Лекция 12

Пространство имен в XML документе

Прежде чем рассмотреть подробно метод startElement, рассмотрим понятие пространства имен.

Концептуально, пространство имен функционирует как Java-оператор package.

У двух классов могут быть одинаковые имена, если они из двух разных пакетов.

Пространства имён состоят из двух частей: префикс и однозначная строка.

Рассмотрим пример:

```
<bookOrder
  xmlns:lit="http://www.literarysociety.org/books"
  xmlns:addr="http://www.usps.com/addresses">
  . . .
 <lit:title>My Life in the Bush of Ghosts</lit:title>
 . . .
 <shipTo>
  <addr:title>Ms.</addr:title>
  <addr:firstName>Linda</addr:firstName>
  <addr:lastName>Lovely</addr:lastName>
   . . .
</bookOrder>
```

Этот документ определяет два пространства имён:

- префикс lit ассоциируется со строкой http://www.literarysociety.org/books;
- префикс addr ассоциируется со строкой http://www.usps.com/addresses;

Когда используется элемент <lit:title> или <addr:title>, ясно, который элемент <title> используется

Когда определяется пространства имён на данном элементе, эти пространства имён могут быть использованы этим элементом и любым элементом внутри него.

Например:

```
<shipTo xmlns:addr="http://www.usps.com/addresses">
   <addr:title>Ms.</addr:title>
    <addr:firstName>Linda</addr:firstName>
    <addr:lastName>Lovely</addr:lastName>
   . . .
</shipTo>
```

Другой эквивалентный вариант данного фрагмента имеет вид:

```
<shipTo>

 <addr:title
    xmlns:addr="http://www.usps.com/addresses"> Ms.
 </addr:title>

 <addr:firstName
  xmlns:addr="http://www.usps.com/addresses"> Linda
 </addr:firstName>

 <addr:lastName
  xmlns:addr="http://www.usps.com/addresses"> Lovely
 </addr:lastName>
 . . .
```

Когда используется префикс пространства имён, пространство имён, ассоциируемое с этим префиксом, должно быть определено в этом элементе или одном из предшествующих элементов.

Определение всех пространств имён в корневом элементе упрощает и сокращает документ.

Использование атрибута xmlns без определения префикса, определяет **пространство имён по умолчанию** для текущего элемента и любого производного элемента, которые не имеют префикс пространства имён.

Пример:

```
<author
   xmlns="http://www.literarysociety.org/authors">


   <lastName>Shakespeare</lastName>
   <firstName>William</firstName>
   <nationality>British</nationality>
   <yearOfBirth>1564</yearOfBirth>
   <yearOfDeath>1616</yearOfDeath>


</author>
```

Так как ни у одного из этих элементов нет префикса пространства имён, анализатор, поддерживающий пространства имён, покажет, что все эти элементы принадлежат пространству имён http://www.literarysociety.org/authors.

Вне элемента <author> это пространство имён по умолчанию больше не определяется.

Сравнение двух пространств имён

Рассмотрим проверку значения пространства имен.

Например, в таблицах стилей XSLT все элементы таблицы стилей должны быть из пространства имён **http://www.w3.org/1999/XSL/Transform**.

Обычно эта строка пространства имён ассоциируется с префиксом xsl, но это не обязательно.

Когда проверяется верность пространства имён для данного элемента, необходимо проверить строку пространства имён, а не префикс пространства имён.

Другими словами,

этот XSLT-элемент верен:

<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

этот XSLT элемент неверен:

<xsl:stylesheet xmlns:xsl="http://ilove-stylesheets.com">

В данном примере префикс пространства имён такой, какой мы ожидаем, но строка пространства имён неверна.

Метод startElement

Рассмотрим подробнее интерфейс метода startElement.

void startElement(String uri, String localName, String qName, Attributes attrs)

Здесь

Квалифицированное имя или qName. Это комбинация информации пространства имен, если оно существует, и собственно имени элемента. qName также включает в себя двоеточие (:), если оно есть - например, lit:title.

URI пространства имен. Пример: URL для пространства имен с именем lit

http://www.literarysociety.org/books.

Локальное имя. Это действительное имя элемента, такое, как note. Если документ не предоставляет информацию о постранствах имен, парсер не сможет определить, какой частью qName является localName.

Любые атрибуты. Атрибуты для элементов передаются как коллекция объектов, как показано в предыдущей лекции.

Следует помнить, что атрибуты никогда не находятся в пространстве имен по умолчанию.

Список Attributes имеет методы, которые позволяют определять пространство имен атрибута, это методы, getURI() и getQName.

Парсер Java по умолчанию не будет сообщать о значениях локальных имен, если специально не включена обработка пространств имен.

Рассмотрим процедуру включения обработки пространства имен:

```
...
try {
SAXParserFactory spfactory =
                                    SAXParserFactory.newInstance();
    spfactory.setValidating(true);
    spfactory.setNamespaceAware(true);
SAXParser saxParser = spfactory.newSAXParser(); ...}
 catch(….){…..}
……..
```

setNamespaceAware(boolean awareness) - установка информированности о пространстве имен,

setValidating(boolean validating) – включение проверки корректности DTD

Древовидная модель DOM

DOM (Dynamic object model) представляет собой некоторый общий интерфейс для работы со структурой документа. Одна из целей разработки заключалась в том, чтобы код, написанный для работы с каким-либо DOM-анализатором, мог работать и с любым другим DOM-анализатором.

DOM-анализатор строит дерево, которое представляет содержимое XML-документа, и определяет набор классов, которые представляют каждый элемент в XML-документе (элементы, атрибуты, сущности, текст и т.д.).

В Java включена поддержка DOM (пакет org.w3c.dom).

Основным объектом DOM является **Node** – некоторый общий элемент дерева. Большинство DOM-объектов унаследовано именно от Node.

Node определяет ряд методов, которые используются для работы с деревом:

getNodeType() – возвращает тип объекта (элемент, атрибут, текст, CDATA и т.д.), список возвращаемых значений имеет вид

ELEMENT_NODE 1
ATTRIBUTE_NODE 2
TEXT_NODE 3
CDATA_SECTION 4
ENTITY_REFERENCE_NODE 5
ENTITY_NODE 6
PROCESSING_INSTRUCTION_NODE 7
COMMENT_NODE 8
DOCUMENT_NODE 9
DOCUMENT_TYPE_NODE 10
DOCUMENT_FRAGMENT_NODE 11
NOTATION_NODE 12

getParentNode() – возвращает объект, являющийся родителем текущего узла Node;

getChildNodes() – возвращает список объектов, являющихся дочерними элементами;

getFirstChild(), **getLastChild()** – возвращает первый и последний дочерние элементы;

getAttributes() – возвращает список атрибутов данного элемента.

getNodeName() и getNodeValue() - используется для извлечения имени и значения каждого атрибута

setNodeValue() - устанавливает значение узла

hasChildNodes() – возвращает true если существуют дочерние узлы.

getChildNodes() - возвращает дочерние элементы(возвращает объект класса **NodeList**)

Интерфейс **NamedNodeMap**

getLength() - возвращает количество элементов

item(int) – извлекает элемент с указанным индексом. Если узел не содержит атрибутов, то возвращается null.

Интерфейс **Document.**

getElementsByTagName(String) - в качестве параметра задается имя элемента, метод возвращает объект класса NodeList (среди его методов также присутствуют getLength() и item(int), как у NamedNodeMap), содержащий ссылки на все элементы с заданным именем.

Интерфейс **Element**

getElementsByTagName(String) - работа этого метода идентична вышерассмотренному, с той разницей, что этот метод ищет элементы только среди вложенных элементов текущего узла.

Рассмотрим разбор документа notepad.xml (будем использовать анализатор XML4J от IBM).

```
import org.w3c.dom.Element;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;
import org.apache.xerces.parsers.DOMParser;
import java.net.URL;
import java.util.Vector;
```

```
public class MyDOMDemo {
  public static String getValue(Element e, String name) {
    NodeList nList = e.getElementsByTagName(name);
    Element elem = (Element) nList.item(0);
    Text t= (Text) elem.getFirstChild();
    return t.getNodeValue();
 }
public static void main(String[] args) {
   Document doc = null;
   DOMParser parser = new DOMParser();
   Vector entries = new Vector();
  try {
    parser.parse("notepad.xml");
    doc = parser.getDocument();
    Element root = doc.getDocumentElement();
    NodeList noteList = root.getElementsByTagName("note");
    Element noteElem;
```

```
for (int i = 0; i <noteList.getLength(); i++) {
  noteElem = (Element) noteList.item(i);
  Myclass e = new Myclass();
  NodeList list = noteElem.getChildNodes();
  Node log = noteElem.getAttributes().item(0);
  e.setLogin(log.getNodeValue());
  e.setName(getValue(noteElem, "name"));
  e.setTel(Integer.parseInt(getValue(noteElem,
                                        "tel")));
  e.setUrl(new URL(getValue(noteElem, "url")));
  Element n =
  (Element)noteElem.getElementsByTagName(
                                    "address").item(0);
```

```
    e.address.setStreet(getValue(n, "street"));
    e.address.setCountry(getValue(n, "country"));
    e.address.setCity(getValue(n, "city"));
    entries.add(e);
   }
  }
  catch (Exception e) {System.out.println(e);}
  for (int i = 0; i < entries.size(); i++)
    System.out.println(((Note)
                    entries.elementAt(i)).toString());}}
```

XML-документы можно не только читать, но и корректировать. Рассмотрим пример

```
import org.jdom.*;
import org.jdom.input.SAXBuilder;
import org.jdom.output.XMLOutputter;
import java.util.*;
import java.io.FileOutputStream;
public class JDOMChanger {
   static void lookForElement(String name,
                  String element, String content, String login) {
    SAXBuilder builder = new SAXBuilder();
  try {
     Document document = builder.build(name);
     Element root = document.getDocumentElement();
     NodeList c = root.getChildNodes();
```

```
  for (int i=0; i<c.getLenght(); i++) {
    Element e = (Element) c.item(i);
    if (e.getAttributeValue("login").equals(login)) {
      e.getChild(element).setText(content);
    }
  }
   XMLOutputter serializer = new XMLOutputter();
   serializer.output(document, new FileOutputStream(name));
   System.out.flush();
 }
 catch (Exception e) { System.out.println(e);}
 }
 public static void main(String[] args) {
  String name = "notepad.xml";
  JDOMChanger.lookForElement(name, "tel", "09", "rom");
 }
}
```

Рассмотрим преобразование файла notepad.xml в html файл с использованием notepad.xsl, который имеет вид

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
  <html>
    <head><title>Notepad Contents</title></head>
   <body>
    <table border="1">
    <tr>
     <th>Login</th>
     <th>Name</th>
     <th>Street</th>
    </tr>
   <xsl:for-each select="notepad/note">
     <tr>
      <td><xsl:value-of select="@login"/></td>(символ @ означает,что далее идет атрибут)
      <td><xsl:value-of select="name"/></td>
     <td><xsl:value-of select="address/street"/></td>
   </tr>
  </xsl:for-each>
```

```
</table>
 </body></html>
</xsl:template>
</xsl:stylesheet>
```

Соответствующий код будет иметь вид:

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamSource;
```

```
public class SimpleTransform {
   public static void main(String[] args) {
   try {
    TransformerFactory tFact =
                            TransformerFactory.newInstance();
    Transformer transformer = tFact.newTransformer(
                                      new StreamSource("notepad.xsl"));
    transformer.transform(
                          new StreamSource("notepad.xml"),
                          new StreamResult("notepad.html"));
   } catch (TransformerException e){
     e.printStackTrace();
   }
 }
}
```

В результате получится HTML-документ следующего вида:

```
<html><head>
<META http-equiv="Content-Type" content="text/html;
    charset=UTF-8">
  <title>Notepad Contents</title>
    </head>
   <body>
   <table border="1">
   <tr>
    <th>Login</th><th>Name</th><th>Street</th>
    </tr><tr>
   <td>rom</td><td>Valera</td><td>Main Str., 35</td>
   </tr><tr>
   <td>goch</td><td>Igor</td><td>Deep Forest, 7</td></tr>
</table></body></html>
```

Рассмотрим проверку документа на корректность средствами языка Java.

```
import java.io.IOException;
import org.xml.sax.SAXException;
import org.apache.xerces.parsers.DOMParser;
import org.xml.sax.SAXNotRecognizedException;
import org.xml.sax.SAXNotSupportedException;
```

```
public class XSDMain {
 public static void main(String[] args) {
  String filename = "students.xml";
  DOMParser parser = new DOMParser();
  try {
    // установка обработчика ошибок
    parser.setErrorHandler(new MyErrorHandler("log.txt"));
   // установка способов проверки с  использованием XSD
   parser.setFeature("http://xml.org/sax/features/validation", true);
   parser.setFeature("http://apache.org/xml/features/validation/schema",

true);
   parser.parse(filename);
  } catch (SAXNotRecognizedException e) {
     e.printStackTrace();
 }
}
}
```

Обработчик ошибок MyErrorHandler имеет вид:

```
import java.io.IOException;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXParseException;
import org.apache.log4j.FileAppender;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;
```

```
public class MyErrorHandler implements
                                        ErrorHandler {
  private Logger logger;
  public MyErrorHandler(String log) throws
                                        IOException {
    //создание регистратора ошибок
    logger = Logger.getLogger("error");
   //установка файла и формата вывода
                                        ошибок
 logger.addAppender(new FileAppender(
                        new SimpleLayout(), log));
}
```

```
public void warning(SAXParseException e) {
  logger.warn(getLineAddress(e) + "-" + e.getMessage());
 }
 public void error(SAXParseException e) {
  logger.error(getLineAddress(e) + " - “ + e.getMessage());
 }
 public void fatalError(SAXParseException e) {
  logger.fatal(getLineAddress(e) + " - “ + e.getMessage());
 }
 private String getLineAddress(SAXParseException e) {
  //определение строки и столбца ошибки
  return e.getLineNumber() + " : “ + e.getColumnNumber();
 }
}
```

Маршалинг и Демаршалинг

Маршаллизация - это процесс преобразования находящихся в памяти данных в формат их хранения.

Так, для технологий Java и XML, маршаллизация представляет собой преобразование некоторого набора Java-объектов в XML-документ.

Таким образом, смысл маршаллизации заключается в преобразовании объектно-ориентированной структуры экземпляров Java-объектов в плоскую структуру XML.

Демаршаллизация - это процесс преобразования данных из формата среды хранения в память, т.е. процесс, прямо противоположный маршаллизации.

Иначе говоря, можно демаршиллизировать XML-документ в Java VM.

Сложность здесь заключается в отображении нужных данных в нужные переменные Java-кода.

Если такое отображение ошибочно, то тогда невозможно получить доступ к данным.

Это, в свою очередь, приведет к еще большим проблемам при попытке обратной маршаллизации данных, причем проблемы быстро нарастают.

Кругооборот данных(round-tripping) является важным термином связывания данных.

Понятие кругооборота данных используется для описания полного цикла перемещения данных - из среды хранения в память и обратно.

Для технологий Java и XML это означает перемещение данных из XML-документа в экземпляры переменных Java и обратно в XML-документ.

Корректный кругооборот данных требует идентичности исходных и полученных XML-документов в предположении, что данные во время этой операции не менялись.

Рассмотрим пример маршализации:

```
class Myclass{
  public int a;
public Myclass(){a=10;}}
@XmlRootElement
class Employee {
      private String code;
      private String name;
      @XmlElement
      private Myclass m;
      private int salary;
      public String getCode() { return code; }

    public void setCode(String code) { this.code = code; }

    public String getName() { return name;  }

    public void setName(String name) { this.name = name;  }

    public int getSalary() {return salary; }


   public void setMyclass(Myclass m1){m=m1;}
    public void setSalary(int population) { this.salary = population;  }}
```

```
class Main {
public static void main(String[] args){
 try{
   JAXBContext context = JAXBContext.newInstance(
                                                    Employee.class);
  Marshaller m = context.createMarshaller();
  m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
    //true означает применит данное свойство.
  Employee object = new Employee();
  Myclass m1=new Myclass();
  object.setCode("CA");
  object.setName("Cath");
  object.setSalary(300);
  object.setMyclass(m1);
  File f=new File(“my.xml”);
   m.marshal(object, f);
 }
 catch(Exception e){};
}
}
```

@XmlRootElement – аннотация использующаяся вместе с классом верхнего уровня или с перечислением. Если класс или перечисление используется с данной аннотацией, то это означает, что его значение представляется как xml элемент.

@XmlElement- аннотация, означающая, что данное поле класса в xml схеме нужно представлять как сложный тип.

В интерфейсе Marshaller определены следующие константы (которые используются в setProperty):

JAXB_ENCODING – данное свойство используется для спецификации выходных данных, те что записываются в xml файл.

JAXB_FORMATTED_OUTPUT – данное свойство говорит о том, что данные записываются в xml файл в форматированном виде.

JAXB_FRAGMENT – данное свойство используется, если записываемые данные должны генерировать событие при разборе данного xml документа SAX парсером (имеется ввиду, событие типа startDocument)

JAXB_NO_NAMESPACE_SCHEMA_LOCATION – данное свойство говорит о том, что в генерируемый xml документ нужно поместить атрибут xsi:noNamespaceSchemaLocation

JAXB_SCHEMA_LOCATION – данный элемент говорит о том, что генерируемый xml документ нужно поместить атрибут xsi:schemaLocation

Файл my.xml примет вид:

```
<?xml version="1.0" encoding="UTF-8" standalon
  e="yes"?>
  <employee>
  <m>
      <a>10</a>
  </m>
   <code>CA</code>
     <name>Cath</name>
     <salary>300</salary>
  </employee>
```

Демаршализация будет иметь вид:

```
class Myclass{
  public int a;
  public Myclass(){a=10;}
}
@XmlRootElement
class Employee {
    private String code;
    private String name;
    private Myclass m;
    private int salary;


   public String getCode() { return code; }

    public void setCode(String code) { this.code = code; }

    public String getName() { return name;  }

    public void setName(String name) { this.name = name;  }

    public int getSalary() {return salary; }


   public void setMyclass(Myclass m1){m=m1;}
    public void setSalary(int population) { this.salary = population;  }
}
```

```
class Main {
  public static void main(String[] args){
   try{
     JAXBContext context = JAXBContext.
                          newInstance(Employee.class);
     Unmarshaller m = context. createUnmarshaller() ;
     File f=new File("my.xml");
     Employee em=(Employee)m.unmarshal(f);
     System.out.println(em.getCode());
   }
   catch(Exception e){};
   }
}
```

Демаршалинг можно автоматизировать при наличии xsd файла.
Рассмотрим файл my.xsd, соответствующий классам Employee и Myclass:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="Employee">
 <xs:complexType>
  <xs:sequence>
    <xs:element name="code" type="xs:string"/>
     <xs:element name="name" type="xs:string"/>
    <xs:element name="m" type="Myclass">
      <xs:complexType name="Myclass">
       <xs:sequence>
       <xs:element name="a" type="xs:integer"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    <xs:element name="salary" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
  </xs:element>
</xs:schema>
```

Генерируем java код соответствующий xsd файлу

xjc –nv my.xsd –d src

(xjc находится в той же директории, где и javac). Директорию src нужно предварительно создать. В итоге в директории будут 3 файла

Employee.java

Myclass.java

ObjectFactory.java

Файл Employee.java имеет вид:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "code",
    "name",
    "m",
    "salary"
})
@XmlRootElement(name = "Employee")
public class Employee {
    @XmlElement(required = true)
     protected String code;
    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    protected Myclass m;
    @XmlElement(required = true)
    protected BigInteger salary;
```

```
public String getCode() {  return code;    }
public void setCode(String value) {this.code = value;
   }
public String getName() {  return name;    }
public void setName(String value) {  this.name =
   value;  }
public Myclass getM() { return m;  }
public void setM(Myclass value) { this.m = value;  }
public BigInteger getSalary() { return salary;  }
public void setSalary(BigInteger value) {
    this.salary = value;  }
}
```

Файл Myclass.java имеет вид:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Myclass", propOrder = {
    "a"
  })
public class Myclass {
    @XmlElement(required = true)
    protected BigInteger a;

    public BigInteger getA() { return a; }

    public void setA(BigInteger value) {
       this.a = value; }
}
```

Файл ObjectFactory.java имеет вид:

```
import javax.xml.bind.annotation.XmlRegistry;

 public class ObjectFactory {
  public ObjectFactory() {  }

  public Myclass createMyclass() {
        return new    Myclass(); }

  public Employee createEmployee() {
       return new Employee(); }
}
```

Тогда демаршализация будет иметь вид:

```
class Main {
  public static void main(String[] args){
   try{
   JAXBContext context = JAXBContext.newInstance(
                                                    Employee.class);
    Unmarshaller m = context. createUnmarshaller() ;
    File f=new File(“my.xml”);
    Employee em=(Employee)m.unmarshal(f);
    System.out.println(em.getCode());
   }
   catch(Exception e){};
 }
}
```

Примечание: для того, чтобы не было исключения, необходимо убрать в аннотации

@XmlRootElement(name = "Employee"), часть name = "Employee",

т.е оставить только

@XmlRootElement

Потоки ввода-вывода

Для того чтобы отвлечься от особенностей конкретных устройств ввода/вывода, в Java употребляется понятие потока (stream).

Считается, что в программу идет входной поток (input stream) символов Unicode или просто байтов, воспринимаемый в программе методами read().

Из программы методами write() или print(), println() выводится выходной поток (output stream) символов или байтов.

При этом неважно, куда направлен поток: на консоль, на принтер, в файл или в сеть, методы write() и print() ничего об этом не знают.

Полное игнорирование особенностей устройств ввода/вывода сильно замедляет передачу информации.

Поэтому в Java выделяется файловый ввод/вывод, вывод на печать, сетевой поток.

Три потока определены в классе system статическими полями in, out и err.

Их можно использовать без всяких дополнительных определений.

Они называются соответственно стандартным вводом (stdin), стандартным выводом (stdout) и стандартным выводом сообщений (stderr).

Эти стандартные потоки могут быть соединены с разными конкретными устройствами ввода и вывода.

Потоки out и err — это экземпляры класса Printstream, организующего выходной поток байтов.

Эти экземпляры выводят информацию на консоль методами print(), println() и write().

Поток err предназначен для вывода системных сообщений программы: трассировки, сообщений об ошибках или, просто, о выполнении каких-то этапов программы.

Поток in — это экземпляр класса InputStream.

Он назначен на клавиатурный ввод с консоли методами read().

Класс InputStream абстрактный, поэтому реально используется какой-то из его подклассов.

В Java предусмотрена возможность создания потоков, направляющих символы или байты не на внешнее устройство, а в массив или из массива,

т. е. связывающих программу с областью оперативной памяти.

Таким образом, в Java есть четыре иерархии классов для создания, преобразования и слияния потоков.

Во главе иерархии четыре класса, непосредственно расширяющих класс Object:

- **Reader** — абстрактный класс, в котором собраны самые общие методы символьного ввода;
- **Writer** — абстрактный класс, в котором собраны самые общие методы символьного вывода;
- **InputStream** — абстрактный класс с общими методами байтового ввода;
- **OutputStream** — абстрактный класс с общими методами байтового вывода.

Классы входных потоков Reader и InputStream определяют по три метода ввода:

read() — возвращает один символ или байт, взятый из входного потока, в виде целого значения типа int; если поток уже закончился, возвращает -1;

read(char[] buf) — заполняет определенный массив buf символами из входного потока;

в классе InputStream используется вместо char[] массив типа byte[] и заполняется он байтами;

метод возвращает фактическое число взятых из потока элементов или -1, если поток уже закончился;

read (char[] buf, int offset, int len) — заполняет часть символьного или байтового массива buf, начиная с индекса offset, число взятых из потока элементов равно len; метод возвращает фактическое число взятых из потока элементов или -1.

Эти методы выбрасывают IOException, если произошла ошибка ввода/вывода.

long skip(long n) "проматывает" поток с текущей позиции на n символов или байтов вперед.

Метод возвращает реальное число пропущенных элементов, которое может отличаться от n, например поток может закончиться.

void mark(int n) помечает текущий элемент потока, к которому затем можно вернуться с помощью метода reset(), но не более чем через n элементов.

void reset() – осуществляет возврат к помеченному элементу

boolean marksupported() - возвращает true, если реализованы методы расстановки и возврата к меткам.

Классы выходных потоков Writer и OutputStream определяют по три почти одинаковых метода вывода:

write (char[] buf) — выводит массив в выходной поток, в классе Outputstream массив имеет тип byte[];

write (char[] buf, int offset, int len) — выводит len элементов массива buf, начиная с элемента с индексом offset;

write (int elem) в классе Writer - выводит 16, а в классе Outputstream 8 младших битов аргумента elem в выходной поток;

В классе Writer есть еще два метода:

write (String s) — выводит строку s в выходной поток;

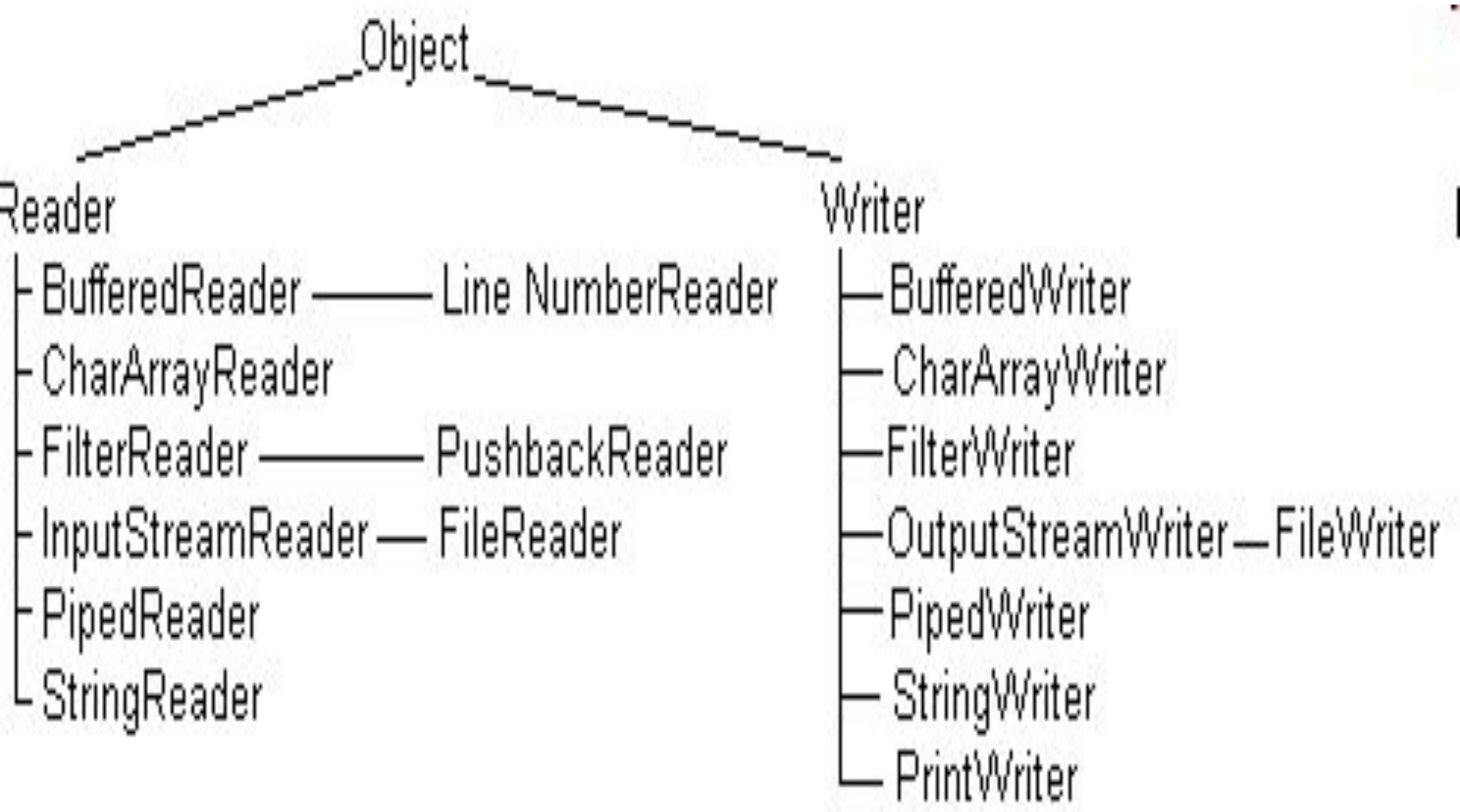
write (String s, int offset, int len) — выводит len символов строки s, начиная с символа с номером offset.

Многие подклассы классов Writer и OutputStream осуществляют буферизованный вывод.

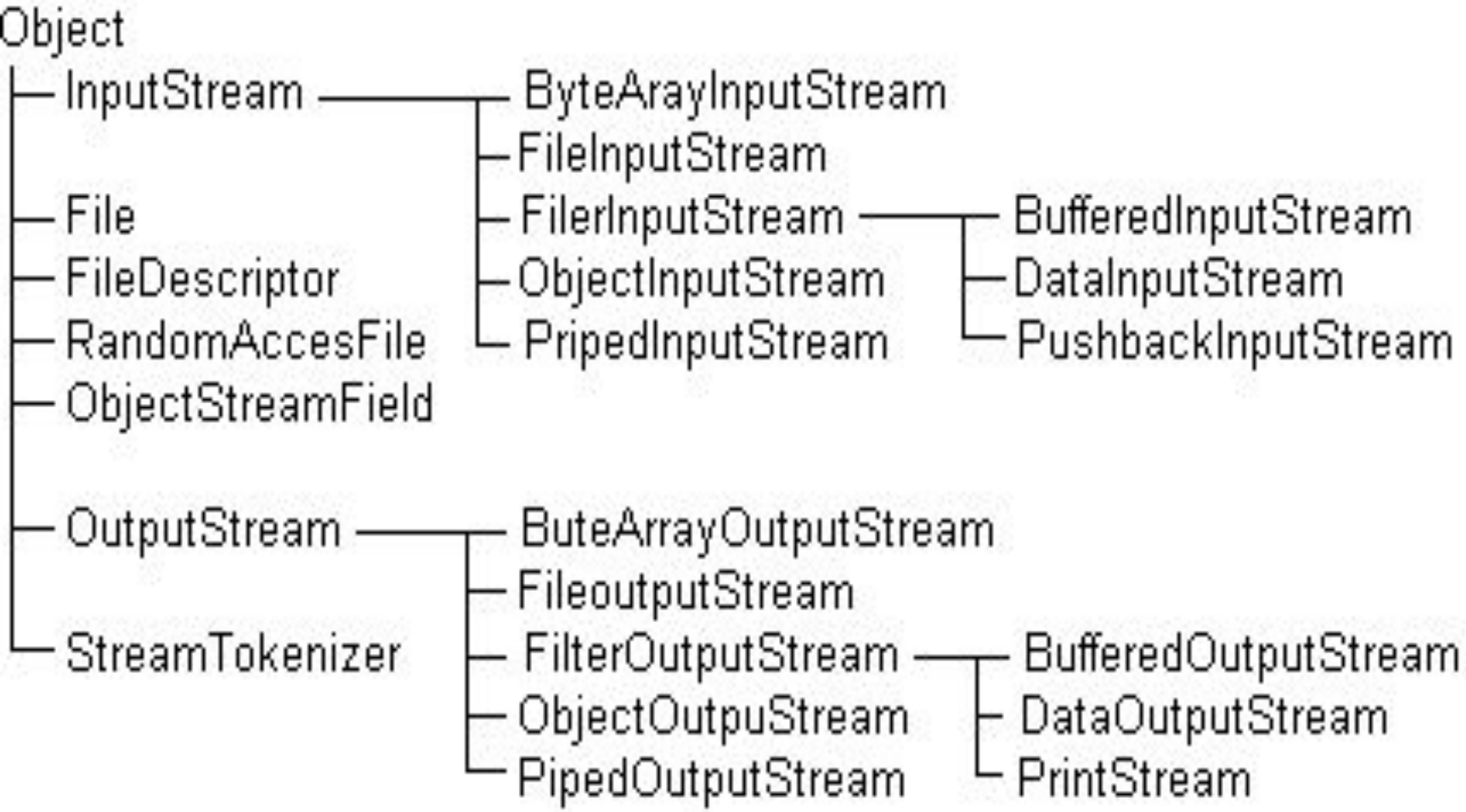
При этом элементы сначала накапливаются в буфере, в оперативной памяти, и выводятся в выходной поток только после того, как буфер заполнится.

По окончании работы с потоком его необходимо закрыть методом **close().**

Классы, входящие в иерархию символьных потоков ввода/вывода:



Классы, входящие в иерархию байтовых потоков потоков ввода/вывода:



Пример. Консольный ввод/вывод.

```
import java.io.*;
public class Main2 {
public static void main(String[] args) {
 try{
  BufferedReader br = new BufferedReader(
          new InputStreamReader(System.in, "Cp866"));
  PrintWriter pw = new PrintWriter(
     new OutputStreamWriter(System.out, "Cp866"),
                                                  true);
  //true –означает, что после вызова pw.println(…)
    можно не вызывать pw.flush().
  String s = "Привет, мир";
  System.out.println("System.out puts: " + s);
  pw.println("PrintWriter puts: " + s) ;
  int c = 0;
```

```
        pw.println("Посимвольный ввод:");
         while((c = br.read()) != -1)
          pw.println((char)c);
          pw.println("Построчный ввод:");
          do{
                s = br.readLine();
                pw.println(s);}
           while(!s.equals("q"));}
 catch(Exception e){};
    }
}
```

Пример. Работа с файлами:

```
import java.io.*;
class FileTest{
public static void main(String[] args){
 try{
   PrintWriter pw = new PrintWriter(
              new OutputStreamWriter(System.out, "Cp866"), true);
   File f = new File("FileTest.java");
   pw.println();
   pw.println("Файл \"" + f.getName() + "\" " + (f.exists()?"":"не ") +
                                                    "существует");
   pw.println("Вы " + (f.canRead()?"":"не ") +
                                                  "можете читать файл");
   pw.println("Вы " + (f.canWrite()?"":"не ") +
                                                "можете записывать в файл");
   pw.println("Длина файла " + f.length() + " б");
   pw.println() ;
```

```
  File d = new File("C:\\Windows");
  pw.println("Содержимое каталога:");
   if (d.exists() && d.isDirectory()) {
    String[] s = d.list();
    for (int i = 0; i < s.length; i++)
        pw.println(s[i]);
   }
  } catch(Exception e){};
 }
}
```

Пример. Буферизованный ввод/вывод.

```
import java.io.*;
 class FileTest1{
  public static void main(String[] args){
   try{
     BufferedReader br = new BufferedReader(
              new InputStreamReader(new FileInputStream("FileTest.java"),
                                                                    "Cp866"));
     BufferedWriter bw = new BufferedWriter(
          new OutputStreamWriter(new FileOutputStream("FileTest2.java"),

     "Cp866"));
      int c = 0;
      while ((c = br.read()) != -1){
        bw.write((char)c);
      }
      br.close();
      bw.close();
      System.out.println("The job's finished.");   }
   catch(Exception e){};
  }
}
```

Пример. Поток простых типов Java

```
import java.io.*;
class Data1{
 public static void main(String[] args) throws IOException{
   DataOutputStream dos = new DataOutputStream (
                                        new FileOutputStream("fib.txt"));
      int a = 1, b = 1, c = 1;
      for (int k = 0; k < 40; k++){
          System.out.print(b + " ");
          dos.writeInt(b);
          a = b;
          b = c;
          c = a + b;}
      dos.close();
      System.out.println("\n");
```

```
DataInputStream dis = new DataInputStream (
                              new FileInputStream("fib.txt"));
      while(true)
      try{
         a = dis.readInt();
         System.out.print(a + " ");}
      catch(Exception e){
         dis.close();
         System.out.println("End of file");
         System.exit(0);
      }
 }
}
```

Каналы обмена информацией

Канал обмена информацией строится следующим образом.

В одном процессе — источнике информации — создается объект класса PipedWriter или PipedOutputStream, в который записывается информация методами write() этих классов.

В другом процессе — приемнике информации — формируется объект класса PipedReader или PipedInputStream.

Он связывается с объектом-источником с помощью конструктора или специальным методом connect(), и читает информацию методами read().

Пример.

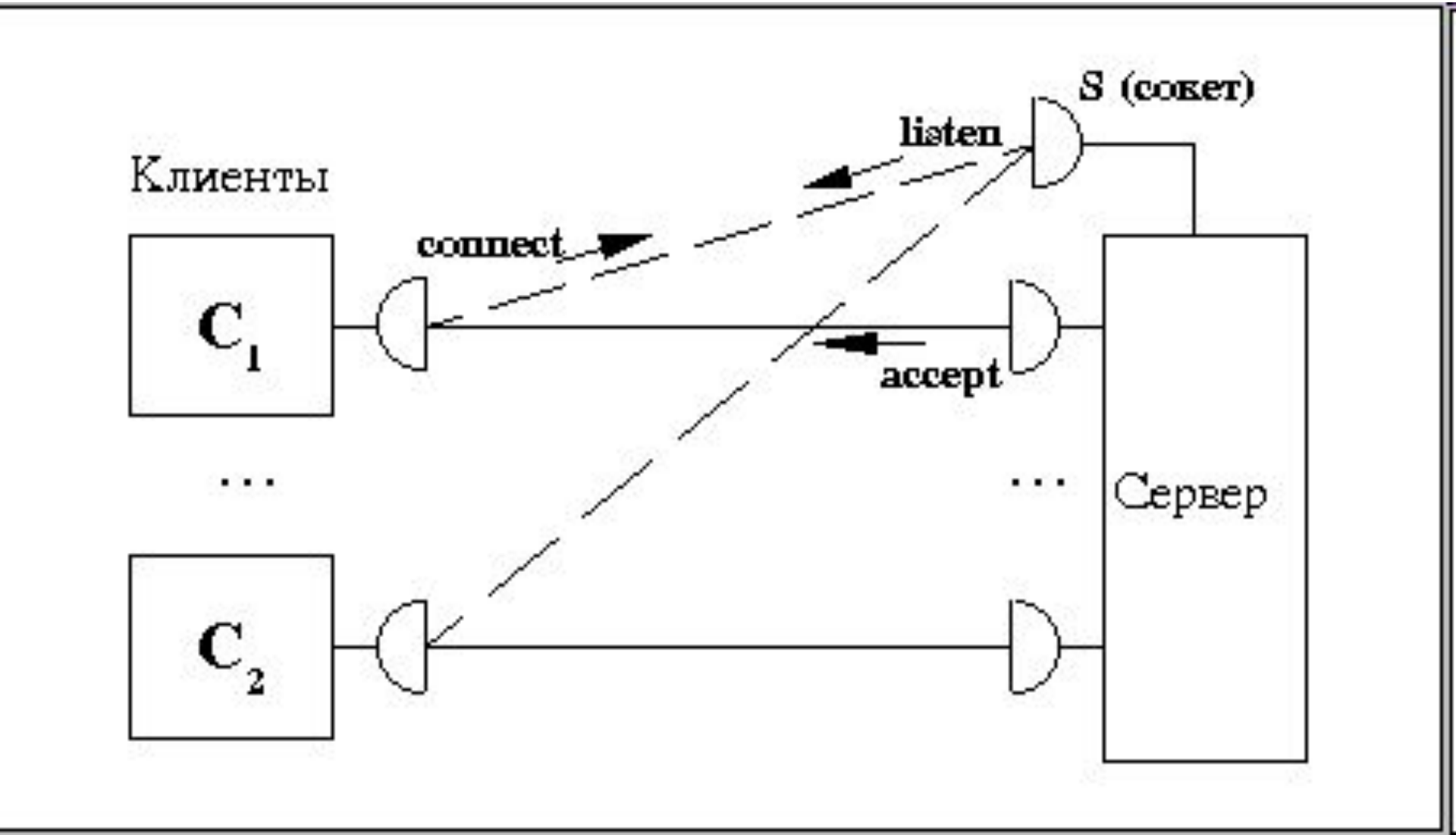
```
import java.io.*;
  class Target extends Thread{
        private PipedReader pr;
   Target(PipedWriter pw){
        try{
           pr = new PipedReader(pw);   }
        catch(Exception e){System.err.println("From Target(): " + e);}
    }
   PipedReader getStream(){ return pr;}
       public void run() {
         while(true)
           try{
             System.out.println("Reading: " + pr.read());}
           catch(Exception e){
             System.out.println("The job's finished.");
             System.exit(0);
           }
    }
}
```

```
class Source extends Thread{
  private PipedWriter pw;
  Source (){
     pw = new PipedWriter();}
  PipedWriter getStream(){ return pw;}
  public void run() {
     for (int k = 0; k < 10; k++)
       try{
          pw.write(k);
         System.out.println("Writing: " + k);}
       catch(Exception e){
         System.err.println("From Source.run(): " + e) ;}
} }
```

```
public class Pipe{
    public static void main(String[] args){
        Source s = new Source();
        Target t = new Target(s.getStream());
        s.start();
        t.start();
    }
}
```

Сокеты в Java

Процесс установления связи между сервером и клиентом имеет вид



В Java для сетевого программирования существует специальный пакет "java.net", содержащий класс java.net.Socket.

Гнёзда монтируются на порт хоста (port).

Порт обозначается числом от 0 до 65535 и логически обозначает место, куда можно пристыковать (bind) сокет.

Если порт на этом хосте уже занят каким-то сокетом, то ещё один сокет туда пристыковать уже не получится.

Таким образом, после того, как сокет установлен, он имеет вполне определённый адрес, символически записывающийся так [host]:[port], к примеру - 127.0.0.1:8888

Клиентский сокет

Сокет инициализируется следующим образом:

public Socket(String host, int port) throws UnknownHostException, IOException

Также полезной будет функция:

public void setSoTimeout(int timeout) throws SocketException

Эта функция устанавливает время ожидания (timeout) для работы с сокетом.

Если в течение этого времени никаких действий с сокетом не произведено (имеется ввиду получение и отправка данных), то он самоликвидируется.

Сокет сервера

Для инициализации сокета на сервере удобно использовать функцию

public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException

После установки сокета, вызывается функция

public Socket accept() throws IOException

Рассмотрим пример.

Клиент-серверное приложение.

Сервер устанавливает сокет на порт 3128, после чего ждёт входящих подключений.

Приняв новое подключение, сервер передаёт его в отдельный вычислительный поток.

В новом потоке сервер принимает от клиента данные, приписывает к ним порядковый номер подключения и отправляет данные обратно к клиенту.

TCP/IP клиент

```
import java.io.*;
import java.net.*;
class SampleClient extends Thread {
  public static void main(String args[]) {
 try {
// открываем сокет и коннектимся к localhost:3128
// получаем сокет сервера
Socket s = new Socket("localhost", 3128);

// берём поток вывода и выводим туда первый аргумент
// заданный при вызове, адрес открытого сокета и его
    порт
//Метод getHostAddress() из класса InetAddress
    возвращает IP хоста в текстовом виде.

args[0] = args[0]+" "+s.getInetAddress().getHostAddress()
    +":"+s.getLocalPort();
s.getOutputStream().write(args[0].getBytes());
```

```
// читаем ответ
byte buf[] = new byte[64*1024];
int r = s.getInputStream().read(buf);
String data = new String(buf, 0, r);
//выводим ответ в консоль
   System.out.println(data); }
catch(Exception e) {
//вывод исключений
  System.out.println("init error: "+e);} }
}
```

TCP/IP сервер

```
import java.io.*;
import java.net.*;
class SampleServer extends Thread {
Socket s;
int num;
public static void main(String args[]) {
  try {
    int i = 0; // счётчик подключений

// привинтить сокет на localhost, порт 3128
    ServerSocket server = new ServerSocket(3128, 0,
                                           InetAddress.getByName("localhost"));
    System.out.println("server is started");

// слушаем порт
    while(true) {
// ждём нового подключения, после чего запускаем обработку клиента
// в новый вычислительный поток и увеличиваем счётчик на единицу
       new SampleServer(i, server.accept());
       i++;
     }
}
```

```
  catch(Exception e) {
    System.out.println("init error: "+e);
  }
 }
public SampleServer(int num, Socket s) {
  // копируем данные
  this.num = num;
   this.s = s;
  // и запускаем новый вычислительный поток
  setDaemon(true);
   setPriority(NORM_PRIORITY);
   start();
}
```

```
public void run() {
 try {
   // из сокета клиента берём поток входящих данных
   InputStream is = s.getInputStream();
   // и оттуда же - поток данных от сервера к клиенту
   OutputStream os = s.getOutputStream();
   // буфер данных в 64 килобайта
  byte buf[] = new byte[64*1024];
  // читаем 64кб от клиента, результат - кол-во реально
     принятых данных
  int r = is.read(buf);
  // создаём строку, содержащую полученную от клиента
     информацию
  String data = new String(buf, 0, r);
  // добавляем данные об адресе сокета:
  data = ""+num+": "+" "+data;
  // выводим данные:
  os.write(data.getBytes());
  s.close(); // завершаем соединение
 }
  catch(Exception e) {System.out.println("init error: "+e);} } }
```

Запускаем сервер

java SampleServer

Запускаем клиент

java SampleClient test1

java SampleClient test2