Laboratory Work #28 (part 1)

Java Input-Output Streams. The Decorator Pattern

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 # 28 (part 1)
Потоки ввода-вывода в Java. Структурный шаблон проектирования Декоратор

Цель работы 

Познакомиться с основами ввода-вывода в Java, со структурным шаблоном проектирования Декоратор и библиотекой потоков ввода-вывода *java.io*, спроектированной и реализованной на базе данного шаблона, а также практически закрепить данные знания на примере разработки интерактивного приложения.

Требования 

1) Скорректировать UML-диаграмму классов и интерфейсов, составляющих архитектуру основного приложения.
2) При проектировании и реализации программы рекомендуется использовать архитектурный шаблон MVC, а также фундаментальные SOLID и GRASP принципы.
3) Для построения необходимых объектов для создания бизнес-объектов программной системы и объектов-принтеров рекомендуется воспользоваться шаблонами Абстрактная фабрика (*The Abstract Factory Pattern*), Фабричный метод (*The Factory Method Pattern*) или Строитель (*The Builder Pattern*).
4) Классы и другие сущности программы должны быть грамотно структурированы по соответствующим пакетам и иметь отражающую их функциональность названия.
5) Приложение должно корректно обрабатывать любые исключительные ситуации, которые могут возникнуть в процессе работы программы.
6) Для подтверждения работоспособности и адекватности программы, вся модель проекта должна быть покрыта соответствующими модульными тестами. Для модульного (*unit*) тестирования рекомендуется использовать тестовый фреймворк *jUnit* или *TestNG*.
7) Необходимо по максимуму пытаться разрабатывать универсальный код.

8) Программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом. Работа с консолью должна быть минимальной.
9) 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. Исходный текст классов и демонстрационной программы рекомендуется также снабжать комментариями.
10) При разработке программ придерживайтесь соглашений по написанию кода на *JAVA* (*Java Code-Convention)*.

Основное задание

В программную систему, разработанную на предшествующем этапе, необходимо внести следующие расширяющие функционал возможности:

- возможность создавать объекты, которыми манипулирует бизнес логика, различными способами (с использованием генератора случайных чисел и «хардкода») и из различных источников данных (из текстового и бинарного файла) – для этого рекомендуется реализовать иерархию компонентов-создателей и сделать из взаимозаменяемыми в тестовом классе (контроллере);
- возможность выводить результирующие данные не только в консоль, но и сохранять их в файл – для этого рекомендуется также создать иерархию принтеров и сделать их взаимозаменяемыми в классе-контроллере.

Дополнительное задание

1) Необходимо разработать эффективную и полезную программу-утилиту, которая бы сравнивала содержимое двух файлов на эквивалентность хранимых данных.
2) Необходимо реализовать свой класс-декоратор, который бы весь ввод-вывод изменял (подменял) таким образом, чтобы все данные читались (записывались) в нижнем или в верхнем регистре.

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

Что нужно запомнить (краткие тезисы)

1. Ввод-вывод в программах на Java осуществляется посредством концепции соответствующих ***потоков (streams) и каналов (channels) ввода-вывода***.
2. ***Поток*** и ***канал*** – это Java-абстракции для организации управления процесса производства и потребления информации в программных системах.
3. ***Поток*** – это процесс перемещения данных между двумя компонентами, один из которых записывает, а другой читает.
4. К примеру, физически файловый поток представляет собой ***ресурс файловой системы*** ассоциированный с программным ***объектом-каналом*** между источником и приёмников данных для соответствующей передачи (или получения) целевой информации.
5. Базовые (основные) классы реализации потоков и каналов ввода-вывода в Java содержаться в двух основных пакетах: ***java.io*** и ***java.nio***. Первый пакет предоставляет «классический» API-интерфейс для ввода-вывода, а второй – альтернативу, которая направлена на гибкость и улучшение производительности платформы Java.
6. Главная разница между потоками пакета *java.io* и каналами пакета *java.nio* заключается в том, что первые ориентированы на работу с байтами, а вторые – на «буферы», которые содержат разные типы данных.
7. В Java ***все потоки унифицированы***, т.е. действуют одинаково, даже если они связаны с разными физическими устройствами. Это даёт возможность применять (использовать) одно и то же поведение к устройством различных типов. Например, одни и те же методы вывода в терминал можно также использовать и для вывода в файл.
8. ***Файл*** – это именованная область памяти.
9. В природе существуют только два типа файлов – бинарные и текстовые. Но помните, на физическом уровне даже текстовый файл состоит из байт. Отличие лишь в том, как анализируются эти байты и интерпретируется данные из него.
10. ***Бинарный файл*** содержит в себе набор байт, которые группируются в блоки, ассоциированные с соответствующими записанными данными. В таких файлах важна последовательность записи и чтения набора байт, т.е. если алгоритм

записи данных в бинарный файл не известен, то будет очень сложно прочитать целевые данные.

11. ***Текстовые файлы*** представляют собой также набор байт, но каждый из них (или несколько из них) представляют собой отдельные символы, согласно установленной таблицы кодировки. Следовательно, такие файлы не сложно анализировать и они будут читаться любыми текстовыми редакторами, даже если данный файл повреждён или некоторые его части отсутствуют.
12. Соответственно, в Java определены два типа потоков для работы с данными: ***байтовые*** *(since JDK 1.0)* и ***символьные*** *(since JDK 1.1 and JDK 5.0)*.
13. На самом низком уровне все механизмы системы ввода-вывода имеют байтовую организацию!
14. В качестве единицы информации для записи или чтения в байтовых потоках выступает **один байт** (восемь бит). Т.е. за один раз можно записать только значение из диапазона от **0** до **255**.
15. В качестве единицы информации для записи или чтения в символьных потоках выступает **символ** в **Unicode**-кодировке (может занимать как один байт, так и два байта, всё зависит от порядкового символа в таблице кодировки). В общем, с помощью символьных потоков можно записать символы в диапазоне с '*\u0000*' до '*0xFFFF*' (с **0** до **65535**).
16. По функционалу байтовые и символьные потоки дублируются, т.к. почти всё идентично.
17. Все классы потоков ввода-вывода имеют общее поведение в виде интерфейса ***java.io.Closeable*** *(since JDK 5.0)*, который содержит метод ***close()*** для уничтожения созданного ресурса (канала) для ввода или вывода данных.
18. Начиная с версии ***JDK 7.0*** интерфейс ***java.io.Closeable*** расширяет новый интерфейс ***java.lang.AutoCloseable*** *(since JDK 7.0)*, который используется для того, чтобы на объектах можно было автоматически вызывать метод ***close()***, если он создаётся с использованием новой конструкции ***try-with-resources***.
19. Все классы потоков вывода дополнительно содержат поведение, декларируемое интерфейсом ***java.io.Flushable*** *(since JDK 5.0)*. Данный интерфейс содержит единственный метод ***flush()***, которые необходимо вызывать перед вызовом метода ***close()*** для гарантированно полной передачи данных.

20. Во главе вершины байтовых потоков ввода-вывода стоят два абстрактных класса ***OutputStream*** и ***InputStream***, которые определяют базовые возможности чтения и записи неструктурированных последовательностей байтов и лежат в основе всех остальных байтовых потоком.
21. Класс ***InputStream*** с помощью трёх перегруженных методов ***read()*** определяет общее поведение для потоков ввода байтовых данных, а класс ***OutputStream*** с помощью трёх перегруженных методов ***write()*** определяет общее поведение для потоков вывода байтовых данных.
22. Во главе вершины символьных потоков ввода-вывода стоят два абстрактных класса ***Reader*** и ***Writer***, которые определяют базовые возможности чтения и записи последовательностей символов с поддержкой Unicode. На из основе строятся остальные символьные потоки.
23. Класс ***Reader*** с помощью четырёх перегруженных методов ***read()*** определяет общее поведение для потоков ввода символьных данных, а класс ***Writer*** с помощью пяти перегруженных методов ***write()*** определяет общее поведение для потоков вывода символьных данных.
24. Классы ***InputStreamReader*** и ***OutputStreamWriter*** являются связующим звеном между байтовыми и символьными потоками. Для преобразования используется целевая кодировка, которая задаётся одним из параметров соответствующих конструкторов.
25. Классы ***DataInputStream*** и ***DataOutputStream*** являются специальными потоковыми фильтрами, позволяющие считывать и записывать многобайтовые типы данных, к примеру, все примитивные данные.
26. Для буферизированного ввода-вывода используются объекты соответствующих классов ***BufferedInputStream***, ***BufferedOutputStream***, ***BufferedReader*** и ***BufferedWriter***.
27. Объекты классов ***PrintStream*** и ***PrintWriter*** упрощают вывода текстовых данных.
28. Объекты ***System.out*** и ***System.err*** являются экземплярами класса *PrintStream*, который является расширение класса *OutputStream*.
29. Объект ***System.in*** является конкретной реализацией потока *InputStream*.
30. Объекты соответствующих классов ***FileInputStream***, ***FileOutputStream***, ***FileReader*** и ***FileWriter*** позволяют работать с файлами в рамках локальной файловой системы.

31. Для работы с наборам символов (таблицей кодировки) в языке Java используют класс ***java.nio.charset.Charset*** и его экземпляры. В частности, статические метода ***forName()*** данного класса возвращает свой экземпляр кодировки, который был указан в виде параметра метода: *Charset utf8 = Charset.forName("UTF-8"),* а метод ***defaultCharset()*** позволяет текущую активную кодировку.

Как узнать доступные кодировки?

```
public static void main(String[] args) {
    SortedMap<String, Charset> map = Charset.availableCharsets();
    for (String code : map.keySet()) {
        System.out.println(code);
    }
}
```

Таблица результата выполнения вышеописанного кода

Big5	IBM500	JIS_X0212-1990	x-IBM1122	x-ISO-2022-CN-CNS
Big5-HKSCS	IBM775	KOI8-R	x-IBM1123	x-ISO-2022-CN-GB
CESU-8	IBM850	KOI8-U	x-IBM1124	x-JISAutoDetect
EUC-JP	IBM852	Shift_JIS	x-IBM1364	x-Johab
EUC-KR	IBM855	TIS-620	x-IBM1381	x-MacArabic
GB18030	IBM857	US-ASCII	x-IBM1383	x-MacCentralEurope
GB2312	IBM860	UTF-16	x-IBM300	x-MacCroatian
GBK	IBM861	UTF-16BE	x-IBM33722	x-MacCyrillic
IBM-Thai	IBM862	UTF-16LE	x-IBM737	x-MacDingbat
IBM00858	IBM863	UTF-32	x-IBM833	x-MacGreek
IBM01140	IBM864	UTF-32BE	x-IBM834	x-MacHebrew
IBM01141	IBM865	UTF-32LE	x-IBM856	x-MacIceland
IBM01142	IBM866	UTF-8	x-IBM874	x-MacRoman
IBM01143	IBM868	windows-1250	x-IBM875	x-MacRomania
IBM01144	IBM869	windows-1251	x-IBM921	x-MacSymbol
IBM01145	IBM870	windows-1252	x-IBM922	x-MacThai
IBM01146	IBM871	windows-1253	x-IBM930	x-MacTurkish
IBM01147	IBM918	windows-1254	x-IBM933	x-MacUkraine
IBM01148	ISO-2022-CN	windows-1255	x-IBM935	x-MS932_0213
IBM01149	ISO-2022-JP	windows-1256	x-IBM937	x-MS950-HKSCS
IBM037	ISO-2022-JP-2	windows-1257	x-IBM939	x-MS950-HKSCS-XP
IBM1026	ISO-2022-KR	windows-1258	x-IBM942	x-mswin-936
IBM1047	ISO-8859-1	windows-31j	x-IBM942C	x-PCK
IBM273	ISO-8859-13	x-Big5-HKSCS-2001	x-IBM943	x-SJIS_0213
IBM277	ISO-8859-15	x-Big5-Solaris	x-IBM943C	x-UTF-16LE-BOM
IBM278	ISO-8859-2	x-euc-jp-linux	x-IBM948	X-UTF-32BE-BOM
IBM280	ISO-8859-3	x-EUC-TW	x-IBM949	X-UTF-32LE-BOM
IBM284	ISO-8859-4	x-eucJP-Open	x-IBM949C	x-windows-50220
IBM285	ISO-8859-5	x-IBM1006	x-IBM950	x-windows-50221
IBM290	ISO-8859-6	x-IBM1025	x-IBM964	x-windows-874
IBM297	ISO-8859-7	x-IBM1046	x-IBM970	x-windows-949
IBM420	ISO-8859-8	x-IBM1097	x-ISCII91	x-windows-950
IBM424	ISO-8859-9	x-IBM1098	x-iso-8859-11	x-windows-iso2022jp
IBM437	JIS_X0201	x-IBM1112	x-JIS0208	

Пример работы с байтовыми потоками в Java

Задание

Необходимо написать тестовую программу, которая бы демонстрировала запись последовательности целых чисел в бинарный файл, а также чтения из него.

Решение

1) Создадим два класса: класс *BinaryWorker* – основной класс для реализации чтения и записи набора целочисленных значений и класс Test – класс для тестирования функционала основного класса.
2) Общий вид декларации основного класса *BinaryWorker*:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class BinaryWorker {
    public static void write(String fileName) {
        OutputStream stream = null;

        try {

            stream = new FileOutputStream(fileName);

            for (int i = 0; i < 100; i++) {
                stream.write(i);
            }

        } catch (IOException exc) {
            // exception handling here
        } finally {
            if (stream != null) {
                try {
                    stream.flush();
                    stream.close();
                } catch (IOException exc) {
                    // exception handling here
                }
            }
        }
    }
    public static void read(String fileName) {
```

Все основные потоки ввода-вывода Java находятся в пакете ***java.io***

Главная абстракция для байтовых потоков записи данных – базовый класс ***OutputStream***

Класс, отвечающий за запись данных в файл ***FileOutputStream***

Основной метод, для записи одного байта

Главное проверяемое исключение (базовый класс всех исключений) в библиотеки потоков ввода-вывода в Java

Метод ***flush()*** гарантирует, что данные будут записаны прежде, чем произойдёт уничтожение ресурса

Любая работа с потоком ввода-вывода в Java должна сопровождаться в конце его очисткой, которая осуществляется в методе ***close()***

```
        InputStream stream = null;

        try {

            stream = new FileInputStream(fileName);
            int data;

            while ((data = stream.read()) != -1) {
                System.out.print(data + " ");
            }

        } catch (IOException exc) {
            // exception handling here
        } finally {
            if (stream != null) {
                try {
                    stream.close();
                } catch (IOException exc) {
                    // exception handling here
                }
            }
        }
    }
}
```

Главная абстракция для байтовых потоков чтения данных – базовый класс ***InputStream***

Класс, отвечающий за чтение данных из файла ***FileInputStream***

Основной метод для чтения одного байта. В случае достижения конца файла метод возвращает -1

Главное проверяемое исключение (базовый класс всех исключений) в библиотеки потоков ввода-вывода в Java

3) Общий вид декларации класса *Test*:

```
public class Test {

    public static void main(String[] args) {

        String fileName = "c:\\test\\numbers.bin";

        BinaryWorker.write(fileName);

        BinaryWorker.read(fileName);
    }
}
```

Обратите внимание, как задается путь к файлу

Пример работы с символьными потоками в Java

Задание

Необходимо написать тестовую программу, которая бы демонстрировала запись последовательности целочисленных значений в текстовый файл, а также чтения из него.

Решение

4) Создадим два класса: класс *CharacterWorker* – основной класс для реализации чтения и записи набора целочисленных значений и класс Test – класс для тестирования функционала основного класса.
5) Общий вид декларации основного класса *CharacterWorker*:

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Reader;
import java.io.Writer;

public class CharacterWorker {

    public static void write(String fileName) {
        Writer stream = null;

        try {
            stream = new FileWriter(fileName);

            for (int i = 0; i < 100; i++) {
                stream.write(i);
            }

        } catch (IOException exc) {
            // exception handling here
        } finally {
            if (stream != null) {
                try {
                    stream.flush();
                    stream.close();
                } catch (IOException exc) {
                    // exception handling here
                }
            }
        }
    }
    public static void read(String fileName) {
```

Главная абстракция для символьных потоков записи данных – базовый класс ***Writer***

Класс, отвечающий за запись данных в файл ***FileWriter***

Основной метод, для записи одного символа

```
        Reader stream = null;

        try {

            stream = new FileReader(fileName);
            int data;

            while ((data = stream.read()) != -1) {
                System.out.print(data + " ");
            }

        } catch (IOException exc) {
            // exception handling here
        } finally {
            if (stream != null) {
                try {
                    stream.close();
                } catch (IOException exc) {
                    // exception handling here
                }
            }
        }
    }
}
```

Главная абстракция для символьных потоков чтения данных – базовый класс ***Reader***

Класс, отвечающий за чтение данных из файла ***FileReader***

Основной метод для чтения одного символа. В случае достижения конца файла метод возвращает -1

6) Общий вид декларации класса *Test*:

```
public class Test {

    public static void main(String[] args) {

        String fileName = "c:\\test\\numbers.txt";

        CharacterWorker.write(fileName);

        CharacterWorker.read(fileName);
    }
}
```

Обратите внимание, как задается путь к файлу

Пример работы с потоками в Java с использованием конструкции *try-with-resources*

Задание

Сделайте рефакторинг основного класса *BinaryWorker* таким образом, чтобы он для создания объектов потоков использовал новую конструкции *try-with-resources*, которая появилась в JDK 7.0.

Решение

Общий вид декларации основного класса *BinaryWorker* с внедрённой в него новой конструкции языка Java *try-with-resources:*

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class BinaryWorker {
    public static void write(String fileName) {

        try (OutputStream stream = new FileOutputStream(fileName)) {

            for (int i = 0; i < 100; i++) {
                stream.write(i);
            }

        } catch (IOException exc) {
            // exception handling here
        }
    }

    public static void read(String fileName) {

        try (InputStream stream = new FileInputStream(fileName)) {

            int data;

            while ((data = stream.read()) != -1) {
                System.out.print(data + " ");
            }

        } catch (IOException exc) {
            // exception handling here
        }
    }
}
```

Любой объект, который реализует интерфейс ***AutoCloseable***, может быть создан при помощи конструкции *try-with-resources*

При выходе из блока *try* у всех объектов, которые создавались с помощью конструкции *try-with-resources* будет автоматически вызван метод ***close()***

Контрольные вопросы

1) Что такое файл?
2) Что такое атрибут файла?
3) Перечислите все возможные атрибуты файла.
4) Что такое файловая система?
5) Основные функции файловой системы?
6) Какие типы файлов в общем случае существуют в компьютерном мире?
7) Опишите основную концепцию использования потоков (*streams*) ввода-вывода в Java для организации процессов хранения, записи, чтения и обмена данными между программами.
8) Что физически представляет собой поток?
9) В каких стандартных Java-пакетах сосредоточены основные реализации потоков ввода-вывода?
10) На базе какого структурного шаблона проектирования базируется библиотека основных классов и объектов ввода-вывода *java.io*?
11) Какие разновидности потоков существуют в Java?
12) Какую организацию имеют все средства ввода-вывода в Java на самом низком уровне?
13) Опишите общее поведение (интерфейсы), которым обладают почти все потоки ввода-вывода в Java.
14) Опишите основные классы исключительных ситуаций для поддержки работы системы ввода-вывода. Какое исключение является центральным для данной системы?
15) Какова основная концепция байтовых потоков ввода-вывода? Единица данных, с которой работают байтовые потоки ввода-вывода в Java?
16) Опишите базовые классы, задающие основную абстракцию поведения байтовых потоков.
17) Опишите основные классы-реализации, задающие основное поведение байтовых потоков с различными источниками данных.
18) Опишите классы-декораторы, задающие дополнительное поведение байтовых потоков для работы с данными.

19) Какова основная концепция символьных потоков ввода-вывода? Единица данных, с которой работают символьные потоки ввода-вывода в Java?
20) На базе какой кодировки реализована работа символьных потоков?
21) Опишите базовые классы, задающие основную абстракцию поведения символьные потоков.
22) Опишите основные классы-реализации, задающие основное поведение символьных потоков с различными источниками данных.
23) Опишите классы-декораторы, задающие дополнительное поведение символьных потоков для работы с данными.
24) В чём отличия байтовых и символьных потоков ввода-вывода, а в чём их сходство?
25) Опишите укрупнённо UML-диаграмму иерархии основных классов системы ввода-вывода в Java и их взаимосвязи.
26) Опишите архитектуру и поведение структурного шаблона проектирования Декоратор (*The Decorator Pattern*). Когда и где его применяют?
27) На каких принципах базируется структурный шаблон проектирования Декоратор?
28) На какую конкретно функциональность библиотеки ввода-вывода *java.io* повлиял структурный шаблон проектирования Декоратор?
29) Что есть общего у архитектур шаблонов проектирования Стратегия и Декоратор?
30) При общей архитектурной схожести обеих шаблонов проектирования Стратегия и Декоратор в чём их принципиальное отличие?