Laboratory Work #29 (part 2)

Java Information Handling and Regular Expressions

LEARN. GROW. SUCCEED.

 Department: <Software of Information Systems and Technologies>
Faculty of Information Technology and Robotics
Belarusian National Technical University
by Viktor Ivanchenko / ivanvikvik@bntu.by / Minsk

ЛАБОРАТОРНАЯ РАБОТА # 29 (part 2)
Обработка текста в Java. Использование регулярных выражений

Цель работы 

Закрепить навыки работы со строками на примере разработки интерактивных приложений для обработки текстовой информации.

Требования 

1) Необходимо создать интерактивное приложение для анализа текста по программированию согласно индивидуальному заданию (необходимо выполнить минимум два задания).
2) Анализируемый текст в процессе работы программы должен быть представлен в виде соответствующих бизнес-объектов (сущностей). К примеру, объект-текст содержит в себе объекты, ассоциированы с блоками текста и программного кода. Блоки текста, в свою очередь, представляют собой объекты-контейнеры, в которых содержатся предложения. А сами предложения являются хранилищами простых слов, знаков препинаний и других частей предложения. Классы, на базе которых будут создаваться вышеописанные объекты, являются классами-сущностями. Они должны содержать в себе только определённую текстовую информации. У них не должно быть никаких методов бизнес логики.
3) Анализ (разбор) текста должен осуществлять специальный утилитный класс-парсер. Именно данный класс должен возвращать бизнес объект (объекты), с которыми и должна работать бизнес логика приложения.
4) Для разбиения тестовой информации необходимо использовать регулярные выражения.
5) Предусмотреть восстановление текста в первоначальный (оригинальный) вид из программных объектов-сущностей. Оригинальный текст и текст, который

получается в результате восстановления, должны полностью совпадать. Это и будет показателем высокого качества разработанного класса-парсера.

6) Спроектировать UML-диаграмму классов и интерфейсов, составляющих архитектуру приложения.
7) При проектировании и реализации программы рекомендуется использовать архитектурный шаблон MVC, а также фундаментальные SOLID и GRASP принципы.
8) Классы и другие сущности программы должны быть грамотно структурированы по соответствующим пакетам и иметь отражающую их функциональность названия.
9) Программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом. Работа с консолью должна быть минимальной.
10) Приложение должно корректно обрабатывать любые исключительные ситуации, которые могут возникнуть в процессе работы программы, а также вести журналирование данных ситуаций. В качестве логгера рекомендуется использовать библиотеку логгирования Apache Log4j.
11) Для подтверждения работоспособности и адекватности программы, вся модель проекта должна быть покрыта соответствующими модульными тестами. Для модульного (*unit*) тестирования рекомендуется использовать тестовый фреймворк jUnit версии 4.0 и младше.
12) Необходимо по максимуму пытаться разрабатывать универсальный код.
13) Программа должна обязательно быть снабжена комментариями на английском языке, в которых необходимо указать краткое предназначение программы, номер лабораторной работы и её название, версию программы, ФИО разработчика, номер группы и дату разработки.
14) Исходный текст классов и демонстрационной программы рекомендуется также снабжать комментариями.
15) При разработке программ придерживайтесь соглашений по написанию кода на JAVA (Java Code-Convention*)*.

Индивидуальное задание

1) Найти наибольшее количество предложений текста, в которых есть одинаковые слова.
2) Вывести все предложения заданного текста в порядке возрастания количества слов в каждом из них.
3) Найти такое слово в первом предложении, которого нет ни в одном из остальных предложений.
4) Во всех вопросительных предложениях текста найти и напечатать без повторений слова заданной длины.
5) В каждом предложении текста поменять местами первое слово с последним, не изменяя длины предложения.
6) Напечатать слова текста в алфавитном порядке по первой букве. Слова, начинающиеся с новой буквы, печатать с красной строки.
7) Рассортировать слова текста по возрастанию доли гласных букв (отношение количества гласных к общему количеству букв в слове).
8) Слова текста, начинающиеся с гласных букв, рассортировать в алфавитном порядке по первой согласной букве слова.
9) Все слова текста рассортировать по возрастанию количества заданной буквы в слове. Слова с одинаковым количеством расположить в алфавитном порядке.
10) Существует текст и список слов. Для каждого слова из заданного списка найти, сколько раз оно встречается в каждом предложении, и рассортировать слова по убыванию общего количества вхождений.
11) В каждом предложении текста исключить подстроку максимальной длины, начинающуюся и заканчивающуюся заданными символами.
12) Из текста удалить все слова заданной длины, начинающиеся на согласную букву.
13) Отсортировать слова в тексте по убыванию количества вхождений заданного символа, а в случае равенства – по алфавиту.
14) В заданном тексте найти подстроку максимальной длины, являющуюся палиндромом, т.е. читающуюся слева направо и справа налево одинаково.

15) Преобразовать каждое слово в тексте, удалив из него все последующие вхождения первой буквы этого слова.
16) Преобразовать каждое слово в тексте, удалив из него все предыдущие вхождения последней буквы этого слова.
17) В некотором предложении текста слова заданной длины заменить указанной подстрокой, длина которой может не совпадать с длиной слова.

Best of LUCK with it, and remember to HAVE FUN while you're learning :)
Victor Ivanchenko

Пример текста для разбора

Autoboxing and Unboxing

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an `int` to an `Integer`, a `double` to a `Double`, and so on. If the conversion goes the other way, this is called unboxing.
Here is the simplest example of autoboxing:

```
Character ch = 'a';
```

The rest of the examples in this section use generics. If you are not yet familiar with the syntax of generics, see the Generics (Updated) lesson.
Consider the following code:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(i);
```

Although you add the `int` values as primitive types, rather than `Integer` objects, to `li`, the code compiles. Because `li` is a list of `Integer` objects, not a list of `int` values, you may wonder why the Java compiler does not issue a compile-time error. The compiler does not generate an error because it creates an `Integer` object from `i` and adds the object to `li`. Thus, the compiler converts the previous code to the following at runtime:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(Integer.valueOf(i));
```

Converting a primitive value (an `int`, for example) into an object of the corresponding wrapper class (`Integer`) is called autoboxing. The Java compiler applies autoboxing when a primitive value is:

- passed as a parameter to a method that expects an object of the corresponding wrapper class;
- assigned to a variable of the corresponding wrapper class.

Consider the following method:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i: li)
        if (i % 2 == 0)
            sum += i;
        return sum;
}
```

Because the remainder (`%`) and unary plus (`+=`) operators do not apply to `Integer` objects, you may wonder why the Java compiler compiles the method without issuing any errors. The compiler does not generate an error because it invokes the `intValue` method to convert an `Integer` to an `int` at runtime:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i : li)
        if (i.intValue() % 2 == 0)
            sum += i.intValue();
        return sum;
}
```

Converting an object of a wrapper type (`Integer`) to its corresponding primitive (`int`) value is called unboxing. The Java compiler applies unboxing when an object of a wrapper class is:

- passed as a parameter to a method that expects a value of the corresponding primitive type;
- assigned to a variable of the corresponding primitive type.

The `Unboxing` example shows how this works:

```
import java.util.ArrayList;
import java.util.List;

public class Unboxing {

    public static void main(String[] args) {
        Integer i = new Integer(-8);

        // 1. Unboxing through method invocation
        int absVal = absoluteValue(i);
        System.out.println("absolute value of " + i + " = " + absVal);

        List<Double> ld = new ArrayList<>();
        ld.add(3.1416);    // Π is autoboxed through method invocation.

        // 2. Unboxing through assignment
        double pi = ld.get(0);
        System.out.println("pi = " + pi);
    }

    public static int absoluteValue(int i) {
        return (i < 0) ? -i : i;
    }
}
```

The program prints the following:

```
absolute value of -8 = 8
pi = 3.1416
```

Autoboxing and unboxing lets developers write cleaner code, making it easier to read.