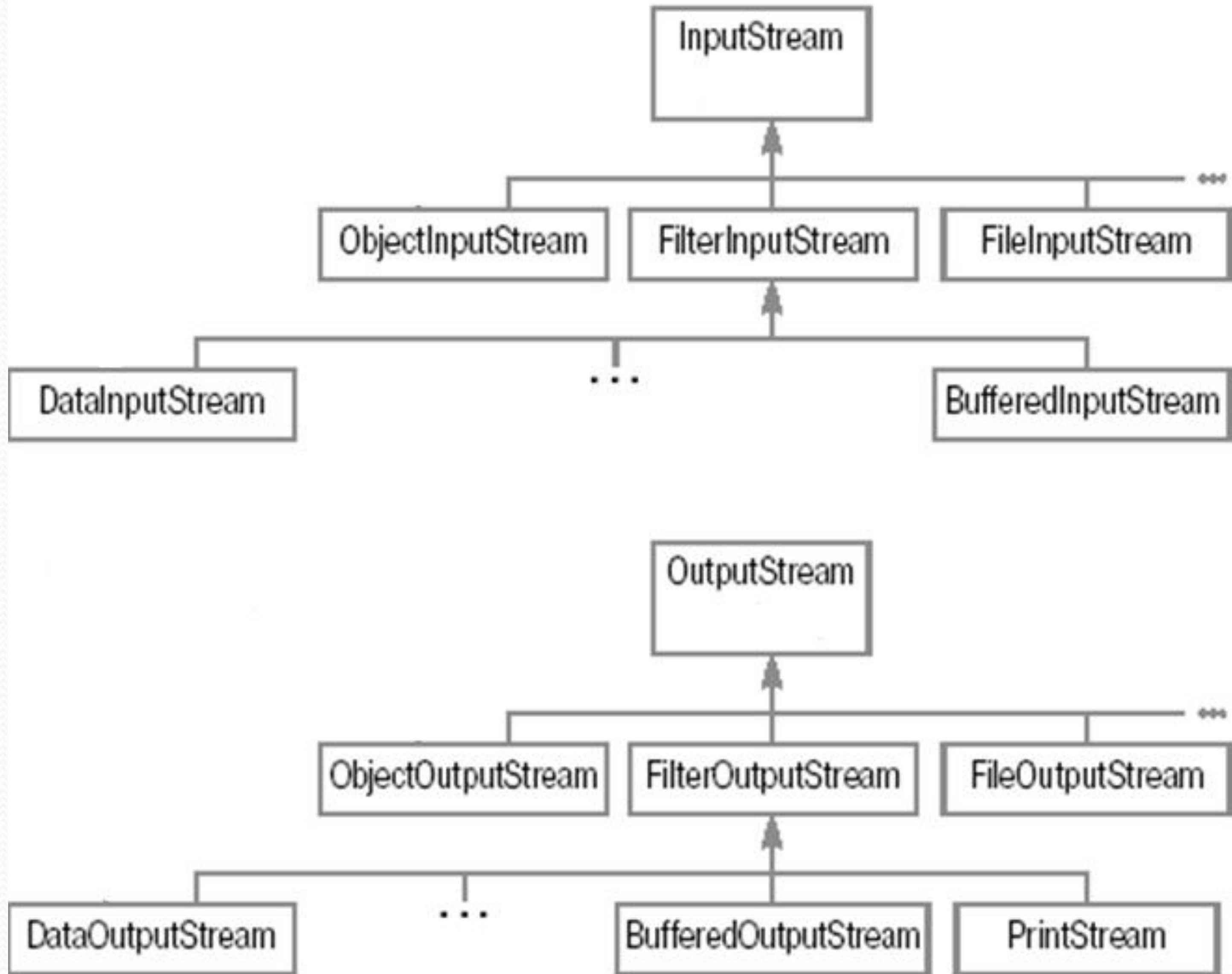
Программирование

Тема 6.2 Работа с файлами в Java

Классы байтовых потоков

- **Фрагмент иерархии классов байтовых потоков**



Классы байтовых потоков

- В Java так же как и в C++ для описания работы по вводу/выводу используется специальное понятие **поток данных (stream).** Поток данных связан с некоторым источником, или приемником, данных, способным получать или предоставлять информацию. Соответственно, потоки делятся на входящие – читающие данные и выходящие – передающие (записывающие) данные. Введение концепции stream позволяет отделить основную логику программы, обменивающейся информацией с любыми устройствами одинаковым образом, от низкоуровневых операций с такими устройствами ввода/вывода.
- В Java *потоки* естественным образом представляются объектами. Описывающие их классы как раз и составляют основную часть пакета java.io. Все классы разделены на две части – одни осуществляют ввод данных, другие – вывод.
- На рисунке слайда представлен фрагмент иерархии классов байтовых потоков. Классы байтовых потоков используются для работы с бинарными файлами.
- Классы входных байтовых *потоков* наследуются от InputStream, а выходных – от OutputStream. Базовые, наиболее универсальные, классы позволяют считывать и записывать информацию в виде набора байт. Чтобы их было удобно применять в различных задачах, java.io содержит также классы, преобразующие любые данные в набор байт.
- Например, если нужно сохранить результаты вычислений – набор значений типа double – в файл, то их можно сначала превратить в набор байт, а затем эти байты записать в файл. Аналогичные действия совершаются и в ситуации, когда требуется сохранить объект (т.е. его состояние) – преобразование в набор байт и последующая их запись в файл. Понятно, что при восстановлении данных в обоих рассмотренных случаях проделываются обратные действия – сначала считывается последовательность байт, а затем она преобразуется в нужный формат.

Классы InputStream и OutputStream

- **Методы класса InputStream:**
  - `int read()`
  - `int read(byte[] buf)`
  - `int read(byte[] buf, int offset, int len)`
  - `available()`
  - `close()`

- **Методы класса OutputStream:**
  - `void write(int)`
  - `void write(byte[] buf)`
  - `void write(byte[] buf, int offset, int len)`
  - `flush()`
  - `close()`

Классы InputStream и OutputStream

- InputStream – это базовый класс для байтовых *потоков* ввода. Методы этого класса необходимы всем классам, которые наследуются от InputStream.
- Метод read() является абстрактным и, поэтому определяется в классах-наследниках. Этот метод предназначен для считывания ровно одного байта из *потока*, однако возвращает при этом значение типа int.. Если достигнут конец *потока*, то есть в нем больше нет информации для чтения, то возвращаемое значение равно -1. Если же считать из *потока* данные не удается из-за каких-то ошибок, или сбоев, будет брошено исключение java.io.IOException. Этот класс наследуется от Exception, т.е. его всегда необходимо обрабатывать явно.
- На практике обычно приходится считывать не один, а сразу несколько байт – то есть массив байт. Для этого используется метод read(), где в качестве параметров передается массив byte[]. Возможна ситуация, когда в *потоке* байт содержится меньше, чем длина массива. Поэтому метод возвращает значение int, указывающее, сколько байт было реально считано. Понятно, что это значение может быть от 0 до величины длины переданного массива.
- Если же мы изначально хотим заполнить не весь массив, а только его часть, то для этих целей используется метод read(), которому, кроме массива byte[], передаются еще два int значения. Первое – это позиция в массиве, с которой следует начать заполнение, второе – количество байт, которое нужно считать.
- Метод available( ) возвращает значение типа int, которое показывает, сколько байт в *потоке* готово к считыванию. При этом не стоит путать количество байт, готовых к считыванию, с тем количеством байт, которые вообще можно будет считать из этого *потока*. Метод available() возвращает число – количество байт, именно на данный момент готовых к считыванию.
- Когда работа с входным *потоком данных* окончена, его следует закрыть. Для этого вызывается метод close(). Этим вызовом будут освобождены все системные ресурсы, связанные с *потоком*.

Классы InputStream и OutputStream

- Класс OutputStream – это базовый класс для байтовых *потоков* вывода.
- В классе OutputStream аналогичным образом определяются три метода write() – один принимающий в качестве параметра int, второй – byte[] и третий –byte[], плюс два int -числа. Все эти методы ничего не возвращают (void).
- Метод write(int) является абстрактным и поэтому реализован в классах-наследниках. Этот метод принимает в качестве параметра int, но реально записывает в *поток* только byte – младшие 8 бит в двоичном представлении. Остальные 24 бита будут проигнорированы. В случае возникновения ошибки этот метод бросает java.io.IOException, как, впрочем, и большинство методов, связанных с вводом-выводом.
- Для записи в *поток* сразу некоторого количества байт методу write() передается массив байт. Или, если мы хотим записать только часть массива, то передаем массив byte[] и два int -числа – отступ и количество байт для записи. Понятно, что если указать неверные параметры – например, отрицательный отступ, отрицательное количество байт для записи, либо если сумма отступ плюс длина будет больше длины массива, – во всех этих случаях кидается исключение IndexOutOfBoundsException.
- Класс выходного *потока* может использовать некоторый внутренний механизм для буферизации (временного хранения перед отправкой) данных. Чтобы убедиться, что данные записаны в *поток*, а не хранятся в буфере, вызывается метод flush(), определенный в OutputStream. В этом классе его реализация пустая, но если какой-либо из наследников использует буферизацию данных, то этот метод должен быть в нем переопределен.
- Когда работа с *потоком* закончена, его следует закрыть. Для этого вызывается метод close(). Этот метод сначала освобождает буфер (вызовом метода flush), после чего *поток* закрывается и освобождаются все связанные с ним системные ресурсы. Закрытый *поток* не может выполнять операции вывода и не может быть открыт заново. В классе OutputStream реализация метода close() не производит никаких действий.
- Итак, классы InputStream и OutputStream определяют необходимые методы для работы с байтовыми *потоками данных*. Эти классы являются абстрактными. Их задача – определить общий интерфейс для классов, которые получают данные из различных источников. Такими источниками могут быть, например, массив байт, файл, строка и т.д.

Классы FileInputStream и FileOutputStream

Пример 1:

```
package primer;
import java.io.*;
class Primer
{ public static void main(String[] args)
  {  byte[] bytesToWrite = {1, 2, 3};
     byte[] bytesReaded = new byte[10];
     String fileName = "E:\\test.txt";
     try
     { // Создать выходной поток
       FileOutputStream outFile = new FileOutputStream(fileName);
       System.out.println("Файл открыт для записи");
       // Записать массив
       outFile.write(bytesToWrite);
       System.out.println("Записано: " + bytesToWrite.length + " байт");
       // По окончании использования должен быть закрыт
       outFile.close();
       System.out.println("Выходной поток закрыт");
```

Классы FileInputStream и FileOutputStream

- **Пример 1:**

```
    FileInputStream inFile = new FileInputStream(fileName);
    System.out.println("Файл открыт для чтения");
    // Узнать, сколько байт готово к считыванию
    int bytesAvailable = inFile.available();
    System.out.println("Готово к считыванию: " + bytesAvailable + " байт");
   // Считать в массив
   int count = inFile.read(bytesReaded,0,bytesAvailable);
  System.out.println("Считано: " + count + " байт");
  for (int i=0;i<count;i++)
     System.out.print(bytesReaded[i]+",");
  System.out.println();
  inFile.close();
  System.out.println("Входной поток закрыт");
}
```

Классы FileInputStream и FileOutputStream

Пример 1:

```
    catch (FileNotFoundException e)
      { System.out.println("Невозможно произвести запись в файл: " + fileName); }
    catch (IOException e)
      { System.out.println("Ошибка ввода/вывода: " + e.toString()); }
  }
}
```

```
run:
Файл открыт для записи
Записано: 3 байт
Выходной поток закрыт
Файл открыт для чтения
Готово к считыванию: 3 байт
Считано: 3 байт
1,2,3,
Входной поток закрыт
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общ
```

Классы FileInputStream и FileOutputStream

- Класс FileInputStream используется для чтения данных из файла. Конструктор такого класса в качестве параметра принимает название файла, из которого будет производиться считывание. При указании строки имени файла нужно учитывать, что она будет напрямую передана операционной системе, поэтому формат имени файла и пути к нему может различаться на разных платформах. Если при вызове этого конструктора передать строку, указывающую на несуществующий файл или каталог, то будет брошено java.io.FileNotFoundException. Если же объект успешно создан, то при вызове его методов read() возвращаемые значения будут считываться из указанного файла.
- Для записи байт в файл используется класс FileOutputStream. При создании объектов этого класса, то есть при вызовах его конструкторов, кроме имени файла, также можно указать, будут ли данные дописываться в конец файла (второй параметр равен true), либо файл будет перезаписан (второй параметр отсутствует, либо равен false). Если указанный файл не существует, то сразу после создания FileOutputStream он будет создан. При вызовах методов write() передаваемые значения будут записываться в этот файл. По окончании работы необходимо вызвать метод close(), чтобы сообщить системе, что работа по записи файла закончена. См. пример на слайде.
- При работе с FileInputStream метод available() практически наверняка вернет длину файла, то есть число байт, сколько вообще из него можно считать.
- В приведенном примере для наглядности закрытие *потоков* производилось сразу же после окончания их использования в основном блоке. Однако лучше закрывать *потоки* в finally блоке.

```
 ... }
 finally {
  try  { inFile.close();  }  catch(IOException e  ){ };  }
```

- Такой подход гарантирует, что *поток* будет закрыт и будут освобождены все связанные с ним системные ресурсы.

Классы BufferedInputStream и BufferedOutputStream

- **Пример 2:**

```
package primer;
import java.io.*;
class Primer
{ public static void main(String[] args)
  {  try
     { String fileName = "E:\\file1";
       InputStream inStream = null;
       OutputStream outStream = null;
       //Записать в файл некоторое количество байт
       long timeStart = System.currentTimeMillis();
       outStream = new FileOutputStream(fileName);
       outStream = new BufferedOutputStream(outStream);
       for(int i=1000000; --i>=0;)
         { outStream.write(i); }
       long time = System.currentTimeMillis() - timeStart;
       System.out.println("Writing time: " + time + " millisec");
       outStream.close();
```

- **Пример 2:**

```
 // Определить время считывания без буферизации
 timeStart = System.currentTimeMillis();
 inStream = new FileInputStream(fileName); while(inStream.read()!=-1){ }
 time = System.currentTimeMillis() - timeStart; inStream.close();
 System.out.println("Direct read time: " + (time) + " millisec");
// Теперь применим буферизацию
timeStart = System.currentTimeMillis();
inStream = new FileInputStream(fileName);
inStream = new BufferedInputStream(inStream);
while(inStream.read()!=-1){ }
time = System.currentTimeMillis() - timeStart;
inStream.close();
System.out.println("Buffered read time: " + (time) + " millisec");
}
```

Классы BufferedInputStream и BufferedOutputStream

- **Пример 2:**

```
    catch (IOException e)
     { System.out.println("IOException: " + e.toString());
       e.printStackTrace();
     }
  }
}
```

```
run:
Writing time: 77 millisec
Direct read time: 2295 millisec
Buffered read time: 39 millisec
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее в
```

Классы BufferedInputStream и BufferedOutputStream

- На практике при считывании с внешних устройств ввод данных почти всегда необходимо буферизировать. Для буферизации данных служат классы надстройки BufferedInputStream и BufferedOutputStream.
- Классы FilterI/OStream являются базовыми для надстроек и определяют общий интерфейс для надстраиваемых объектов. Потоки-надстройки не являются источниками данных. Они лишь модифицируют (расширяют) работу надстраиваемого *потока*.
- BufferedInputStream содержит массив байт, который служит буфером для считываемых данных. То есть когда байты из *потока* считываются либо пропускаются (метод skip()), сначала заполняется буферный массив, причем, из надстраиваемого *потока* загружается сразу много байт, чтобы не требовалось обращаться к нему при каждой операции read или skip.
- BufferedOutputStream предоставляет возможность производить многократную запись небольших блоков данных без обращения к устройству вывода при записи каждого из них. Сначала данные записываются во внутренний буфер. Непосредственное обращение к устройству вывода и, соответственно, запись в него, произойдет, когда буфер заполнится. Инициировать передачу содержимого буфера на устройство вывода можно и явным образом, вызвав метод flush(). Так же буфер освобождается перед закрытием потока. При этом будет закрыт и надстраиваемый поток (так же поступает BufferedInputStream).
- Пример на слайде наглядно демонстрирует повышение скорости считывания данных из файла с использованием буфера. В данном случае не производилось никаких дополнительных вычислений, занимающих процессорное время, только запись и считывание из файла. При этом считывание с использованием буфера заняло в 10 (!) раз меньше времени, чем аналогичное без буферизации. Для более быстрого выполнения программы запись в файл производилась с буферизацией, однако ее влияние на скорость записи нетрудно проверить, убрав из программы строку, создающую BufferedOutputStream.
- Классы BufferedI/OStream добавляют только внутреннюю логику обработки запросов, но не добавляют никаких новых методов.

Классы DataInputStream и DataOutputStream

- **Пример 3:**

```
package primer;
import java.io.*;
class Primer
{ public static void main(String[] args)
  { try
    { String fileName = "E:\\file2";
      InputStream inStream = null;
      OutputStream outStream = null;
      outStream = new FileOutputStream(fileName);
      outStream = new BufferedOutputStream(outStream);
      DataOutputStream outData = new DataOutputStream(outStream);

     outData.writeByte(128);
      // этот метод принимает аргумент int, но записывает
      // лишь младший байт
     outData.writeInt(128);
     outData.writeLong(128);
     outData.writeDouble(128.5);
     outData.close();
```

Классы DataInputStream и DataOutputStream

- **Пример 3:**

```
inStream = new FileInputStream(fileName);
inStream = new BufferedInputStream(inStream);
DataInputStream inData = new DataInputStream(inStream);
System.out.println("Чтение в правильной последовательности: ");
System.out.println("readByte: " + inData.readByte());
System.out.println("readInt: " + inData.readInt());
System.out.println("readLong: " + inData.readLong());
System.out.println("readDouble: " + inData.readDouble());
inData.close();
System.out.println("Чтение в измененной последовательности:");
inStream = new FileInputStream(fileName);
inStream = new BufferedInputStream(inStream);
inData = new DataInputStream(inStream);
System.out.println("readInt: " + inData.readInt());
System.out.println("readDouble: " + inData.readDouble());
System.out.println("readLong: " + inData.readLong());
inData.close();
}
```

Классы DataInputStream и DataOutputStream

- **Пример 3:**

```
  catch (Exception e)
   { System.out.println("Impossible IOException occurs: " + e.toString());
     e.printStackTrace();
   }
 }
}
```

```
run:
Чтение в правильной последовательности:
readByte: -128
readInt: 128
readLong: 128
readDouble: 128.5
Чтение в измененной последовательности:
readInt: -2147483648
readDouble: -0.0
readLong: -9205252016509550592
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0
```

Классы DataInputStream и DataOutputStream

- До сих пор речь шла только о считывании и записи в *поток данных* в виде byte. Для работы с другими *примитивными типами данных* классы-фильтры DataInputStream и DataOutputStream. Их место в иерархии классов ввода/вывода можно увидеть на диаграмме классов второго слайда.
- Классы DataInputStream и DataOutputStream, соответственно, реализуют методы считывания и записи значений всех примитивных типов. При этом происходит конвертация этих данных в набор byte и обратно. Чтение необходимо организовать так, чтобы данные запрашивались в виде тех же типов, в той же последовательности, как и производилась запись. Если записать, например, int и long, а потом считывать их как short, чтение будет выполнено корректно, без исключительных ситуаций, но числа будут получены совсем другие. Это наглядно показано в примере на слайде.

Классы ObjectInputStream и ObjectOutputStream

- **Пример 4:**

```
package primer;
import java.io.*;

class Person implements Serializable
{ private String name;    // фио
  private int age;        // возраст
  public Person() {}
  public Person(String name, int age)
  {this.name = name;
   this.age = age;
  }
  public void setage(int age){this.age = age;}
  public void output()
    { System.out.printf("%-10s  %d\n",name,age);}
  // запись в файл
  void diskOut(ObjectOutputStream oos)throws IOException
  {   oos.writeObject(this);}
  // чтение из файла
  Person  diskIn(ObjectInputStream ois)throws IOException, ClassNotFoundException
  {   Person p =(Person)ois.readObject();
      return p;
  }
 }
```

Классы ObjectInputStream и ObjectOutputStream

- **Пример 4:**

```
class Primer
{ public static void main(String[] args)
  { try
    { Person[] p1 = {new Person("Иванов Иван", 20),
                     new Person("Петров Петр", 20)};
      Person[] p2 = new Person[2];
      String fileName = "E:\\file3";
      FileOutputStream fos = new FileOutputStream(fileName);
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      for(int i=0; i<2;i++)
        p1[i].diskOut(oos);
      oos.close();
      FileInputStream fis = new FileInputStream(fileName);
      ObjectInputStream ois = new ObjectInputStream(fis);
      int k=0;
      while(fis.available() > 0)
       { p2[k]=new Person();
        p2[k]=p2[k].diskIn(ois);
        k++;
       }
      ois.close();
```

Пример 4:

```
        System.out.println("Массив p1:");
        for(int i=0; i<2;i++)
            p1[i].output();
        System.out.println("Массив p2:");
        for(int i=0; i<k;i++)
            p2[i].output();
        p2[1].setage(25);
        System.out.println("Измененный массив p2:");
        for(int i=0; i<k;i++)
            p2[i].output();
        System.out.println("Массив p1:");
        for(int i=0; i<2;i++)
            p1[i].output();
    }
    catch (IOException e)
    { System.out.println(e);}
    catch (ClassNotFoundException e)
    { System.out.println(e);}
  }
}
```

```
run:
Массив p1:
Иванов Иван   20
Петров Петр   20
Массив p2:
Иванов Иван   20
Петров Петр   20
Измененный массив p2:
Иванов Иван   20
Петров Петр   25
Массив p1:
Иванов Иван   20
Петров Петр   20
СБОРКА УСПЕШНО ЗАВЕРШЕН
```

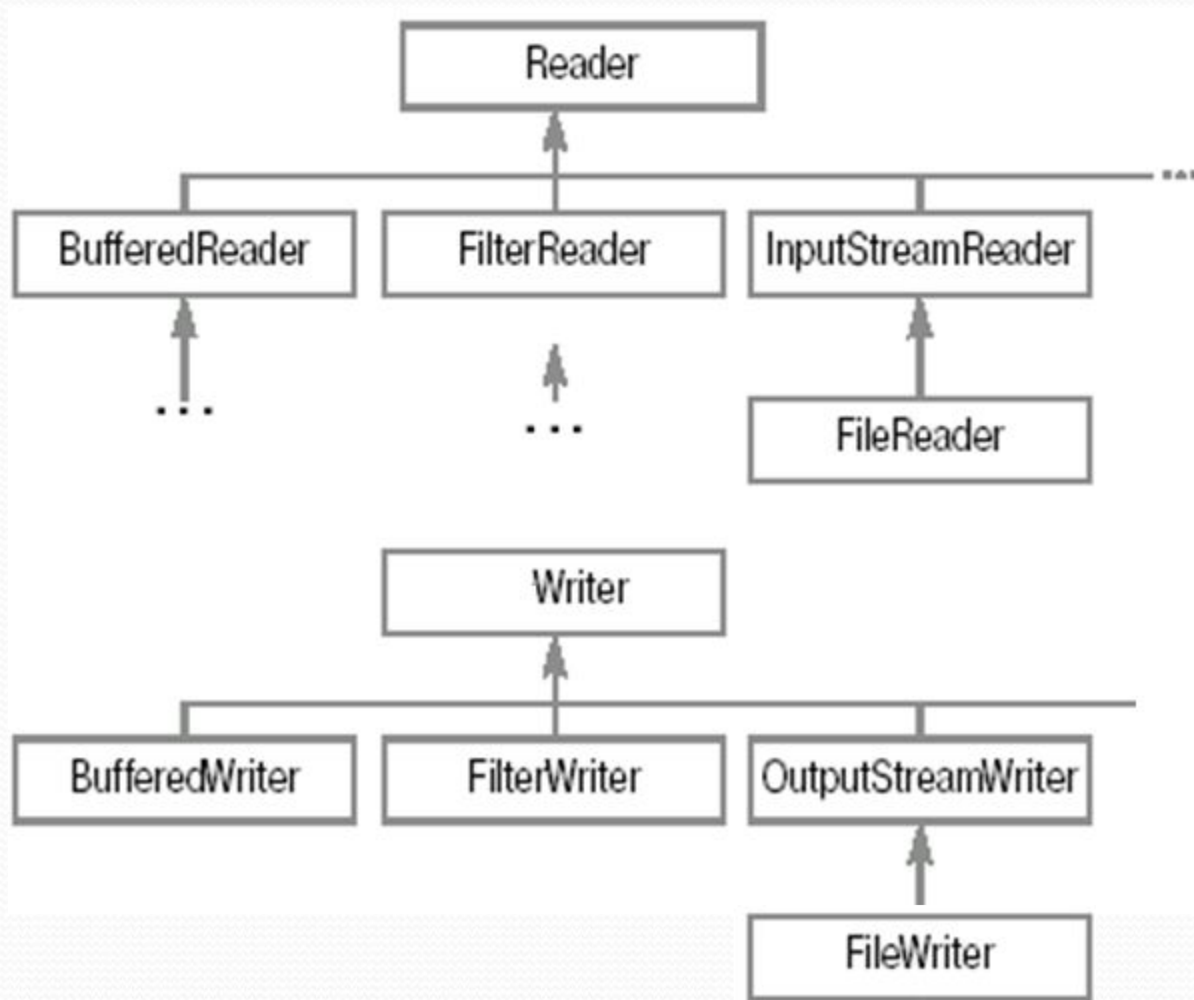
Классы ObjectInputStream и ObjectOutputStream

- Для объектов процесс преобразования в последовательность байт и обратно организован несколько сложнее – объекты имеют различную структуру, хранят ссылки на другие объекты и т.д. Поэтому такая процедура получила специальное название - ***сериализация*** (serialization), обратное действие, – то есть воссоздание объекта из последовательности байт – ***десериализация***.
- Поскольку сериализованный объект – это последовательность байт, которую можно легко сохранить в файл, передать по сети и т.д., то и объект затем можно восстановить на любой машине, вне зависимости от того, где проводилась *сериализация*. Разумеется, Java позволяет не задумываться при этом о таких факторах, как, например, используемая операционная система на машине-отправителе и получателе. Такая гибкость обусловила широкое применение *сериализации* при создании распределенных приложений.
- Для представления объектов в виде последовательности байт определены классы ObjectInputStream и ObjectOutputStream.
- Эти классы используют стандартный механизм *сериализации*, который предлагает JVM. Для того, чтобы объект мог быть сериализован, класс, от которого он порожден, должен реализовывать интерфейс java.io.Serializable. В этом интерфейсе не определен ни один метод. Он нужен лишь для указания, что объекты класса могут участвовать в *сериализации*. При попытке сериализовать объект, не имеющий такого интерфейса, будет брошен java.io.NotSerializableException.
- Чтобы начать *сериализацию* объекта, нужен выходной *поток* OutputStream, в который и будет записываться сгенерированная последовательность байт. Этот *поток* передается в конструктор ObjectOutputStream. Затем вызовом метода writeObject() объект сериализуется и записывается в выходной *поток*. См. пример на слайде.
- Восстановленный объект не совпадает с исходным по имени, но равен сериализованному по значению.
- Рассмотрим основные исключения, которые может генерировать метод readObject() класса ObjectInputStream.

Классы ObjectInputStream и ObjectOutputStream

- Предположим, объект некоторого класса TestClass был сериализован и передан по сети на другую машину для восстановления. Может случиться так, что у считывающей JVM на локальном диске не окажется описания этого класса (файл TestClass.class). Поскольку стандартный механизм *сериализации* записывает в *поток* байт лишь состояние объекта, для успешной *десериализации* необходимо наличие описание класса. В результате будет брошено исключение ClassNotFoundException.
- Причина появления java.io.StreamCorruptedException вполне очевидна из названия – неправильный формат входного *потока*. Предположим, происходит попытка считать сериализованный объект из файла. Если этот файл испорчен, то стандартная процедура *десериализации* даст сбой. Эта же ошибка возникнет, если считать некоторое количество байт (с помощью метода read) непосредственно из надстраиваемого *потока* InputStream. В таком случае ObjectInputStream снова обнаружит сбой в формате данных и будет брошено исключение java.io.StreamCorruptedException.
- Если при считывании будет вызван метод readObject, а в исходном *потоке* следующим на очереди записано значение примитивного типа, будет брошено исключение java.io.OptionalDataException. Очевидно, что для корректного восстановления данных из *потока* их нужно считывать именно в том порядке, в каком были записаны.

- **Фрагмент иерархии классов символьных потоков:**



Классы символьных потоков

- Рассмотренные классы – наследники InputStream и OutputStream – работают с байтовыми данными. Если с их помощью записывать или считывать текст, то сначала необходимо сопоставить каждому символу его числовой код. Такое соответствие называется кодировкой.
- Известно, что Java использует кодировку Unicode, в которой символы представляются двухбайтовым кодом. Байтовые *потоки* зачастую работают с текстом упрощенно – они просто отбрасывают старший байт каждого символа. В реальных же приложениях могут использовать различные кодировки (даже для русского языка их существует несколько). Поэтому в версии Java 1.1 появился дополнительный набор классов, основывающийся на типах Reader и Writer. Их иерархия представлена на слайде.
- Эта иерархия очень схожа с аналогичной для байтовых *потоков* InputStream и OutputStream. Главное отличие между ними – Reader и Writer работают с *потоком* символов (char). Только чтение массива символов в Reader описывается методом read(char[]), а запись в Writer – write(char[]).

Классы символьных потоков

- **Таблица соответствия основных классов для байтовых и символьных потоков:**

Байтовый поток	Символьный поток
InputStream	Reader
OutputStream	Writer
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
Нет аналога	InputStreamReader
Нет аналога	OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
PrintStream	PrintWriter
DataInputStream	Нет аналога
DataOutputStream	Нет аналога
ObjectInputStream	Нет аналога
ObjectOutputStream	Нет аналога

Классы символьных потоков

- В <u>таблице на слайде</u> приведены соответствия классов для байтовых и символьных *потоков*.
- Как видно из таблицы, <u>различия крайне незначительны и предсказуемы</u>.
- Например, конечно же, отсутствует преобразование в символьное представление примитивных типов Java и объектов (DataInput/Output, ObjectInput/Output). <u>Добавлены классы-мосты, преобразующие символьные *потоки* в байтовые: InputStreamReader и OutputStreamWriter</u>. Именно на их основе реализованы FileReader и FileWriter. Метод available() класса InputStream в классе Reader отсутствует, он заменен методом ready(), возвращающим булевое значение, – готов ли *поток* к считыванию (то есть будет ли считывание произведено без блокирования).
- В остальном же использование символьных *потоков* идентично работе с байтовыми *потоками*.
- <u>Классы-мосты InputStreamReader и OutputStreamWriter</u> при преобразовании символов также используют некоторую кодировку. Ее можно задать, передав в конструктор в качестве аргумента ее название. Если оно не будет соответствовать никакой из известных кодировок, будет брошено исключение UnsupportedEncodingException. Вот некоторые из корректных значений этого аргумента (чувствительного к регистру!) для распространенных кодировок: "Cp1251", "UTF-8" и т.д.

Пример 5:

```
package primer;
import java.io.*;

class Primer
{ public static void main(String[] args)
     { String fileName = "E:\\file.txt";
      //Строка, которая будет записана в файл
      String data = "Строка для записи и чтения.\n";
      try
       { FileWriter fw = new FileWriter(fileName);
         BufferedWriter bw = new BufferedWriter(fw);
         System.out.println("Write some data to file: " + fileName);
         // Несколько раз записать строку
        for(int i=(int)(Math.random()*10);--i>=0;)
           bw.write(data);
        bw.close();
```

Пример 5:

```
            // Считываем результат
            FileReader fr = new FileReader(fileName);
            BufferedReader br = new BufferedReader(fr);
            String s = null;
            int count = 0;
            System.out.println("Read data from file: " + fileName);
            // Считывать данные, отображая на экран
           while((s=br.readLine())!=null)
              System.out.println("row " + ++count + " read:" + s);
           br.close();
        }
      catch (IOException e)
       { System.out.println(e);}
    }
}
```

```
run:
Write some data to file: E:\file.txt
Read data from file: E:\file.txt
row 1 read:Строка для записи и чтения.
row 2 read:Строка для записи и чтения.
```

Пример 6: Применение класса Scanner для для расчета среднего арифметического чисел, считываемых из текстового файла

```
package primer;
import java.util.*;
import java.io.*;
class Primer
{ public static void main(String args[]) throws IOException
  { int count = 0;
   double sum = 0.0;
   FileWriter fout = new FileWriter("E:\\Test.txt");
   fout.write("2 3,4 5 6 7,4 9,1 10,5 done");
   fout.close();
   FileReader fin = new FileReader("E:\\Test.txt");
   Scanner src = new Scanner(fin);
```

- **Пример 6:**

```
    while(src.hasNext())
    { if(src.hasNextDouble())
        { sum += src.nextDouble();
          count++;
        }
      else { String str = src.next();
             if(str.equals("done")) break;
               else {
                     System.out.println("File format error.");
                     return;
                    }
           }
    }
fin.close();
System.out.println("Average is " + sum / count);
  }
}
```

```
run:
Average is 6.2
СБОРКА УСПЕШНО ЗАВЕРШЕНА
```

Пример 7: Применение класса Scanner для считывания данных разных типов, хранящихся в текстовом файле

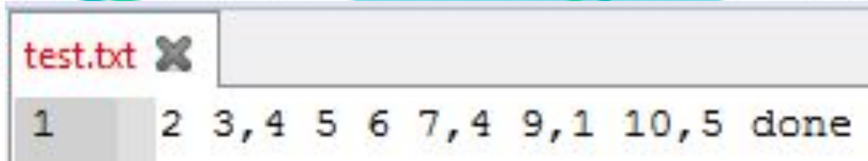
```
package primer;
import java.util.*;
import java.io.*;
class Primer {
public static void main(String args[]) throws IOException
{ int i;
  double d;
  boolean b;
  String str;
  FileWriter fout = new FileWriter("E:\\test.txt");
  fout.write("Testing Scanner 10 12,2 one true two false");
  fout.close();
  FileReader fin = new FileReader("E:\\test.txt");
  Scanner src = new Scanner(fin);
```

- **Пример 7:**

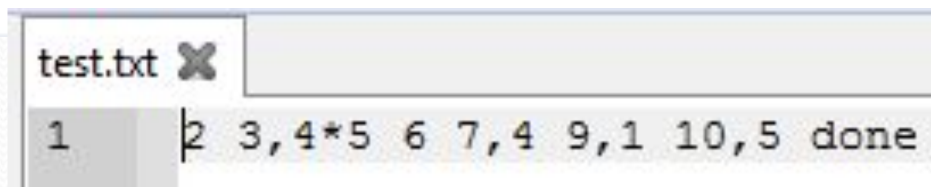
```
while(src.hasNext())
{ if (src.hasNextInt())
   { i = src.nextInt();
     System.out.println("int: " + i);
   }
  else
    if(src.hasNextDouble())
      { d = src.nextDouble();
        System.out.println("double: " + d);
      }
     else if(src.hasNextBoolean())
           { b = src.nextBoolean();
             System.out.println("boolean: " + b);
           }
          else { str = src.next();
                 System.out.println("String: " + str);
               }
 }
 fin.close();
}
}
```

```
run:
String: Testing
String: Scanner
int: 10
double: 12.2
String: one
boolean: true
String: two
boolean: false
```

- **Пример 8:**


test.txt
1 2 3,4 5 6 7,4 9,1 10,5 done

```
package primer;
import java.io.*;
public class Primer
{ public static void main(String[] args)
    { try
      { RandomAccessFile myRAF=new RandomAccessFile("E:\\Test.txt","rw");
        try
         {myRAF.seek(5);
          myRAF.write('*');
          myRAF.close();
         }
         catch(IOException e)
          { System.out.println("Cannot write a char");}
      }
      catch(FileNotFoundException e)
        { System.out.println("File Not Found"); }
  }
}
```


test.txt
1 2 3,4*5 6 7,4 9,1 10,5 done

Контрольные вопросы

1. Классы FileInputStream и FileOutputStream: назначение, основные методы. Пример использования классов.
2. Классы BufferedInputStream и BufferedOutputStream: назначение, основные методы. Пример использования классов.
3. Классы DataInputStream и DataOutputStream: назначение, примеры использования.
4. Классы ObjectInputStream и ObjectOutputStream: назначение, примеры использования.
5. Классы FileReader и FileWriter: назначение, основные методы, примеры использования.
6. Использование класса Scanner для работы с файлами, примеры.