Индивидуальная работа № 4. ВВОД-ВЫВОД. ПАКЕТ JAVA.IO

Цель работы. Работа посвящена пакету JAVA.IO, поддерживающему операции ввода-вывода.

6.1. Теоретический материал

Как известно всем программистам с давних времен, большинство программ не может выполнять свою работу, не имея доступа к внешним данным. Данные извлекаются из источника ***ввода.*** Результат программы направляется в ***вывод.***

На языке Java эти понятия определяются очень широко. Например, источником ввода или местом вывода может служить **сетевое соединение, буфер памяти или дисковый файл** — всеми ими можно манипулировать при помощи классов ввода- вывода Java. Хотя физически они совершенно различны, все эти устройства описываются единой абстракцией — ***потоком.***

Поток - это логическая сущность, которая выдает или получает информацию. Поток присоединен к физическому устройству при помощи системы ввода- вывода Java. Все потоки ведут себя похоже, даже несмотря на то, что физические устройства, к которым они присоединены, в корне отличаются.

6.1.1 Класс Console

Ранее (в Java SE 6) был добавлен класс Console. Он используется для чтения и записи информации на консоли, если таковая существует, и реализует интерфейс Flushable.

Класс Console, прежде всего, введен для удобства, поскольку большая часть его функциональных возможностей доступна через объекты System.in и System.out. Однако его применение позволяет упростить некоторые виды консольных итераций, особенно при чтении строк с консоли.

Класс Console **не поддерживает конструкторов.** Его объект получают вызовом метода System.console ().

static System.console()

Если консоль доступна, возвращается ссылка на нее. В противном случае возвращается значение **null.** Консоль не будет доступна во всех классах, поэтому если возвращается значение **null,** консольные операции ввода-вывода невозможны.

Класс Console определяет методы, перечисленные в табл. 19.6. Обратите внимание на то, что методы ввода, такие как метод readLine **(),** передают исключение IO- Exception, когда возникают ошибки ввода. Класс исключения **ioError** происходит от класса Error и означает сбой ввода-вывода, который происходит вне контроля вашей программы. То есть обычно вы не будете перехватывать исключение **ioError**. Откровенно говоря, если исключение **ioError** возникнет в процессе обращения к консоли, обычно это свидетельствует о катастрофическом сбое системы.

Также обратите внимание на методы readPassword **()**, которые позволяют приложению считывать пароль, не отображая его на экране. Читая пароли, следует "обнулять" как массив, содержащий строку, введенную пользователем, так и массив, содержащий правильный пароль, с которым нужно сравнить первую строку. Это уменьшает шансы вредоносной программы получить пароль при помощи сканирования памяти.

ния памяти.

Таблица 6.1. Методы, определенные в классе **Console**

Метод	Описание
`void flush()`	Выполняет физическую запись буферизованного вывода на консоль
`Console format(String` *формСтрока*`, Object...` *аргументы*`)`	Выводит на консоль *аргументы*, используя формат, указанный в *формСтрока*
`Console printf(String` *формСтрока*`, Object...` *аргументы*`)`	Выводит на консоль *аргументы*, используя формат, указанный в *формСтрока*
`Reader reader()`	Возвращает ссылку на объект класса, производного от класса `Reader`, соединенный с консолью
`String readLine()`	Читает и возвращает строку, введенную с клавиатуры. Ввод прекращается нажатием клавиши <Enter>. Если достигнут конец входного потока консоли, возвращается значение `null`. В случае сбоя передается исключение `IOError`
`String readLine(String` *формСтрока*`, Object...` *аргументы*`)`	Отображает строку приглашения, форматированную в соответствии с *формСтрока* и *аргументы*, затем читает и возвращает строку, введенную с клавиатуры. Ввод прекращается, когда пользователь нажимает клавишу <Enter>. Если достигнут конец входного потока консоли, возвращается значение `null`. В случае сбоя передается исключение `IOError`
`char[ ] readPassword()`	Читает и возвращает строку, введенную с клавиатуры. Ввод прекращается нажатием клавиши <Enter>. При этом строка не отображается. Если достигнут конец входного потока консоли, возвращается значение `null`. В случае сбоя передается исключение `IOError`
`char[ ] readPassword(String` *формСтрока*`, Object...` *аргументы*`)`	Отображает строку приглашения, форматированную в соответствии с *формСтрока* и *аргументы*, а затем читает и возвращает строку, введенную с клавиатуры. Ввод прекращается, когда пользователь нажимает клавишу <Enter>. При этом строка не отображается. Если достигнут конец входного потока консоли, возвращается значение `null`. В случае сбоя передается исключение `IOError`
`PrintWriter writer()`	Возвращает ссылку на объект класса, производного от класса `Writer`, ассоциированный с консолью

Рассмотрим пример, демонстрирующий класс **Console** в действии.

```
import java.io.*;
class ConsoleDemo {
public static void main(String args[])   {
String str;
Console con;
// Получить ссылку на консоль.
con = System.console();
// Если нет доступной консоли,  выход,
```

```
if (con == null)  return;
// Прочесть строку и отобразить ее.
str = con.readLine("Введите строку:   ") ;
con.printf("Вот ваша строка;  %s\n",  str);
}
}
```

Вывод этого примера.
Введите строку: Это тест. Вот ваша строка: Это тест.

6.1.2. Сериализация

Сериализация — это процесс записи состояния объекта в байтовый поток. Она удобна, когда нужно сохранить состояние вашей программы в области постоянного хранения, такой как файл. Позднее вы можете восстановить эти объекты, используя процесс десериализации.

Сериализация также необходима в реализации *дистанционного вызова методов* **(Remote Method Invocation — RMI).**

RMI позволяет объекту Java на одной машине обращаться к методу объекта Java на другой машине. Объект может быть применен как аргумент этого дистанционного метода. Посылающая машина сериализует объект и передает его. Принимающая машина десериализует его.

Предположим, что объект, подлежащий сериализации, ссылается на другие объекты, которые, в свою очередь, имеют ссылки на еще какие-то объекты. Такой набор объектов и отношений между ними формирует ориентированный граф. В этом графе могут присутствовать и циклические ссылки. Иными словами, объект X может содержать ссылку на объект Y, а объект Y — обратную ссылку на X. Объекты также могут содержать ссылки на самих себя.

Средства сериализации и десериализации объектов устроены так, что могут корректно работать во всех этих сценариях. Если вы попытаетесь сериализовать объект, находящийся на вершине такого графа объектов, то все прочие объекты, на которые имеются ссылки, также будут рекурсивно найдены и сериализованы. Аналогично во время процесса десериализации все эти объекты и их ссылки корректно восстанавливаются.

Ниже приведен обзор интерфейсов и классов, поддерживающих сериализацию.

Интерфейс Serializable

Только объект, реализующий интерфейс Serializable**,** может быть сохранен и восстановлен средствами сериализации. Интерфейс Serializabl**e** не содержит никаких членов. Он просто используется для того, чтобы указать, что класс может быть сериализован. Если класс является сериализуемым, все его подклассы также сериализуемы.

Переменные, объявленные как **transient,** не сохраняются средствами сериализации. Не сохраняются также статические переменные.

Интерфейс Externalizable

Средства Java для сериализации и десериализации спроектированы так, что большая часть работы по сохранению и восстановлению состояния объекта выполняется автоматически. Однако бывают случаи, когда программисту нужно управлять

этим процессом. Например, может оказаться желательным использовать технологии сжатия и шифрования. Интерфейс Externalizable предназначен именно для таких ситуаций.

Интерфейс Externalizable определяет следующие два метода.

void readExternal(Obj eetInput *входнойПоток)*
throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput *выходнойПоток)*
throws IOException

В этих методах ***входнойПоток*** — это байтовый поток, из которого объект может быть прочитан, а ***выходнойПоток*** — байтовый поток, куда он записывается.

Интерфейс ObjectOutput

Интерфейс ObjectOutput расширяет интерфейсы AutoCloseable и DataOutpit**,** поддерживает сериализацию объектов. Он определяет методы, показанные в табл. 6.2. Особо отметим метод writeObject **().** Он вызывается для сериализации объекта. В случае ошибок все методы этого интерфейса передают исключение IOException**.**

Таблица 6.2. Методы, определенные в интерфейсе **ObjectOutput**

Метод	Описание
`void close()`	Закрывает вызывающий поток. Последующие попытки записи передадут исключение `IOException`
`void flush()`	Финализирует выходное состояние, чтобы очистить все буферы. То есть все выходные буферы сбрасываются

Окончание табл. 19.7

Метод	Описание
`void write(byte буфер[])`	Записывает массив байтов в вызывающий поток
`void write(byte буфер[], int смещение, int колБайтов)`	Записывает диапазон *колБайтов* байт из массива *буфер* начиная с *буфер[смещение]*
`void write(int b)`	Записывает одиночный байт в вызывающий поток. Из аргумента *b* записывается только младший байт
`void writeObject(Object объект)`	Записывает объект *объект* в вызывающий поток

Класс ObjectOutputStream

Класс ObjectOutputStream расширяет класс OutputStream и реализует интерфейс ObjectOutput**.** Этот класс отвечает за запись объекта в поток. Конструктор его выглядит так.

ObjectOutputStream(OutputStream *выходнойПоток)* throws IOException

Аргумент ***выходнойПоток*** представляет собой выходной поток, в который мо-

гут быть записаны сериализуемые объекты. Закрытие объекта класса ObjectOutputStream приводит к закрытию также внутреннего потока, определенного аргументом *выходнойПоток*.

Несколько часто используемых методов класса перечислено в табл. 63. В случае ошибки все они передают исключение IOException. Присутствует также класс PutField, вложенный в класс ObjectOutputStream. Он обслуживает запись постоянных полей, и описание его применения выходит за рамки настоящей книги.

Таблица 6.3. Некоторые из наиболее часто используемых методов класса **ObjectOutputStream**

Метод	Описание
`void close()`	Закрывает вызывающий поток. Последующие попытки записи передадут исключение `IOException`. Внутренний поток тоже закрывается
`void flush()`	Финализирует выходное состояние, так что все буферы очищаются. То есть все выходные буферы сбрасываются
`void write(byte буфер[])`	Записывает массив байтов в вызывающий поток
`void write(byte буфер[], int смещение, int колБайтов)`	Записывает диапазон *колБайтов* байт из массива *буфер* начиная с *буфер[смещение]*
`void write(int b)`	Записывает одиночный байт в вызывающий поток. Из аргумента *b* записывается только младший байт
`void writeBoolean(boolean b)`	Записывает значение типа `boolean` в вызывающий поток
`void writeByte(int b)`	Записывает значение типа `byte` в вызывающий поток. Записываемый байт — младший из аргумента *b*
`void writeBytes(String строка)`	Записывает байты, составляющие строку `str`, в вызывающий поток
`void writeChar(int c)`	Записывает значение типа `char` в вызывающий поток
`void writeChars(String строка)`	Записывает символы, составляющие строку *строка*, в вызывающий поток
`void writeDouble(double d)`	Записывает значение типа `double` в вызывающий поток
`void writeFloat(float f)`	Записывает значение типа `float` в вызывающий поток
`void writeInt(int i)`	Записывает значение типа `int` в вызывающий поток
`void writeLong(long l)`	Записывает значение типа `long` в вызывающий поток
`final void writeObject(Object объект)`	Записывает объект *объект* в вызывающий поток
`void writeShort(int i)`	Записывает значение типа `short` в вызывающий поток

Интерфейс ObjectInput

Интерфейс ObjectInput расширяет интерфейсы AutoCloseable и DataInput и определяет методы, перечисленные в табл. 6.4. Он поддерживает сериализацию объектов. Особо стоит отметить метод readObject **().** Он вызывается для десериализации объекта. В случае ошибок все эти методы передают исключение IOException**.** Метод readObject **()** также может передать исключение ClassNotFoundException**.**

Таблица 6.4. Методы, определенные **в** интерфейсе **objectInput**

Метод	Описание
`int available()`	Возвращает количество байтов, которые доступны во входном буфере в настоящий момент
`void close()`	Закрывает вызывающий поток. Последующие попытки чтения вызовут передачу исключения `IOException`. Внутренний поток тоже закрывается
`int read()`	Возвращает целочисленное представление следующего доступного байта ввода. При достижении конца файла возвращается значение `-1`
`int read(byte буфер[])`	Пытается прочитать до *буфер.длина* байт в *буфер*, начиная с *буфер[смещение]*, возвращая количество байтов, которые удалось прочитать. При достижении конца файла возвращается значение `-1`
`int read(byte буфер[], int смещение, int колБайтов)`	Пытается прочитать до *колБайтов* байт в *буфер*, начиная с *буфер[смещение]*, возвращая количество байтов, которые удалось прочитать. При достижении конца файла возвращается значение `-1`
`Object readObject()`	Читает объект из вызывающего потока
`Long skip(long numBytes)`	Игнорирует (т.е. пропускает) *колБайтов* байт вызывающего потока, возвращая количество действительно пропущенных байтов

Класс ObjectInputStream

Этот класс расширяет класс InputStream реализует интерфейс ObjectInput. Класс ObjectInputStream отвечает за чтение объектов из потока. Ниже показан конструктор этого класса.

ObjectInputStream(InputStream *входнойПоток)* throws IOException

Аргумент ***входнойПоток*** — это входной поток, из которого должен быть прочитан сериализованный объект. Закрытие объекта класса ObjectInputStream приводит к закрытию также внутреннего потока, определенного аргументом ***входнойПоток***.

Несколько часто используемых методов этого класса показано в табл. 6.5. В случае ошибок все они передают исключение **IOException.** Метод readObject() также может передать исключение **ClassNotPoundException.** Также в классе ObjectInputStream присутствует вложенный класс по имени GetField. Он обслуживает чтение постоянных полей, и описание его применения выходит за рамки настоящей книги.

Таблица 6.5. Часто используемые методы, определенные в классе **ObjectInputStream**

Метод	Описание
`int available()`	Возвращает количество байтов, доступных в данный момент во входном буфере
`void close()`	Закрывает вызывающий поток. Последующие попытки чтения вызовут передачу исключения `IOException`. Внутренний поток тоже закрывается
`int read()`	Возвращает целочисленное представление следующего доступного байта ввода. При достижении конца файла возвращается значение `-1`
`int read(byte` *буфер*`[], int` *смещение*`, int` *колБайтов*`)`	Пытается прочитать до *колБайтов* байт в *буфер*, начиная с *буфер*`[`*смещение*`]`, возвращая количество байтов, которые удалось прочитать. При достижении конца файла возвращается значение `-1`
`Boolean readBoolean()`	Читает и возвращает значение типа `boolean` из вызывающего потока
`byte readByte()`	Читает и возвращает значение типа `byte` из вызывающего потока
`char readChar()`	Читает и возвращает значение типа `char` из вызывающего потока
`double readDouble()`	Читает и возвращает значение типа `double` из вызывающего потока
`double readFloat()`	Читает и возвращает значение типа `float` из вызывающего потока
`void readFully(byte` *буфер*`[])`	Читает *буфер*`.`*длина* байт в *буфер*. Возвращает управление, только когда все байты прочитаны
`void readFully(byte` *буфер*`[],` `int` *смещение*`, int` *колБайтов*`)`	Читает *колБайтов* байт в *буфер*, начиная с *буфер*`[`*смещение*`]` Возвращает управление, только когда прочитано *колБайтов* байт

Окончание табл. 19.10

Метод	Описание
`int readInt()`	Читает и возвращает значение типа `int` из вызывающего потока
`int readLong()`	Читает и возвращает значение типа `long` из вызывающего потока
`final Object readObject()`	Читает и возвращает объект из вызывающего потока
`short readShort()`	Читает и возвращает значение типа `short` из вызывающего потока
`int readUnsignedByte()`	Читает и возвращает значение типа `unsigned byte` из вызывающего потока
`int readUnsignedShort()`	Читает и возвращает значение типа `unsigned short` из вызывающего потока

Пример сериализации

В следующей программе показано, как использовать сериализацию и десериа-

лизацию объектов. Начинается она с создания экземпляра объекта My Class**.** Этот объект имеет три переменные экземпляра типа String, int и double**.** Именно эту информацию мы хотим сохранять и восстанавливать.

В программе создается объект класса FileOutputStream**,** который ссылается на файл по имени " **serial**", и для этого файлового потока создается объект класса ObjectOutputStream**.** Метод writeObject**()** этого объекта класса ObjectOutputStream используется затем для сериализации объекта. Объект выходного потока очищается и закрывается.

Далее создается объект класса FilelnputStream**,** который ссылается на файл по имени " **serial**", и для этого файлового потока создается объект класса ObjectInputStream**.** Метод **readObject** () класса ObjectInputStream используется для последующей десериализации объекта. После этого входной поток закрывается.

Обратите внимание на то, что объект MyClass определен с реализацией интерфейса Serializable. Если бы этого не было, передалось бы исключение NotSerializableException**.** Поэкспериментируйте с этой программой, объявляя некоторые переменные экземпляра MyClass как transient**.** Эти данные не будут сохраняться при сериализации.

```
// Демонстрация сериализации
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7.
import java.io.*;
public class SerializationDemo {
public static void main(String args[])   {
// Сериализации объекта
try  ( ObjectOutputStream objOStrm =
new ObjectOutput Stream(new FileOutputStream (" serial") )   )
{
MyClass object1 = new MyClass("Hello",  -7,  2.7e10);
System.out.println("object1:   " + object1);
objOStrm.writeObject(object1);
}
catch(IOException e)   {
System.out.println("Исключение во время сериализации  :   "  + e) ;
}
// Десериализация объекта
try  { ObjectlnputStream objIStrm =
new ObjectlnputStream(new FilelnputStream("serial"))   )
{
MyClass object2 =  (MyClass) objIstrm.readObject () ;
System.out.println("object2:   " + object2);
}
catch(Exception e)   {
System.out.println ("Исключение во время сериализации:  "  + e) ;
System.exit (0);
}
}
}
```

```
class MyClass implements Serializable {
String s;
int i;
double d;
public MyClass(String s,  int i, double d)   {
this.s =s;
this.i = i;
this.d = d;
}
public String toString()   {
return  "s="  + s +   ";   i="  +  i  +   " ;   d="  + d;
}
}
```

Эта программа демонстрирует идентичность переменных экземпляра объектов **object**1 и **object2.** Вот ее вывод.

Object1: s=Hello; i=-7; d=2.7E10

object2: s=Hello; i=-7; d=2.7E10

6.1.3 Преимущества потоков

Потоковый интерфейс ввода-вывода в Java предоставляет чистую абстракцию для сложных и зачастую обременительных задач. Композиция классов фильтрующих потоков позволяет динамически строить собственные настраиваемые потоковые интерфейсы, которые отвечают вашим требованиям к передаче данных. Программы Java, использующие эти абстрактные высокоуровневые классы — InputStream, OutputStream, Reader и Writer**,** — будут корректно функционировать в будущем, даже когда появятся новые усовершенствованные конкретные потоковые классы. Эта модель работает очень хорошо, когда мы переключаемся от набора потоков на основе файлов к сетевым потокам и потокам сокетов. И наконец, сериализация объектов играет важную роль в программах Java различных типов. Классы сериализации ввода-вывода Java обеспечивают переносимое решение этой непростой задачи.

6.2 Контрольный пример

Откройте Java-проект, созданный в лабораторной работе №4. Изменим набор классов **Item, Book, MusicCD, MusicDVD** и **TestItem** для организации работы со сканером и возможностью ввода и вывода данных в внешние файлы.

Изменим наши классы так, чтобы данные для новых объектов классов пользователь ввод сам с клавиатуры в консоле. Также добавим для удобства общее пользовательское меню для работы с классами, в котором организуем ввод данных для создания объектов классов.

Откройте первый класс – **Book.class**. Добавим возможность в класс ввода данных пользователем с клавиатуры.

Для возможности работы со сканером подключите библиотеку **import java.util.Scanner**;

Для работы со сканером создайте следующую процедуру:

```
public static Book makeNewBook(Scanner in) {
```

```
in.skip("\r\n");
System.out.println("Программное создание объекта класса  Book....");
System.out.println("Введите название книги ....");
String newnaimenovanie = in.nextLine();
System.out.println("Введите фамилию автора ....");
String newname = in.nextLine();
System.out.println("Введите год издательства ....");
int newgod = in.nextInt();
System.out.println("Введите издательство ....");
String newizdatelstvo = in.nextLine();
System.out.println("Введите цену ....");
float newprice = in.nextFloat();
System.out.println("Введите кол-во страниц ....");
int newdlitelnost = in.nextInt();


return new Book (newnaimenovanie, newname, newgod, newizdatelstvo, newprice, newdlitelnost);  // возвращение введенных значений в конструктор класса
}
```

Далее самостоятельно измените аналогично классы **MusicCD, MusicDVD**.

Затем добавим пользовательское меню в класс **TestItem**.
Подключите библиотеку **import java.util.Scanner**;
Создайте следующую процедуру, которая выведет меню на экран:

```
private static void showMenu() {
        System.out.println("\n");
        System.out.println("1 - Show collection");
        System.out.println("2 - Add new Book");
        System.out.println("3 - Add new MusicCD");
        System.out.println("4 - Add new MusicDVD");
        System.out.println("5 - Save program");
        System.out.println("6 - Load program");
        System.out.println("7 - Exit");
}
```

Далее, для того, чтобы организовать возможность пользователю выбирать элементы меню с клавиатуры реализуем работу со сканером. Для этого внутри класса **TestItem** в главной процедуре **main** объявим сканер и реализуем работу с меню. Для этого измените процедуру **main** на следующую:

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

        Scanner in  = new Scanner(System.in).useDelimiter("\r\n");
        showMenu();
```

```
                while(true) {
                        int menuItem = in.nextInt();

                        if (menuItem == 7) {
                                System.out.println("Bye");
                                System.exit(0);

                        } else if (menuItem == 1) {
                                //пока ничего не делаем

                        } else if (menuItem == 2) {
                                Book newBook = Book.makeNewBook(in);
                                System.out.println("Информация про книгу:
"+newBook.toString());


                        } else if (menuItem == 3) {
                        MusicCD newMusicCD= MusicCD.makeNewMusicCD(in);
                        System.out.println("Информация про музыку:
"+newMusicCD.toString());

                        } else if (menuItem == 4) {
                    MusicDVD newMusicDVD= Mu-
sicDVD.makeNewMusicDVD(in);
                                System.out.println("Информация про фильм:
"+newMusicDVD.toString());

                        } else if (menuItem == 5) {
                                //      сохранить данные в файл

                        } else if (menuItem == 6) {

                                // извлечь данные из файла

                        }
                        showMenu();
                }

        }
```

Далее, организуем работу с **коллекциями**. Например, мы хотим вводить не по одному объекту, а хотим, чтобы каждый новый объект добавлялся в коллекцию **Book, MusicCD, MusicDVD.**

Создадим специальный класс для работы с коллекцией. Создайте и добавьте в свой пакет новый класс **Compar.class**.

```
import java.util.*;
```

```
public class Compar {

        private Comparator<Item> comparator = null;  // объявление будущей сортировки

        private ArrayList<Item> item = new ArrayList<Item>();  // создание коллекции списка-массива типа класса Item – общего родительского класса

        public void setItem(ArrayList<Item> item) {  // присвоение коллекции
                this.item = item;
        }

        public ArrayList<Item> getItem(){   // возвращение коллекции
                return item;
        }

        public void add(Item item_service) {
        // добавление элементов в коллекцию
                this.item.add(item_service);
        }

        public void setComparator(Comparator<Item> comparator) {
                this.comparator = comparator;
        }

        public void show() {
    // отображение на экран всей коллекции
                if (this.comparator != null)
                   Collections.sort(this.item, this.comparator);   // сортировка коллекции

                Iterator<Item> iter = this.item.iterator();   // установка итератора на начало коллекции
                while (iter.hasNext()) {   // пока не конец коллекции
                        System.out.println(iter.next());       // вывод объектов коллекции
                }
        }
}
```

Далее изменим базовый родительский класс для реализации интерфейса сортировки по коллекции.

Для этого откройте класс **Item.class** вверху подключите библиотеку **import java.util.*;**, чтобы были доступны интерфейсы **Collection, ArrayList, Iterator, Comparator**.

Далее измените объявление класса на следующее:

```
public class Item implements Comparable<Item>
```

Далее необходимо описать в классе метод сортировки, допустим мы будем сортировать коллекцию по возрастанию цены продукции. Для этого добавьте еще один метод в класс:

```
public int compareTo(Item another) {
return (int)(this.price - another.getPrice());
}
```

Далее изменим класс TestItem в главной процедуре main. Измените библиотеку на import java.util.*;, чтобы были доступны интерфейсы Collection, ArrayList, It-erator, Comparator.

Внутри процедуры main создайте новый объект класса Compar:

```
Compar compar = new Compar();
```

Затем изменим меню. Для того, чтобы обеспечить создание новых объектов и добавление их в коллекцию нужно вызвать метод compar.add(obj).

Например, для объектов книг меню изменится на:

```
else if (menuItem == 2) {
        Book newBook = BookmakeNewBook(in);
        System.out.println("Информация про книгу:"+newBook.toString());
        compar.add(newBook);
}
```

Аналогично выполните изменение меню для объектов классов **MusicCD** и **MusicDVD**.

Далее, чтобы вывести всю отсортированную коллекцию на экран нужно вызвать метод **compar.show()**. Поэтому измените соответствующее меню на:

```
else if (menuItem == 1) {
        compar.show(); }
```