Класс Locale

Представляет определенный географический, политический или культурный регион (местность).

Конструкторы:

```
public Locale (String language, // ISO 639
          String country); // ISO 3166
public Locale (String language,
          String country,
                String variant);
```

Класс Locale

Пример:

```
Locale current = new Locale("en", "US");
Locale loc = new Locale("ru", "RU", "koi8r");
```

Класс Locale

Методы:

```
public String getLanguage();
public String getCountry();
public String getVariant();
public static Locale getDefault();
public static void setDefault(Locale loc);
```

Класс Locale

Метод Locale.getDefault() возвращает значение Locale, используемое по умолчанию. Установить его можно следующим образом:

- с помощью системных свойств user.language и user.region
- с помощью метода Locale.setDefault()

Получить список возможных комбинаций языка и страны можно с помощью статического метода getAvailableLocales() различных классов, которые используют форматирование с учетом местных особенностей.

Например:

```
Locale list[] = DateFormat.getAvailableLocales();
```

Класс ResourceBundle

Абстрактный класс, предназначенный для хранения наборов зависящих от местности ресурсов.

Используется один из его потомков:

`ListResourceBundle`

или

`PropertyResourceBundle`

Класс ResourceBundle

Это набор связанных классов с единым базовым именем, и различающихся суффиксами, задающими язык, страну и вариант. Например:

```
MsgBundle
MsgBundle_ru
MsgBundle_en_US
MsgBundle_fr_CA_UNIX
```

Класс ResourceBundle

Основные методы:

```
public static final ResourceBundle getBundle(
 String name)
 throws MissingResourceException;

public static final ResourceBundle getBundle(
 String name, Locale locale)
 throws MissingResourceException;
```

- возвращают объект одного из подклассов ResourceBundle с базовым именем name и местностью, заданной объектом locale или взятой по умолчанию. При отсутствии ресурса осуществляется поиск подходящего путем последовательного исключения суффиксов, при неудаче инициируется исключение:

`MissingResourceException.`

Класс ListResourceBundle

Абстрактный класс, управляющий ресурсами с помощью списка. Используется созданием набора классов, расширяющих ListResourceBundle, для каждой поддерживаемой местности и определения метода getContents().

Класс ListResourceBundle

Пример:

```
public class MsgBundle_ru extends ListResourceBundle {
   public Object[][] getContents() {
    return contents;
   }
   public Object[][] contents = {
    { "greeting", "Привет!" },
    { "inquiry", "Как дела?" },
    { "farewell", "До свидания!" }
   };
}
```

Класс ListResourceBundle

Массив **`contents`** содержит список пар ключ-значение, причем ключ должен быть объектом типа **`String`**, а значение - **`Object`**.

Объект класса **`ListResourceBundle`** можно получить вызовом статического метода **`ResourceBundle.getBundle()`**.

```
ResourceBundle messages =
    ResourceBundle.getBundle(
       "MsgBundle",
       new Locale("ru", "RU"));
```

Класс ListResourceBundle

Поиск классов осуществляется в следующей последовательности:

1. **MsgBundle_ru_RU.class**
2. **MsgBundle_ru.class**
3. **MsgBundle.class**

Для получения значения объекта используется метод getObject:

```
String s = (String)
   messages.getObject("greeting");
```

Класс PropertyResourceBundle

Абстрактный класс, управляющий ресурсами с помощью набора свойств. Используется в случаях, когда локализуемые объекты имеют тип String. Ресурсы хранятся отдельно от кода, поэтому для добавления новых ресурсов не требуется перекомпиляция.

Пример файла Msg_ru.properties:

```
greeting = Привет!
inquiry = Как дела?
farewell = До свидания!
```

Класс PropertyResourceBundle

Объект класса `PropertyResourceBundle` можно получить вызовом статического метода `ResourceBundle.getBundle()`:

```
ResourceBundle messages =  ResourceBundle.getBundle( "Msg",
        new Locale("ru", "RU"));
```

Если `getBundle()` не может найти соответствующий класс, производится поиск файлов с расширением `.properties` в той же последовательности, как и для `ListResourceBundle`:

1. **Msg_ru_RU.properties**
2. **Msg_ru.properties**
3. **Msg.properties**

Класс PropertyResourceBundle

Для получения значения свойства используется метод getString:

```
String s = messages.getString("greeting");
```

Класс NumberFormat

Абстрактный класс, позволяющий форматировать числа, денежные единицы, проценты в соответствии с форматом, принятым в определенной местности.

Форматирование осуществляется в 2 этапа:

1. получение требуемого экземпляра класса с помощью одного из методов getNumberInstance, getCurrencyInstance, getPercentInstance.
2. вызов метода format() для получения отформатированной строки.

Список допустимых местностей, для которых определены форматы, можно получить с помощью статического метода

```
public Locale[] NumberFormat.getAvailableLocales();
```

Класс NumberFormat

Числа:

```
NumberFormat formatter =
 NumberFormat.getNumberInstance(
     Locale.GERMANY);
String result = formatter.format(123456.789);
```

Денежные единицы:

```
NumberFormat formatter =
 NumberFormat.getCurrencyInstance(
     Locale.FRANCE);
String result = formatter.format(4999.99);
```

Проценты:

```
NumberFormat formatter =
 NumberFormat.getPercentInstance(
     Locale.US);
String result = formatter.format(.75);
```

Класс DecimalFormat

Позволяет создавать собственные форматы для чисел, денежных единиц и процентов.

Порядок использования:

1. Вызывается конструктор с шаблоном в качестве аргумента

```
String pattern = "###,##0.##";
DecimalFormat formatter =
    new DecimalFormat(pattern);
```

2. Вызывается метод format() для получения отформатированной строки

```
String s = formatter.format(123123.456);
```

Класс DecimalFormat

Значения символов шаблона:

Символ	***Значение***
0	цифра
#	цифра, или пробел в случае нуля
.	десятичный разделитель
,	групповой разделитель
;	разделитель форматов
-	префикс отрицательного числа
%	процент (значение умножается на 100)
?	промилле (значение умножается на 1000)
¤	заменяется обозначением денежной единицы (международным если удвоен) и в формате вместо десятичного будет использован денежный разделитель
X	любой другой символ в префиксе или суффиксе
'	используется для экранирования специальных символов в префиксе или суффиксе

Класс DecimalFormatSymbols

Для изменения значения используемых по умолчанию разделителей в классе DecimalFormat используется класс DecimalFormatSymbols.

Конструкторы:

```
public DecimalFormatSymbols();
public DecimalFormatSymbols(Locale locale);
```

Класс DecimalFormatSymbols

Методы:

```
public void setZeroDigit(char c);
public void setGroupingSeparator(char c);
public void setDecimalSeparator(char c);
public void setPerMill(char c);
public void setPercent(char c);
public void setDigit(char c);
public void setNaN(char c);
public void setInfinity(char c);
public void setMinusSign(char c);
public void setPatternSeparator(char c);
```

Имеются соответствующие методы get() для получения установленных значений.

Класс DecimalFormatSymbols

Для передачи значений разделителей объект **`DecimalFormatSymbols`** передается конструктору класса **`DecimalFormat`** в качестве аргумента:

```
DecimalFormatSymbols symbols =
 new DecimalFormatSymbols();
symbols.setGroupingSeparator(" ");
DecimalFormat formatter =
 new DecimalFormat("#,##0.00", symbols);
String s = formatter.format(4.50);
```

Класс DateFormat

Абстрактный класс, позволяющий форматировать дату и время в соответствии с форматом, принятым в определенной местности.

Список допустимых местностей, для которых определены форматы, можно получить с помощью статического метода

```
public Locale[] DateFormat.getAvailableLocales();
```

Стили форматирования определяются константами:

DateFormat.DEFAULT, DateFormat.SHORT, DateFormat.MEDIUM, DateFormat.LONG, DateFormat.FULL

Класс DateFormat

Форматирование осуществляется в 2 этапа:

1. Получение требуемого экземпляра класса с помощью одного из методов
   - `getDateInstance`
   - `getTimeInstance`
   - `getDateTimeInstance`
2. Вызов метода format() для получения отформатированной строки.

Класс DateFormat

Дата:

```
DateFormat fmt= DateFormat.getDateInstance(
 DateFormat.SHORT, Locale.UK);
String result = fmt.format(new Date());
```

Время:

```
DateFormat fmt= DateFormat.getTimeInstance(
 DateFormat.LONG, Locale.FRANCE);
String result = fmt.format(new Date());
```

Дата и время:

```
DateFormat f= DateFormat.getDateTimeInstance(
 DateFormat.FULL, DateFormat.FULL, Locale.US);
String result = f.format(new Date());
```

Класс SimpleDateFormat

Позволяет создавать форматы для даты и времени.

Порядок использования:

1. Вызывается конструктор с шаблоном в качестве аргумента:

```
SimpleDateFormat f= new SimpleDateFormat(
    "K:mm EEE MMM d ''yy");
```

2. Вызывается метод format() для получения отформатированной строки:

```
String s = f.format(new Date());
```

Класс SimpleDateFormat

Шаблоны SimpleDateFormat

Символ	Значение	Тип	Пример
G	обозначение эры	текст	AD
y	год	число	1996
M	месяц года	текст/число	July или 07
d	число месяца	число	23
h	часы (1-12)	число	5
H	часы (0-23)	число	22
m	минуты	число	45
s	секунды	число	31
S	миллисекунды	число	978
E	день недели	текст	Tuesday
D	номер дня в году	число	189
F	день недели в месяце	число	2 (2nd Wed in July)
w	неделя в году	число	27
W	неделя в месяце	число	2
a	знак AM/PM	текст	PM
k	часы (1-24)	число	24
K	часы (0-11)	число	0
z	временная зона	текст	GMT

Класс DateFormatSymbols

Используется для изменения названий месяцев, дней недели и других значений в классе SimpleDateFormat.

Конструкторы:

```
public DateFormatSymbols();
public DateFormatSymbols(Locale locale);
```

Класс DateFormatSymbols

Методы:

```
public void setEras(String newValue[]);
public void setMonths(String newValue[]);
public void setShortMonths(String newValue[]);
public void setWeekDays(String newValue[]);
public void setShortWeekDays(String newValue[]);
public void setAmPmStrings(String newValue[]);
public void setZoneStrings(String newValue[]);
public void setPatternChars(String newValue[]);
```

Имеются соответствующие методы get() для получения установленных значений.

Класс DateFormatSymbols

Для передачи значений разделителей объект DateFormatSymbols передается конструктору класса SimpleDateFormat в качестве аргумента:

```
DateFormatSymbols symbols =
 new DateFormatSymbols();
String weekdays[] = {"Пн","Вт","Ср","Чт","Пт","
Сб","Вс"};
symbols.setShortWeekDays(weekdays);
SimpleDateFormat f=
 new SimpleDateFormat("E", symbols);
String s = f.format(new Date());
```

Класс MessageFormat

Используется для выдачи сообщений на различных языках с включением изменяющихся объектов.

Использование класса:

1. Выделение переменных объектов в сообщении:

```
At 1:15 PM on April 13, 1998, we detected 7 spaceships
on the planet Mars.
```

2. Помещение шаблона сообщения в ResourceBundle:

```
ResourceBundle messages =
    ResourceBundle.getBundle(
        "MessageBundle", currentLocale);
```

Класс MessageFormat

содержимое файла **MessageBundle.properties**:

```
template = At {2,time,short} on {2,date,long}, we detected
{1,number,integer} spaceships on the planet {0}.
planet = Mars
```

3. Установка аргументов сообщения:

```
Object[] args = {
 messages.getString("planet"),
 new Integer(7),
 new Date()
}
```

Класс MessageFormat

4. Создание объекта MessageFormat:

```
MessageFormat formatter =
 new MessageFormat(
    messages.getString("template"));
formatter.setLocale(currentLocale);
```

5. Форматирование сообщения:

```
String s = formatter.format(args);
```

Класс MessageFormat

Синтаксис аргументов MessageFormat:

```
{ индекс аргумента, [ тип, [ стиль ] ] }
```

Индекс задает порядковый индекс аргумента в массиве объектов (0-9).

Типы и стили аргументов:

Возможные типы	Возможные стили
number	currency, percent, integer, шаблон числа
date	short, long, full, medium, шаблон даты
time	short, long, full, medium, шаблон времени

Класс ChoiceFormat

Используется для задания возможности выбора различных элементов в зависимости от значения параметров.

Использование класса:

1. Выделение переменных объектов в сообщении:

```
There are no files on disk C.
There is one file on disk C.
There are 3 files on disk C.
```

Класс ChoiceFormat

2. Помещение шаблона сообщения в ResourceBundle:
содержимое файла MessageBundle.properties:

```
template = There {0} on disk {1}.
no = are no files
one = is one file
many = are {2} files
```

3. Создание объекта ChoiceFormat:

```
double limits[] = {0,1,2}
String choices[] = { messages.getString("no"),
          messages.getString("one"),
          messages.getString("many") }
ChoiceFormat choice =
 new ChoiceFormat(limits, choices);
```

Класс ChoiceFormat

4. Создание объекта MessageFormat:

```
MessageFormat formatter =
 new MessageFormat(
    messages.getString("template"));
formatter.setLocale(currentLocale);
Format[] formats = { choice, null,
       NumberFormat.getInstance()};
formatter.setFormats(formats);
```

5. Установка аргументов сообщения:

```
Object[] args = { 1, "C", 1 };
```

6. Форматирование сообщения

```
String s = formatter.format(args);
```

Ввод/вывод

Класс File

Для работы с физическим файлами и каталогами на внешних носителях, в Java используются классы из пакета `java.io`.

Класс `File` предназначен для хранения и обработки каталогов и имен файлов. Не содержит методы для работы с содержимым файла, но позволяет манипулировать такими свойствами, как права доступа, дата и время создания, путь в иерархии каталогов, создание, удаление файла, изменение его имени и каталога и т.д.

Класс File

Примеры создания объектов класса File:

```
File myFile = new File(”\\com\\myfile.txt”);
File myDir = new File( ”c:\\jdk1.7.0\\src\\java\\io”);
File myFile = new File(myDir, ”File.java”);
File myFile = new File(”c:\\com”, ”myfile.txt”);
File myFile = new File(
   new URI(”htmp://www.bsu.by/index.html”));
```

При создании объекта класса File любым из конструкторов компилятор не выполняет проверку на существование физического файла с заданным путем.

Класс File

Когда неизвестно, в какой системе будет выполняться код, предусмотрены специальные поля в классе File:

```
public static final String separator;
public static final char separatorChar;
```

С помощью этих полей можно задать путь, универсальный в любой системе:

```
File myFile = new File(
 File.separator + ”com” + File.separator +
 ”myfile.txt” );
```

Класс File

Предусмотрен еще один тип разделителей – для директорий переменной PATH:

```
public static final String pathSeparator;
public static final char pathSeparatorChar;
```

К примеру, для ОС Unix значение `pathSeparator=":"`, а для Windows – `pathSeparator=";"`.

Класс File

Пример:

```
/* FileTest.java */
package fpack;
import java.io.*;
import java.util.*;

public class FileTest {
  public static void main(String[] args) {
  //с объектом типа File ассоциируется файл на диске FileTest2.java
    File fp =
        new File("fpack"+ File.separator+ "FileTest2.java");
    if (fp.exists()) {
      System.out.println(fp.getName() + " существует");
```

Класс File - пример

```
if(fp.isFile()) {   //если объект - дисковый файл
  System.out.println("Путь к файлу:\t"+ fp.getPath());
  System.out.println("Абсолютный путь:\t"+
                     fp.getAbsolutePath());
  System.out.println("Размер файла:\t"+ fp.length());
  System.out.println("Последняя модификация :\t"+
                      new Date(fp.lastModified()));
  System.out.println("Файл доступен для чтения:\t"+
                     fp.canRead());
  System.out.println("Файл доступен для записи:\t"+
                     fp.canWrite());
  System.out.println("Файл удален:\t"+ fp.delete());
}
```

Класс File - пример

```
} else
  System.out.println("файл "+ fp.getName()+
                     " не существует");
  try {
    if ( fp.createNewFile())
      System.out.println("Файл "+ fp.getName()+
                         " создан");
  } catch(IOException e) {
    System.err.println(e);
  }

  //в объект типа File помещается каталог\директория
  // в корне проекта должен быть создан каталог
com.learn
  // с несколькими файлами

  File dir = new File("com"+ File.separator+ "learn");
```

Класс File - пример

```
    if (dir.exists() && dir.isDirectory())
      /*если объект является каталогом и если этот каталог
   существует*/
      System.out.println("каталог "+ dir.getName()+ " существует");
    File[] files = dir.listFiles();
    for(int i = 0; i < files.length; i++) {
      Date date = new Date(files[i].lastModified());
      System.out.print("\n"+ files[i].getPath()+ " \t| "+
                   files[i].length()+ "\t| "+ date.toString());
    }
    // метод listRoots() возвращает доступные корневые каталоги
    File root = File.listRoots()[1];
    System.out.printf("\n%s %,d из %,d свободно.",
                      root.getPath(), root.getUsableSpace(),
                      root.getTotalSpace());
  }
}
```

Класс File - пример

В результате файл `FileTest2.java` будет очищен, а на консоль выведено:

```
FileTest2.java существует
Путь к файлу: pack\FileTest2.java
Абсолютный путь: D:\workspace\pack\FileTest2.java
Размер файла: 2091
Последняя модификация : Fri Mar 31 12:26:50 EEST 2010
Файл доступен для чтения: true
Файл доступен для записи: true
Файл удален: true
Файл FileTest2.java создан
каталог learn существует
com\learn\bb.txt | 9 | Fri Mar 24 15:30:33 EET 2010
com\learn\byte.txt| 8 | Thu Jan 26 12:56:46 EET 2010
com\learn\cat.gif | 670 | Tue Feb 03 00:44:44 EET 2011
C:\ 3 665 334 272 из 15 751 376 896 свободно.
```

Класс File

Каталог класса `File` имеет дополнительное свойство - просмотр списка имен файлов с помощью методов `list()`, `listFiles()`, `listRoots()`.

Потоки ввода-вывода

Поток данных (stream) – это абстрактный объект для получения или передачи данных единым способом, независимо от связанного с потоком источника или приемника данных.

Иерархия потоков в Java

Потоки реализуются классами пакета java.io.

Делятся на две больших группы — потоки ввода, и потоки вывода. Потоки ввода связаны с источниками данных, потоки вывода — с приемниками данных.

Кроме того, потоки можно разделить на байтовые и символьные. Единицей обмена для байтовых потоков является байт, для символьных — символ Unicode.

Иерархия потоков в Java

Базовые потоки ввода:

- **`InputStream`** – байтовый
- **`Reader`** -- символьный

Базовые потоки вывода:

- **`OutputStream`** – байтовый
- **`Writer`** -- символьный

Кроме этих основных потоков, в пакет входят специализированные потоки, предназначенные для работы с различными источниками или приемниками данных, а также преобразующие потоки, предназначенные для преобразования информации, поступающей на вход потока, и выдачи ее на выход в преобразованном виде.

Класс InputStream

Класс InputStream – это абстрактный входной поток байтов, предок для всех входных байтовых потоков.

Конструктор:

`InputStream();`

`Создает входной байтовый поток.`

Методы:

`abstract int read() throws IOException;`

Читает очередной байт данных из входного потока. Значение должно быть от 0 до 255. При достижении конца потока возвращается -1. При ошибке ввода-вывода генерируется исключение.

Класс InputStream

```
int read(byte[] buf);
```

Читает данные в буфер и возвращает количество прочитанных байтов.

```
int read(byte[] buf, int offset, int len);
```

Читает не более len байтов в буфер, заполняя его со смещением offset, и возвращает количество прочитанных байтов

```
void close();
```

Закрывает поток.

Класс InputStream

```
int available();
```

Возвращает количество доступных на данный момент байтов для чтения из потока.

```
long skip(long n);
```

Пропускает указанное количество байтов из потока.

```
boolean markSupported();
```

Проверка на возможность повторного чтения из потока.

Класс InputStream

`void mark(int limit);`

Устанавливает метку для последующего повторного чтения. limit – размер буфера для операции повторного чтения.

`void reset();`

Возвращает указатель потока на предварительно установленную метку. Дальнейшие вызовы метода read() будут снова возвращать данные, начиная с заданной метки.

Класс OutputSrteam

Класс OutputSrteam – это абстрактный выходной поток байтов, предок для всех выходных байтовых потоков.

Конструктор:

```
OutputSrteam();
```

Создает выходной байтовый поток.

Класс OutputSrteam

Методы:

```
abstract void write(int n) throws IOException;
```

Записывает очередной байт данных в выходной поток. Значащими являются 8 младших битов, старшие - игнорируются. При ошибке ввода-вывода генерируется исключение.

```
void write(byte[] buf);
```

Записывает в поток данные из буфера.

Класс OutputSrteam

```
void write(byte[] buf, int offset, int len);
```

Записывает в поток len байтов из буфера, начиная со смещения offset.

```
void close();
```

Закрывает поток.

```
void flush();
```

Заставляет освободить возможный буфер потока, отправляя на запись все записанные в него данные.

Класс Reader

Абстрактный входной поток символов, предок для всех входных символьных потоков.

Конструктор:

```
Reader();
```

Создает входной символьный поток

Класс Reader

Методы:

```
abstract int read() throws IOException;
```

Читает очередной символ Unicode из входного потока. При достижении конца потока возвращается -1. При ошибке ввода-вывода генерируется исключение.

```
int read(char[] buf);
```

Читает данные в буфер и возвращает количество прочитанных символов.

Класс Reader

```
int read(char[] buf, int offset, int len);
```

Читает не более len символов в буфер, заполняя его со смещением offset, и возвращает количество прочитанных символов

```
void close();
```

закрывает поток

```
int available();
```

возвращает количество доступных на данный момент символов для чтения из потока

Класс Reader

```
long skip(long n);
```

пропускает указанное количество символов из потока

```
boolean markSupported();
```

проверка на возможность повторного чтения из потока

```
void mark(int limit);
```

устанавливает метку для последующего повторного чтения. limit – размер буфера для операции повторного чтения

Класс Reader

```
void reset();
```

возвращает указатель потока на предварительно установленную метку. Дальнейшие вызовы метода read() будут снова возвращать данные, начиная с заданной метки.

Класс Writer

Абстрактный выходной поток символов, предок для всех выходных символьных потоков.

Конструктор:

```
Writer();
```

создает выходной символьный поток

Класс Writer

Методы:

```
abstract void write(int n) throws IOException;
```

Записывает очередной символ Unicode в выходной поток. Значащими являются 16 младших битов, старшие - игнорируются. При ошибке ввода-вывода генерируется исключение.

```
void write(char[] buf);
```

Записывает в поток данные из буфера.

Класс Writer

```
void write(char[] buf, int offset, int len);
```

Записывает в поток len символов из буфера, начиная со смещения offset

```
void close();
```

закрывает поток

```
void flush();
```

заставляет освободить возможный буфер потока, отправляя на запись все записанные в него данные.

Специализированные потоки

Конструкторы этих потоков в качестве аргумента принимают ссылку на источник или приемник данных - файл, массив, строку. Методы для чтения и записи данных - read() для входных потоков, write() -для выходных потоков.

Конвейер имеет особенность, что источником данных для входного конвейера является выходной конвейер, и наоборот. Обычно конвейеры используются для обмена данными между двумя потоками выполнения (Thread).

Специализированные потоки

В пакет java.io входят потоки для работы со следующими основными типами источников и приемников данных:

Файл:

`FileInputStream` - входной байтовый поток

`FileOutputStream` - выходной байтовый поток

`FileReader` - входной символьный поток

`FileWriter` - выходной символьный поток

Специализированные потоки

Массив:

- `ByteArrayInputStream` - входной байтовый поток
- `ByteArrayOutputStream` - выходной байтовый поток
- `CharArrayReader` - входной символьный поток
- `CharArrayWriter` - выходной символьный поток

Специализированные потоки

Строка:

- StringReader - входной символьный поток
- StringWriter - выходной символьный поток

Специализированные потоки

Конвейер:

- `PipedInputStream` - входной байтовый поток
- `PipedOutputStream` - выходной байтовый поток
- `PipedReader` - входной символьный поток
- `PipedWriter` - выходной символьный поток

Специализированные потоки

Пример чтения данных из файла:

```
FileReader f = new FileReader(“myfile.txt”);
char[] buffer = new char[512];
f.read(buffer);
f.close();
```

Преобразующие потоки

Этот тип потоков выполняет некие преобразования над данными других потоков. Конструкторы таких классов в качестве аргумента принимают поток данных.

Преобразующие потоки

Классы **`BufferedInputStream`**, **`BufferedOutputStream`**, **`BufferedReader`** и **`BufferedWriter`** предназначены для буферизации ввода-вывода. Они позволяют читать и записывать данные большими блоками. При этом обмен данными со стороны приложения ведется с буфером, а по мере необходимости в буфер из источника данных подгружается новая порция данных, либо из буфера данные переписываются в приемник данных.

Класс **`BufferedReader`** имеет дополнительный метод **`readLine()`** для чтения строки символов, ограниченной разделителем строк.

Класс **`BufferedWriter`** имеет дополнительный метод **`newLine()`** для вывода разделителя строк.

Преобразующие потоки

Классы **InputStreamReader** и **OutputStreamWriter** предназначены для преобразования байтовых потоков в символьные и наоборот. Кодировка задается в конструкторе класса. Если она опущена, то используется системная кодировка, установленная по умолчанию).

В конструктор класса **InputStreamReader** передается как аргумент объект класса **InputStream**, а в конструктор класса **OutputStreamWriter** – объект класса **OutputStream**. Методы **read()** и **write()** этих классов аналогичны методам классов **Reader** и **Writer**.

Преобразующие потоки

Пример использования:

Вариант 1:

```
FileInputStream f = new FileInputStream(“myfile.txt”);
InputStreamReader isr = new InputStreamReader(f);
BufferedReader br = new BufferedReader(isr);
br.readLine();
```

Вариант 2:

```
BufferedReader br = new BufferedReader(
          new InputStreamReader(
          new FileInputStream(“myfile.txt”)));
br.readLine();
```

Преобразующие потоки

Классы `DataInputStream` и `DataOutputStream` предназначены для записи и чтения примитивных типов данных и содержат методы `readBoolean()`, `readInt()`, `readDouble()`, `writeFloat()`, `writeByte()` и другие подобные методы.

Для успешного чтения таких данных из потока `DataInputStream` они должны быть предварительно записаны с помощью соответствующих методов `DataOutputStream` в том же порядке.

Преобразующие потоки

Классы `PrintStream` и `PrintWriter` предназначены для форматированного вывода в поток вывода. В них определено множество методов `print()` и `println()` с различными аргументами, которые позволяют напечатать в поток аргумент, представленный в текстовой форме (с использованием системной кодировки).

В качестве аргумента может использоваться любой примитивный тип данных, строка и любой объект. Методы `println` добавляют в конце разделитель строк.

Стандартные потоки

Класс `java.lang.System` содержит 3 поля, представляющих собой стандарные консольные потоки:

`InputStream System.in` – стандартный поток ввода

`PrintStream System.out` - стандартный поток вывода

`PrintStream System.err` - стандартный поток ошибок

Стандартные потоки

Имеется возможность перенаправлять данные потоки с помощью методов `System.setIn, System.setOut, System.setErr.`

Пример чтения данных с клавиатуры и вывода в окно терминала:

```
BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in));
String s = br.readLine();
System.out.println("Введена строка : " + s);
System.out.println("Длина строки : " + s.length);
```

java.lang.AutoClosable и java.io.Closable

```
public interface AutoCloseable {
   void close() throws Exception;
}

public interface Closeable extends AutoCloseable {
    public void close() throws IOException;
}
```

try с ресурсами

Для классов реализующих интерфейс AutoClosable допустима специальная форма оператора try:

```
try ( спецификация ресурса
        [; спецификация ресурса] ) {
}
[блоки catch и finally]
```

До выхода из блока try, метод close() вызывается автоматически для ресурсов объявленных в try.

try с ресурсами

Пример:

```
try (FileInputStream fin = new FileinputStream(
                                        args[0]);
      FileOutputStream fout = new FileOutputStream(
                                        args[1]))
{
   while( true ) {
       int i = fin.read();
       if ( i == -1 ) break;
       fout.write( i );
   }
}
// files are closed here
```

Сериализация объектов

Сериализация объектов - запись объекта со всеми полями и ссылками на другие объекты в виде последовательности байтов в поток вывода с последующим воссозданием (десериализацией) копии этого объекта путем чтения последовательности байтов сохраненного объекта из потока ввода.

Сериализация объектов

Интерфейс java.io.Serializable:

Интерфейс-метка, указывающий на то, что реализующий его класс может быть сериализован. Поля класса, не требующие сериализации, должны иметь модификатор `transient`.

При использовании Serializable десериализация происходит следующим образом: под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор объекта при этом не вызывается.

Сериализация объектов

При сериализации объекта класса, реализующего интерфейс Serializable, учитывается порядок объявления полей в классе. Поэтому при изменении порядка десериализация пройдет некорректно. На стадии компиляции, в каждый класс реализующий Serializable добавляется поле

```
private static final long serialVersionUID;
```

Это поле содержит уникальный идентификатор версии сериализованного класса. Оно вычисляется по содержимому класса – полям, их порядку объявления, методам, их порядку объявления.

Сериализация объектов

Это поле записывается в поток при сериализации класса. Это единственный случай, когда static-поле сериализуется.

При десериализации значение этого поля сравнивается с имеющимся у класса в виртуальной машине. Если значения не совпадают, инициируется исключение java.io.InvalidClassException. Соответственно, при любом изменении в классе это поле поменяет свое значение.

Вместо реализации интерфейса Serializable можно реализовать Externalizable.

Интерфейс java.io.Externalizable

Предназначен для реализации классами, которым требуется нестандартное поведение при сериализации. В интерфейсе описаны 2 метода:

```
void writeExternal(ObjectOutput out);
void readExternal(ObjectInput in);
```

При использовании этого интерфейса в поток автоматически записывается только идентификация класса. Сохранить и восстановить всю информацию о состоянии экземпляра должен сам класс. Для этого должны быть переопределены методы writeExternal() и readExternal() интерфейса Externalizable. Эти методы должны обеспечить сохранение состояния, описываемого полями самого класса и его суперкласса.

Интерфейс java.io.Externalizable

При восстановлении Externalizable-объекта экземпляр создается вызовом конструктора без аргументов, затем вызывается метод readExternal(), поэтому в классе должен быть пустой конструктор. Для сохранения состояния вызываются методы ObjectOutput, с помощью которых можно записать как примитивные, так и объектные значения.

Для чтения и записи в поток значений отдельных полей объекта можно использовать соответственно методы внутренних классов:

`ObjectInputStream.GetField`

`ObjectOutputStream.PutField.`