    }
    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
    @Override
    public String toString() {
        return "Abonent [id=" + id + ", phone=" + phone + ", lastname=" + last-
        name + "]";
    }
    }
```

Параметры соединения можно задавать несколькими способами: с помощью прямой передачи значений в коде класса, а также с помощью файлов properties или xml. Окончательный выбор производится в зависимости от конфигурации проекта. Чтение параметров соединения с базой данных и получение соединения следует вынести в отдельный класс. Класс ConnectorDB использует файл ресурсов database.properties, в котором хранятся, как правило, параметры подключения к БД, такие как логин и пароль доступа.

Например:

```
db.driver = com.mysql.jdbc.Driver
db.user = root
db.password = pass
db.poolsize = 32
db.url = jdbc:mysql://localhost:3306/testphones
db.useUnicode = true
db.encoding = UTF-8
```

Код класса ConnectorDB выглядит следующим образом:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.ResourceBundle;
public class ConnectorDB {
    public static Connection getConnection() throws SQLException {
```

```
        ResourceBundle resource = ResourceBundle.getBundle("database");
        String url = resource.getString("db.url");
        String user = resource.getString("db.user");
        String pass = resource.getString("db.password");
        return DriverManager.getConnection(url, user, pass);
        }
}
```

В таком случае получение соединения с БД сведется к вызову

```
Connection cn = ConnectorDB.getConnection();
```

Метаданные

Существует целый ряд методов интерфейсов ResultSetMetaData и DatabaseMetaData для интроспекции объектов. С помощью этих методов можно получить список таблиц, определить типы, свойства и количество столбцов БД [4]. Для строк подобных методов нет. Получить объект ResultSetMetaData можно следующим образом:

```
ResultSetMetaData rsMetaData = rs.getMetaData();
```

Некоторые методы интерфейса ResultSetMetaData:

- int getColumnCount() – возвращает число столбцов набора результатов объекта ResultSet;
- String getColumnName(int column) – возвращает имя указанного столбца объекта ResultSet;
- int getColumnType(int column) – возвращает тип данных указанного столбца объекта ResultSet и т. д.

Получить объект DatabaseMetaData можно следующим образом:

```
DatabaseMetaData dbMetaData = cn.getMetaData();
```

Некоторые методы весьма обширного интерфейса DatabaseMetaData:

- String getDatabaseProductName() – возвращает название СУБД;
- String getDatabaseProductVersion() – возвращает номер версии СУБД;
- String getDriverName() – возвращает имя драйвера JDBC;
- String getUserName() – возвращает имя пользователя БД;
- String getURL() – возвращает местонахождение источника данных;
- ResultSet getTables() – возвращает набор типов таблиц, доступных для данной БД, и т. д.

Подготовленные запросы и хранимые процедуры

Для представления запросов существует еще два типа объектов: PreparedStatement и CallableStatement.

Объекты первого типа используются при выполнении часто повторяющихся запросов SQL. Такой оператор предварительно готовится и хранится в объекте, что ускоряет обмен информацией с базой данных при многократном выполнении однотипных запросов.

Второй интерфейс используется для выполнения хранимых процедур, созданных средствами самой СУБД.

При использовании PreparedStatement невозможен sql injection attacks. То есть если существует возможность передачи в запрос информации в виде строки, то следует использовать для выполнения такого запроса объект PreparedStatement. Для подготовки SQL-запроса, в котором отсутствуют конкретные параметры, используется метод prepareStatement(String sql) интерфейса Connection, возвращающий объект PreparedStatement [4]:

```
String sql =
"INSERT INTO phonebook(idphonebook, lastname, phone) VALUES(?, ?, ?)";
PreparedStatement ps = cn.prepareStatement(sql);
```

Установка входных значений конкретных параметров этого объекта производится с помощью методов setString(int index, String x), setInt(int index, int x) и подобных им, после чего и осуществляется непосредственное выполнение запроса методами int executeUpdate(), ResultSet executeQuery().

Интерфейс CallableStatement расширяет возможности интерфейса PreparedStatement и обеспечивает выполнение хранимых процедур.

Хранимая процедура – это в общем случае именованная последовательность команд SQL, рассматриваемая как единое целое и выполняющаяся в адресном пространстве процессов СУБД, которые можно вызвать извне (в зависимости от политики доступа используемой СУБД). В данном случае хранимая процедура будет рассматриваться в более узком смысле – как последовательность команд SQL, хранимых в БД и доступных любому пользователю этой СУБД. Механизм создания и настройки хранимых процедур зависит от конкретной базы данных [2–4]. Для создания объекта CallableStatement вызывается метод prepareCall() объекта Connection.

Интерфейс CallableStatement позволяет исполнять хранимые процедуры, которые находятся непосредственно в БД. Одна из особенностей этого процесса заключается в том, что CallableStatement способен обрабатывать не только входные (IN), но и выходные (OUT) и смешанные (INOUT) параметры. Тип выходного параметра должен быть зарегистрирован с помощью метода registerOutParameter(). После установки входных и выходных параметров вызываются методы execute(), executeQuery() или executeUpdate().

Пусть в БД существует хранимая процедура getlastname, которая по уникальному номеру телефона для каждой записи в таблице phonebook будет возвращать соответствующее ему имя:

```
CREATE PROCEDURE getlastname (p_phone IN INT, p_lastname OUT VARCHAR) AS BEGIN
SELECT lastname INTO p_lastname FROM phonebook WHERE phone = p_phone;
END
```

Тогда для получения имени через вызов данной процедуры необходимо исполнить java-код следующего вида:

```
final String SQL = "{call getlastname (?, ?)}";
CallableStatement cs = cn.prepareCall(SQL);
// передача значения входного параметра
cs.setInt(1, 1658468);
// регистрация возвращаемого параметра
cs.registerOutParameter(2, java.sql.Types.VARCHAR);
cs.execute();
String lastName = cs.getString(2);
```

В JDBC также существует механизм batch-команд, который позволяет запускать на исполнение в БД массив запросов SQL вместе, как одну единицу:

```
// turn off autocommit
cn.setAutoCommit(false);
Statement st = con.createStatement();
st.addBatch("INSERT INTO phonebook VALUES (55, 5642032, 'Ивано')");
st.addBatch("INSERT INTO location VALUES (260, 'Minsk')");
st.addBatch("INSERT INTO student_department VALUES (500, 130)");
// submit a batch of update commands for execution
int[ ] updateCounts = stmt.executeBatch();
```

Если используется объект PreparedStatement, batch-команда состоит из параметризованного SQL-запроса и ассоциируемого с ним множества параметров.

Метод executeBatch() интерфейса PreparedStatement возвращает массив чисел, причем каждое характеризует число строк, которые были изменены конкретным запросом из batch-команды.

Пусть существует список объектов типа Abonent со стандартным набором методов getТип()/setТип() для каждого из его полей и необходимо внести их значения в БД. Многократное использование методов execute() или executeUpdate()

становится неэффективным, и в данном случае лучше использовать схему batch-команд:

```
try {
ArrayList abonents = new ArrayList<>(); // заполнение списка
PreparedStatement statement = con.prepareStatement("INSERT INTO phone-book VALUES(?,?,?)");
for (Abonent abonent : abonents) {
statement.setInt(0, abonent.getId());
statement.setInt(1, abonent.getPhone());
statement.setString(2, abonent.getLastname());
statement.addBatch();
}
updateCounts = statement.executeBatch();
} catch (BatchUpdateException e) {
        e.printStackTrace();
}
```

5.2 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Результаты выполнения практического задания.
5 Выводы.

5.3 Задания к лабораторной работе №5

Используя интерфейс JDBC, совместно с СУБД MySQL создать приложение согласно варианту:

1 Приложение «Справочник телефонных номеров».
2 Приложение для учета товаров на складе.
3 Приложение для учета билетов в кинотеатре.
4 Приложение для учета билетов автовокзала.
5 Приложение учета заявок по ремонту бытовой техники.
6 Приложение учета заказов в интернет-магазине.
7 Приложение для учета успеваемости студентов.
8 Приложение «Электронный отдел кадров».
9 Приложение «Расписание движения поездов».
10 Приложение для учета оказываемых предприятием услуг.
11 Приложение для учета книг в библиотеке.
12 Приложение для учета курсов валют.
13 Приложение для учета футбольных матчей.

14 Приложение для учета выигрышных лотерейных билетов.
15 Приложение для учета результатов тестирования.

Контрольные вопросы

1 Что такое JDBC?
2 Что такое драйвер базы данных?
3 Как выполнить запрос к базе данных?
4 Что называют параметром запроса?
5 С помощью каких интерфейсов можно получить список таблиц, определить типы, свойства и количество столбцов БД?
6 Какой метод возвращает имя указанного столбца в БД?
7 Какой метод возвращает местонахождение источника данных?
8 Для чего используются интерфейс PreparedStatement?
9 Что такое хранимая процедура?
10 Что позволяет использовать интерфейс CallableStatement?

ЛАБОРАТОРНАЯ РАБОТА №6. СЕРВЛЕТЫ

Цель работы: изучить применение сервлетов, научиться использовать интерфейсы для разработки сервлетов.

6.1 Теоретическая часть

Приложение Java запускается и выполняется в контейнере сервера приложений. Клиент общается с таким приложением посредством веб-браузера. Никаких дополнительных приложений на стороне клиента устанавливать не требуется. Сервлеты поддерживаются виртуальной машиной JVM, что предотвращает утечки памяти и обеспечивает функционирование garbage collection. Каждому клиенту сервлет выделяет независимый поток выполнения. Клиент посылает приложению HTTP-запрос, сервлет генерирует ответ и возвращает его клиенту в виде html-документа [4–6].

Характеристика сервлета:

- компонент приложений Java Enterprise Edition;
- загружается веб-сервером в контейнер;
- выполняется на стороне сервера;
- обрабатывает клиентские запросы;
- динамически генерирует ответы на запросы;
- находится в состоянии ожидания, если запросы отсутствуют;
- принимает запросы от других сервлетов (Servlet chaining);
- поддерживает соединения с ресурсами.

Наибольшее распространение получили сервлеты, обрабатывающие клиентские запросы по протоколу HTTP. Технология сервлетов является оболочкой протокола HTTP и поддерживает его как транспорт передачи данных от клиента серверу и обратно. Контейнер сервлетов поддерживает также протокол HTTPS (HTTP и SSL) для защищаемых запросов [4, 6].

Сервлеты в промышленном программировании используются:

- для приема входящих данных от клиента;
- взаимодействия с бизнес-логикой системы;
- динамической генерации ответа клиенту.

Все сервлеты реализуют общий интерфейс **Servlet** из пакета javax.servlet. Для обработки HTTP-запросов в web можно воспользоваться в качестве базового класса абстрактным классом HttpServlet из пакета javax.servlet.http.

Жизненный цикл сервлета начинается с его инициализации и загрузки в память контейнером сервлетов при старте контейнера либо в ответ на первый клиентский запрос. Сервлет готов к обслуживанию любого числа запросов. Завершение существования происходит при выгрузке его из контейнера.

Первым вызывается метод init(). Он дает сервлету возможность инициализировать данные и подготовиться для обработки запросов. Чаще всего в этом методе размещается код, кэширующий данные фазы инициализации.

После этого сервлет можно считать запущенным: он находится в ожидании запросов от клиентов. Появившийся запрос обслуживается методом service (HttpServletRequest request, HttpServletResponse response) сервлета, вызываемого контейнером, а все параметры запроса упаковываются в экземпляр request интерфейса HttpServletRequest, передаваемый в сервлет. Еще одним параметром этого метода является экземпляр response интерфейса HttpServletResponse, в который загружается информация для передачи клиенту. Для каждого нового клиента при обращении к сервлету создается независимый поток, в котором производится вызов метода service(). Метод service() предназначен для одновременной обработки множества запросов.

При выгрузке приложения из контейнера, т. е. по окончании жизненного цикла сервлета, вызывается метод destroy(), в теле которого следует помещать код освобождения занятых сервлетом ресурсов.

Преимуществом сервлетов перед CGI или ASP является быстродействие, переносимость на различные платформы, использование объектно-ориентированного языка высокого уровня Java, который расширяется большим числом классов и программных интерфейсов.

Сервлеты поддерживаются серверами приложений WebSphere, Weblogic, JBoss, Oracle OC4J Server, Glassfish, Geronimo, Apache Tomcat, а также контейнерами сервлетов tomcat, jetty, grizzly и являются частью платформы JEE.

Сервлеты реализуют интерфейс **Servlet**, в котором, кроме рассмотренных выше методов service(), init(), destroy(), предусмотрена реализация метода ServletConfig getServletConfig(), – он возвращает объект, содержащий параметры конфигурации сервлета, и дает доступ к среде выполнения [4–6].

При разработке сервлетов в качестве суперкласса в большинстве случаев используется не интерфейс Servlet, а абстрактный класс HttpServlet, отвечающий за обработку запросов HTTP.

Метод **service()** класса HttpServlet служит диспетчером для других методов, каждый из которых обрабатывает методы доступа к соответствующим ресурсам. В спецификации HTTP определены методы: GET, HEAD, POST, PUT, DELETE, OPTIONS и TRACE. Наиболее часто употребляются методы GET и POST, передающие на сервер запросы, а также параметры для их выполнения.

Метод **GET** (method="GET") используется для запроса содержимого указанного ресурса, изображения или гипертекстового документа. Вместе с запросом могут передаваться дополнительные параметры как часть URI, значения могут выбираться из полей формы или передаваться непосредственно через URL. При этом запросы кэшируются и имеют ограничения на размер. Этот метод является основным методом взаимодействия браузера клиента и веб-сервера [4, 6].

Метод **POST** используется для передачи пользовательских данных в содержимом HTTP-запроса на сервер. Пользовательские данные упакованы в тело запроса согласно полю заголовка Content-Type и/или включены в URI запроса. При использовании метода POST под URI подразумевается ресурс, который будет обрабатывать запрос [4, 6].

Метод **PUT** схож с методом POST, а отличием является то, что здесь URI подразумевает ресурс, который будет создан или сохранен на сервере в результате выполнения PUT-запроса.

Метод **DELETE** предназначен для удаления целевого ресурса. Оба эти действия на некоторых серверах могут запрещаться из-за угрозы внутренней безопасности.

Метод **HEAD** предполагает возврат сервером такого же ответа, как и при использовании GET, но без тела ответа. Метод обычно используется, чтобы проверить существование ресурса либо узнать, изменился ли запрашиваемый ресурс с момента последнего обращения.

Метод **OPTIONS** должен возвращать информацию о возможностях веб-сервера или параметрах соединения для конкретного ресурса.

Метод **TRACE** возвращает клиенту запрос в том виде, в каком он пришел на сервер, – используется для отладки, определяя заголовки, добавляемые промежуточными серверами, а также для тестирования настроек соединения.

Методы **HEAD**, **OPTIONS** и **TRACE**, как правило, реализуются сервером веб-приложения прозрачно для программиста и не требуют какой-либо специальной обработки. Поэтому обычно при разработке веб-приложений они не рассматриваются.

Методы **GET** и **HEAD** по принятым соглашениям не должны иметь другого назначения, кроме как передача информации клиенту, – они должны быть «безопасными» (safe). То есть в результате их выполнения на сервере не должно возникать никаких «побочных эффектов», которые повлияют на состояние ресурсов сервера, – таких, как создание, изменение, удаление данных. Также подразумевается, что запросы с «безопасными» методами могут быть выполнены браузером клиента в автоматическом режиме, без специального разрешения пользователя [4, 6]. Так как ссылки через элемент в коде HTML по умолчанию подразумевают использование браузером GET-запросов, то довольно часто можно встретить нарушение «безопасности» GET-запросов из-за использования элемента для выполнения каких-либо действий, изменяющих состояние сервера, например, ссылки следующего вида:

```
<a href="http://example.com/do?action=delete">Delete database</a>
```

Далеко не всегда такое нарушение влечет негативные последствия, однако т. к. инфраструктура глобальных сетей предполагает соблюдение соглашения «безопасности» GET-запросов, то при взаимодействии со следующими факторами могут возникать негативные последствия [4–7]:

1 Кэширующие прокси. Так как GET-запрос (в отличие от POST) может быть кэширован промежуточным кэширующим прокси-сервером, то при обработке запроса прокси может не выполнить запрос к целевому серверу, а сразу вернуть результат. Таким образом, если запрос подразумевал какое-то действие над данными на сервере, то это действие не будет выполнено [6, 7].

2 Предзагрузка. Некоторые браузеры для ускорения навигации выполняют загрузку страниц и других ресурсов по ссылкам еще до того, как пользователь выполнил переход по этим ссылкам. В этом случае при нарушении безопасности браузер выполнит запрос, который не был инициирован пользователем. Например, если на странице расположены ссылки, удаляющие какие-то объекты через GET-запросы, то при открытии этой страницы браузер может вызвать удаление всех этих объектов [6, 7].

3 Поисковые системы и другие клиенты, действующие в автоматическом режиме. Разнообразные автоматические клиенты не могут отличить запросы, которые выполняют какие-либо действия, от запросов, которые просто возвращают информацию. Поэтому они обычно ориентируются по методу запроса: GET-запросы выполняются и используются для дальнейшего анализа, а POST-запросы игнорируются. При нарушении безопасности GET-запросов возникает ситуация, аналогичная предзагрузке: выполняются те действия, которые при обычном взаимодействии с приложением не должны были бы выполняться [6, 7].

4 Нарушение авторизации. Если выполнение действия предполагает наличие определенных полномочий, то использование GET-запроса для этого предоставляет злоумышленнику возможность воспользоваться привилегиями другого пользователя (например, путем размещения ссылки с нужным GET-запросом в <img> элементе на произвольном сайте, который может посетить пользователь, обладающий нужными полномочиями с идентификацией через cookies), и тем самым выполнить неавторизованное действие [6, 7].

Поэтому важно понимать различие между методами GET и POST и использовать GET лишь там, где выполняется только непосредственная передача данных без выполнения активных действий.

Методы GET, HEAD, PUT и DELETE должны быть идемпотентными (idempotence) в том смысле, что повторное (более одного раза) выполнение запросов с помощью этих методов будет иметь тот же эффект, что и однократное выполнение.

По умолчанию параметры передаются в запрос в формате

?имя1=значение&имя2=значение

Однако форматы упаковки параметров могут быть самые разные, например, в случае передачи файлов с использованием формы

enctype="multipart/form-data"

Также часто вместо формата представления параметров вида

/article.jsp?id=1234&page=4

может быть использована так называемая Friendly URL-форма, более понятная для человека, не являющегося программистом, и должна быть представлена следующим образом:

/article/1234/page/4 либо /article_1234/page_4

Однако такая форма представления параметров обычно требует дополнительной работы со стороны программиста.

В задачу метода service() класса HttpServlet входит анализ полученного через запрос метода доступа к ресурсам и вызов метода со сходным именем, где перед именем добавляется префикс do: doGet(), doPost(), doHead(), doPut(), doDelete(), doOptions() и doTrace(). Разработчик должен переопределить нужный метод, разместив в нем функциональную логику [4, 7].

Механизмами протокола HTTP не предусмотрено сохранение информации о своем клиенте. Однако веб-приложение может требовать информацию о клиенте. Это касается веб-приложения, реализация которого предусматривает аутентификацию клиента без необходимости ее подтверждения при смене страниц.

Существуют и примитивные механизмы получения сведений о клиенте:

- файлы cookie – текстовые файлы, ассоциирующиеся с приложением и сохраняемые на компьютере клиента;
- добавление в строку GET-запроса дополнительных параметров;
- скрытые HTML-теги вида

<input type="hidden" name ="userType" value ="admin">

которые необходимо добавлять в каждую страницу приложения для сохранения целостности ее архитектуры.

В следующем примере приведен готовый к выполнению, но не к практическому применению, шаблон сервлета:

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.annotation.WebServlet;
@WebServlet("/FirstServlettest")
public class FirstServlet extends HttpServlet {
        public FirstServlet() {
                super();
        }
        public void init() throws ServletException {
        }
```

```
        protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
                response.setContentType("text/html");
                response.getWriter().print("This is " + this.getClass().getName() +
                ", using the GET method");
        }
        protected void doPost(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
                response.setContentType("text/html");
                response.getWriter().print("This is " + this.getClass().getName() +
                ", using the POST method");
        }
        public void destroy() {
                super.destroy();
        }
}
```

Начиная с версии Servlet API 3.0 регистрация сервлетов в файле-дескрипторе *web.xml* необязательна [5, 6]. Конфигурация определяется с помощью аннотации *@WebServlet* и в расширенном виде с указанием имени сервлета в контексте:

```
@WebServlet(name = "FirstServletname", urlPatterns = {"/FirstServlettest"})
public class FirstServlet extends HttpServlet {
}
```

Аннотация @WebServlet может содержать и другие методы-члены, о чем будет сказано ниже. Также аннотированию подлежат методы реализации функциональности сервлета.

Практика включения HTML-кода в код сервлета не считается хорошей, т. к. эти действия отвлекают сервлет от его основной роли – контроллера приложения. Это приводит к росту объема кода сервлета, который на определенном этапе становится неконтролируемым и реализует вследствие этого антишаблон «Волшебный сервлет». Приведенный выше сервлет имеет признаки антишаблона, т. к. содержит метод print() для формирования кода HTML. Сервлет должен использоваться только для контроля реализации бизнес-логики приложения и обязан быть отделен как от непосредственного формирования текста ответа на запрос, так и от данных, необходимых для этого. Обычно для формирования ответа на запрос применяются возможности JSP, JSPX, JSF и XHTML. Признаки наличия антишаблонов все же будут встречаться ниже, но это отступление сделано только с точки зрения компактности примеров.

Сервлет является компонентом веб-приложения, который будет называться FirstProject и размещаться в папке /WEB-INF/classes проекта.

Запуск контейнера сервлетов и размещение проекта

Здесь и далее применяется веб-сервер Apache Tomcat в качестве обработчика страниц JSP и сервлетов. Последняя версия может быть загружена с сайта jakarta.apache.org [4, 7].

При установке Tomcat предложит значение порта по умолчанию 8080, но во избежание конфликтов с иными Application Server рекомендуется присвоить другое значение, например, 8087.

Ниже приведены необходимые действия по запуску сервлета с помощью сервера Tomcat, который установлен в каталоге /Apache Software Foundation/Tomcat [версия]. В этом же каталоге размещаются следующие подкаталоги:

1) /bin – содержит файлы запуска монитора и контейнера сервлетов вида tomcat[номер].exe, tomcat[номер]w.exe и некоторые необходимые для этого библиотеки;

2) /lib – содержит библиотеки служебных классов, в частности, Servlet API и JSP API; используемые внешние библиотеки (если они есть), упакованные в JAR-файлы;

3) /conf – содержит конфигурационные файлы, в частности, конфигурационный файл сервера server.xml;

4) /logs – в этот каталог записываются log-файлы, инициированные сервером;

5) /webapps – в этот каталог помещаются папки с веб-приложениями, содержащие в свою очередь сервлеты и другие компоненты конкретного приложения. В каталог /webapps необходимо поместить папку /FirstProject с вложенным в нее сервлетом FirstServlet. Кроме того, папка /FirstProject должна содержать каталог /WEB-INF, в котором помещаются подкаталоги:

а) /classes – содержит class-файл сервлета servlet.FirstServlet;

б) /src – содержит исходный файл сервлета FirstServlet.java (опционально);

в) а также web.xml – дескриптор доставки (развертывания) приложения, который располагается в каталоге /WEB-INF.

В файле web.xml для версий Servlet API, не превышающих 2.5, необходимо прописать имя и путь к сервлету. Кроме того, в дескрипторном файле можно определять параметры инициализации, MIME-типы, mapping сервлетов и JSP, стартовые страницы и страницы с сообщениями об ошибках, а также параметры для безопасной авторизации и аутентификации. Этот файл можно сконфигурировать так, что путь к сервлету в браузере не будет совпадать с истинным именем класса сервлета.

Например:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns=http://java.sun.com/xml/ns/javaee
```

```
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation=
"http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
id="WebApp_ID" version="2.5">
<display-name>FirstProject</display-name>
<servlet>
<display-name>FirstServletdisplay</display-name>
<servlet-name>FirstServletname</servlet-name>
<servlet-class>first.servlet.FirstServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>FirstServletname</servlet-name>
<url-pattern>/FirstServlettest</url-pattern>
</servlet-mapping>
<session-config>
<session-timeout>30</session-timeout>
</session-config>
<welcome-file-list>
<welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

Здесь указано имя сервлета FirstServletname, путь к откомпилированному классу сервлета FirstServlet.class, а также URL-имя сервлета FirstServlettest, по которому происходит его вызов.

Для версии сервлетов 3.0 теги мэппинга не нужны, т. к. эта информация определяется аннотацией [4–6]:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation=
"http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID" version="3.0">
<display-name>FirstProject</display-name>
<session-config>
<session-timeout>30</session-timeout>
</session-config>
<welcome-file-list>
<welcome-file>index.jsp</welcome-file>
```

```
</welcome-file-list>
</web-app>
```

Таким образом, требуется выполнить следующие действия:

1) компиляцию сервлета с указанием в пути к архиву servlet-api.jar;

2) полученный файл класса FirstServlet.class поместить в каталог /FirstProject/ WEB-INF/classes/servlet;

3) в каталог /FirstProject/WEB-INF поместить файл конфигурации web.xml;

4) в каталог /FirstProject поместить файл стартовой JSP-страницы index.jsp;

5) разместить папку /FirstProject в каталог /webapps сервера Tomcat;

6) стартовать Apache Tomcat;

7) запустить браузер и ввести адрес

http://localhost:8080/FirstProject/FirstServlettest

При обращении к сервлету из другого компьютера вместо localhost следует указать IP-адрес или имя компьютера;

8) если вызывать сервлет из index.jsp, то тег FORM должен выглядеть следующим образом:

```
<form action="FirstServlettest">
        <input type="submit" value="Execute">
</form>
```

Для запуска проекта в браузере набирается строка

http://localhost:8080/FirstProject/index.jsp или
http://localhost:8080/FirstProject/

Простая JSP-страница

Технология Java Server Pages (JSP) обеспечивает разделение динамической и статической частей страницы, результатом чего является возможность изменения дизайна страницы, не затрагивая динамическое содержание. Это свойство используется при разработке и поддержке страниц, т. к. дизайнерам нет необходимости знать, как работать с динамическими данными [4, 6, 8].

JSP-код состоит из специальных тегов и выражений, которые указывают контейнеру соответствие между тегами и java-кодом для генерации сервлета или его части. Таким образом, поддерживается документ, который одновременно содержит и статическую страницу, и теги Java, управляющие страницей. Статические части HTML-страниц посылаются в виде строк в метод write() [6, 8].

Динамические части включаются прямо в код сервлета. С момента формирования ответа сервера страница ведет себя как обычная HTML-страница с ассоциированным сервлетом.

Чтобы создать простейшую JSP-страницу, достаточно взять HTML-страницу и заменить расширение html на jsp. Только в этом случае для запуска страницы необходимы и специальный application server с контейнером сервлетов и особое размещение самой страницы.

Взаимодействие сервлета и JSP

Страницы JSP и сервлеты никогда не следует использовать в информационных системах друг без друга. Причиной являются принципиально различные роли, которые играют данные компоненты в приложении. Страница JSP ответственна за формирование пользовательского интерфейса и отображение информации, переданной с сервера. Сервлет выполняет роль контроллера запросов и ответов, т. е. принимает запросы от всех связанных с ним JSP-страниц, вызывает соответствующую бизнес-логику для их (запросов) обработки и в зависимости от результата выполнения решает, какую JSP поставить этому результату в соответствие [6, 8].

При необходимости расширения функциональности системы не следует создавать дополнительные сервлеты. Сервлет в приложении должен быть один. При создании нового учебного приложения во избежание путаницы всегда следует создавать новый проект.

6.2 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Результаты выполнения практического задания.
5 Выводы.

6.3 Задания к лабораторной работе №6

Создать сервлет и взаимодействующие с ним классы и JSP-страницы, выполняющие действия, согласно варианту:

1 Генерация таблиц по переданным параметрам: заголовок, количество строк и столбцов, цвет фона.

2 Вычисление тригонометрических функций в градусах и радианах с указанной точностью, выбор функций должен осуществляться через выпадающий список.

3 Поиск слова, введенного пользователем. Поиск и определение частоты встречаемости осуществляется в текстовом файле, расположенном на сервере.

4 Вычисление объемов тел (параллелепипед, куб, сфера, тетраэдр, тор, шар, эллипсоид и т. д.) с точностью и параметрами, указываемыми пользователем.

5 Поиск и/или замена информации в коллекции по ключу (значению).

6 Выбор текстового файла из архива файлов по разделам (поэзия, проза, фантастика и т. д.) и его отображение.

7 Выбор изображения по тематике (природа, автомобили, дети и т. д.) и его отображение.

8 Вывод информации о среднесуточной температуре воздуха за месяц, заданной в виде списка, хранящегося в файле.

Определить:

а) среднемесячную температуру воздуха;

б) количество дней, когда температура была выше среднемесячной;

в) количество дней, когда температура опускалась ниже 0 ºС;

г) три самых теплых дня.

9 Игра с сервером в карточную игру «Двадцать одно».

10 Реализация адаптивного текста из цепочки в пять вопросов.

11 Определение значения полинома в заданной точке. Степень полинома и его коэффициенты вводятся пользователем.

12 Вывод фрагментов текстов шрифтами различного размера. Размер шрифта и количество строк задаются на стороне клиента.

13 Информация о точках на плоскости хранится в файле. Выбрать все точки, наиболее приближенные к заданной прямой. Параметры прямой и максимальное расстояние от точки до прямой вводятся на стороне клиента.

14 Осуществить сортировку введенного пользователем массива целых чисел. Числа вводятся через запятую.

15 Реализовать игру с сервером в крестики-нолики.

Контрольные вопросы

1 Назовите свойства и функции сервлета.

2 Для каких целей используются сервлеты в промышленном программировании?

3 Из чего состоит «жизненный цикл» сервлета?

4 Какие методы определены в спецификации HTTP?

5 Что такое «безопасные методы»?

6 В чем отличие методов GET и POST?

7 Перечислите механизмы получения сведений о клиенте.

8 Какие основные подкаталоги размещаются в главном каталоге сервера Apache Tomcat?

9 Из чего состоит JSP-код?

ЛАБОРАТОРНАЯ РАБОТА №7. JSP

Цель: изучить основы JSP и научиться создавать JSP-страницы.

7.1 Теоретическая часть

Технология Java Server Pages (JSP) была разработана компанией Sun Microsystems (в настоящее время поглощена компанией Oracle), чтобы облегчить создание страниц с динамическим содержанием [4, 8].

В то время как сервлеты наилучшим образом подходят для выполнения контролирующей функции приложения в виде обработки запросов и определения содержания и вида ответа, страницы JSP выполняют функцию отображения результатов работы приложения в виде текстовых документов типа HTML, XML, WML и некоторых других. JSP поддерживают как JavaScript, так и HTML-теги. JavaScript обычно используется, чтобы добавить функциональные возможности на уровне HTML-страницы [6, 8].

Принято разделять динамическое и статическое содержимое JSP.

Динамические ресурсы. Результаты их деятельности изменяются во время выполнения приложения. Обычно представлены в виде выражений Expression Language, библиотек тегов и тегов разработчика.

Статические ресурсы. Не изменяются сами в процессе работы (HTML, JavaScript, изображения и т. д.). Смысл разделения динамического и статического содержания в том, что статические ресурсы могут находиться под управлением HTTP-сервера, в то время как динамические нуждаются в движке (JSP Engine) и в большинстве случаев в доступе к уровню данных.

Рекомендуется разделить и разрабатывать параллельно две части приложения: часть, состоящая только из динамических ресурсов, и часть, состоящая только из статических ресурсов.

Некоторые преимущества использования JSP-технологии над другими методами создания динамического содержания страниц:

- **разделение динамического и статического содержания**. Возможность разделить логику приложения и дизайн веб-страницы снижает сложность разработки веб-приложений и упрощает их поддержку;
- **независимость от платформы**. Технологии Java не зависят от платформы, следовательно, JSP могут выполняться практически на любом веб-сервере. Разрабатывать JSP можно на любой платформе;
- **многократное использование компонентов**. Использование JavaBeans и Enterprise JavaBeans (EJB) позволяет многократно использовать компоненты, что ускоряет создание веб-приложений;

- **теги**. Спецификация JSP содержит библиотеку стандартных тегов JSTL, позволяет разработчику создавать собственные теги, кроме того, теги обеспечивают возможность использования JavaBean и обращение к классам бизнес-логики.

Содержимое Java Server Pages (теги HTML, теги JSP и скрипты) переводится в сервлет код-сервером. Этот процесс ответственен за трансляцию как динамических, так и статических элементов, объявленных внутри файла JSP. Об архитектуре сайтов, использующих JSP/Servlet-технологии, часто говорят как о thin-client (использование ресурсов клиента незначительно), потому что большая часть логики выполняется на сервере [6, 8].

В спецификации JSP 1.2 были объявлены только пять основных тегов:

1) <%@ директива %> – используется для установки параметров серверной страницы JSP;

2) <%! объявление %> – содержит поля и методы, которые вызываются в expression-блоке и становятся полями и методами генерируемого сервлета (нежелателен в современном программировании). Объявление не должно производить запись в выходной поток out страницы, но может быть использовано в скриплетах и выражениях;

3) <% скриплет %> – вживление java-кода в JSP-страницу (нежелателен). Скриптлеты обычно используют маленькие блоки кода и выполняются во время обработки запроса клиента. Когда все скриплеты собираются воедино в том порядке, в котором они записаны на странице, они должны представлять собой правильный код языка программирования. Контейнер помещает код Java в метод _jspService() на этапе трансляции;

4) <%= вычисляемое выражение %> – содержит операторы языка Java, которые вычисляются, после чего результат вычисления преобразуется в строку String и посылается в поток out (нежелателен);

5) <%-- JSP-комментарий --%> – комментарий, который не отображается в исходных кодах JSP-страницы после этапа выполнения.

Ниже приведен простой пример «вживления» java-кода в JSP-страницу:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html><head><title>JSP-страница</title></head><body>
<%! private int count = 0;
 String version = new String("1.2");
 private String getName() {return "Old Style";} %>
<% out.println("Значение count: "); %>
<%= count++ %>
<br/>
<% out.println("Значение count после инкремента: " + count); %>
<br/>
<% out.println("Старое значение version: "); %>
<%= version %>
```

```
<br/>
<% version = getName();
        out.println("Новое значение version: " + version); %>
</body></html>
```

Такая страница трудна для понимания несмотря на элементарность действий. В конце прошлого тысячелетия существовала техника создания страницы на основе одного скриплета, в который вставлялся большой фрагмент кода, что в корне противоречило концепции JSP. В современных приложениях скриплеты, выражения и объявления не применяются вовсе из-за нарушения «теговой» структуры страницы [6, 8].

Жизненный цикл

Процессы, выполняемые с файлом JSP при первом вызове [6, 8]:

1) браузер делает запрос к странице JSP;

2) JSP-engine анализирует содержание файла JSP и создает сервлет с кодом, основанным на исходном тексте файла JSP, при этом engine транслирует статическое содержимое в методы вывода и помещает его в метод _jspService(). Полученный сервлет будет ответственен за генерацию статических элементов, определенных во время разработки. Динамические элементы транслируются в java-код;

3) код сервлета компилируется в файл *.class. и загружается в контейнер. В итоге сервлет на основе JSP установлен и готов к работе;

4) выполняется метод init() сервлета;

5) вызывается метод _jspService(), и сервлет логически исполняется, формируя экземпляр response;

6) комбинация статического HTML и графики вместе с результатами исполнения динамических элементов, определенных в оригинале JSP, пересылаются браузеру через выходной поток объекта ответа HttpServletResponse.

Следующие обращения к файлу JSP просто вызовут метод _jspService() сервлета. Сервлет используется до тех пор, пока сервер не будет остановлен или сервлет не будет выгружен из контейнера. Результат работы JSP можно легко представить, зная правила трансляции JSP в сервлет, в частности, в его метод _jspService() [6–9].

Если рассмотреть преобразование в сервлет простейшей страницы JSP, отправляющей в браузер приветствие

```
<html><head>
<title>Simple</title>
</head>
<body>
<jsp:text>Hello, Bender</jsp:text>
</body></html>
```

то в результате запуска в браузер будет выведено: «Hello, Bender», а код сервлета на базе JSP будет выглядеть следующим образом:

```
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
public final class simple_jsp extends org.apache.jasper.runtime.HttpJspBase
implements org.apache.jasper.runtime.JspSourceDependent {
	private static java.util.List _jspx_dependants;
	public Object getDependants() {
		return _jspx_dependants;
	}
	public void _jspService(HttpServletRequest request, HttpServletResponse re-
sponse)
	throws java.io.IOException, ServletException {
		JspFactory _jspxFactory = null;
		PageContext pageContext = null;
		HttpSession session = null;
		ServletContext application = null;
		ServletConfig config = null;
		JspWriter out = null;
		Object page = this;
		JspWriter _jspx_out = null;
		PageContext _jspx_page_context = null;
		try {
			_jspxFactory = JspFactory.getDefaultFactory();
			response.setContentType("text/html");
			pageContext = _jspxFactory.getPageContext(this, request, re-
			sponse, null, true, 8192, true);
			_jspx_page_context = pageContext;
			application = pageContext.getServletContext();
			config = pageContext.getServletConfig();
			session = pageContext.getSession();
			out = pageContext.getOut();
			_jspx_out = out;
			out.write("<html><head>\r\n");
			out.write("<title>Simple</title>\r\n");
			out.write("</head>\r\n");
			out.write("<body>\r\n");
			out.write("Hello, Bender\r\n");
			out.write("</body></html>");
		} catch (Throwable t) {
```

```
                    if (!(t instanceof SkipPageException)){
                    out = _jspx_out;
                    if (out != null && out.getBufferSize() != 0)
                            out.clearBuffer();
                    if (_jspx_page_context != null) _jspx_page_context.handlePageEx-
                    ception(t);
                }
        } finally {
                if (_jspxFactory != null)
                        _jspxFactory.releasePageContext(_jspx_page_context);
                }
        }
}
```

Следует обратить внимание на объявление в методе _jspService() целого ряда локальных переменных. Ко всем этим переменным возможен доступ прямо из текста JSP по имени в выражениях Expression Language.

Неявные объекты в expression language

JSP expression language определяет свое множество неявных объектов, пересекающееся с множеством неявных объектов для JSP. Страница JSP с помощью скриплетов всегда имеет доступ ко всем локальным переменным и параметрам метода _jspService() сервлета, создаваемым jsp-engine на основе этой же страницы [6–8]. Наиболее очевидные:

- pageContext – определяет контекст JSP-страницы и предоставляет доступ к другим неявным объектам. Объект класса javax.servlet.jsp.PageContext. Область видимости в пределах страницы;
- servletContext – определяет контекст для сервлета JSP страницы и любого веб-компонента, содержащегося в этом приложении;
- param – содержит все параметры, переданные странице в виде параметров формы или параметров адресной строки;
- paramValues – содержит список всех значений параметров, переданных на страницу;
- initParam – содержит конфигурационные параметры страницы, заданные в файле web.xml;
- cookie – содержит список переменных, переданных с файлом cookie;
- request – представляет запрос клиента. Обычно объект является экземпляром класса, реализующего интерфейс javax.servlet.http.HttpServletRequest. Для протокола, отличного от HTTP, это будет объект реализации интерфейса javax.servlet.ServletRequest. Область видимости в пределах запроса;
- response – представляет собой ответ клиенту. Обычно объект является экземпляром класса, реализующего интерфейс javax.servlet.http.

HttpServletResponse. Для протокола, отличного от HTTP, это будет объект реализации интерфейса javax.servlet.ServletResponse. Область видимости в пределах страницы;

- config – содержит параметры конфигурации сервлета и является экземпляром класса javax.servlet.ServletConfig. Область видимости в пределах страницы;
- session – создается контейнером в соответствии с протоколом HTTP и является экземпляром класса javax.servlet.http.HttpSession, предоставляет информацию о сессии клиента, если такая была создана. Область видимости в пределах сессии;
- out – содержит выходной поток сервлета. Информация, посылаемая в этот поток, передается клиенту. Объект является экземпляром класса javax.servlet.jsp.JspWriter. Область видимости в пределах страницы;
- page – явная ссылка this для текущего экземпляра данной страницы, является объектом java.lang.Object. Область видимости в пределах страницы;
- exception – представляет исключение одного из подклассов java.lang.Throwable, которое передается странице, помеченной как error page. Область видимости в пределах страницы ошибок.

К неявным объектам возможно и обращение по имени, не допускающее никаких сокращений.

Существуют также объекты, которые обеспечивают доступ к различным областям видимости, а именно: pageScope, requestScope, sessionScope, applicationScope.

Стандартные элементы action

Большинство стандартных тегов, содержащих символ **%**, в современном программировании не применяются. Используются action-теги версии JSP 2.0.

Они позволяют создавать правильные JSP-документы. В то же время декларации, выражения и скриптлеты остаются под негласным запретом из-за явного внедрения java-кода в JSP [6–8].

Action-теги:

1 <jsp:include> позволяет включать статические и динамические ресурсы в контекст генерируемой страницы при запросе вида

<jsp:include page="относительный URL" flush="true"/>

Включаемая страница не может объявлять собственные заголовки, определяющие тип или кодировку.

2 <jsp:declaration> – объявление, аналогичен тегу <%! %> (нежелателен).

3 <jsp:scriptlet> – скриптлет, аналогичен тегу <% %> (нежелателен).

4 <jsp:expression> – выражение, аналогичен тегу <%= %> (нежелателен).

5 <jsp:text> – вывод текста.

6 <jsp:useBean> – объявление экземпляра компонента Java Bean или класса, обладающего public-конструктором по умолчанию. Если экземпляр с указанным идентификатором не существует, то он будет создан в области видимости scope со значением page (страница), request (запрос), session (сессия) или application (приложение). Объявляется, как правило, с атрибутами id (имя объекта), class (полное имя класса), type (по умолчанию class):

```
<jsp:useBean id="ob" scope="request" class="test.Message"/>
```

что идентично java-коду:

```
test.Message ob = new test.Message();
```

Создан объект ob класса Message, и в дальнейшем через этот объект можно вызывать доступные методы класса. По умолчанию область видимости устанавливается в значение page. Специфика компонентов JavaBean в том, что если компонент имеет поле text, экземпляр компонента имеет параметр text, а метод, устанавливающий значение, должен называться setText(String txt), возвращающий значение – getText().

```
public class Message {
private Integer id;
private String text = "";
public Message() {
}
public Message(Integer id, String text) {
        this.id = id;
        this.text = text;
}
public void setText(String text) {
        this.text = text;
}
public String getText() {
        return text;
}
public Integer getId() {
        return id;
}
public void setId(Integer id) {
        this.id = id;
}
@Override
public String toString() {
```

```
        return "Message [id=" + id + ", text=" + text + "]";
    }
}
```

В теге useBean можно создавать только объекты классов, у которых определен конструктор по умолчанию.

7 <jsp:setProperty> позволяет устанавливать значения полей указанного в атрибуте text объекта. Если установить значение property в «*», то значения свойств компонента JavaBean будут установлены таким образом, что будет определено соответствие между именами параметров и именами методов – установщиков (setter) компонента:

```
<jsp:setProperty name="ob" property="text" value="hello" />
```

что идентично java-коду:

```
ob.setText("hello");
```

или с автоматическим преобразованием строки в число

```
<jsp:setProperty name="ob" property="id" value="717" />
```

что идентично

```
ob.setId(717);
```

что возможно только для интегральных и логических (boolean) типов;

8 <jsp:getProperty> извлекает значения поля указанного объекта, преобразует его в строку и отправляет в неявный объект out:

```
<jsp:getProperty name="ob" property="text" />
```

9 <jsp:forward> позволяет передать запрос другой странице или сервлету:

```
<jsp:forward page="относительный URL"/>
```

10 <jsp:plugin> замещается тегом <OBJECT> или <EMBED> в зависимости от типа браузера, в котором будет выполняться подключаемый апплет или Java Bean.

11 <jsp:params> группирует параметры внутри тега jsp:plugin.

12 <jsp:param> добавляет параметры в объект запроса, например, в элементах forward, include, plugin.

13 <jsp:fallback> указывает содержимое, которое будет использоваться браузером клиента, если подключаемый модуль не сможет запуститься.

Используется внутри элемента plugin.

В качестве примера можно привести следующий фрагмент:

```
<jsp:plugin type="bean | applet" code="test.com.ReadParam" width="250" height="250">
        <jsp:params>
            <jsp:param name="bNumber" value="7" />
            <jsp:param name="state" value="true" />
        </jsp:params>
        <jsp:fallback>
            <p> unable to start plugin </p>
        </jsp:fallback>
</jsp:plugin>
```

Код апплета и пакет, в котором он объявлен, должен быть расположен в корне папки /WEB-INF, а не в папке /classes.

Элементы <jsp:attribute>, <jsp:body>, <jsp:invoke>, <jsp:doBody>, <jsp:element>, <jsp:output> используются в основном при включении в страницу пользовательских тегов и библиотек тегов.

JSP-документ

Современные тенденции оформления клиентской части стремятся к формату XML, например, в виде xhtml. Такой формат менее подвержен грамматическим ошибкам разработчика, т. к. осуществляется проверка на well-formed [6–8].

Предпочтительно создавать JSP-страницу в виде JSP-документа – корректного XML-документа, который ссылается на определенное пространство имен, содержит стандартные действия JSP, пользовательские теги и теги ядра JSTL, XML-эквиваленты директив JSP. В JSP-документе вышеперечисленные пять тегов неприменимы, поэтому их нужно заменять стандартными действиями и корректными тегами. JSP-документы необходимо сохранять с расширением .jspx [6–8].

Директива page для обычной JSP:

```
<%@ page contentType="text/html"%>
```

Директива page для JSP-документа:

```
<jsp:directive.page contentType="text/html" />
```

Если к директиве page добавить атрибут session="false", то атрибуты сессии будут недоступны.

Директива include для обычной JSP:

```
<%@ include file="header.jspf"%>
```

Директива include для JSP-документа:

```
<jsp:include file="header.jspf" />
```

Директива taglib для обычной JSP:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Директива taglib для JSP-документа:

```
<jsp:root xmlns:c="http://java.sun.com/jsp/jstl/core"/>
```

Выше были представлены директива и action-тег include. При включении статических ресурсов в страницу оба механизма работают идентично. При включении динамических ресурсов (другой JSP или JSP-фрагмента) директива include сразу конвертирует их непосредственно в создаваемый сервлет и включает в процесс исполнения – таким образом, исходная и включаемая страницы могут пользоваться одним и тем же пространством имен. Применение action-тега include формирует вид запрашиваемой страницы уже после выполнения сгенерированного сервлета и статически включает в его объект response [6–8].

Таким образом, для включения динамического содержимого в JSP-страницу каждый раз, когда та получает запрос, используется директива include. В этом случае включаемые сегменты имеют доступ к объектам page, request, session и application исходной страницы и ко всем атрибутам, которые имеют эти объекты. Если использовать action-тег jsp:include, то изменения включаемого сегмента отразятся только после изменения исходной страницы (контейнер JSP перекомпилирует исходную страницу) [6–8].

Для иллюстрации отличий директивы и action-тега include следует объявить некоторую jsp с обращением к переменной с именем calendar, которая на данной странице не объявлялась, а именно:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<html>
<head><title>Fragment</title></head>
<body>
        <hr/>
```

```
        Время в миллисекундах: ${calendar.timeInMillis}
        <hr/>
</body>
</html>
```

Теперь в index.jsp будут использованы и директива, и action-тег:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<html>
<head><title>Index</title></head>
<body>
<jsp:useBean id="calendar" scope="page" class="java.util.GregorianCalendar"/>
        Directive
        <%@ include file="time.jsp"%>
        <br/>
        Action-tag
        <jsp:include page="time.jsp"></jsp:include>
</body>
</html>
```

В результате руководствуемся следующим: если страница вставлена с помощью директивы include, то ее объявления получают доступ к переменным в области видимости основной страницы.

Другой способ импорта статического содержимого представляет тег <c:import> из библиотеки Java Standard Tag Library (JSTL).

Expression Language

В JSP API введено понятие Expression Language (EL). EL используется для упрощения доступа к данным (атрибутам, параметрам и т. п.), хранящимся в различных областях видимости: page, request, session, application, и вычисления простых выражений [6–8].

EL вызывается при помощи конструкции ${имя_атрибута}.

Начиная с версии спецификации JSP 2.0, EL является частью JSP и поддерживается без всяких сторонних библиотек.

EL-идентификатор ссылается на переменную, возвращаемую вызовом вида pageContext.findAttribute(имя_атрибута). В общем случае переменная может быть сохранена в любой области видимости: page (PageContext), request (HttpServletRequest), session (HttpSession), application (ServletContext). Перечислены в порядке возрастания длительности существования. Если переменная не найдена, возвращается null. Значение null в поток вывода не передается. Также

возможен доступ к параметрам запроса через предопределенные объекты param и paramValues, а также к заголовкам запроса через requestHeaders и некоторым другим, как уже было сказано.

Данные, как правило, состоят из объектов, соответствующих спецификации JavaBeans, или представляют собой коллекции, такие как List, Set, Map, массивы и др. EL предоставляет доступ к этим объектам при помощи операторов · и **[]**. Способ применения этих операторов зависит от типа объекта. К элементу массива или списка доступ производится обычной индексацией с оператором **[]**. К элементам типа Map – либо указанием ключа после оператора ·, либо обращением к ключу также по имени, но в операторе **[]**. С помощью оператора · можно вызывать другие методы класса, к которому принадлежит объект. В этом случае сигнатура метода должна быть представлена полностью без каких-либо сокращений [6–8].

Типы EL-операторов

Стандартные операторы отношения: == (или eq), != (или ne), < (или lt), > (или gt), <= (или le), >= (или ge).

Арифметические операторы: +, –, *, / (или div), % (или mod).

Логические операторы: && (или and), || (или or), ! (или not).

Оператор empty используется для проверки значения переменной на null, или пустое значение. Термин «пустое значение» зависит от типа проверяемого объекта. Это понятие включает нулевую длину для строки или нулевой размер для коллекции или массива.

Если в JSP отсутствует объявление или доступ к объекту ob типа Message, то при попытке обращения ${ob} в результате ничего выведено не будет. Поток вывода игнорирует значения типа null. Более того, при обращении к полю, т. е. get-методу несуществующего объекта вида ${ob.text}, генерации исключения не произойдет. Если же объект существует в какой-либо области видимости, а обращение происходит к несуществующему полю, то будет сгенерировано исключение [6–8].

Обработка ошибок

При выполнении веб-приложений, как и любых других, могут возникать ошибки и исключительные ситуации. Возникает вопрос об их обработке. Если исключение в java-коде не обрабатывается, то оно попадает в контейнер сервлетов и получает код 500 с генерацией сообщения вида «500 Internal Server Error». Вопрос обработки ошибок возникает также при вызове сервлетом метода sendError(500) на объекте HttpServletResponse. Такие действия производятся, как правило, для исключений времени выполнения. При отсутствии запрашиваемой страницы генерируется ошибка с кодом 404. При запрете доступа – код 403. При

отсутствии ответа сервера в течение определенного времени – код 504. При передаче слишком длинной строки запроса – код 414 [8, 9].

Для обработки исключений в зависимости от типа в приложении могут использоваться обычные JSP-страницы, специальные JSP для обработки ошибок или HTML-страницы. Для настройки соответствия ошибок и обработчиков используется элемент error-page файла web.xml.

Страница, вызываемая при ошибках, может иметь статический вид, но при необходимости сообщает о типе и месте возникшего исключения в понятной для клиента приложения форме. Информация о коде и типе ошибки, а также о месте ее возникновения извлекается из неявного объекта pageContext, точнее, из его поля errorData, являющегося экземпляром класса javax.servlet.jsp.ErrorData.

7.2 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Результаты выполнения практического задания.
5 Выводы.

7.3 Задания к лабораторной работе №7

Построить веб-систему, используя варианты задания из лабораторной работы №5. Разработанная система должна поддерживать заданную функциональность:

1) на основе сущностей предметной области создать классы, их описывающие;

2) классы и методы должны иметь названия, отражающие их функциональность, и быть грамотно структурированы по пакетам;

3) информацию о предметной области хранить в БД, для доступа использовать API JDBC с использованием пула соединений, стандартного или разработанного самостоятельно. В качестве СУБД рекомендуется MySQL или Derby;

4) приложение должно поддерживать работу с кириллицей (быть многоязычной), в том числе и при хранении информации в БД;

5) используя сервлеты и JSP, реализовать функции, предложенные в постановке конкретной задачи;

6) в страницах JSP применять библиотеку JSTL;

7) код должен содержать комментарии.

Контрольные вопросы

1 Назовите динамические и статические ресурсы JSP.
2 Назовите основные теги спецификации JSP 1.2.

3 Опишите процессы, выполняемые с файлом JSP при первом вызове.
4 Назовите неявные объекты в expression language.
5 Назовите стандартные элементы action.
6 Что представляет собой JSP-документ?
7 Для чего используется Expression Language?
8 Какие существуют EL-операторы?
9 Перечислите основные коды ошибок, возникающих в веб-приложениях.

ЛАБОРАТОРНАЯ РАБОТА №8. ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Цель: ознакомиться с основными шаблонами проектирования, научиться применять их при проектировании и разработке приложений.

8.1 Теоретическая часть

В задачах проектирования информационных систем, классов, их составляющих, при распределении обязанностей и способов взаимодействия объектов этих классов перед программистом возникает серьезная проблема. Неоптимальный выбор может сделать системы и их отдельные компоненты непригодными для поддержки, восприятия, повторного использования и расширения. Систематизация приемов программирования и принципов организации классов получила название шаблона (паттерна) [10–12].

Разработано огромное количество шаблонов, как вариаций первопроходцев, так и совершенно самостоятельных решений.

Шаблоны очень хорошо выражают преимущества объектно-ориентированного подхода, в частности, знаменитых «трех китов»: наследования, полиморфизма и инкапсуляции.

При решении задач программирования с применением шаблонов следует придерживаться следующих рекомендаций:

1 Изучение шаблонов приносит пользу вне зависимости от того, как часто они используются при программировании.

2 Не следует торопиться с применением шаблонов при решении новой задачи.

3 Шаблон в чистом виде (классическом, из учебника), как правило, не применим – применимы только вариации.

4 Применяя шаблон, следует начинать с его простейшей реализации, и лишь затем вносить изменения, адаптируясь к конкретной ситуации.

5 Если после применения шаблона код стал хуже, то шаблон лучше убрать.

Наиболее общие принципы объектно-ориентированного проектирования, применяемого при создании диаграммы классов и распределения обязанностей между ними, систематизированы в шаблонах **GRASP** (General Responsibility Assignment Software Patterns). Ниже будут сформулированы некоторые базовые способы и стандартные решения, придерживаясь которых можно создавать хорошо структурированный и понятный код.

Шаблон Expert

Все классы делятся на две большие группы: носители информации и производящие действия. Классов-носителей значительно меньше, но они играют роль отображения сущности реального мира в системе анимирования «неживых» сущностей и, как правило, должны быть представителями Java Beans.

Классы – носители информации – в общем не должны манипулировать значениями своих полей, не считая установки, извлечения и поддержки функциональности по контракту, навязанной классом Object.

При проектировании классов на первом этапе необходимо определить общий принцип распределения обязанностей между классами проекта, а именно: в первую очередь определить кандидатов в информационные эксперты – классы, обладающие информацией, требуемой для выполнения функциональности программной системы.

Простые эксперты определять достаточно просто, например, Student, Item, Order и т. д. Однако часто в процессе разработки возникает потребность дополнять класс атрибутами, и в этом случае необходимо сделать правильный выбор. Например, в подсистеме прохождения назначенного теста некоторому классу необходимо знать число вопросов, на которые получен ответ на текущий момент времени в процессе тестирования. Какой класс должен отвечать за знание количества вопросов, на которые дан ответ на текущий момент времени при прохождении теста, если определены следующие классы?

Пример:

```
public class Test {
        private int idTest;
        private String testName;
        private int questNumber;
        private long time;
        // реализация конструкторов и методов
}

public class Quest {
        private int idQuest;
        private int idTest;
        // реализация конструкторов и методов
}
```

Количество вопросов из теста, на которые дан ответ, есть число созданных объектов класса Quest. Такую информацию можно поместить в описание класса Test, но в данной ситуации это приведет к загромождению класса. Итак, класс Test ответственен за общие характеристики теста; класс CurrentStateTest будет владеть информацией о текущем состоянии теста:

```
public class CurrentStateTest {
        private int idT;
        private int idS;
        private int idCurrentQuest;
        private long timeRemain;
```

```
        private Queue idListQuest;
        // конструкторы и методы
}
```

Класс CurrentStateTest достаточно серьезно отличается от класса Test и довольно просто воспринимается с первого взгляда.

Преимущества следования шаблону **Expert** [10–13]:

- сохранение инкапсуляции информации при назначении ответственности классам, которые уже обладают необходимой информацией для обеспечения своей функциональности;
- уклонение от новых зависимостей способствует обеспечению низкой степени связанности между классами (Low Coupling);
- добавление соответствующего метода или внутреннего класса способствует высокому зацеплению (Highly Cohesive) классов, если класс уже обладает информацией для обеспечения необходимой функциональности.

Однако назначение чрезмерно большого числа ответственностей классу при использовании шаблона **Expert** может привезти к получению слишком сложных классов, которые перестанут удовлетворять шаблонам **Low Coupling** и **High Cohesion**.

Шаблон Creator

Существует большая вероятность того, что класс станет проще, если он будет большую часть своего жизненного цикла ссылаться на создаваемые объекты [10, 11].

После определения информационных экспертов следует определить классы, ответственные за создание нового экземпляра некоторого класса: следует назначить классу B обязанность создавать экземпляры класса A при выполнении одного из следующих условий:

- класс B содержит или получает данные инициализации (has the initializing data), которые будут передаваться объектам класса A при его создании;
- класс B записывает или активно использует (records or closely uses) экземпляры объектов A;
- класс B агрегирует (aggregate) объекты A;
- класс B содержит (contains) объекты A;
- классы B и A относятся к одному и тому же типу, и их экземпляры составляют, агрегируют, содержат или напрямую используют другие экземпляры того же класса.

Если выполняется одно из указанных условий, то класс B – создатель (creator) объектов A.

Инициализация объектов – стандартный процесс. Грамотное распределение обязанностей при проектировании позволяет создать слабо связанные независимые простые классы и компоненты.

В соответствии с шаблоном необходимо найти класс, который должен отвечать за создание нового экземпляра объекта Quest, например, агрегирующий экземпляры объектов Quest.

Поскольку объект BuildTest использует объект Quest, согласно шаблону **Creator** он является кандидатом на выполнение обязанности, связанной с созданием экземпляров объектов Quest. В этом случае обязанности могут быть распределены следующим образом [11, 14–16]:

```
public class BuildTest {
        // поля, методы
        public void buildTest(Queue q) {
                q.add(makeQuest(параметры));
                // реализация
        }
        private Quest makeQuest(параметры) {
                // реализация
                return new Quest(параметры);
        }
}
```

Шаблон **Creator** способствует низкой зависимости между классами (Low Coupling), т. к. экземпляры класса, которым необходимо содержать ссылку на некоторые объекты, должны создавать эти объекты. Если класс создает некоторый объект самостоятельно, то тем самым он перестает быть зависимым от класса, отвечающего за создание объектов для него [14–16].

Шаблон Low Coupling

Степень связанности классов определяет, насколько класс связан с другими классами и какой информацией о других классах он обладает. При проектировании отношений между классами следует распределить обязанности таким образом, чтобы степень связанности оставалась низкой [14–16].

Наличие классов с высокой степенью связанности нежелательно, т. к.:

- изменения в связанных классах приводят к локальным изменениям в данном классе;
- затрудняется понимание каждого класса в отдельности;
- усложняется повторное использование, поскольку для этого требуется дополнительный анализ классов, с которыми связан данный класс.

Учебный курс содержит тесты, которые состоят из вопросов. Пусть требуется создать экземпляр класса Quest и добавить его в экземпляр класса Test. В предметной области регистрация объекта Test выполняется объектом Course.

Далее экземпляр объекта Course должен передать сообщение makeQuest() объекту Test. Это значит, что в текущем тесте были получены идентификаторы

всех вопросов, составляющих тест, и становится возможным создание объектов типа Quest и наполнение собственно теста:

```
public class Course {
        // поля, методы
        public void makeTest(int id) {
                Test test = new Test(параметры);
                // реализация while(условие) {
                        Quest quest = new Quest(параметры);
                        // реализация test.addQuest(quest);
                }
        }
}
public class Test {
        // поля , методы
        public void addQuest(Quest quest){
                // реализация
        }
}
```

При таком распределении обязанностей предполагается, что класс Course связан с классом Quest.

Рассмотрим второй вариант распределения обязанностей с устранением класса Course от создания объектов вопросов:

```
public class Course {
        // поля
        public void makeTest() {
                Test test = new Test(параметры);
                // реализация
                test.createTest();
                // реализация
        }
}

public class Test {
        // поля
        public void createTest() {
                // реализация
                while(условие) {
                        Quest quest = new Quest(параметры);
                        // реализация
                }
```

```
        }
}
```

Какой из методов проектирования, основанный на распределении обязанностей, обеспечивает низкую степень связанности?

В обоих случаях предполагается, что объекту Test должно быть известно о существовании объекта Quest.

При использовании первого способа, когда объект Quest создается объектом Course, между этими двумя объектами добавляется новая связь, тогда как второй способ степень связывания объектов не увеличивает. Более предпочтителен второй способ, т. к. он обеспечивает низкую связываемость.

В ООП имеются некоторые стандартные способы связывания объектов A и B [14–16]:

- объект A содержит атрибут, который ссылается на экземпляр объекта B;
- объект A содержит метод, который ссылается на экземпляр объекта B, что подразумевает использование B в качестве типа параметра, локальной переменной или возвращаемого значения;
- класс объекта A является подклассом объекта B;
- B является интерфейсом, а класс объекта A реализует этот интерфейс.

Шаблон Low Coupling нельзя рассматривать изолированно от других шаблонов (Expert, Creator, High Cohesion). Не существует абсолютной меры для определения слишком высокой степени связывания.

Преимущества следования шаблону Low Coupling:

- изменение компонентов класса мало сказывается на других классах;
- принципы работы и функции компонентов можно понять, не изучая другие классы.

Шаблон High Cohesion

С помощью этого шаблона можно обеспечить возможность управления сложностью, распределив обязанности, поддерживая высокую степень зацепления [10, 11].

Зацепление – мера специализированности класса на своих обязанностях. При высоком зацеплении обязанности класса тесно связаны между собой, и класс не выполняет работ непомерных объемов. Класс с низкой степенью зацепления выполняет много разнородных действий или не связанных между собой обязанностей.

Возможно возникновение следующих проблем:

- класс труден в понимании, т. к. необходимо уделять внимание несвязным (неродственным) идеям;
- класс сложен в поддержке и повторном использовании из-за того, что он должен быть использован вместе с зависимыми классами;
- класс ненадежен, постоянно подвержен изменениям.

Классы со слабым зацеплением выполняют обязанности, которые можно легко распределить между другими классами.

Пусть необходимо создать экземпляр класса Quest и связать его с заданным тестом. Какой класс должен выполнять эту обязанность? В предметной области сведения о вопросах на текущий момент времени при прохождении теста записываются в объекте Course, согласно шаблону для создания экземпляра объекта Quest можно использовать объект Course. Тогда экземпляр объекта Course сможет отправить сообщение makeTest() объекту Test. За прохождение теста отвечает объект Course, т. е. объект Course частично несет ответственность за выполнение операции makeTest(). Однако если и далее возлагать на класс Course обязанности по выполнению все новых функций, связанных с другими системными операциями, то этот класс будет слишком перегружен и будет обладать низкой степенью зацепления.

Этот шаблон необходимо применять при оценке эффективности каждого проектного решения.

Виды зацепления:

1) очень слабое зацепление – единоличное выполнение множества разнородных операций:

```
public class Initializer {
        public void createTCPServer(String port) {
                // реализация
        }
        public int connectToDB (URL url) {
                // реализация
        }
        public void initXMLDocument(String fileName) {
                // реализация
        }
}
```

2) слабое зацепление – единоличное выполнение сложной задачи из одной функциональной области:

```
public class NetLogicCreator {
        public void createTCPServer() {
                // реализация
        }
        public void createTCPClient() {
                // реализация
        }
        public void createUDPServer() {
                // реализация
```

```
    }
    public void createUDPClient() {
        // реализация
    }
}
```

3) среднее зацепление – несложные обязанности в нескольких различных областях, логически связанных с концепцией этого класса, но не связанных между собой:

```
public class TCPServerHelper {
    public void createTCPServer() {
        // реализация
    }
    public void receiveData() {
        // реализация
    }
    public void sendData() {
        // реализация
    }
    public void compression() {
        // реализация
    }
    public void decompression() {
        // реализация
    }
}
```

4) высокое зацепление – среднее количество обязанностей из одной функциональной области при взаимодействии с другими классами:

```
public class TCPServerCreator {
    public void createTCPServer() {
        // реализация
    }
}
public class DataTransmission {
    public void receiveData() {
        // реализация
    }
    public void sendData() {
        // реализация
    }
```

```
}

public class CodingData {
        public void compression() {
                // реализация
        }
        public void decompression() {
                // реализация
        }
}
```

Если обнаруживается, что используется слишком негибкий дизайн, который сложен в поддержке, следует обратить внимание на классы, которые не обладают свойством зацепления или зависят от других классов. Эти классы легко узнаваемы, поскольку они сильно взаимосвязаны с другими классами или содержат множество неродственных методов. Как правило, классы, которые не обладают сильной зависимостью от других классов, обладают свойством зацепления и наоборот. При наличии таких классов необходимо реорганизовать их структуру таким образом, чтобы они по возможности не являлись зависимыми и обладали свойством зацепления [11, 14–16].

Шаблон Controller

Одной из базовых задач при проектировании информационных систем является определение класса, отвечающего за обработку системных событий. При необходимости посылки внешнего события прямо объекту приложения, которое обрабатывает это событие, как минимум один из объектов должен содержать ссылку на другой объект, что может послужить причиной очень негибкого дизайна, если обработчик событий зависит от типа источника событий или источник событий зависит от типа обработчика событий [14–16].

В простейшем случае зависимость между внешним источником событий и внутренним обработчиком событий заключается исключительно в передаче событий. Довольно просто обеспечить необходимую степень независимости между источником событий и обработчиком событий, используя интерфейсы. Интерфейсов может оказаться недостаточно для обеспечения поведенческой независимости между источником и обработчиком событий, когда отношения между этими источником и обработчиком достаточно сложны [14–16].

Можно избежать зависимости между внешним источником событий и внутренним обработчиком событий путем введения между ними дополнительного объекта, который будет работать в качестве посредника при передаче событий. Этот объект должен быть способен справляться с любыми другими сложными аспектами взаимоотношений между объектами.

Согласно шаблону **Controller** производится делегирование обязанностей по обработке системных сообщений классу при соблюдении следующих условий:

- если он представляет всю организацию или всю систему в целом (внешний контроллер);
- если он представляет активный объект из реального мира, который может участвовать в решении задачи (контроллер роли);
- если он представляет искусственный обработчик всех системных событий прецедента и называется прецедент **Handler** (контроллер прецедента). Для всех системных событий в рамках одного прецедента используется один и тот же контроллер.

Controller – это класс, не относящийся к интерфейсу пользователя и отвечающий за обработку системных событий. Использование объекта-контроллера обеспечивает независимость между внешними источниками событий и внутренними обработчиками событий, их типом и поведением. Выбор определяется зацеплением и связыванием [14–16].

Антишаблон – когда раздутый контроллер (набор, глобальный сервлет) выполняет слишком много обязанностей.

Признаки антишаблона:

- в системе имеется единственный класс контроллера, получающий все системные сообщения, которых поступает слишком много (внешний или ролевой контроллер);
- контроллер имеет много полей (информации) и методов (ассоциаций), которые необходимо распределить между другими классами.

Антишаблоны

Рассмотрим антишаблоны.

Big ball of mud – «большой ком грязи» – термин для системы или просто программы, которая не имеет хоть немного различимой архитектуры. Как правило, включает в себя более одного антишаблона. Этим страдают системы, разработанные людьми без подготовки в области архитектуры ПО.

Software Bloat – «распухание ПО» – термин, используемый для описания тенденций развития новейших программ в направлении использования бóльших объемов системных ресурсов (место на диске, ОЗУ), чем предшествующие версии. В более общем контексте применяется для описания программ, которые используют больше ресурсов, чем необходимо.

Yo-Yo problem – «проблема йо-йо» – возникает, когда необходимо разобраться в программе, иерархия наследования и вложенность вызовов методов которой очень длинны и сложны. Программисту вследствие этого необходимо лавировать между множеством различных классов и методов, чтобы контролировать поведение программы. Термин происходит от названия игрушки йо-йо.

Magic Button – «магическая кнопка» – возникает, когда код обработки формы сконцентрирован в одном месте и, естественно, никак не структурирован.

Magic Number – «магическое число» – наличие в коде многократно повторяющихся одинаковых чисел или чисел, объяснение происхождения которых отсутствует.

Gas Factory – «газовый завод» – необязательный сложный дизайн для простой задачи.

Analiys paralisys – «паралич анализа» в разработке ПО – проявляет себя через чрезвычайно длинные фазы планирования проекта, сбора необходимых для этого артефактов, программного моделирования и дизайна, которые не имеют особого смысла для достижения итоговой цели.

Interface Bloat – «распухший интерфейс» – термин, используемый для описания интерфейсов, которые пытаются вместить в себя все возможные операции над данными.

Smoke And Mirrors – термин «дым и зеркала» – используемый, чтобы описать программу либо функциональность, которая еще не существует, но выставляется за таковую. Часто используется для демонстрации финального проекта и его функциональности.

Improbability Factor – «фактор неправдоподобия» – ситуация, при которой в системе наблюдается некоторая проблема. Часто программисты знают о проблеме, но им не разрешено ее исправить отчасти из-за того, что шанс всплыть наружу у этой проблемы очень мал. Как правило (следуя закону Мерфи), она всплывает и наносит ущерб.

Creeping featurism – «расползание функций» – используется для описания ПО, которое выставляет напоказ вновь разработанные элементы, доводя до высокой степени ущербности по сравнению с ними другие аспекты дизайна – простоту, компактность и отсутствие ошибок. Как правило, существует вера в то, что каждая новая маленькая черта информационной системы увеличит ее стоимость.

Accidental complexity – «случайная сложность» – проблема в программировании, которой легко можно было бы избежать, возникает вследствие неправильного понимания проблемы или неэффективного планирования.

Ambiguous viewpoint – «неопределенная точка зрения» – объектно-ориентированные модели анализа и дизайна представляются без внесения ясности в особенности модели. Изначально эти модели обозначаются с точки зрения визуализации структуры программы. Двусмысленные точки зрения не поддерживают фундаментального разделения интерфейсов и деталей представления.

Boat anchor – «корабельный якорь» – часть бесполезного компьютерного «железа» (единственное применение которого – отправить на утилизацию). Этот термин появился в то время, когда компьютеры были больших размеров. В настоящее время термин «корабельный якорь» означает классы и методы, которые по различным причинам не имеют какого-либо применения в приложении и, в принципе, бесполезны. Они только отвлекают внимание от действительно важного кода.

Busy spin – «холостой цикл» – техника, при которой процесс непрерывно проверяет изменение некоторого состояния, например, ожидает ввода с клавиатуры или разблокировки объекта. В результате повышается загрузка процессора, ресурсы которого можно было бы перенаправить на исполнение другого процесса. Альтернативным путем является использование сигналов. Большинство ОС поддерживают погружение потока в состояние сна до тех пор, пока ему не отправит сигнал другой поток в результате изменения своего состояния.

Caching Failure – «кэширование ошибки» – тип программного бага (bug), при котором приложение сохраняет (кэширует) результаты, указывающие на ошибку даже после того, как она исправлена. Программист исправляет ошибку, но флаг ошибки не меняет своего состояния, поэтому приложение все еще не работает.

8.2 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Результаты выполнения практического задания.
5 Выводы.

8.3 Задания к лабораторной работе №8

Используя варианты задания из лабораторной работы №5, разработать программу, в основе которой будет прослеживаться явное присутствие одного (или нескольких) антишаблона. Привести методы улучшения кода разработанной программы (отсутствие антишаблонов).

Контрольные вопросы

1 Опишите особенности применения шаблонов при решении задач программирования.
2 Назовите преимущества следования шаблону Expert.
3 Назовите условия, при которых одному классу назначается обязанность создавать экземпляры другого класса.
4 Почему нежелательно наличие классов с высокой степенью связанности?
5 Приведите стандартные способы связывания двух объектов.
6 Что такое «зацепление» класса?
7 Назовите свойства класса, при которых ему производится делегирование обязанностей по обработке системных сообщений.
8 Назовите признаки антишаблона.
9 Приведите примеры антишаблонов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Java Database Connectivity. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Java_Database_Connectivity.

2 JDBC API в Java – обзор и туториал. Javenue – Программирование на Java. Информационные технологии [Электронный ресурс]. – Режим доступа : http://www.javenue.info/post/java-jdbc-api.

3 Блинов, И. Н. Java 2. Практическое руководство / И. Н. Блинов, В. С. Романчик. – Минск : УниверсалПресс, 2005. – 400 с.

4 Блинов, И. Н. Java. Промышленное программировние / И. Н. Блинов, В. С. Романчик. – Минск : УниверсалПресс, 2007. – 704 с.

5 Сервлет (Java). Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Сервлет_(Java).

6 Перри, Б. У. Java сервлеты и JSP. Сборник рецептов / Б. У. Перри. – М. : Кудиц-пресс, 2009. – 768 с.

7 Тейт, Б. Горький вкус Java / Б. Тейт. – СПб. : Питер, 2003. – 334 с.

8 JavaServer Pages. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/JavaServer_Pages.

9 Буч, Г. Язык UML. Руководство пользователя / Г. Буч, Дж. Рамбо, А. Джекобсон. – М. : ДМК, 2000. – 432 с.

10 GRASP. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/GRASP.

11 Design Patterns. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/ Design_Patterns.

12 Ларман, К. Применение UML 2.0 и шаблонов пректирования. Введение в объектно-ориентированный анализ и проектирование / К. Ларман. – 3-е изд. – СПб. : Изд. дом. «Вильямс», 2012. – 736 с.

13 Эдди, С. Э. XML : справочник / С. Э. Эдди. – СПб. : Питер, 2000. – 480 с.

14 Стелтинг, С. Java. Применение шаблонов Java / С. Стелтинг, О. Маасен. – М. : Вильямс, 2002. – 576 с.

15 Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [и др.]. – СПб. : Питер, 2007. – 366 с.

16 Кириевски, Д. Рефакторинг с использованием шаблонов / Д. Кириевски. – М. : Вильямс, 2008. – 400 с.

17 Объектно-ориентированные языки программирования – Объектно-ориентированное программирование [Электронный ресурс]. – Режим доступа : https://bourabai.ru/alg/oop11.htm.

18 Java – прогопедия [Электронный ресурс]. – Режим доступа : http://progopedia.ru/language/java.

19 Основная информация о языке Java – Университет ИТМО [Электронный ресурс]. – Режим доступа : https://neerc.ifmo.ru/wiki/index.php?title.

20 Java. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Java.

21 Java – StudFiles [Электронный ресурс]. – Режим доступа : https://studfiles.net/preview/2874314/page:4.

22 Паттерны для новичков: MVC vs MVP vs MVVM – Хабрахабр [Электронный ресурс]. – Режим доступа : https://habrahabr.ru/post/215605.

23 Паттерн MVP – Professor Web. Net & Web Programming [Электронный ресурс]. – Режим доступа : https://professorweb.ru/my/level36/36_4.php.

24 Model-View-ViewModel. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Model-View-ViewModel.

25 Паттерн HMVC в веб-разработке – Хабрахабр [Электронный ресурс]. – Режим доступа : https://habrahabr.ru/post/212065.

26 Инкапсуляция. Наследование. Полиморфизм. [Электронный ресурс]. – Режим доступа : http://codrob.ru/lesson/26.

27 Модификаторы доступа и инкапсуляция METANIT.COM. Сайт о программировании [Электронный ресурс]. – Режим доступа : https://metanit.com/java/tutorial/3.3.php.

Св. план 2019, поз. 4

Учебное издание

Шнейдеров Евгений Николаевич
Писарчик Андрей Юрьевич
Казючиц Владислав Олегович

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ЯЗЫКЕ JAVA. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

ПОСОБИЕ

Редакторы *Е. В. Иванюшина, Е. С. Юрец*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *В. М. Задоля*

Подписано в печать 06.03.2023. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 5,58. Уч.-изд. л. 6,0. Тираж 50 экз. Заказ 36.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изадний №1/238 от 23.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск