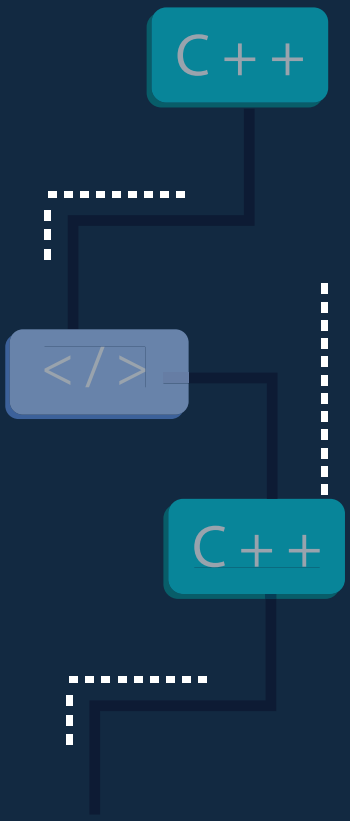

top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

Язык программирования JAVA

Урок №10

Работа с файлами

Содержание

1. Потоки ввода-вывода

Ввод-вывод в Java реализован с помощью специализированных классов, называемых классами потокового ввода-вывода. Потоком ввода-вывода называется объект какого-либо потокового класса. В остальной части этого урока будем использовать просто термин ***поток***. Это не должно вызывать у вас путаницы с потоками выполнения (объектами класса Thread) или с потоками (производными от интерфейса Stream<T>), рассмотренными в предыдущей части урока. Наша задача заключается в рассмотрении семейства классов потокового ввода-вывода.

Прежде всего, выделим основные характеристики потока. Поток может иметь направление передачи данных. Есть потоки передающие данные из приложения вовне. Такие потоки называются выходными потоками. Есть потоки, передающие данные извне в приложение. Они называются входными. А есть потоки двунаправленные, способные передавать данные в обоих направлениях.

Следующая важная характеристика потока – это объем данных передаваемых потоком за одну операцию ввода-вывода. Есть потоки байтовые, передающие данные побайтово. Еще есть потоки символьные, передающие данные как символы какой-либо кодировки, например, UTF-8.

Также потоки можно характеризовать тем, откуда данные попадают в поток и куда они выводятся потоком. Например, есть потоки, работающие с файлами на диске, или с консолью, или с оперативной памятью, или с сокетами.

Иерархия потоковых классов состоит из двух групп классов. Одна группа представляет собой байтовые потоки,

другая – символьные. Базовыми классами для байтовых потоков являются классы InputStream и OutputStream, а для символьных – Reader и Writer. На диаграмме внизу показаны взаимосвязи между основными классами байтовой иерархии. Базовыми в этой иерархии являются абстрактные классы InputStream и OutputStream.

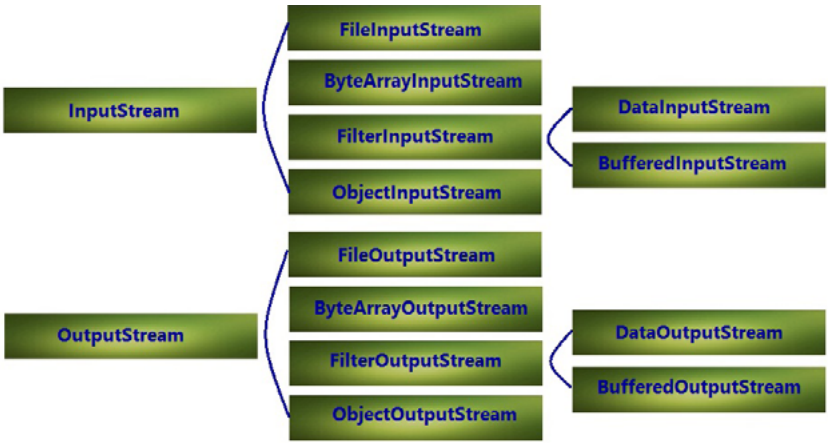

Рис. 1. Байтовые потоки ввода-вывода

На следующей диаграмме приведены основные символьные потоковые классы. Базовыми в этой иерархии являются абстрактные классы Reader и Writer.

Рис. 2. Символьные потоки ввода-вывода

Рассмотрим все эти потоковые классы подробнее. Однако перед знакомством с потоковыми классами будет по-

лезно рассмотреть еще один класс. Это класс File, который не является потоковым классом, но играет важную роль в вводе–выводе при работе с файлами и папками.

Класс File

Этот класс работает непосредственно с файловой системой и является незаменимым при работе с файлами и папками на диске. Класс File не имеет отношения к процессам чтения или записи. Он содержит информацию о самих файлах и папках. Экземпляр класса File, можно ассоциировать с файлом или директорией, после чего, использовать методы класса для получения информации о директории или файле, для изменения характеристик папок или файлов и конечно же, для создания новых объектов файловой системы.

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/io/File.html.

File имеет такие конструкторы:

```
File (File parent, String child) создает новый объект
                    с именем child по тому же пути,
                    где размещен объект parent;
File (String path) создает новый объект по пути path;
File (String parent, String child)  создает новый
                    объект с именем child по пути
                    parent;
File (URI uri)      создает новый объект по
                    абстрактному пути, полученному
                    путем конвертации uri.
```

Прежде всего, вы должны понять следующее: если вы создали объект класса File, это вовсе не означает автоматического создания файла или папки на диске. Чтобы создать файл или папку, соответствующие объекту File, надо вызывать специальные методы.

Создайте новый проект и добавьте в метод main() такой код:

```
String fileName = "test.txt";
String fullName = "";
String dirName = System.getProperty("user.dir");
fullName = dirName + File.separator + fileName;
System.out.println("File path : " + fullName);
File file = new File(fullName);
```

Мы создали объект класса файл, указав в качестве пути корневую папку текущего проекта. Обратите внимание, что для получения пути к корневой папке проекта мы использовали свойство "user.dir". А вот для разделителя между именем папки и именем файла мы использовали поле File.separator. Запомните, что в классе File хранятся необходимые свойства, позволяющие не обращаться к вызову метода System.getProperty(). Запустите проект, а затем перейдите в его корневую папку и убедитесь, что файл test.txt не создан. Это то, о чем мы говорили: создание объекта класса File не приводит автоматически к созданию файла на диске.

Добавьте к коду такое продолжение:

```
if ( ! file.exists()){
    try {
        if( file.createNewFile() )
```

```
            System.out.println("File created!");
        else
            System.out.println("Something Wrong!");
    } catch (IOException ex) {
        Logger.getLogger(Filetest.class.getName()).
               log(Level.SEVERE, null, ex);
    }
}else{
    System.out.println("File already exists!");
}
```

Выполните проект снова. Вывод этого кода будет таким:

```
File path: C:\Users\admin\Documents\NetBeansProjects\
    filetest\test.txt.
File created!
```

Перейдите в текущую папку проекта и убедитесь, что теперь файл test.txt создан. Мы рассмотрели пример создания файла на диске. Обратите внимание на то, что мы ничего в созданный файл не записывали. Класс File не умеет читать или писать. Для этих целей применяются потоковые классы.

Для создания папок этот класс использует методы mkdir() и mkdirs(). Первый метод позволяет создать только одну папку, а второй – цепочку вложенных папок. Добавьте к проекту такой код и выполните проект снова.

```
String dirname = dirName + "/tmp/user/java/bin";
File d = new File(dirname);
// Create directories now.
d.mkdirs();
```

Теперь вы увидите в текущей папке проекта всю цепочку вложенных папок: "/tmp/user/java/bin". А вот если бы вы вызвали метод mkdir(), то никакие папки созданы бы не были, потому, что этот метод требует указания только одной папки, без вложенных папок.

В этом классе собрано еще много полезных методов. Например, таких как:

```
deleteOnExit()   если вызвать этот метод для объекта
                 класса File, то файл на диске, асco-
                 циированный с этим объектом, будет
                 удален автоматически, при завершении
                 работы JVM;
getFreeSpace()   возвращает кол-во свободных байт на том
                 диске, где расположен текущий объект;
getTotalSpace()  возвращает общий размер диска,
                 на котором расположен текущий объект;
setReadonly()    делает файл доступным только для чтения.
```

В дальнейшем мы часто будем пользоваться этим классом.

Байтовый ввод-вывод

В базовых абстрактных классах байтовой иерархии (InputStream и OutputStream) описаны основные методы для чтения, записи, закрытия потока и некоторые другие:

Методы в классе InputStream

```
read()          Возвращает текущий доступный символ из
                входного потока в виде целого значения.
read(byte b[]) Читает b.length байт из входного
                потока в массив b. Возвращает количество
                прочитанных из потока байт.
```

```
read(byte b[], int start, int len) Читает len байт из
            входного потока в массив b, но
            располагает их в массиве, начиная с
            элемента start. Возвращает количество
            прочитанных байт.
skip(long n) Пропускает во входном потоке n байт.
            Возвращает количество пропущенных байт.
available()  Возвращает количество байт в потоке,
            доступных для чтения.
mark(int readlimit)  Ставит метку в текущей позиции
            входного потока, которую можно
            Будет использовать до тех пор, пока из
            потока не будет прочитано readlimit байтов.
reset()      возвращает указатель потока на
            установленную ранее метку.
markSupported()  Возвращает true, если данный поток
            поддерживает операции mark и reset.
close()      Закрывает входной поток.
```

Методы в классе OutputStream:

```
write(int b) Записывает один байт в выходной поток.
write(byte b[])  Записывает в выходной поток весь
            указанный массив байтов.
write(byte b[], int start, int len) Записывает в
            поток часть массива – len байтов,
            начиная с элемента b[start].
flush()      Очищает выходные буферы
            буферизированных потоков.
close()      Закрывает выходной поток.
```

При работе с вводом-выводом важную роль играют исключения. Всегда надо учитывать такие исключе-

ния, как IOException (и производное от него исключение FileNotFoundException) и SecurityException. Вы уже привыкли к тому, что среда разработки Java очень бережно относится к обработке исключительных ситуаций. Поэтому отметьте тот факт, что код, использующий ввод-вывод должен заключаться в try-catch блоки. Кроме традиционных try и catch блоков, очень полезно будет использовать необязательный finally блок, в котором удобно будет выполнять обязательное закрытие потоков. Альтернативой этому может быть использование улучшенного блока try с ресурсами:

```
try( resource ){
    //IO processing
}
```

FileInputStream и FileOutputStream

После этой предварительной информации рассмотрим побайтовую работу с файлами.

Сначала рассмотрим несколько примеров превращения входного потока в файл. В первом примере предполагается, что данные во входной поток попадают из файла на диске, и что размер этого файла позволяет прочитать его сразу. Если же мы не можем полагать, что размер данных во входном потоке приемлем, чтобы прочитать его за один раз, то надо использовать подход, продемонстрированный во втором примере.

Пример 1:

```
InputStream in=null;
OutputStream out=null;
```

```
byte[] buffer=null;
try {
    in = new FileInputStream(new File("test.txt"));
    buffer = new byte[in.available()];
    in.read(buffer);
    File file = new File("outputFile.tmp");
    out = new FileOutputStream(file);
    out.write(buffer);
} catch (FileNotFoundException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
} catch (IOException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
}

finally{
    try {
        in.close();
        out.close();
    } catch (IOException ex) {
        Logger.getLogger(Filetest.class.getName()).
               log(Level.SEVERE, null, ex);
    }
}
```

Рассмотрим приведенный выше код. Мы создаем объект in типа InputStream и связываем его с существующим файлом на диске. Это и есть наш входной поток, через который данные будут поступать в наше приложение. Понятно, что источником данных для этого потока является файл, с которым мы связали наш поток. Этот файл уже должен находиться в текущей папке вашего приложения. Обратите внимание, что конструктору потока мы передаем объект типа File. Затем мы создаем массив, в кото-

рый хотим прочитать содержимое потока. Мы выбрали поток, который читает данные побайтово, поэтому тип массива – byte[]. Поскольку мы договорились, что размер потока допускает чтение за один раз, мы должны сделать размер массива таким, чтобы все данные разместились в нем. Смотрите, как мы указываем размерность массива при создании – используя потоковый метод available(). Затем методом read() за один вызов читаем все данные из потока в созданный байтовый массив.

Мы хотим создать копию прочитанного файла, поэтому создаем выходной поток out типа OutputStream. Снова, используя класс File, ассоциируем выходной поток с файлом на жестком диске. Вы понимаете, что для выходного потока можно указать не существующий файл, и поток создаст этот файл самостоятельно. И, наконец, за один вызов метода write() от объекта выходного потока, записываем содержимое нашего массива в файл. Проверьте текущую паку своего приложения и убедитесь, что выходной файл создался и что он является копией исходного файла.

Отметьте важный момент: мы выполняем чтение и запись файл побайтово, поэтому таким способом можем копировать не только текстовые файлы, а и бинарные. Положите в текущую папку проекта какой-нибудь графический файл, свяжите его со входным потоком и запустите приложение. Будет создана копия вашего графического файла.

Обратите внимание на try-catch-finally блоки, без которых этот код работать не будет. У вас есть альтернатива. Посмотрите, как выглядит этот же код, но с улучшенным try блоком с двумя ресурсами:

```
File file = new File("outputFile2.tmp");
byte[] buffer=null;
try (InputStream in =
    new FileInputStream(new File("1.jpg"));
    OutputStream out = new FileOutputStream(file);){
    buffer = new byte[in.available()];
    in.read(buffer);
    out.write(buffer);
}
```

Этот код немного короче, так как отсутствуют явные вызовы метода close(). Однако, вам придется использовать инструкции throws, чтобы кто-то обработал возможные исключения FileNotFoundException и IOException или добавлять catch блоки.

Теперь рассмотрим случай, когда мы не уверены, что размер данных во входном потоке допускает чтение всех данных целиком. В этом случае будем читать данные в цикле, порциями.

Пример 2:

```
InputStream in=null;
OutputStream out=null;
byte[] buffer=new byte[8*1024];
try {
    in = new FileInputStream(new File("2.jpg"));
    File file = new File("outputFile2.tmp");
    out = new FileOutputStream(file);
    int bytesRead=0;
    while ((bytesRead = in.read(buffer)) != -1) {
        out.write(buffer, 0, bytesRead);
    }
} catch (FileNotFoundException ex) {
    Logger.getLogger(Filetest.class.getName()).
```

```
            log(Level.SEVERE, null, ex);
}

catch (IOException ex) {
    Logger.getLogger(Filetest.class.getName()).
            log(Level.SEVERE, null, ex);
}

finally{
    try {
        in.close();
        out.close();
    } catch (IOException ex) {
         Logger.getLogger(Filetest.class.getName()).
                log(Level.SEVERE, null, ex);
    }
}
```

Отличия этого примера состоят в том, что мы выполняем чтение входного потока в цикле и на каждой итерации прочитанную порцию данных сразу записываем в выходной поток. Это избавляет нас от необходимости создавать большой массив для чтения, чтобы в него поместились все данные. В приведенном случае размер массива для чтения выбирается произвольно и влияет только на число итераций, которые придется выполнить, чтобы скопировать все данные. Отметьте использование переменной bytesRead, в которую метод read() возвращает количество прочитанных байт. Во всех итерациях (если их будет больше одной) метод read() будет заполнять весь массив buffer и значение bytesRead будет совпадать с размером массива buffer. А на последней итерации будет прочитана оставшаяся порция данных, которая, как

правило, будет меньше размера массива. Поэтому очень важно, в методе write() в третьем параметре указывать именно количество фактически прочитанных байт, а не размер массива.

Вы уже увидели, что байтовые потоки позволяют работать как с текстовыми, так и с бинарными файлами. Какие еще возможности дает нам использование этих потоков? В предыдущих примерах мы выполняли чтение в массив, однако есть возможность выполнять чтение буквально побайтово. Это, конечно медленнее, но иногда бывает необходимо обрабатывать каждый прочитанный байт. Рассмотрим такой пример.

Пример 3:

```
FileInputStream in = null;
FileOutputStream out = null;

try {
    in = new FileInputStream(new File("test.txt"));
    File file = new File("outputFile3.txt");
    out = new FileOutputStream(file);
    int c;

    while ((c = in.read()) != -1) {
        if(c < 65)out.write(c);
    }

} catch (FileNotFoundException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
}
catch (IOException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
```

```
}
finally{
    try {
        in.close();
        ut.close();
    } catch (IOException ex) {
        Logger.getLogger(Filetest.class.getName()).
                log(Level.SEVERE, null, ex);
    }
}
```

В этом примере мы снова выполняем чтение входного потока в цикле. Но теперь используем метод read(), который читает один байт и возвращает код прочитанного байта. В этом же цикле мы выполняем простую проверку – если прочитанный символ не буква, а цифра или разделитель, мы записываем его в файл, в противном случае – не записываем. Возможно, этот пример надуманный, но он наглядно демонстрирует логику побайтовой обработки. Не поленитесь, вставьте во входной поток файл со знаками препинания и цифрами и посмотрите, что получится в выходном файле. Объясните увиденное.

ByteArrayInputStream и ByteArrayOutputStream

Во всех предыдущих примерах мы рассматривали классы FileInputStream и FileOutputStream, которые являются реализациями базовых абстрактных классов InputStream и OutputStream. Рассмотрим еще другие байтовые потоки, производные от этих двух абстрактных классов. Очень часто в Java возникает необходимость преобразовать какой-нибудь объект в массив байт. Скоро вы будет ра-

ботать с сокетами и при использовании протокола UDP это преобразование будет актуальным постоянно. Рассмотрим пример преобразования изображения в байтовый массив. Использование потока – не единственный способ выполнить такое преобразование, но знать этот прием необходимо.

Пример 4:

```
File fnew=new File("2.jpg");
try{
    BufferedImage bImage=ImageIO.read(fnew);
    ByteArrayOutputStream baos = new
             ByteArrayOutputStream();
    ImageIO.write(bImage, "jpg", baos );
    byte[] imageInByte=baos.toByteArray();
} catch (IOException ex) {
    Logger.getLogger(Filetest.class.getName()).
            log(Level.SEVERE, null, ex);
}
```

В приведенном примере мы с помощью класса ImageIO читаем графический файл в объект типа BufferedImage. Затем создаем объект байтового потока ByteArrayOutputStream, который хранит свои данные в оперативной памяти. Записываем объект типа BufferedImage в этот поток. Дальше остается преобразовать поток в байтовый массив с помощью метода toByteArray(). Вы еще не оценили этот пример? Тогда продолжим. Мы пока ничего не делаем с полученным байтовым массивом. Давайте исправим эту ситуацию. Очень часто возникает необходимость выполнить действие, которое условно можно назвать так "преобразование выходного потока

во входной". Другими словами, в одной части кода вы записали объект в выходной поток, а в другом месте вам надо этот объект прочитать. В Java самый удобный способ выполнить такое действие – использовать потоки ByteArrayOutputStream и ByteArrayInputStream. Сначала объект надо записать в выходной поток. Затем преобразовать этот поток в байтовый массив и передать его, куда надо. Затем байтовый массив преобразовать во входной поток и прочитать его.

Добавьте в состав приложения класс, который будет принимать массив байт с изображением и выводить это изображение во фрейме:

```
public class ImageFrame {
    BufferedImage image = null;
    JFrame form = null;

    public ImageFrame(byte[] imageInByte)
           throws IOException
    {
        image = ImageIO.read(new
                ByteArrayInputStream(imageInByte));
        form = new JFrame();
        form.setSize(image.getWidth(),
                image.getHeight());
        form.setAlwaysOnTop(true);

        JPanel pn = new JPanel(){
        @Override
        public void paint(Graphics g) {
            super.paint(g);
            g.drawImage(image, 0, 0, image.
                getWidth(), image.getHeight(), null);
        }};
```

```
        pn.setSize(image.getWidth(),
                image.getHeight());
        form.add(pn);
        form.setVisible(true);
    }
}
```

Обратите внимание, что в конструкторе класса мы создаем входной поток ByteArrayInputStream вокруг массива байт, который перед этим был получен из выходного потока ByteArrayOutputStream. Теперь добавьте одну строку в предыдущий код:

```
File fnew=new File("2.jpg");
try{
    BufferedImage bImage=ImageIO.read(fnew);
    ByteArrayOutputStream baos=new
        ByteArrayOutputStream();
    ImageIO.write(bImage, "jpg", baos );
    byte[] imageInByte=baos.toByteArray();
    ImageFrame imf =new ImageFrame(imageInByte);
}
catch (IOException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
}
```

Запустите приложение и вы получите фрейм с изображением. Просто создание объекта нашего класса ImageFrame приводит к созданию фрейма, размеры которого совпадают с размерами переданного изображения, и выводу этого фрейма поверх других окон. У меня это выглядит так:

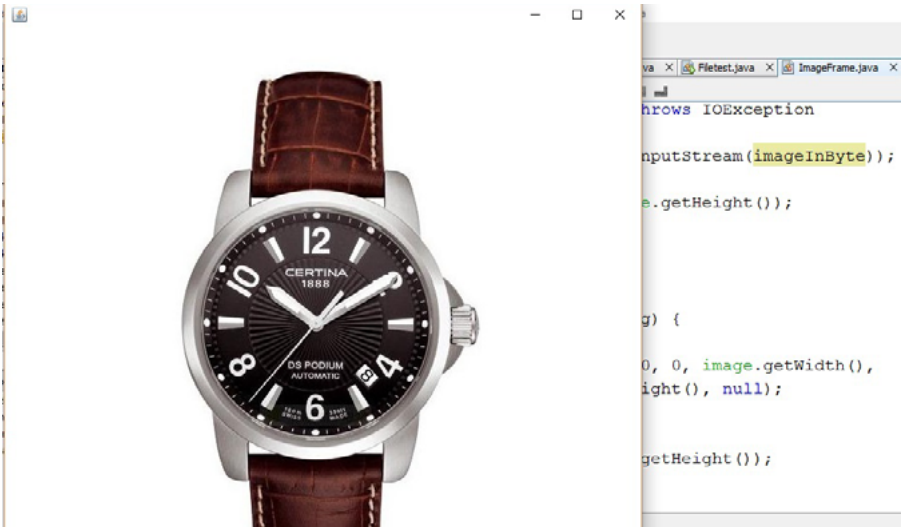

Рис. 3. Изображение во фрейме

Рассмотрим еще пример преобразования строки с помощью байтового потока в оперативной памяти.

Пример 5:

```
String line="This is a sample string to be capitalized";
ByteArrayInputStream bais =
          new ByteArrayInputStream(line.getBytes());
int ch;
StringBuilder sb = new StringBuilder();
while ( (ch = bais.read()) != -1) {
    sb.append(Character.toUpperCase((char) ch));
}
System.out.println("Capitalized string: " +
    sb.toString());
```

В этом примере мы преобразовываем строку в байтовый поток, а затем выполняем побайтовое чтение потока и одновременно требуемое преобразование прочитанных

данных. Поскольку при преобразовании мы создаем новую строку, здесь удобнее использовать класс StringBuilder.

ObjectInputStream и ObjectOutputStream

Перейдем к рассмотрению следующих из байтовых потоков – ObjectInputStream и ObjectOutputStream. Эти классы являются незаменимыми, когда вам надо записывать или читать объекты каких-либо классов. А такая необходимость возникает очень часто. Рассмотрим примеры использования этих потоков.

Пример 6:

```
FileOutputStream fout=null;
ObjectOutputStream oout=null;
try {
    fout = new FileOutputStream("fish.txt");
    Fish f = new Fish("salmon",2.5,180);
    oout = new ObjectOutputStream(fout);
    oout.writeObject(f);
} catch (FileNotFoundException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
} catch (IOException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
}
finally
{
    try {
        oout.close();
    } catch (IOException ex) {
        Logger.getLogger(Filetest.class.getName()).
               log(Level.SEVERE, null, ex);
    }
}
```

Разберем этот код. Сначала мы создаем уже знакомый нам объект потока FileOutputStream и связываем его с файлом "fish.txt", в который хотим записать объект класса Fish. Потом создаем сам объект, который будем записывать. А вот теперь начинается нечто новое. Мы создаем объект потока ObjectOutputStream вокруг ранее созданного объекта потока FileOutputStream. В таких случаях говорят, что поток ObjectOutputStream оборачивает поток FileOutputStream. Теперь наш поток ObjectOutputStream тоже связан с файлом "fish.txt", но он обладает качествами, которые отсутствуют у потока FileOutputStream. Когда вы выполняете обертку одного потока вокруг другого, вы добавляете потоку новые возможности. В нашем случае, мы теперь можем записывать в поток объекты любых классов, потому, что у потока появился метод writeObject(), которого не было у потока FileOutputStream. Запустите это приложение и обратите внимание на ошибку, которая возникла при выполнении. Дело в том, что если мы хотим записывать объекты какого-либо класса в потоки, то этот класс должен удовлетворять некоторым условиям. А именно – этот класс должен наследовать интерфейсу Serializable. Вы уже привыкли, что если класс наследует интерфейсу, то он должен реализовать все методы, объявленные в интерфейсе. Однако в Java существуют интерфейсы, к которых не объявлены никакие методы. Такие интерфейсы называются маркерами. Интерфейс Serializable является маркером, поэтому в классе Fish достаточно добавить implements Serializable и никаких методов реализовывать не надо. Добавьте в класс Fish наследование, запустите приложение и убедитесь, что наш выходной файл создан и в него что-то записано. Не

пытайтесь читать этот файл. Он бинарный. Мы с вами подробно разберем то, что сейчас сделали в последнем разделе нашего урока, когда будем говорить о сериализации. Поэтому сейчас просто запомните: чтобы записывать и читать объекты класса, надо наследовать этот класс от Serializable.

Давайте прочитаем то, что записали в файл "fish.txt".

Пример 7:

```
FileInputStream fin = null;
try {
    fin = new FileInputStream(new File("fish.txt"));
    ObjectInputStream oin = new ObjectInputStream(fin);
    Fish f = (Fish) oin.readObject();
    System.out.println(f);
} catch (FileNotFoundException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
} catch (IOException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
} catch (ClassNotFoundException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
} finally {
    try {
        fin.close();
    } catch (IOException ex) {
        Logger.getLogger(Filetest.class.getName()).
               log(Level.SEVERE, null, ex);
    }
}
```

В этом примере вам все должно быть понятно. Мы снова создаем поток ObjectInputStream как обертку вокруг

потока FileInputStream и используем метод readObject() для чтения объекта из файла. Поскольку этот метод возвращает тип Object, необходимо выполнять явное преобразование типа к требуемому типу данных. Затем мы выводим в консоль описание прочитанного объекта, чтобы убедиться, что чтение прошло успешно. В нашем классе перегружен метод toString(), поэтому мы просто вызываем println().

Вывод этого кода будет таким:

```
salmon  weight:2.5  price:180.0
```

BufferedInputStream и BufferedOutputStream

Перейдем к рассмотрению еще двух оберточных потоков – BufferedInputStream и BufferedOutputStream. Вы должны знать, что операции вставки и извлечения данных из потока являются очень медленными. Если мы имеем дело с байтовыми потоками, то такие операции выполняются для каждого обрабатываемого байта. Чем больше байт – тем медленнее обработка данных. Эта проблема присутствует не только в Java, а во всех языках программирования, где используется потоковый ввод-вывод. И способ бороться с этой проблемой известен давно. Это – буферизация потоков. Что такое буферизированный поток? Рассмотрим буферизированный входной поток. Если вы вводите в этот поток данные, они не идут сразу непосредственно в поток. Они накапливаются в специальном буфере. И только, когда весь буфер заполнен, выполняется вставка всех данных в поток за одну операцию. Таким образом, вместо вставки каждого байта непосредственно в поток, байты

накапливаются в буфере. Вставка в буфер – это простая операция не требующая много времени и ресурсов для выполнения. А затем за одну операцию вставки в поток, в поток вставляются все данные из буфера. Таким образом сокращается количество операций вставки в поток и экономится время. Аналогично работает буферизированный выходной поток. Данные не выводятся из потока сразу, они накапливаются в буфере. Когда буфер заполнен, все за один раз выводится из потока.

Работа с буферизированными потоками обладает некоторой спецификой. Рассмотрим выходной поток. Предположим, в буфер складываются данные для вывода в файл. Эти данные закончились, а буфер еще не заполнился. Та часть данных, которая осталась в буфере в файл может не записаться. Поэтому в таких потоках есть специальный метод flush(), позволяющий принудительно вывести данные из буфера, даже, если буфер еще не заполнен.

В Java буферизация потоков выполняется, как обертка потоками BufferedInputStream и BufferedOutputStream. Рассмотрим примеры использования этих потоков.

Пример 8:

```
String text = "This lines of text should be written
                in file\r\n"
    + "using buffered stream.\r\n"
    + "Just one more line.\r\n";
try(FileOutputStream out=new FileOutputStream("notes.txt");
    BufferedOutputStream bos =
            new BufferedOutputStream(out))
{
    byte[] buffer = text.getBytes();
```

```
    bos.write(buffer, 0, buffer.length);
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}
```

В этом примере мы используем try блок с ресурсами и уже известным вам приемом создаем буферизированную обертку вокруг обычного потока FileOutputStream. Теперь запись в файл будет выполнена несколько быстрее, чем при отсутствии буферизации. Кстати, имейте в виду, что массив buffer, используемый в коде, не имеет никакого отношения к буферу потока. Это просто массив, из которого выполняется вставка данных в поток. Буфер потока скрыт во внутренней реализации. Его размер равен 8192 байт, но это значение может изменяться в зависимости от реализации. Положитесь пока на разработчиков в том, что они не советуют нам управлять этим размером.

Выполним чтение с использованием буферизации.

Пример 9:

```
try(FileInputStream fin =
        new FileInputStream(new File("notes.txt"));
    BufferedInputStream bis =
        new BufferedInputStream(fin)){
    int c;
    while((c=bis.read())!=-1){
        System.out.print((char)c);
    }
}
catch(Exception e){
    System.out.println(e.getMessage());
}
```

Вывод этого кода будет таким:

```
This lines of text should be written in file
using buffered stream.
Just one more line.
```

Как вы понимаете, чтение этого файла тоже выполнялось быстрее, благодаря наличию буферизации.

Давайте теперь рассмотрим символьный ввод вывод.

Символьный ввод-вывод

При написании приложения очень часто приходится иметь дело с текстом. Для этого и существуют символьные потоки. Вы уже видели, что байтовые потоки позволяют спокойно читать и писать текстовые файлы. Возникает вопрос – зачем для ввода-вывода текста нужны еще другие потоки? Причин здесь несколько. В символьных потоках, с которыми мы начинаем знакомство, при чтении мы получаем из потока не байт, а символ char. Размер этого символа будет зависеть от кодировки. Кроме того, символьные потоки содержат ряд методов, очень удобных при работе с текстом. Например, метод, позволяющий прочитать из текстового файла одну строку. Этот метод сам поймет, какая у строки длина и вернет нам прочитанную строку в переменную типа String.

FileReader и FileWriter

Пожалуй, самым удобным способом читать текстовые файлы является использование класса FileReader. Правда, при этом необходимо учитывать, что этот класс будет читать файл только в вашей текущей кодировке

и изменить ее не сможет. Как изменить кодировку, мы посмотрим в следующем примере. Давайте сейчас прочитаем какой-нибудь текстовый файл и создадим его копию, записав в нее только строки с нечетными номерами.

Предположим, у нас есть файл lines.txt с таким содержимым:

```
First line.
This is the second line in the file.
Here is the third line.
And the fourth one.
And this is the fifth.
```

Пример 1:

```
FileReader fr = null;
FileWriter fw = null;

try {
    fr = new FileReader("lines.txt");
    fw = new FileWriter("lines1.txt");
    BufferedReader br = new BufferedReader(fr);
    String line="";
    int lineCounter=0;
    while((line = br.readLine()) != null) {
        if( (lineCounter++) % 2 == 0) {
            System.out.println(line);
            fw.write(line+System.getProperty("line.
            separator"));
        }
    }
} catch (FileNotFoundException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
```

```
} catch (IOException ex) {
    Logger.getLogger(Filetest.class.getName()).
           log(Level.SEVERE, null, ex);
} finally {
    try {
        fr.close();
        fw.close();
    } catch (IOException ex) {
        Logger.getLogger(Filetest.class.getName()).
            log(Level.SEVERE, null, ex);
    }
}
```

В этом примере вам все уже знакомо. Создаем два потока для чтения и записи. Связываем входной поток с существующим файлом "lines.txt", а выходному потоку указываем имя файла "lines1.txt", который должен быть создан. Оборачиваем входной поток потоком BufferedReader для создания буферизации. Самое интересное происходит в цикле, где выполняется чтение. Мы вызываем метод readLine() который за один вызов считывает из файла текущую строку, возвращает ее в переменную line и перемещает указатель в файле на следующую строку. Этот метод мы получили из потока BufferedReader. Когда все строки файла будут прочитаны, readLine() занесет в line значение null и цикл остановится. Выполните приложение. Вывод этого кода в выходной файл "lines1.txt" будет таким:

```
First line.
Here is the third line.
And this is the fifth.
```

Запустите наше приложение несколько раз подряд, а затем откройте выходной файл "lines1.txt". Его содержимое будет таким же, как и после первого запуска. Это значит, что при каждом запуске кода этот файл перезаписывается. А можно ли сделать так, чтобы при повторных запусках приложения выполнялось дозаписывание в файл? Да, это сделать очень просто. Добавьте в строке создания потока fw второй параметр со значением true:

```
fw = new FileWriter("lines1.txt", true);
```

Выполните приложение и убедитесь, что теперь новые строки добавляются в файл "lines1.txt".

В этом примере мы работали целиком со строками. Однако FileReader и FileWriter позволяют работать с символами на "низком уровне". Давайте немного изменим код записи нашего файла. Будем проверять все символы в строке и заменять "e" на "E".

```
char buffer[] = new char[line.length()];
line.getChars(0, line.length(), buffer, 0);
for (int i=0; i < buffer.length; i ++) {
    if(buffer[i]=='e')
        fw.write('E');
    else
        fw.write(buffer[i]);
    }
    fw.write(System.getProperty("line.separator"));
```

В этом примере мы преобразовываем прочитанную из файла строку в символьный массив, затем в цикле просматриваем все символы в созданном массиве и выполняем необходимую обработку. Мы снова вызываем для

записи метод write(), но это уже не тот write(), который мы использовали перед этим. Тогда мы передавали этому методу строку, а сейчас – char.

Вывод этого кода в выходной файл "lines1.txt" будет таким:

```
First linE.
HErE is thE third linE.
And this is thE fifth.
```

Конечно, в этом примере мало прикладного смысла, но много пользы с методологической точки зрения.

InputStreamReader и OutputStreamWriter

Перейдем к рассмотрению потоков, которые позволяют работать с текстовыми файлами в символьном режиме и при этом управлять кодировкой. Вы уже понимаете, что тип String в Java предназначен для работы с текстом, а тип byte[] – для работы с бинарным контентом. Тот факт, что мы с вами читали текстовые файлы с помощью байтовых потоков, не должен вводить вас в заблуждение. Байты – это не текст. Но байты могут представлять собой символы текста в какой-нибудь кодировке. И для того, чтобы правильно получить из байт то, что они представляют, надо уметь работать с кодировками. Все случаи, рассмотренные нами до сих пор, использовали кодировку по умолчанию. У меня это – UTF–8. У большинства из вас, я полагаю – тоже. Проделайте такой эксперимент. Сохраните входной файл "lines.txt" в кодировке, отличной от UTF–8. Например – в ANSI. И еще занесите в этот файл пару строк не на английском языке. Выполните наше

приложение еще раз и посмотрите, что будет выведено в консольное окно и что будет записано в выходной файл "lines1.txt". Вы увидите и там и там нечитаемые символы. Почему это произошло? Потому, что потоки FileReader и FileWriter считают, что все символы закодированы в UTF–8, а на самом деле у нас сейчас символы в другой кодировке. Прочитать символы, записанные в кодировке, отличной от кодировки по умолчанию поможет поток InputStreamReader. А записать символы в другой кодировке поможет поток OutputStreamWriter. Эти потоки превращают байтовый ввод-вывод в символьный, позволяя при этом указывать требуемую кодировку.

Пример 2:

```
try (FileInputStream fis =
         new FileInputStream(new File("lines.txt"));
InputStreamReader reader =
         new InputStreamReader(fis, "windows-1251");
FileOutputStream fs =
         new FileOutputStream(new File("lines2.txt"));
OutputStreamWriter writer =
         new OutputStreamWriter(fs, "UTF-8")) {
    int c;
    while ((c = reader.read()) != -1) {
        System.out.print((char) c);
        writer.write(c);
    }
} catch (FileNotFoundException ex) {
    Logger.getLogger(Filetest.class.getName()).
         log(Level.SEVERE, null, ex);
} catch (IOException ex) {
    Logger.getLogger(Filetest.class.getName()).
         log(Level.SEVERE, null, ex);
}
```

Обратите внимание, как создается поток InputStream-Reader. Он создается как обертка вокруг байтового потока FileInputStream, но при создании мы указываем конструктору InputStreamReader требуемую нам кодировку. Я указал "windows-1251" потому, что это кодовая страница для кириллицы, которую я использовал в файле "lines.txt". Если вы использовали другую кодировку – укажите имя кодовой страницы для нее. Например, для английского языка в ANSI кодировке имя кодовой страницы будет "windows-1252". В приведенном примере мы читаем символы в ANSI, а записываем их уже в UTF–8. Запустите этот пример и убедитесь, что и в консольном окне и в выходном файле "lines2.txt" будут читабельные символы, а сам этот файл будет в кодировке UTF–8.

В предыдущем примере мы читали входной файл и записывали выходной по одному символу. Однако, мы могли бы воспользоваться буферизацией и получить за это возможность читать и писать построчно. Другими словами, мы можем создать поток BufferedReader вокруг потока InputStreamReader и получить в свое распоряжение метод readLine(). Подобным образом мы можем создать поток BufferedWriter вокруг потока OutputStreamWriter и получить возможность пользоваться методом write(), выполняющим запись строками. Посмотрите на немного измененный код предыдущего примера. Обратите внимание, что правильная обработка кодировок остается.

Пример 3:

```
try (FileInputStream fis =
          new FileInputStream(new File("lines.txt"));
```

```
    InputStreamReader reader =
          new InputStreamReader(fis, "windows-1251");
    BufferedReader br=new BufferedReader(reader);
    FileOutputStream fs =
          new FileOutputStream(new File("lines2.txt"));
    OutputStreamWriter writer =
          new OutputStreamWriter(fs, "UTF-8");
    BufferedWriter bw=new BufferedWriter(writer)) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
        bw.write(line+System.getProperty("line.
                separator"));
    }
} catch (FileNotFoundException ex) {
    Logger.getLogger(Filetest.class.getName()).
          log(Level.SEVERE, null, ex);
} catch (IOException ex) {
    Logger.getLogger(Filetest.class.getName()).
          log(Level.SEVERE, null, ex);
}
```

System.in, System.out и System.error

В java существует три потока, которые создаются автоматически при запуске JVM. Другими словами, вам не надо в своих программах создавать объекты этих потоков явно, они уже созданы в классе System. Класс System является final классом и в нем определены интересующие нас три статических поля: in, out и err.

```
public final class System {
    static PrintStream out;
    static PrintStream err;
```

```
    static InputStream in;
    ...
}
```

Каждый из этих потоков выполняет предопределенную роль:

- ***System.in*** предназначен для ввода в приложение данных, заносимых с клавиатуры;
- ***System.out*** предназначен для вывода данных из приложения в консольное окно;
- ***System.err*** предназначен для вывода данных об ошибках в консольное окно. Некоторые среды разработки, как например Eclipse, выводят сообщения этого потока красным цветом.

Несмотря на специфику этих потоков, поскольку они также являются объектами потоковых классов, то могут взаимодействовать с другими потоками, оборачиваться ими и т.п. Рассмотрим несколько примеров.

Ввод данных с клавиатуры в String:

```
BufferedReader inp = new BufferedReader(new
                     InputStreamReader(System.in));
String line = inp.readLine();
```

Создаем обертку InputStreamReader вокруг System.in, а затем вокруг этого, добавляем еще обертку BufferedReader, чтобы иметь возможность работать со строками. Теперь, символы, вводимые с клавиатуры, будут заноситься в строку line. В этом примере мы используем для чтения метод readLine(), который позволяет читать сразу вводимую

строку. Еще можно использовать метод read(), который считывает только один введенный с клавиатуры символ и возвращает int код этого символа.

Вывод в консольное окно потоками System.out и System.err вы уже видели много раз. В указанном примере мы используем вывод в out при нормальном выполнении кода, и вывод в err – при возникновении исключения:

```
try {
    InputStream input = new FileInputStream("users.txt");
    System.out.println("File opened successfully!");
} catch (IOException e){
    System.err.println("File opening error:");
    e.printStackTrace();
}
```

Как вы видите, поток out связан с консольным окном, он инициализируется так по умолчанию. Однако, при необходимости out можно перенаправить методом setOut(), например в файл:

```
try {
    System.setOut(new PrintStream(
            new FileOutputStream("out.txt")));
    System.out.println("The output is redirected into
                        file now!");
} catch(Exception e)
{
    System.err.println("File opening error:");
    e.printStackTrace();
}
```

Теперь метод println() будет направлять свой вывод не в консольное окно, а в указанный файл.

2. Сериализация объектов

Сериализация – это процесс превращения объекта какого-либо класса в байтовый поток. Этот поток, помимо данных объекта, будет содержать информацию о типе объекта и о типе данных, хранящихся в объекте. Объект, превращенный в поток, можно записать в файл, передать по сети, занести в таблицу БД. Затем, сериализованный объект можно будет извлечь из потока. Процесс превращения потока в объект в оперативной памяти, называется десериализацией. Необходимо отметить, что и сериализация и десериализация не независят от JVM. Это значит, что можно сериализовать объект на одной платформе, а затем десериализовать его на другой платформе.

В Java существует два способа выполнения сериализации: использование интерфейса Seriallizable и использование интерфейса Externalizable.

Интерфейс Serializable

Рассмотрим первый способ сериализации. Если вы хотите иметь возможность сериализовать объекты какого-либо класса, вы должны наследовать этот класс от интерфейса–маркера Serializable. Этот интерфейс не описывает никаких методов, поэтому дополнительно реализовывать в классе ничего не надо.

Полное описание этого интерфейса можете посмотреть на официальном сайте по адресу: http://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html.

Для выполнения сериализации и десериализации в Java используются потоки ObjectInputStream и Object

OutputStream. Посмотрим, как это работает. Создадим два метода. Один будет выполнять сериализацию. Этот метод будет получать в качестве параметра объект, который надо сериализировать и имя файла, в который надо записать сериализованный объект. Второй метод будет получать имя файла, содержащего сериализованный объект, и возвращать объект, десериализованный из этого файла.

```
public static void serialize(Object obj,
            String fileName) {
    FileOutputStream fos = null;
    try {
       fos = new FileOutputStream(fileName);
       ObjectOutputStream oos =
             new ObjectOutputStream(fos);
       oos.writeObject(obj);
       fos.close();
    } catch (FileNotFoundException ex) {
       System.err.println("File not found:");
       ex.printStackTrace();
    } catch (IOException ex) {
       System.err.println("Input/Output error:");
       ex.printStackTrace();
    } finally {
       try {
           fos.close();
       } catch (IOException ex) {
           System.err.println("Input/Output error:");
           ex.printStackTrace();
       }
    }
}
```

Логика работы этого метода для сериализации проста. Создается символьный поток для вывода в файл. Вокруг этого потока создается поток ObjectOutputStream. Затем выполняется запись в поток переданного объекта. Метод универсальный и будет работать с объектами любого типа. Запись выполняется методом writeObject(), который мы рассмотрели выше.

```
public static Object deserialize(String fileName) {
    FileInputStream fis = null;
    Object obj = null;
    try {
        fis = new FileInputStream(fileName);
        ObjectInputStream ois =
             new ObjectInputStream(fis);
        obj = ois.readObject();
        ois.close();
    } catch (FileNotFoundException ex) {
        System.err.println("File not found:");
        ex.printStackTrace();
    } catch (IOException ex) {
        System.err.println("Input/Output error:");
        ex.printStackTrace();
    } catch (ClassNotFoundException ex) {
        System.err.println("Class not found:");
        ex.printStackTrace();
    } finally {
        try {
             fis.close();
        } catch (IOException ex) {
             System.err.println("Input/Output error:");
             ex.printStackTrace();
        }
    }
    return obj;
}
```

Метод для десериализации выполняет обратную работу. Создает символьный поток для чтения файла, оборачивает этот поток потоком ObjectInputStream, и выполняет чтение методом readObject().

Использовать эти методы можно таким образом:

```
Fish f = new Fish("salmon",2.5,180);
    serialize(f,"fish.txt");
    Fish ff=(Fish)deserialize("fish.txt");
    System.out.println(ff);
```

Вывод этого кода будет таким:

```
salmon weight:2.5  price:180.0
```

Это значит, что в переменной ff находится объект класса Fish.

Иногда в литературе в связи с сериализаций можно встретить термин персистентность. Что это такое? Обычно, объект существует в оперативной памяти максимум до тех пор, пока выполняется приложение. При завершении приложения мусорщик удаляет в оперативной памяти все объекты. Так вот, сериализация позволяет сохранить объект в файл, а затем позже десериализовать его и продолжить работу с объектом. Таким образом, жизненный цикл объекта уже не ограничивается временем выполнения приложения. Это и есть персистентность – продление времени жизни объекта за пределы времени выполнения приложения.

Как видите, в самой сериализации нет ничего сложного. Однако, необходимо учитывать некоторые детали.

Сериализовать можно только объекты классов, которые наследуют интерфейсу Serializable. При этом, если в вашем классе есть поля типа других классов, они тоже должны наследовать интерфейс Serializable, чтобы сериализация выполнилась успешно. А что произойдет, если в классе будут поля несериазлизуемого типа? Например, если бы в нашем классе Fish, было поле типа Thread, то при попытке сериализовать объект такого класса, мы бы получили исключение NotSerializableException. Это случилось бы потому, что класс Thread не наследует интерфейсу Serializable и, следовательно, не может быть сериализован.

В Java большинство классов поддерживают сериализацию, хотя есть и несериализуемые классы. Как быть в ситуации, когда вам надо сериализовать объект класса, а у этого объекта есть несериализуемое поле? В этом случае, такое несериализуемое поле можно обозначить спецификатором transient. Это приведет к тому, что объект будет сериализован, но без этого поля. Этот же спецификатор можно использовать возле любых полей, которые вы не хотите включать в сериализацию. Например, если бы мы не хотели сериализовать цену рыб, надо было бы в классе пометить поле price спецификатором transient:

```
public class Fish implements Serializable
{
    private String name;
    private double weight;
    transient private double price;
    ...
}
```

Сериализация выполнилась бы успешно, но при десериализации поле price было бы равно нулю, т.е. инициализировано значением по умолчанию для своего типа.

Если вам надо сериализовать несколько объектов за один раз, вы можете занести эти объекты в сериализуемую коллекцию и сериализовать коллекцию целиком. Затем десериализовать ее и вытащить оттуда все элементы по одному.

Мы с вами рассмотрели основы сериализации с использованием интерфейса Serializable. Теперь заглянем вовнутрь и поговорим о том, как все это работает, насколько это эффективно и безопасно. Как известно, The God is in the details. Вот об этих мелочах сейчас и поговорим.

Как выполняется процесс записи объекта в поток? Из объекта с помощью Reflection извлекается каждое поле, проверяется не является ли оно transient и записывается в поток. При десериализации для воссоздаваемого объекта выделяется необходимая память и в нее записываются значения сериализованных полей. Если в объекте существуют transient поля, они записываются в десериализируемый объект со значениями по умолчанию для своего типа. Обратите внимание, что ***при десериализации объекта конструктор не используется.*** Надо еще учесть то, что static поля так же, как и transient, ***не сериализуются***. А вот поля с модификатором final сериализуются, как обычные.

А теперь предположим, что наш класс Fish наследует классу BaseFish и интерфейсу Serializable. А вот родительский класс BaseFish не наследует интерфейсу Serializable. Мы хотим сериализвать объект дочернего класса Fish,

который наследует несериализуемому базовому классу BaseFish. Выполнится ли такая сериализация? Сериализация выполнится в любом случае, но вот при десериализации могут возникнуть ошибки. Дело в том, что при десериализации вызывается конструктор без параметров родительского класса, а конструктор производного класса, как мы отметили ранее – не вызывается. Конструктор без параметров родительского класса должен инициализировать поля родительского класса, переданные в дочерний класс. А поля дочернего класса инициализируются не конструктором, а данными из потока. Поэтому, если у вас в базовом классе BaseFish не будет конструктора без параметров – вы получите ошибку. Если такой конструктор будет – все выполнится успешно. Вспомните об этом, когда будете сериализовать объекты производных классов.

Помните, что такое Java bean? Программисты, изучающие Java после знакомства с другими языками, часто удивляются требованию оформлять свои классы, как Java bean. Я думаю, теперь вы понимаете, что Java bean – это не просто требование стиля. В архитектуре языка Java для классов, оформленных, как Java bean становятся доступными очень многие опции.

Что можно сказать о безопасности данных при сериализации и десериализации? Вы уже должны привыкнуть к тому, что когда создаете объект класса в конструкторе, то выполняете проверку присваиваемых полям данных. Такую же проверку вы выполняете в сеттерах. Одно из ключевых требований ООП – контролируемый доступ к данным класса. Другими словами, вы должны поза-

ботиться о том, чтобы в каком-нибудь объекте класса не оказалось 32 число какого-либо месяца, чтобы не оказалась отрицательная цена у товара или что-то в таком роде. Но сейчас вы понимаете, что есть еще один источник появления объектов – десериализация. И это накладывает на программиста некоторые новые обязанности. Дело в том, что вы не можете быть уверены в правильности данных, из которых создаете десериализованный объект. Сериализованные данные можно намеренно или ненамеренно изменить или повредить. Поэтому возьмите себе за правило: после выполнения десериализации проверяйте правильность полученного объекта и если проверка не выполнилась – выбрасывайте исключение.

Идем дальше. У нас сейчас в файле "fish.txt" хранится сериализованный объект класса Fish. Теперь представим, что после выполнения этой сериализации мы немного изменили класс Fish, например, добавили такой метод для проверки цены:

```
public boolean checkPrice()
{
    return this.price>0;
}
```

Если вы сейчас попытаетесь выполнить десериализацию из файла "fish.txt":

```
Fish ff=(Fish)deserialize("fish.txt");
System.out.println(ff);
```

вы получите такую исключительную ситуацию:

```
may 06, 2017 3:30:31 PM filetest.Filetest deserialize
null
SEVERE: null
java.io.InvalidClassException: filetest.Fish;
local class incompatible: stream classdesc
serialVersionUID = -4492936822364132922,
local class serialVersionUID = 127677858221654826
```

Обратите внимание, мы не изменяли поля класса, просто добавили один метод! Почему же десериализация не выполнилась? Дело в том, что в каждый класс, который наследует Serializable, на этапе компиляции автоматически добавляется такое поле:

```
private static final long serialVersionUID
```

Это поле на основе содержимого класса, создает уникальный хеш идентификатор версии класса. При создании этого идентификатора учитывается все: поля класса, методы класса и даже порядок их объявления в классе. И если вы измените в классе хоть что-нибудь – идентификатор класса изменится и не пройдет проверку при десериализации, выполненной с этим же классом до его изменения! Если вас не устраивает такое поведение при десериализации, можно использовать следующую хитрость: явно объявите в своем классе такое поле и инициализируйте его произвольным значением:

```
private static final long serialVersionUID = 585945749688L;
```

Такое поле не будет изменять свое значение при изменениях класса. В официальной документации Java, начиная с 5 версии дается прямой совет: явно добавлять в класс, наследующий Serializable, поле serialVersionUID. Поэтому у вас есть возможность выбора: оставить контроль версии класса при десериализации или нет.

Интерфейс Externalizable

Мы с вами рассмотрели сериализацию и десериализацию с использованием интерфейса Serializable. Вы видели, что в этом случае все происходит автоматически, и мы не можем прямо вмешиваться в процессы сохранения объекта в поток и его извлечения из потока. Если такая ситуация нас не устраивает и мы хотим полностью управлять процессами сериализации и десериализации, тогда в нашем распоряжении интерфейс Externalizable.

Полное описание этого интерфейса можете посмотреть на официальном сайте по адресу: http://docs.oracle.com/javase/8/docs/api/java/io/Externalizable.html#writeExternal-java.io.ObjectOutput.

В отличии от интерфейса Serializable, в котором не описанио ни одного метода, в Externalizable описано два метода: writeExternal() и readExternal(). При реализации первого метода надо выполнить запись объекта в поток, при реализации второго – чтение объекта из потока. При создании процесса записи и чтения мы можем выполнять любую логику, которую считаем необходимой при сериализации и десериализации объектов нашего класса. Рассмотрим пример, где поработаем с двойником класса

Fish с именем FishEx. Наследуем этот класс от Externalizable и реализуем абстрактные методы:

```
public class FishEx implements Externalizable
{
    private String name;
    private double weight;
    transient private double price;

...

    @Override
    public void writeExternal(ObjectOutput out)
           throws IOException {
        out.writeUTF(this.name);
        out.writeDouble(this.price);
        out.writeDouble(this.weight);
    }

    @Override
    public void readExternal(ObjectInput in)
          throws IOException,
    ClassNotFoundException {
        this.name=in.readUTF();
        this.price=in.readDouble();
        this.weight=in.readDouble();
    }
}
```

Обратите внимание, что чтение объекта из потока в методе readExternal() должно коррелировать с записью этого объекта в поток в методе writeExternal(). Если вы при чтении, например, перепутаете очередность считывания полей – вы получите неверный объект при десериализации. Теперь создадим два метода, которые будут

выполнять сериализацию объекта нового класса в файл и его десериализацию. Эти методы очень похожи на методы, использованные при рассмотрении интерфейса Serializable:

```
public static void serializeEx(FishEx obj,
              String fileName) {
    FileOutputStream fos = null;
    try {
        fos = new FileOutputStream(fileName);
        ObjectOutputStream oos =
               new ObjectOutputStream(fos);
        obj.writeExternal(oos);
        fos.close();
    } catch (FileNotFoundException ex) {
        System.err.println("File not found:");
        ex.printStackTrace();
    } catch (IOException ex) {
        System.err.println("Input/Output error:");
        ex.printStackTrace();
    } finally {
        try {
            fos.close();
        } catch (IOException ex) {
            System.err.println("Input/Output error:");
            ex.printStackTrace();
        }
    }
}

public static Object deserializeEx(String fileName) {
    FileInputStream fis = null;
    FishEx obj = null;
    try {
        fis = new FileInputStream(fileName);
        ObjectInputStream ois = new ObjectInputStream(fis);
        obj.readExternal(ois);
```

```
            ois.close();
        } catch (FileNotFoundException ex) {
            System.err.println("File not found:");
            ex.printStackTrace();
        } catch (IOException ex) {
            System.err.println("Input/Output error:");
            ex.printStackTrace();
        } catch (ClassNotFoundException ex) {
            System.err.println("Class not found:");
            ex.printStackTrace();
        } finally {
            try {
                fis.close();
            } catch (IOException ex) {
                System.err.println("Input/Output error:");
                ex.printStackTrace();
            }
        }
        return obj;
}
```

Теперь проверим, как все это работает:

```
FishEx f = new FishEx("salmon",2.5,180);
serialize(f,"fishex.txt");
FishEx fe=(FishEx)deserialize("fishex.txt");
System.out.println(fe);
```

При выполнении этого кода вы должны получить ошибку. Дело в том, что в отличии от десериализации с использованием интерфейса Serializable, при использовании интерфейса Externalizable десериализируемый объект создается конструктором по умолчанию. А уже затем, созданный этим конструктором объект, вызывает

метод readExternal() и заполняет свои поля данными из потока. Поэтому, ***в каждом классе, который реализует интерфейс Externalizable, должен быть конструктор без параметров***. Даже больше – такой конструктор должен быть у всех потомков этого класса, поскольку они тоже наследуют интерфейсу Externalizable.

Добавим в класс FishEx такой конструктор:

```
public FishEx()
{
    this.name="noname";
    this.weight=0;
    this.price=0;
}
```

и снова попробуем выполнить наш код. В этот раз все успешно выполнилось и мы увидели в консольном окне десериализованный объект:

```
salmon  weight:2.5  price:180.0
```

И еще одно. Я думаю, вы понимаете, что спецификатор transient при использовании Externalizable является лишним, поскольку вы сами решаете, какое поле сериализовать, а какое нет.