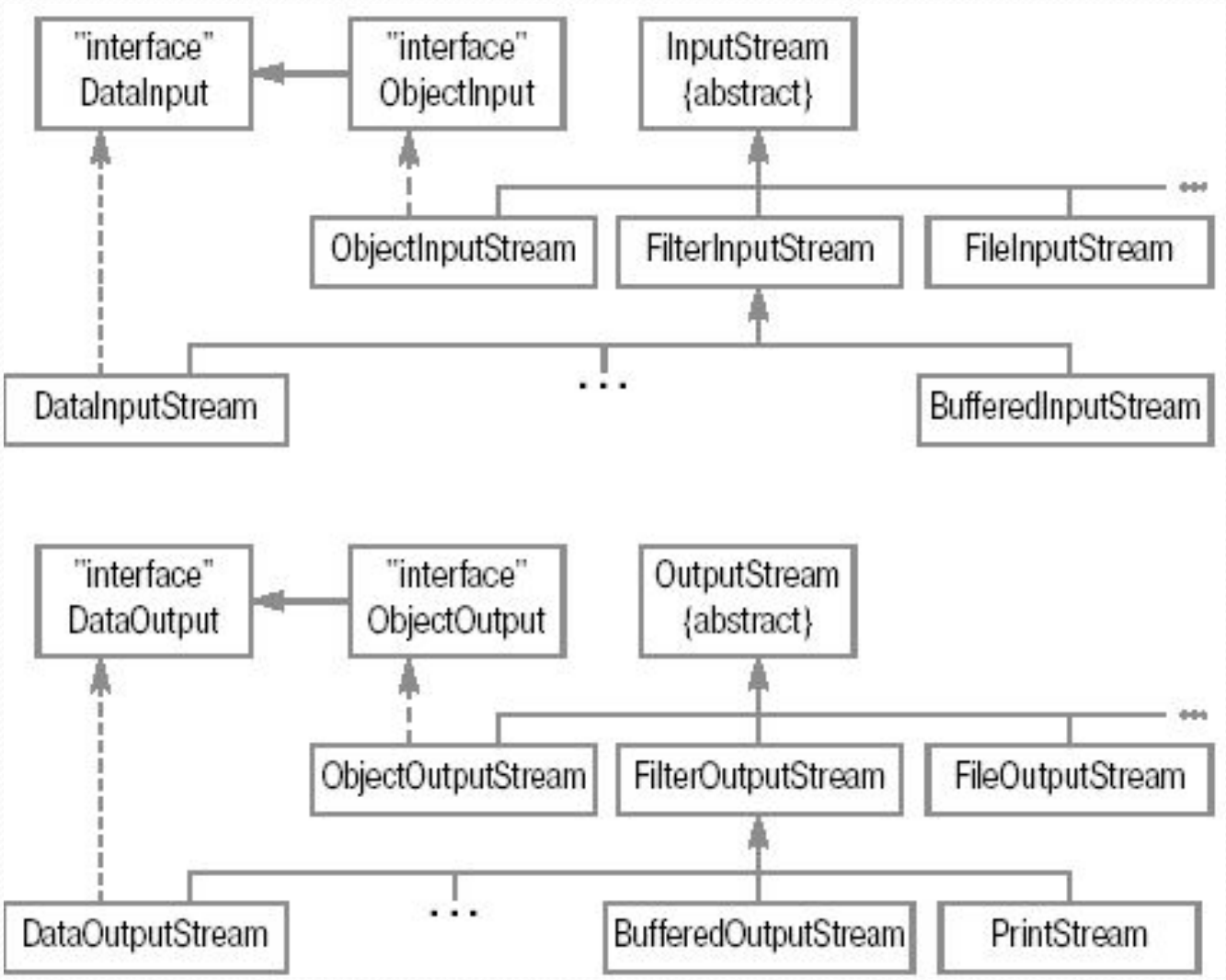
Программирование на Java
Тема 4.2 Потоки ввода-вывода

- **Иерархия классов ввода-вывода**



Система ввода-вывода. Потоки данных

- Подавляющее большинство программ обменивается данными с внешним миром. Так, например, программа может считывать данные с клавиатуры и записывать их в файл, или же наоборот - считывать данные из файла и выводить их на экран. Таким образом, устройства, откуда может производиться считывание информации, могут быть самыми разнообразными – файл, клавиатура, входящее сетевое соединение и т.д. То же касается и устройств вывода – это может быть файл, экран монитора, принтер, исходящее сетевое соединение и т.п. В конечном счете, все данные в компьютерной системе в процессе обработки передаются от устройств ввода к устройствам вывода.
- Обычно часть вычислительной платформы, которая отвечает за обмен данными, так и называется – система ввода/вывода. В Java она представлена пакетом java.io (input/output). Реализация системы ввода/вывода осложняется не только широким спектром источников и получателей данных, но еще и различными форматами передачи информации. Ею можно обмениваться в двоичном представлении, символьном или текстовом, с применением некоторой кодировки. Зачастую для повышения производительности применяется буферизация.
- В Java 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. Они довольно разнообразны и отвечают за различную функциональность. Все классы разделены на две части – одни осуществляют ввод данных, другие – вывод.
- Существующие стандартные классы помогают решить большинство типичных задач. Минимальной "порцией" информации является, как известно, бит, принимающий значение 0 или 1 (это понятие также удобно применять на самом низком уровне, где данные передаются электрическим сигналом; условно говоря, 1 представляется прохождением импульса, 0 – его отсутствием).

Система ввода-вывода. Потоки данных

- Традиционно используется более крупная единица измерения – байт, объединяющая 8 бит. Таким образом, значение, представленное одним байтом, находится в диапазоне от 0 до 2^8-1=255, или, если использовать знак, – от -128 до +127. Примитивный тип byte в Java в точности соответствует последнему – знаковому диапазону.
- Базовые, наиболее универсальные, классы позволяют считывать и записывать информацию именно 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.
- На рисунке слайда представлены иерархии классов ввода/вывода. Как и говорилось, все типы поделены на две группы. Представляющие входные *потоки* классы наследуются от InputStream, а выходные – от OutputStream.

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 *потоков* ввода, т.е. чтения. Соответственно, он описывает базовые методы для работы с байтовыми *потоками данных*. Эти методы необходимы всем классам, которые наследуются от InputStream.
- Метод read() (без аргументов) является абстрактным и, соответственно, должен быть определен в классах-наследниках. Этот метод предназначен для считывания ровно одного байта из *потока*, однако возвращает при этом значение типа int. В том случае, если считывание произошло успешно, возвращаемое значение лежит в диапазоне от 0 до 255 и представляет собой полученный байт (значение int содержит 4 байта и получается простым дополнением нулями в двоичном представлении). Обратите внимание, что полученный таким образом байт не обладает знаком и не находится в диапазоне от -128 до +127, как примитивный тип byte в Java. 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[]. При выполнении этого метода в цикле производится вызов абстрактного метода read() (определенного без параметров) и результатами заполняется переданный массив. Количество байт, считываемое таким образом, равно длине переданного массива. Но при этом может так получиться, что данные в *потоке* закончатся еще до того, как будет заполнен весь массив. То есть возможна ситуация, когда в *потоке данных* (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. Такой подход, когда для получения данных передается массив и два int числа – offset (смещение) и length (длина), является довольно распространенным и часто встречается не только в пакете java.io.

Классы InputStream и OutputStream

- При вызове методов read() возможно возникновение такой ситуации, когда запрашиваемые данные еще не готовы к считыванию. Например, если мы считываем данные, поступающие из сети, и они еще просто не пришли. В таком случае нельзя сказать, что данных больше нет, но и считать тоже нечего - выполнение останавливается на вызове метода read() и получается "зависание".
- Чтобы узнать, сколько байт в *потоке* готово к считыванию, применяется метод available(). Этот метод 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*.
- Точно так же, как InputStream – это базовый класс для *потоков* ввода, класс OutputStream – это базовый класс для *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 должен быть 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 исключениеIndexOutOfBoundsException.

Классы InputStream и OutputStream

- Реализация *потока* может быть такой, что данные записываются не сразу, а хранятся некоторое время в памяти. Например, мы хотим записать в файл какие-то данные, которые получаем порциями по 10 байт, и так 200 раз подряд. В таком случае вместо 200 обращений к файлу удобней будет скопить все эти данные в памяти, а потом одним заходом записать все 2000 байт. То есть 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:

```
byte[] bytesToWrite = {1, 2, 3};
byte[] bytesReaded = new byte[10];
String fileName = "d:\\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

```
// Создать входной поток
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

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

- **Результат работы программы:**

```
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
```

Классы FileInputStream и FileOutputStream

- Рассмотрим классы - реализации потоков данных на примере классов 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. Но не стоит закладываться на это при написании программ, которые должны устойчиво работать на различных платформах,– метод available() возвращает число байт, которое может быть на данный момент считано без блокирования. Тот факт, что, скорее всего, это число и будет длиной файла, является всего лишь частным случаем работы на некоторых платформах.
- В приведенном примере для наглядности закрытие *потоков* производилось сразу же после окончания их использования в основном блоке. Однако лучше закрыват ь*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:

```
try { String fileName = "d:\\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();
```

Классы BufferedInputStream и BufferedOutputStream

```
// 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

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

- Результат работы программы:

```
Writing time: 359 millisec
Direct read time: 6546 millisec
Buffered read time: 250 millisec
```

Классы FilterInputStream и FilterOutputStream и их наследники

- Задачи, возникающие при вводе/выводе весьма разнообразны - это может быть считывание байтов из файлов, объектов из файлов, объектов из массивов, буферизованное считывание строк из массивов и т.д. В такой ситуации решение с использованием простого наследования приводит к возникновению слишком большого числа подклассов. Более эффективно применение надстроек (в ООП этот шаблон называется адаптер) Надстройки – наложение дополнительных объектов для получения новых свойств и функций. Таким образом, необходимо создать несколько дополнительных объектов – адаптеров к классам ввода/вывода. В java.io их еще называют фильтрами. При этом надстройка-фильтр включает в себя интерфейс объекта, на который надстраивается, поэтому может быть, в свою очередь, дополнительно надстроена.
- В java.io интерфейс для таких надстроек ввода/вывода предоставляют классы FilterInputStream (для входных *потоков*) и FilterOutputStream (для выходных *потоков*). Эти классы унаследованы от основных базовых классов ввода/вывода – InputStream и OutputStream, соответственно.
Конструктор FilterInputStream принимает в качестве параметра объек тInputStream и имеет модификатор доступа protected.
- 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*.
- *На практике при считывании с внешних устройств ввод данных почти всегда необходимо буферизировать*. Для буферизации данных служат классы BufferedInputStream и BufferedOutputStream.
- 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. Также класс BufferedInputStream добавляет поддержку методов mark() и reset(). Эти методы определены еще в классе InputStream, но там их реализация по умолчанию бросает исключение IOException. Метод mark() запоминает точку во входном потоке, а вызов метода reset() приводит к тому, что все байты, полученные после последнего вызова mark(), будут считываться повторно, прежде, чем новые байты начнут поступать из надстроенного входного потока.

Классы FilterInputStream и FilterOutputStream и их наследники

- 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

```
try { ByteArrayOutputStream out = new ByteArrayOutputStream();
      DataOutputStream outData = new DataOutputStream(out);
     outData.writeByte(128);
 // этот метод принимает аргумент int, но записывает
 // лишь младший байт
 outData.writeInt(128);
 outData.writeLong(128);
 outData.writeDouble(128);
 outData.close();
 byte[] bytes = out.toByteArray();
 InputStream in = new ByteArrayInputStream(bytes);
 DataInputStream inData = new DataInputStream(in);
 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();
```

Классы DataInputStream и DataOutputStream

```
System.out.println("Чтение в измененной последовательности:");
 in = new ByteArrayInputStream(bytes);
 inData = new DataInputStream(in);
 System.out.println("readInt: " + inData.readInt());
 System.out.println("readDouble: " + inData.readDouble());
 System.out.println("readLong: " + inData.readLong());
 inData.close();
} catch (Exception e) {
      System.out.println("Impossible IOException occurs: " +
  e.toString());
      e.printStackTrace();
}
```

Классы DataInputStream и DataOutputStream

- **Результат выполнения программы:**

```
Чтение в правильной последовательности:
 readByte: -128
 readInt: 128
 readLong: 128
readDouble: 128.0

Чтение в измененной последовательности:
 readInt: -2147483648
 readDouble: -0.0
 readLong: -9205252085229027328
```

Классы FilterInputStream и FilterOutputStream и их наследники

- До сих пор речь шла только о считывании и записи в *поток данных* в виде byte. Для работы с другими *примитивными типами данных* Java определены интерфейсыDataInput и DataOutput и их реализации – классы-фильтры DataInputStream и DataOutputStream. Их место в иерархии классов ввода/вывода можно увидеть на диаграмме классов второго слайда.
- Интерфейсы DataInput и DataOutput определяют, а 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.

- Стандартная сериализация

- Интерфейсы ObjectInput и ObjectOutput
- Классы ObjectInputStream и ObjectOutputStream
- интерфейс java.io.Serializable
- Пример:
- сериализация объекта

```
ByteArrayOutputStream os = new ByteArrayOutputStream();
Object objSave = new Integer(1);
ObjectOutputStream oos = new ObjectOutputStream(os);
oos.writeObject(objSave);
byte[] bArray = os.toByteArray(); // запись в массив
```

- десериализация объекта из массива

```
ByteArrayInputStream is = new ByteArrayInputStream(bArray);
ObjectInputStream ois = new ObjectInputStream(is);
Object objRead = ois.readObject();
```

- Стандартная сериализация

- Интерфейсы ObjectInput и ObjectOutput
- Классы ObjectInputStream и ObjectOutputStream
- интерфейс java.io.Serializable
- Пример:
- Проверка восстановленного объекта на идентичность исходному

```
System.out.println("readed object is: " + objRead.toString());
System.out.println("Object equality is: " +
(objSave.equals(objRead)));
System.out.println("Reference equality is: " +
(objSave==objRead));
```

- Результат выполнения кода:

```
readed object is: 1
Object equality is: true
Reference equality is: false
```

Сериализация объектов

- 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 унаследованные от DataInput и DataOutput интерфейсы ObjectInput и ObjectOutput, соответственно. В java.io имеются реализации этих интерфейсов – классыObjectInputStream 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.
- Как обычно, для упрощения в примере была опущена обработка ошибок. Однако, *сериализация* (*десериализация*) объектов довольно сложная процедура, поэтому возникающие сложности не всегда очевидны. Рассмотрим основные исключения, которые может генерировать метод readObject() класса ObjectInputStream.

Сериализация объектов

- 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 исключени eClassNotFoundException.
- 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.
- Поскольку ObjectOutput наследуется от DataOutput, ObjectOutputStream может быть использован для последовательной записи нескольких значений как объектных, так и примитивных типов в произвольной последовательности. 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.

Сериализация объектов

- Стандартная сериализация
- Интерфейсы ObjectInput и ObjectOutput
- Классы ObjectInputStream и ObjectOutputStream
- интерфейс java.io.Serializable
- Пример:

```
// Родительский класс, не реализующий Serializable
public class Parent
{ public String firstName;
  private String lastName;
  public Parent()
   { System.out.println("Create Parent");
     firstName="old_first";
     lastName="old_last";
   }
  public void changeNames()
   { firstName="new_first";
     lastName="new_last";
   }
  public String toString()
  { return super.toString()+",first="+firstName+",last="+lastName;
    }
}
```

Стандартная сериализация

```
// Класс Child, впервые реализовавший Serializable
public class Child extends Parent implements Serializable
 { private int age;
   public Child(int age)
    { System.out.println("Create Child"); this.age=age; }
  public String toString()
    { return super.toString()+",age="+age; }
 }
// Наследник Serializable-класса
public class Child2 extends Child
 { private int size;
   public Child2(int age, int size)
    { super(age);
      System.out.println("Create Child2");
      this.size=size;
    }
  public String toString()
    { return super.toString()+",size="+size; }
 }
```

Стандартная сериализация

```
// Запускаемый класс для теста
 public class Test
  { public static void main(String[] arg)
    { try
       { FileOutputStream fos=new FileOutputStream("output.bin");
         ObjectOutputStream oos=new ObjectOutputStream(fos);
         Child c = new Child(2);
         c.changeNames(); System.out.println(c);
         oos.writeObject(c); oos.writeObject(new Child2(3, 4));
         oos.close();
         System.out.println("Read objects:");
         FileInputStream fis=new FileInputStream("output.bin");
         ObjectInputStream ois=new ObjectInputStream(fis);
         System.out.println(ois.readObject());
         System.out.println(ois.readObject());
         ois.close();
    } catch (Exception e) { // упрощенная обработка для краткости
            e.printStackTrace(); } } }
```

Результат выполнения примера:

```
Create Parent
Create Child
Child@ad3ba4,first=new_first,last=new_last,age=2
Create Parent
Create Child
Create Child2
Read objects:
Create Parent
Child@723d7c,first=old_first,last=old_last,age=2
Create Parent
Child2@22c95b,first=old_first,last=old_last,age=3,size=4
```

Сериализация объектов

- Итак, *сериализация* объекта заключается в сохранении и восстановлении состояния объекта. В Java в большинстве случаев состояние описывается значениями полей объекта. Причем, что важно, не только тех полей, которые были явно объявлены в классе, от которого порожден объект, но и унаследованных полей.
- Рассмотрим как стандартный механизм сериализации решает следующие проблемы:
  - Нам передается выходной *поток*, в который нужно записать состояние нашего объекта. С помощью DataOutput интерфейса можно легко сохранить значения всех доступных полей (будем для простоты считать, что они все примитивного типа). Однако в большинстве случаев в родительских классах могут быть объявлены недоступные нам поля (например, private). Тем не менее, такие поля, как правило, играют важную роль в определении состояния объекта, так как они могут влиять на результат работы унаследованных методов. Как же сохранить их значения?
  - С другой стороны, не меньшей проблемой является восстановление объекта. Как говорилось раньше, объект может быть создан только вызовом его конструктора. У класса, от которого порожден десериализуемый объект, может быть несколько конструкторов, причем, некоторые из них, или все, могут иметь аргументы. Какой из них вызвать? Какие значения передать в качестве аргументов?
- Рассмотрим подробнее работу с интерфейсом Serializable. Заметим, что класс Object не реализует этот интерфейс. Таким образом, существует два варианта – либо *сериализуемый* класс наследуется от Serializable - класса, либо нет. Первый вариант довольно прост. Если родительский класс уже реализовал интерфейс Serializable, то наследникам это свойство передается автоматически, то есть все объекты, порожденные от такого класса, или любого его наследника, могут быть сериализованы.
- Если же наш класс впервые реализует Serializable в своей ветке наследования, то его суперкласс должен отвечать специальному требованию – у него должен быть доступный конструктор без параметров. Именно с помощью этого конструктора будет создан десериализуемый объект и будут проинициализированы все поля, унаследованные от классов, не наследующих Serializable.
- Рассмотрим пример на слайде. В этом примере объявлено 3 класса. Класс Parent не реализует Serializable и, следовательно, не может быть сериализован. В нем объявлено 2 поля, которые при создании получают значения, содержащие слово "old" ("старый").

Сериализация объектов

- Кроме этого, объявлен метод, позволяющий модифицировать эти поля. Он выставляет им значения, содержащие слово "new" ("новый'). Также переопределен метод toString(), чтобы дать возможность узнать значения этих полей.
- Поскольку класс Parent имеет доступный конструктор по умолчанию, его наследник может реализовать интерфейс Serializable. Обратите внимание, что у самого класса Child такого конструктора уже нет. Также объявлено поле и модифицирован метод toString().
- Наконец, класс Child2 наследуется от Child, а потому автоматически является допустимым для *сериализации*. Аналогично, имеет новое поле, значение которого отображает toString().
- Запускаемый класс Test сериализует в файл output.bin два объекта. Обратите внимание, что у первого из них предварительно вызывается метод changeNames(), который модифицирует значения полей, унаследованных от класса Parent.
- Во всех конструкторах вставлена строка, выводящая сообщение на консоль. Так можно отследить, какие конструкторы вызываются во время *десериализации*. Видно, что для объектов, порожденных от Serializable -классов, конструкторы не вызываются вовсе. Идет обращение лишь к конструктору без параметров не-Serializable -суперкласса.
- Сравним значения полей первого объекта и его копии, полученной десериализацией. Поля, унаследованные от не- Serializable -класса (firstName, lastName), не восстановились. Они имеют значения, полученные в конструкторе Parent без параметров. Поля, объявленные в Serializable -классе, свои значения сохранили. Это верно и для второго объекта – собственные поля Child2 и унаследованные от Child имеют точно такие же значения, что и до *сериализации*. Их значения были записаны, а потом считаны и напрямую установлены из *потока данных*.

Исключение поля объекта из сериализации:

```
class Account implements java.io.Serializable
{ private String name;
  private String login;
  private transient String password;
 /* объявление других элементов класса ... */
}
```

Сериализация объектов

- Иногда в классе есть поля, которые не должны участвовать в *сериализации*. Тому может быть несколько причин. Например, это поле малосущественно (временная переменная) и сохранять его нет необходимости. Если сериализованный объект передается по сети, то исключение такого поля из *сериализации* позволяет уменьшить нагрузку на сеть и ускорить работу приложения.
- Некоторые поля хранят значения, которые не будут иметь смысла при пересылке объекта на другую машину, или при воссоздании его спустя какое-то время. Например, сетевое соединение, или подключение к базе данных, в таких случаях нужно устанавливать заново.
- Затем, в объекте может храниться конфиденциальная информация, например, пароль. Если такое поле будет сериализовано и передано по сети, его значение может быть перехвачено и прочитано, или даже подменено.
- Для исключения поля объекта из *сериализации* его необходимо объявить с модификатором *transient*.

- У класса Account поле password в *сериализации* участвовать не будет и при восстановлении оно получит значение по умолчанию (в данном случае null).
- Особого внимания требуют статические поля. Поскольку они принадлежат классу, а не объекту, они не участвуют в *сериализации*. При восстановлении объект будет работать с таким значением static -поля, которое уже установлено для его класса в этой JVM.

Сериализация объектов

- До этого мы рассматривали объекты, которые имеют поля лишь примитивных типов. Если же *сериализуемый* объект ссылается на другие объекты, их также необходимо сохранить (записать в *поток* байт), а при *десериализации* – восстановить. Эти объекты, в свою очередь, также могут ссылаться на следующие объекты. При этом важно, что если несколько ссылок указывают на один и тот же объект, то этот объект должен быть сериализован лишь однажды, а при восстановлении все ссылки должны вновь указывать на него одного. Например, *сериализуемый* объект A ссылается на объекты B и C, каждый из которых, в свою очередь, ссылается на один и тот же объект D. После *десериализации* не должно возникать ситуации, когда B ссылается на D1, а C – на D2, где D1 и D2 – равные, но все же различные объекты.
- Для организации такого процесса стандартный механизм *сериализации* строит граф, включающий в себя все участвующие объекты и ссылки между ними. Если очередная ссылка указывает на некоторый объект, сначала проверяется – нет ли такого объекта в графе. Если есть – объект второй раз не сериализуется. Если нет – новый объект добавляется в граф.
- При построении графа может встретиться объект, порожденный от класса, не реализующего интерфейс Serializable. В этом случае *сериализация* прерывается, генерируется исключение java.io.NotSerializableException.

- Граф сериализации
- Пример:

```
import java.io.*;
 class Point implements Serializable
{ double x;
  double y;
  public Point(double x, double y)
   { this.x = x; this.y = y; }
  public String toString()
   { return "("+x+","+y+") reference = "+super.toString(); }
}
class Line implements Serializable
 { Point point1;
   Point point2;
   int index;
   public Line()
     { System.out.println("Constructing empty line"); }
```

Сериализация объектов

```
Line(Point p1, Point p2, int index)
 { System.out.println("Constructing line: " + index);
   this.point1 = p1;
   this.point2 = p2;
   this.index = index;
 }
 public int getIndex() { return index; }
 public void setIndex(int newIndex) { index = newIndex; }
 public void printInfo()
  { System.out.println("Line: " + index);
    System.out.println(" Object reference: " + super.toString());
    System.out.println(" from point "+point1);
    System.out.println(" to point "+point2);
  }
}
```

Сериализация объектов

```
public class Main
 { public static void main(java.lang.String[] args)
   { Point p1 = new Point(1.0,1.0);
     Point p2 = new Point(2.0,2.0);
     Point p3 = new Point(3.0,3.0);
     Line line1 = new Line(p1,p2,1);
     Line line2 = new Line(p2,p3,2);
     System.out.println("line 1 = " + line1);
     System.out.println("line 2 = " + line2);
     String fileName = "d:\\file";
     try{ // записываем объекты в файл
          FileOutputStream os = new FileOutputStream(fileName);
          ObjectOutputStream oos = new ObjectOutputStream(os);
          oos.writeObject(line1); oos.writeObject(line2);
          // меняем состояние line1 и записываем его еще раз
          line1.setIndex(3);
          //oos.reset();
           oos.writeObject(line1);
       // закрываем потоки (достаточно закрыть только поток-надстройку)
           oos.close();
```

Сериализация объектов

```
    // считываем объекты
     System.out.println("Read objects:");
     FileInputStream is = new FileInputStream(fileName);
     ObjectInputStream ois = new ObjectInputStream(is);
     for (int i=0; i<3; i++)
      { // Считываем 3 объекта
        Line line = (Line)ois.readObject();
        line.printInfo();
      }
      ois.close();
   } catch(ClassNotFoundException e)
      { e.printStackTrace(); }
     catch(IOException e) { e.printStackTrace(); }
  }
}
```

- **Результат выполнения примера:**

```
Constructing line: 1
Constructing line: 2
line 1 = Line@7d39
line 2 = Line@4ec
Read objects:
Line: 1
   Object reference: Line@331e
   from point (1.0,1.0) reference=Point@36bb
   to point (2.0,2.0) reference=Point@386e
Line: 2
   Object reference: Line@6706
   from point (2.0,2.0) reference=Point@386e
   to point (3.0,3.0) reference=Point@68ae
Line: 1
   Object reference: Line@331e
   from point (1.0,1.0) reference=Point@36bb
   to point (2.0,2.0) reference=Point@386e
```

Результат выполнения примера (без комментария в строке //oos.reset();):

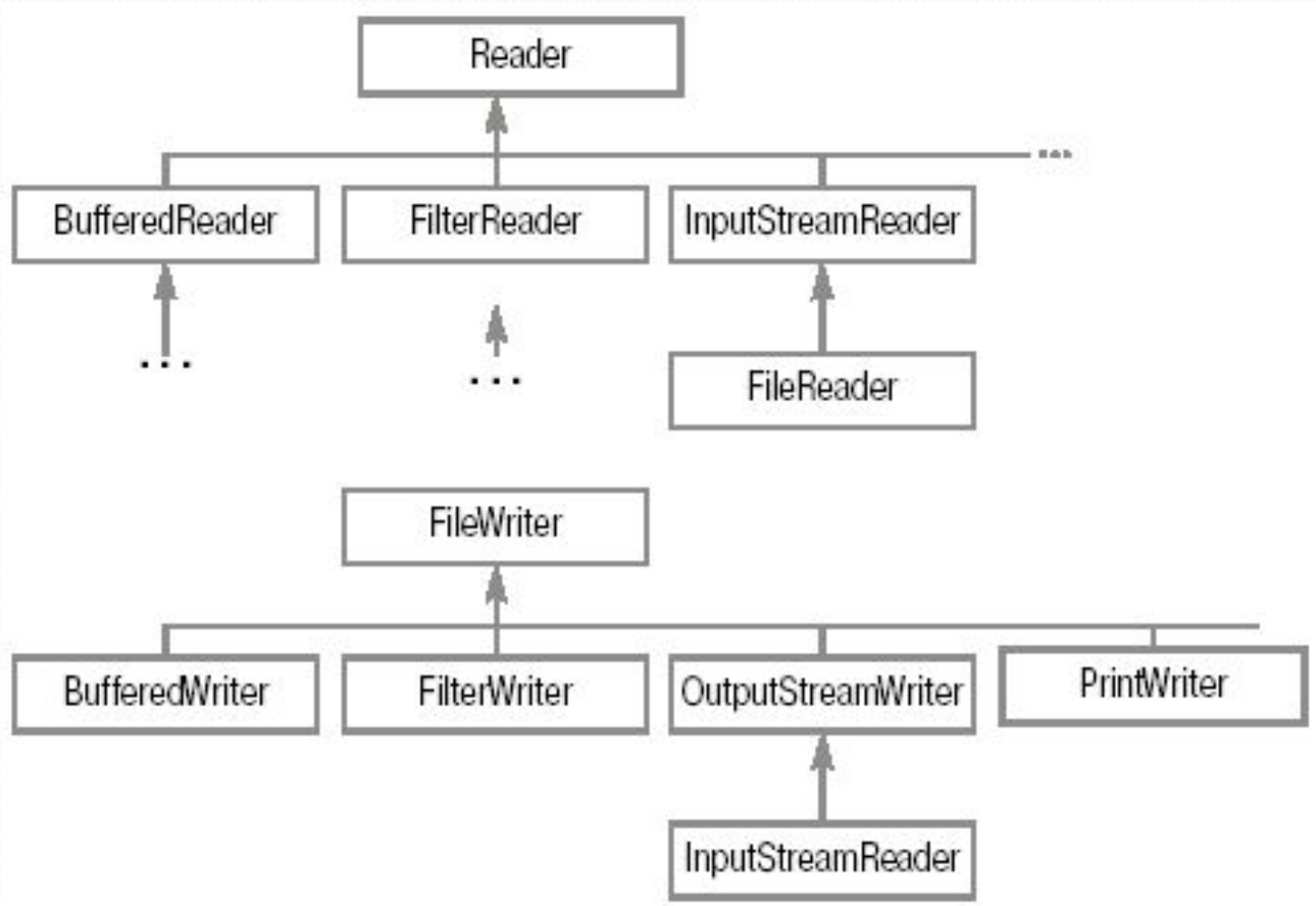
```
Constructing line: 1
Constructing line: 2
line 1 = Line@ea2dfe
line 2 = Line@7182c1
Read objects:
Line: 1
   Object reference: Line@a981ca
   from point (1.0,1.0) reference=Point@1503a3
   to point (2.0,2.0) reference=Point@a1c887
Line: 2
   Object reference: Line@743399
   from point (2.0,2.0) reference=Point@a1c887
   to point (3.0,3.0) reference=Point@e7b241
Line: 3
   Object reference: Line@67d940
   from point (1.0,1.0) reference=Point@e83912
   to point (2.0,2.0) reference=Point@fae3c6
```

Сериализация объектов

- В программе примера работа идет с классом Line (линия), который имеет 2 поля типа Point (линия описывается двумя точками). Запускаемый класс Main создает два объекта класса Line, причем, одна из точек у них общая. Кроме этого, линия имеет номер (поле index). Созданные линии (номера 1 и 2) записываются в *поток*, после чего одна из них получает новый номер (3) и вновь сериализуется.
- Из результата выполнения примера видно, что после восстановления у линий сохраняется общая точка, описываемая одним и тем же объектом (хеш-код 386e).
- Третий записанный объект идентичен первому, причем, совпадают даже объектные ссылки. Несмотря на то, что при записи третьего объекта значение index было изменено на 3, в десериализованном объекте оно осталось равным 1. Так произошло потому, что объект, описывающий первую линию, уже был задействован в *сериализации* и, встретившись во второй раз, повторно записан не был.
- Чтобы указать, что сеанс *сериализации* завершен, и получить возможность передавать измененные объекты, у ObjectOutputStream нужно вызвать метод reset(). В рассматриваемом примере для этого достаточно убрать комментарий в строке
- //oos.reset();
- Если теперь запустить программу, то можно увидеть, что третий объект получит номер 3.
- Однако это будет уже новый объект, ссылка на который отличается от первой считанной линии. Более того, обе точки будут также описываться новыми объектами. То есть в новом сеансе все объекты были записаны, а затем восстановлены заново.

Классы Reader и Writer и их наследники

Иерархия классов Reader и Writer :



Классы Reader и Writer и их наследники

- 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 Reader и Writer и их наследники

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 Reader и Writer и их наследники

- В таблице на слайде приведены соответствия классов для байтовых и символьных *потоков*.
- Как видно из таблицы, различия крайне незначительны и предсказуемы.
- Например, конечно же, отсутствует преобразование в символьное представление примитивных типов Java и объектов (DataInput/Output, ObjectInput/Output). Добавлены классы-мосты, преобразующие символьные *потоки* в байтовые: InputStreamReader и OutputStreamWriter. Именно на их основе реализованы FileReader и FileWriter. Метод available() классаInputStream 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*.
- Классы-мосты InputStreamReader и OutputStreamWriter 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:

```
String fileName = "d:\\file.txt";
//Строка, которая будет записана в файл
String data = "Some data to be written and read.\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();
```

Классы Reader и Writer и их наследники

```
// 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(Exception e)
     { e.printStackTrace(); }
```

Пример Найти количество вхождений заданного слова в текст:

```
import java.lang.*;
import java.io.*;
import java.util.*;

public class Main
{
   public static void main(String[] args) throws
  Exception
   {
     String str, word, si;
     int k = 0;
     Scanner in = new Scanner(System.in);
     System.out.println("Введите слово:");
     word = in.nextLine();

     BufferedReader r = new BufferedReader(
                                new FileReader("info.txt"));
```

Строки

```
while ((str = r.readLine()) != null)
 {  System.out.println(str);
    StringTokenizer st = new StringTokenizer(str, "
  \t\n\r,.-");
      while(st.hasMoreTokens())
       { si = st.nextToken(); // Получаем слово
      if (word.equalsIgnoreCase(si))
          k++;
       }
 }
  System.out.println("Количество вхождений  слова = " + k);
   }
}
```

```
Введите слово:
string1
String1 string2, string1. String3 string1.
String4 - string1.
Количество вхождений  слова = 4
```

Получение свойств файла

- Класс File
- Создание каталога:

```
mkdir(), mkdirs(),
```

- Удаление пустого каталога , удаление файла:

```
delete()
```

- Длина файла в байтах:

```
length()
```

- Переименование файла:

```
renameTo(File newName)
```

Получение свойств файла

- Если каталог с указанным в конструкторе путем не существует, его можно создать логическим методом mkdir(). Этот метод возвращает true, если каталог удалось создать. Логический метод mkdirs() создает еще и все несуществующие каталоги, указанные в пути.
- Пустой каталог удаляется методом delete ().
- Для файла можно получить его длину в байтах методом length (), время последней модификации в секундах с 1 января 1970 г. методом lastModified(). Если файл не существует, эти методы возвращают нуль.
- Файл можно переименовать логическим методом renameTo(File newName) или удалить логическим методом delete (). Эти методы возвращают true, если операция прошла удачно.
- Если файл с указанным в конструкторе путем не существует, его можно создать логическим методом createNewFile(), возвращающим true, если файл не существовал, и его удалось создать, и false, если файл уже существовал.

- На следующем слайде показан пример использования класса File.

Получение свойств файла

- **Пример .** Определение свойств файла и каталога :

```
import java.io.*;
import java.util.*;
class FileTest
{ public static void main(String[] args) throws IOException
    {  Scanner in = new Scanner(System.in);
       File f = new File("FileTest.Java"); System.out.println();
       System.out.println("Файл \"" + f.getName() + "\" " +
                  (f.exists()?"":"не ") + "существует");
       System.out.println("Вы " + (f.canRead()?"":"не ")
                  + "можете читать файл");
       System.out.println("Вы " + (f.canWrite()?"":"не ") +
                   "можете записывать в файл");
       System.out.println("Длина файла " + f.length() + " б");
       System.out.println() ;
       File d = new File("D:\\pr_java");
       System.out.println("Содержимое каталога:");
       if (d.exists() && d.isDirectory())
         {  String[] s = d.list();
            for (int i = 0; i < s.length; i++)
             System.out.println(s[i]);
          }
    }
}
```

```
Файл "FileTest.Java" существует
Вы можете читать файл
Вы можете записывать в файл
Длина файла 990 б

Содержимое каталога:
Dog.class
Fahrenheit.class
Fahrenheit.java
FileTest.class
FileTest.java
geany_run_script.bat
Hello.class
Hello.java
HelloWorld.class
HelloWorld.java
JavaTest.java
Main.class
Main.java
Pi.class
Pi.java
pr
Progr.class
Progr.java
```

Консольный ввод-вывод

- **Потоковые объекты:**
  - `System.in` – объект класса `InputStream`;
  - `System.out` – объект класса `PrintStream`;
- **Вывод данных:** метод `System.out.println( );`

```
System.out.println("Hello World"); // вывод строки
System.out.println(245); // вывод числовой константы

// Вывод строковой или численной переменной
int var = 245;        String text = "Hello World";
System.out.println(text); System.out.println(var);

// Вывод произвольного сочетания переменных и констант
System.out.println("Hello" + "number = " + var);

// Вывод вещ. чисел с ограничением числа знаков после запятой
double f = 2.567432;
NumberFormat nf = NumberFormat.getInstance();
nf.setMaximumFractionDigits(2);
System.out.println(nf.format(f));
```

Консольный ввод-вывод

- Для работы с консолью в классе java.lang.System определены так называемые стандартные потоки, предназначенные для ввода данных с клавиатуры, вывода обычных сообщений и сообщений об ошибках.
- Стандартный поток ввода определен следующим образом:

  public static final InputStream in;
- Для вывода обычных сообщений используется стандартный поток вывода:

  public static final PrintStream out;
- Класс PrintStream используется для конвертации и записи строк в байтовый поток. В нем определены методы print(...) и println(...) , принимающие в качестве аргумента различные примитивные типы Java, а также тип Object. При вызове передаваемые данные будут сначала преобразованы в строку вызовом метода String.valueOf(), после чего записаны в поток. При записи символов в виде байт используется кодировка, принятая по умолчанию в операционной системе. Этот класс объявлен как *deprecated*, то есть использовать его не рекомендуется, поскольку работа с кодировками требует особого подхода (зачастую у двухбайтовых символов Java старший байт просто отбрасывается). Поэтому в версии Java 1.1 появился дополнительный набор классов, основывающийся на типах Reader и Writer. В частности, вместо PrintStream теперь рекомендуется применять PrintWriter. Однако старый класс продолжает активно использоваться, поскольку статические поля out и err класса System имеют именно это тип.
- Для ограничение числа знаков после запятой при выводе вещественных чисел необходимо создать обьект класса **NumberFormat** и вызвать его метод **setMaximumFractionDigits()**, с помощью которого можно указать, сколько следует выводить знаков после запятой. Класс **NumberFormat** входит в состав пакета **java.text**.
- В метод **println()** передается результат форматирования переменных с помощью метода **format()** класса **NmberFomat.**

Консольный ввод-вывод

- **Ввод данных с клавиатуры:**

```
// установить связь с  вводимыми с клавиатуры данными
import java.io.*
    …
BufferedReader bReader = new
BufferedReader (new InputStreamReader(System.in));

// ввод строки данных
String cStr;
cStr = bReader.readLine();

// преобразование строки данных
int iNum = Integer.parseInt(cStr);
float fNum = Float.parseFloat(cStr);
double dNum = Double.parseDouble(cStr);
```

Консольный ввод-вывод

- Считывание вводимых с клавиатуры данных производится в программе на Java в два этапа:
  - установить связь с вводимыми с клавиатуры данными;
  - преобразовать вводимые пользователем знаки в данные соответствующего типа.
- Для того, чтобы установить связь с вводимыми с клавиатуры данными используются три различных класса: **BufferedReader, InputStreamReader** и **InputStream**, объекты которых взаимодействуют между собой и возвращают значение в переменную bReader.
- **System.in** – стандартный байтовый поток для ввода данных (объект класса **InputStream).**
- **InputStreamReader** - объект класса InputStreamReader обеспечивает преобразование из байтового потока в символьный поток: он читает байты и декодирует их в символы, используя указанную кодировку.
- **BufferedReader** - Чтение текста из посимвольного входного потока с использованием буферизации.
- Переменная **bReader** имеет тип класса **BufferedReader**, из которого мы используем только метод **readLine( )**, необходимый для получения введенной пользователем с клавиатуры строки.
- Преобразование строки, которая содержит некое число, в соответствующее численное значение производится с помощью специальных методов, которые существуют для каждого элементарного типа данных. Необходимым условием при вызове данных методов является то, что строка **cStr** должна соответствовать подразумеваемому виду. Если метод получает строку ”**Hello**”, то при всем своем желании компилятор не сможет преобразовать ее в число!

Контрольные вопросы

1. Понятие исключения. Иерархия классов исключений.
2. Как организуется обработка исключительных ситуаций с использованием стандартных классов исключений?
3. Создание собственных классов исключений и их использование для обработки исключительных ситуаций.
4. Консольный ввод-вывод: используемые классы и объекты, примеры.
5. Классы и потоки файлового ввода-вывода: назначение и примеры использования.
6. Сериализация объектов: понятие и назначение, примеры.