Исключения

```
package testExceptions;
import ncStudying.DataOnly;
public class Test1 {
public static int doSomething(DataOnly d){
   return d.k;}
public static void main(String[] arg){
  doSomething(null);
}}
```

Исключениями или исключительными ситуациями (состояниями) называются ошибки, возникшие в программе во время её работы. Все исключения в Java являются объектами. Поэтому они могут порождаться не только автоматически при возникновении исключительной ситуации, но и создаваться самим разработчиком.

```
public class Test1 {
public int doSomething(DataOnly d){
if (d==null) throw new IllegalArgumentException();
else return d.k;}
public static void main(String[] arg){
new Test1().doSomething(null);}}
```

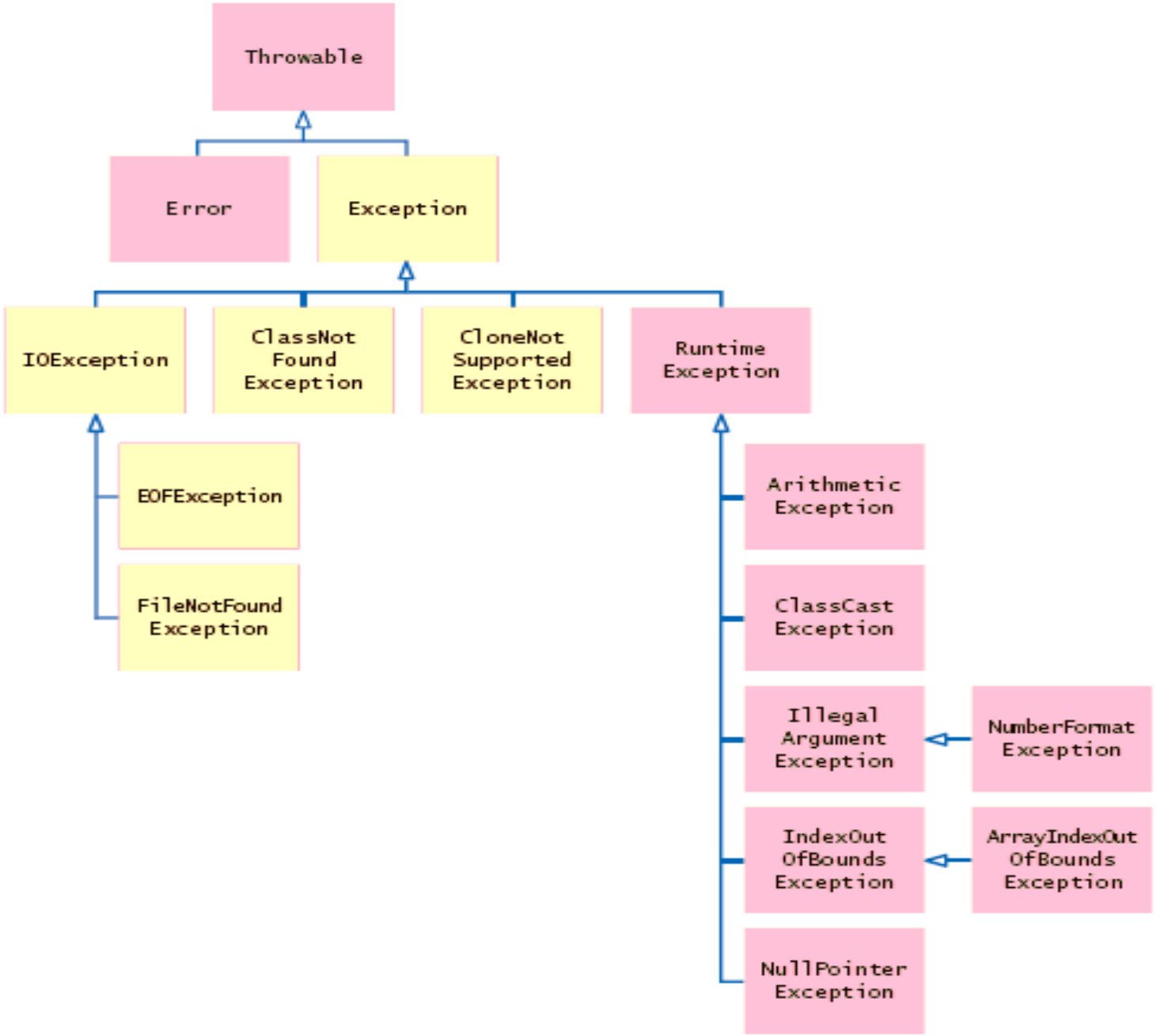
Исключения

Если вы «находитесь» внутри метода и инициируете исключение (или это делает другой вызванный метод), этот метод завершит работу при возникновении исключения. Но если вы не хотите, чтобы оператор ***throw*** завершил работу метода, разместите в методе специальный блок для перехвата исключения — так называемый блок ***try***. Этот блок представляет собой простую область действия, которой предшествует ключевое слово **try.**

Обработчик исключений, создается для каждого исключения, которое вы хотите перехватить. Обработчики исключений размещаются прямо за блоком *try* и обозначаются ключевым словом *catch*:

```
try {
    // Часть программы, способная возбуждать исключения
    } catch(Type1 id1) {
    // Обработка исключения Type1
    } catch(Type2 id2) {
    // Обработка исключения Type2
    } catch(Type3 id3) {}
    // Обработка исключения Type3
```

Иерархия исключений



Исключения

```
try {
    } catch(IOException ex) {

    } catch(FileNotFoundException ex) {
    }
```

Порядок, в котором появляются обработчики очень важен, поскольку, обработка передается первому подходящему обработчику.

Среди исключений можно выделить ошибки, проверенные и непроверенные (checked/unchecked) исключения.

Error – критические ошибки, который могут возникнуть в системе (например, StackOverflowError). Как правило обрабатывает их система. Если они возникают, то приложение закрывается, так как при данной ситуации работа не может быть продолжена.

Exception – это проверенные исключения. Это значит, что если метод бросает исключение, которое унаследовано от Exception (напр. IOException), то этот метод должен быть обязательно заключен в блок try-catch. Проверенные (checked) исключения означают, что исключение можно было предвидеть и, соответственно, оно должно быть обработано, работа приложения должна быть продолжена. Пример такого исключения — это попытка создать новый файл, который уже существует (IOException). В данному случае, работа приложения должна быть продолжена и пользователь должен получить уведомление, по какой причине файл не может быть создан.

RuntimeException – это непроверенные исключения. Они возникают во время выполнения приложения. К таким исключениям относится, например, NullPointerException. Они не требуют обязательного заключения в блок try-catch. Когда RuntimeException возникает, это свидетельствует о ошибке, допущенной программистом. Поэтому данное исключение не нужно обрабатывать, а нужно исправлять ошибку в коде

Регистрация исключений

```
public class TestExRegist {
  public void createFile(String st){
    if (st.equals("con"))
            throw new IllegalArgumentException();
   }
  public void createFile2(String st) {
    if (st.equals("con"))
             throw new IOException();
     }}
```

В языке *Java* необходимо сообщать программисту, вызывающему ваш метод, об исключениях, которые данный метод способен возбуждать.

```
 public void createFile2(String st) throws IOException {
    if (st.equals("con")) throw new IOException();}
```

Обойти спецификацию исключений невозможно — если ваш метод возбуждает исключения и не обрабатывает их, компилятор предложит либо обработать исключение, либо включить его в спецификацию.

Трассировка стека

```
static void f() {
   try {
     throw new Exception();
   } catch (Exception e) {
     for(StackTraceElement ste : e.getStackTrace())
       System.out.println(ste.getMethodName());}}
 static void g() { f(); }
 static void h() { g(); }
 public static void main(String[] args) {
  f();
  System.out.println("----------------------------------");
  g();
  System.out.println("----------------------------------");
  h();}
```

Метод **getStackTrace*()***. возвращает массив элементов трассировки, каждый из которых представляет один кадр стека. Нулевой элемент представляет вершину стека, то есть последний вызванный метод последовательности (точка, в которой был создан и инициирован объект ***Throwable***).

Создание собственных исключений

Для создания собственного класса исключения необходимо определить его производным от уже существующего типа — желательно наиболее близкого к ситуации (хоть это и не всегда возможно). В простейшем случае создается класс с конструктором по умолчанию:

```
class SimpleException extends Exception {}
```

Прежде всего, нужно четко определить ситуации, в которых будет возникать ваше собственное исключение.

После этого можно написать класс-исключение. Его имя, по соглашению, должно завершаться словом Exception. Как правило, этот класс состоит только из двух конструкторов и переопределения методов **tostring()** и **getMessage()**.

Цепочки исключений

Зачастую необходимо перехватить одно исключение и возбудить следующее, не потеряв при этом информации о первом исключении — это называется цепочкой исключений (*exception chaining*). Конструкторам всех подклассов ***Throwable*** может передаваться объект-причина (*cause*). Предполагается, что причиной является изначальное исключение и передача ее в новый объект обеспечивает трассировку стека вплоть до самого его начала, хотя при этом создается и возбуждается новое исключение. Единственными подклассами класса ***Throwable***, принимающими объект-причину в качестве аргумента конструктора, являются три основополагающих класса исключений: ***Error*** (используется виртуальной машиной (*JVM*) для сообщений о системных ошибках), ***Exception*** и ***RuntimeException***. Для организации цепочек из других типов исключений придется использовать метод ***initCause()***, а не конструктор.

Цепочки исключений

```
public class TestExceptionChaining{
    public int doSomething(DataOnly d){
      if (d==null) throw new IllegalArgumentException();
      else return d.k;}

    public void doAnotherThing(DataOnly d){
        try{
            doSomething(d);
        }
        catch (IllegalArgumentException e){
            System.out.println("Illegal");
            throw new RuntimeException(e);
        }

    }
   public static void main(String[] arg){
    TestExceptionChaining t = new TestExceptionChaining();
    t.doAnotherThing(null);
  }
}
```

finally

```
try {
    } catch(A al) {
       // Обработчик для ситуации A
    } catch(B bl) {
       // Обработчик для ситуации B
    } catch(C cl) {
       // Обработчик для ситуации C
    } finally {
      // Действия, производимые в любом случае
    }
```

Finally выполняться независимо от того, было или нет возбуждено исключение внутри блока try.

Блок ***finally*** необходим тогда, когда в исходное состояние вам необходимо вернуть что-то другое, а не память. Это может быть, например, открытый файл или сетевое подключение, подключение к БД

finally

Поскольку секция ***finally*** выполняется всегда, важные завершающие действия будут выполнены даже при возврате из нескольких точек метода:

```
public class MultipleReturns {
  public static void f(int i) {
    print("Initialization that requires cleanup");
    try {
      print("Point 1"); if(i == 1) return;
      print("Point 2"); if(i == 2) return;
      print("Point 3");if(i == 3) return;
      print("End");return;
    } finally {
      print("Performing cleanup");
    }
  }
  public static void main(String[] args) {
    for(int i = 1; i <= 4; i++)f(i);}
```

finally и потеря исключений

```
class VeryImportantException extends Exception {
  public String toString() {
     return "A very important exception!";}}
class HoHumException extends Exception {
  public String toString() {
    return "A trivial exception";}}
public class LostMessage {
  void f() throws VeryImportantException {
    throw new VeryImportantException();}
  void dispose() throws HoHumException {
    throw new HoHumException();}
  public static void main(String[] args) {
    try {
      LostMessage lm = new LostMessage();
      try {lm.f();}
      finally {lm.dispose();}
      }
      catch(Exception e) {
      System.out.println(e);}}}
```

finally и завершение

Плохо!!!

```
OutputStream stream = openOutputStream();
// что-то делаем со stream
stream.close();
```

Плохо!!!

```
OutputStream stream = openOutputStream();
try {
  // что-то делаем со stream
} finally {
  stream.close();
}
```

finally и завершение

Совсем Плохо!!!

```
 OutputStream stream = openOutputStream();
try {
   // что-то делаем со stream
} finally {
   try {
      stream.close();
   } catch (Throwable unused) {
      // игнорируем}}
```

Хорошо!

```
 OutputStream stream = openOutputStream();
try {// что-то делаем со stream}
catch(OutputStreamException ex){//что-то делаем}
finally {
    try {stream.close();} catch (SomeCloseException ex) {
      // Обрабатываем}}
```

```
try (OutputStream stream = openOutputStream())
   { // что-то делаем со stream }
```

finally отменяет выход

```
public class TestExceptionChaining{
  static int doTest(int n) {
    for (int i = 0; i < n; i++) {
      System.out.println("i = " + i);
      try {
        if (i % 3 == 0) {throw new Exception();}
      } catch (Exception e) {
        System.out.println("Exception!"); return i;
      } finally {
        System.out.println("Finally block");
        if (i % 3 == 0) {
          if (i < 5) {
          System.out.println("Cancel exception");
          continue;
          } else {
         System.out.println("OK Done");return 42;}
        }
      }
    }
    return -1;}
```

finally отменяет выход

```
public static void main(String[] args) {
    System.out.println("doTest(2) = " + doTest(2));
    System.out.println();
    System.out.println("doTest(10) = " + doTest(10));
  }
```

```
i = 0  Exception!  Finally block  Cancel exception, please
i = 1  Finally block
doTest(2) = -1

i = 0  Exception!  Finally block Cancel exception, please
i = 1 Finally block
i = 2 Finally block
i = 3 Exception! Finally block Cancel exception, please
i = 4 Finally block
i = 5 Finally block
i = 6 Exception! Finally block OK, now everything is done
doTest(10) = 42
```

Исключения и наследование

В переопределенном методе можно возбуждать только те исключения, которые были описаны в методе базового класса.

Так нельзя!

```
class AException extends Exception{};
class BException extends Exception{};

class A{
    public void doSomething() throws AException{
        throw new AException();
    }
}

class B extends A{
    @Override
    public void doSomething() throws BException{
        throw new BException();

    }
}
```

Идентификация исключений

```
class MyException extends Exception{};
class MyException1 extends MyException{};

public class InitTest{
public static void test(int i)
                          throws MyException, MyException1{
  if (i==1)throw new MyException();
  else throw new MyException1();
}
 public static void main(String[] arg){
  try {
  test(1);
  } catch (MyException e) {
  e.printStackTrace();
  } catch (MyException1 e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
  }
 }
}
```

Преобразование контролируемых исключений в неконтролируемые

```
try { // ... делаем что-нибудь полезное }
catch (НеЗнаюЧтоДелатьСЭтимКонтролируемымИсключением e) {
  throw new RuntimeException(e);
}
```

Решение идеально подходит для тех случаев, когда вы хотите «подавить» контролируемое исключение: вы не «съедаете» его, вам не приходится описывать его в своей спецификации исключений, и благодаря цепочке исключений вы не теряете информацию об исходном исключении.

Описанная методика позволяет игнорировать исключение и пустить его «всплывать» вверх по стеку вызова без необходимости писать блоки ***try-catch*** и (или) спецификации исключения. Впрочем, при этом вы все равно можете перехватить и обработать конкретное исключение, используя метод ***getCause()***

Ввод/вывод. Класс File

Класс File реализует ряд полезных средств, позволяющих манипулировать именами файлов, например разделять имена на компоненты или запрашивать дополнительную информацию о файле с заданным именем.

Объект File служит для представления имени файла (наличие файла как такового, вообще говоря, не обязательно). Например, чтобы ответить на вопрос существует ли файл с определенным именем, достаточно создать объект File на основе строки, содержащей имя, и вызвать метод exists этого объекта.

```
import java.io.File;
import java.util.Arrays;
public class DirList {
  public static void main(String[] args) {
    File path = new File(".");String[] list;
    list = path.list();
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);}}
```

Класс File

Объекты File могут быть созданы с помощью любого из трех конструкторов:

```
public  File(String path)
```

Создает объект File на основе строки, path, содержащей полное имя файла.

```
public File(String dirName,   String name)
```

Создает объект File, представляющий файл с именем name, который размещен в каталоге dirName. Если параметр dirName равен null, рассматривается только параметр name. Если dirName содержит пустую строку, считается, что файл name должен располагаться в каталоге, предусматриваемом по умолчанию значениями настроек операционной системы.

```
public File(File fileDir,   String name)
```

Создает объект File, представляющий файл с именем name; файл размещен в каталоге, наименование которого определяется объектом fileDir типа File.

Класс File

```
File src = new File("data.txt");
println("getName() = " + src.getName()) ;
println("getPath() = " + src.getPath()) ;
println("getAbsolutePath()  = "+ src.getAbsolutePath());
println("getCanonicalPath()  = "+ src.getCanonicalPath());
println("getParent()  = " + src.getParent()) ;
```

- **exists()** возвращает true, если файл существует в файловой системе;
- **canRead()** возвращает true, если файл существует и его содержимое может быть считано;
- **canWrite()** возвращает true, если файл существует и его содержимое может быть прочитанным
- **isFile()** возвращает true, если файл не является каталогом или иным специальным объектом файловой системы;
- **isDirectory()** возвращает true, если файл является каталогом;
- **isAbsolute()** возвращает true, если файл адресуется с помощью абсолют ного имени;
- **isHidden()** возвращает true, если файл является скрытым.

Класс File

`public long length()` Возвращает длину файла в байтах либо нуль, если файл не существует.

`public boolean renameTo(File newName)` переименовывает файл, возвращая значение true, если результат операции успешен.

`public boolean delete()` Удаляет файл или каталог, задаваемый текущим объектом File, и возвращает значение true при успешном завершении операции. Каталог, подлежащий удалению, должен быть пустым.

`public boolean createNewFile()` Создает новый пустой файл с именем, определяемым текущим объектом File. Возвращает значение false, если файл уже существует либо не может быть создан.

`public boolean mkdir()` Создает каталог с именем, определяемым текущим объектом File, и возвращает значение true при успешном завершении операции.

Public static File createTempFile(String prefix, String suffix, File directory) throws IOException

Создает новый пустой файл в заданном каталоге directory; при выборе имени файла учитываются указанные значения префикса (prefix) и расширения имени, или суффикса, (suffix). При успешном завершении метод возвращает сгенерированное имя файла и гарантирует его уникальность на протяжении текущего сеанса работы виртуальной машины (т.е.ни этот метод, ни любой из его вариантов, будучи выполненными в течение одного сеанса работы виртуальной машины, не возвратят того же им ни временного файла дважды). Длина строки prefix не должна быть меньше трех символов; в противном случае выбрасывается исключени типа illegalArgumentException. Рекомендуется употреблять в качестве префиксов имен временных файлов краткие и выразительные слова или аббревиатуры, такие как, например, "mail" или "abc". Если в качестве значения параметра suffix передается null, будет использовано расширение имени файла ".tmp". Поскольку предопределенных символов, которые выполняли бы функцию разделителя между именем файла и его расширением, не существует, любой требуемый разделитель, такой как ‘ . ‘ должен быть включен в состав строки суффикса явно. Если значение параметра directory равно null, создаваемый файл будет размещен в каталоге, который по умолчанию предлагается операционной системой для хранения временных файлов

Класс File

`public void deleteOnExit( )`

Обращается к системе с запросом на удаление файла при завершении сеанса работы виртуальной машины. Если запрос выдан, он не может быть отозван и будет удовлетворен только при нормальном окончании сеанса.

```
File lockFile = new File("database.Ick");
boolean  haveLock = lockFile.createNewFile();
 if  (!haveLock)
       throw new SomethingInaccessibleException();
lockFile.deleteOnExit();
  try {
      useSomething() ;
}  finally {
      lockFile.delete();
   }
```

Потоки ввода/вывода

В библиотеках ввода/вывода часто используется абстрактное понятие потока (*stream*) — произвольного источника или приемника данных, который способен производить или получать некоторую информацию. Поток скрывает детали низкоуровневых процессов, происходящих с данными непосредственно в устройствах ввода/вывода.

Классы библиотеки ввода/вывода *Java* разделены на две части — одни осуществляют ввод, другие вывод. В этом можно убедиться, просмотрев документацию *JDK*. Все классы, производные от базовых классов ***InputStream*** или ***Reader***, имеют методы с именами ***read()*** для чтения одиночных байтов или массива байтов. Аналогично, все классы, производные от базовых классов ***OutputStream*** или ***Writer***, имеют методы с именами ***write()*** для записи одиночных байтов или массива байтов.

InputStream/OutputStream

InputStream — абстрактный класс, задающий используемую в Java модель входных потоков. Все методы этого класса при возникновении ошибки возбуждают исключение IOException.

read() возвращает представление очередного доступного символа во входном потоке в виде целого. Обнаружив конец возвращает -1.

read(byte b[]) пытается прочесть максимум b.length байтов из входного потока в массив b. Возвращает количество байтов, в действительности прочитанных из потока.

read(byte b[], int off, int len) пытается прочесть максимум len байтов, расположив их в массиве b, начиная с элемента off. Возвращает количество реально прочитанных байтов.

skip(long n) пытается пропустить во входном потоке n байтов. Возвращает количество пропущенных байтов.

available() возвращает количество байтов, доступных для чтения в настоящий момент.

close() закрывает источник ввода. Последующие попытки чтения из этого потока приводят к возбуждению IOException.

InputStream/OutputStream

mark(int readlimit) ставит метку в текущей позиции входного потока, которую можно будет использовать до тех пор, пока из потока не будет прочитано readlimit байтов.

reset() возвращает указатель потока на установленную ранее метку.

markSupported() возвращает true, если данный поток поддерживает операции mark/reset.

OutputStream — абстрактный класс. Он задает модель выходных потоков Java.

write(int b) записывает один байт в выходной поток.

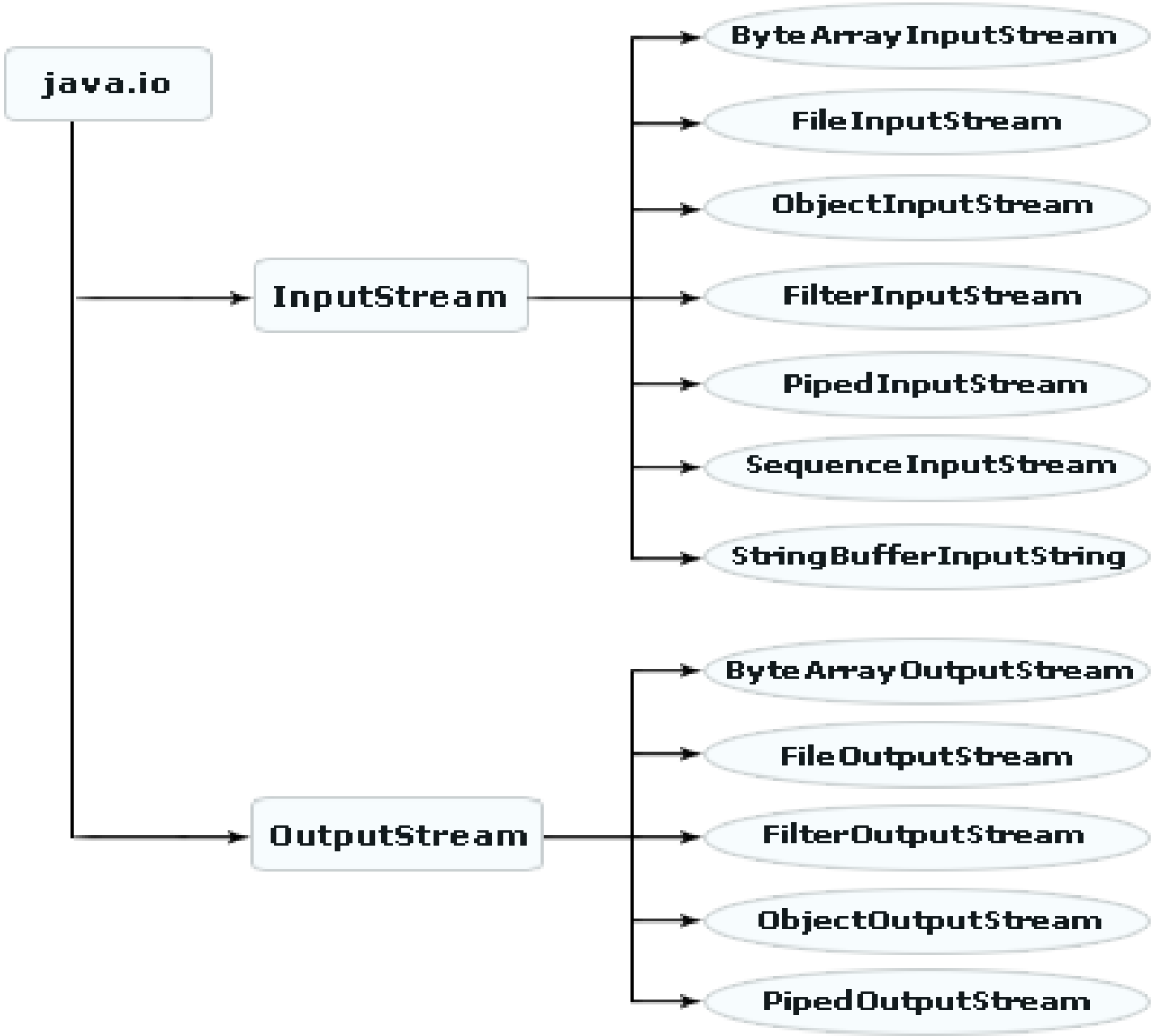
write(byte b[]) записывает в выходной поток весь указанный массив байтов.

write(byte b[], int off, int len) записывает в поток часть массива — len байтов, начиная с элемента b[off].

flush() очищает любые выходные буферы, завершая операцию вывода.

close() закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать IOException.

Иерархия InputStream/OutputStream



FileInputStream

```
int ba;
InputStream f1 = new FileInputStream("data.txt");
ba = f1.available();
System.out.println("Total Available Bytes: " + ba);
System.out.println("First 1/4 of the file: read()");
for (int i=0; i < ba/4; i++) {
      System.out.println((char) f1.read());}
System.out.println("Total Still Available: " +
                                        f1.available());
System.out.println("Reading the next 1/8: read(b[])");
byte b[] = new byte[ba/8];
if (f1.read(b) != b.length) {"Something bad happened";}
String tmpstr = b.toString();
System.out.println(tmpstr);
System.out.println("Skipping another 1/4: skip()");
f1.skip(ba/4);
System.out.println( "Still Available: " + f1.available());
System.out.println("Reading 1/16 into the end of array");
System.out.println("Still Available: " + f1.available());
f1.close();
```

FileOutputStream

```
public static byte getInput()[] throws Exception {
byte buffer[] = new byte[12];
for (int i=0; i<12; i++){
      buffer[i]=(byte)System.in.read();}
return buffer;}
byte buf[] = getInput();
OutputStream f0 = new FileOutputStream("file1.txt");
OutputStream f1 = new FileOutputStream("file2.txt");
OutputStream f2 = new FileOutputStream("file3.txt");
for (int i=0; i < 12; i += 2) { f0.write(buf[i]);}
f0.close();
f1.write(buf);
f1.close();
f2.write(buf, 12/4, 12/2);
f2.close();
}
```

В том случае, если надо добавлять байты в конец существующего файла, нужно использовать конструктор с 2 параметрами

```
OutputStream f0 = new FileOutputStream("file1.txt”,true);
```

ByteArrayInputStream

ByteArrayInputStream - это реализация входного потока, в котором в качестве источника используется массив типа byte. У этого класса два конструктора, каждый из которых в качестве первого параметра требует байтовый массив.

```
public ByteArrayInputStream(byte buf[]);
public ByteArrayInputStream( byte buf[], int offset, int
length);
```

Первый конструктор получает через единственный параметр ссылку на массив, который будет использован для создания входного потока. Второй позволяет дополнительно указать смещение offset и размер области памяти length, которая будет использована для создания потока.

```
byte[] bytes = {1,-1,0};
ByteArrayInputStream in = new ByteArrayInputStream(bytes);
int readedInt = in.read(); // readedInt=1
System.out.println("first element read is: " + readedInt);
readedInt = in.read(); // readedInt=255. Однако
//(byte)readedInt даст значение -1
readedInt = in.read(); // readedInt=0
```

ByteArrayOutputStream

У класса ByteArrayOutputStream — два конструктора. Первая форма конструктора создает буфер размером 32 байта. При использовании второй формы создается буфер с размером, заданным параметром конструктора (в приведенном ниже примере — 1024 байта):

```
OutputStream out0 = new ByteArrayOutputStream();
OutputStream out1 = new ByteArrayOutputStream(1024);
```

В классе ByteArrayOutputStream определено несколько методов:

```
public void reset();
public int size();
public byte[] toByteArray();
public void writeTo(OutputStream out);
```

Метод reset сбрасывает счетчик байт, записанных в выходной поток. С помощью метода size можно определить количество байт данных, записанных в поток.

Метод toByteArray позволяет скопировать данные, записанные в поток, в массив байт. Этот метод возвращает ссылку на созданный для этой цели массива.

С помощью метода writeTo вы можете скопировать содержимое данного потока в другой выходной поток, ссылка на который передается методу через параметр.

DataOutputStream

DataInputStream и DataOutputStream относятся к так называемым фильтровым классам, то есть классам, задающим фильтры для чтения и записи определенных форматов данных. Фильтровые классы не работают сами по себе, а принимают или отсылают данные простым потокам FileInputStream, FileOutputStream и т. д. Обычное создание потока вывода данных на базе класса DataOutputStream сводится к одной строке:

```
DataOutStream is = new DataOutStream ( new
                   FileOutputStream ( "data.dat" ));
```

После того как поток создан, в него можно выводить форматированные данные. Для этого в арсенале класса DataOutputStream имеется целый набор методов writeXXX() для записи различных данных, где XXX - название типа данных.

```
dos.writeDouble(doubleVar);
dos.writeInt(intVar);
dos.writeChars(StringVar);
dos.close();
```

DataInputStream

Для чтения «форматированных» данных применяется класс ***DataInputStream***, ориентированный на ввод/вывод байтов, а не символов. В данном случае необходимо использовать классы иерархии ***InputStream***, а не их аналоги на основе класса ***Reader***.

Методы чтения readXXX() класса DataInputStream практически полностью соответствуют методам writeXXX() класса DataOutputStream

```
DataInputStream dis = new DataInputStream ( new
                          FileInputStream ("data.dat" ));
doubleVar = dis.readDouble();
intVar = dis.readInt();
StringVar = dis.readLine();
dis.close();
```

DataInputStream/DataOutputStream

Если данные записываются в выходной поток ***DataOutputStream***, язык *Java* гарантирует, что эти данные в точно таком же виде будут восстановлены входным потоком ***DataInputStream*** — невзирая на платформу, на которой производится запись или чтение.

Единственным надежным способом записать в поток ***DataOutputStream*** строку (***String***) так, чтобы ее можно было потом правильно считать потоком ***DataInputStream***, является кодирование *UTF-8*, реализуемое методами ***readUTF()*** и ***writeUTF()***.

Метод ***writeDouble()*** записывает число ***double*** в поток, а соответствующий ему метод ***readDouble()*** затем восстанавливает его. Но, чтобы правильно интерпретировать любые данные, вы должны точно знать их расположение в потоке; Поэтому данные в файле должны иметь определенный формат, или вам придется использовать дополнительную информацию, показывающую, какие именно данные находятся в определенных местах.

RandomAccessFile

Класс ***RandomAccessFile*** предназначен для работы с файлами, содержащими записи известного размера, между которыми можно перемещаться методом ***seek()***, а также выполнять операции чтения и модификации. Записи не обязаны иметь фиксированную длину; вы просто должны уметь определить их размер и то, где они располагаются в файле.

Конструктор этого класса требует второй аргумент устанавливающий режим использования файла: только для чтения (строка «***r***») или для чтения и для записи (строка «***rw***»).

Прямое позиционирование допустимо только для класса ***RandomAccessFile***, и работает оно только в случае файлов. Класс ***BufferedInputStream*** позволяет вам пометить некоторую позицию потока методом ***mark()***, а затем вернуться к ней методом ***reset()***. Однако эта возможность ограничена (позиция запоминается в единственной внутренней переменной) и потому нечасто востребована.

```
public class UsingRandomAccessFile {
  static String file = "rtest.dat";
  static void display() throws IOException {
    RandomAccessFile rf = new RandomAccessFile(file, "r");
    for(int i = 0; i < 7; i++)
      System.out.println(
        "Value " + i + ": " + rf.readDouble());
    System.out.println(rf.readUTF());
    rf.close();}
  public static void main(String[] args)
  throws IOException {
    RandomAccessFile rf = new RandomAccessFile(file, "rw");
    for(int i = 0; i < 7; i++)
      rf.writeDouble(i*1.414);
    rf.writeUTF("The end of the file");
    rf.close();
    display();
    rf = new RandomAccessFile(file, "rw");
    rf.seek(5*8);
    rf.writeDouble(47.0001);
    rf.close();
    display();}}
```

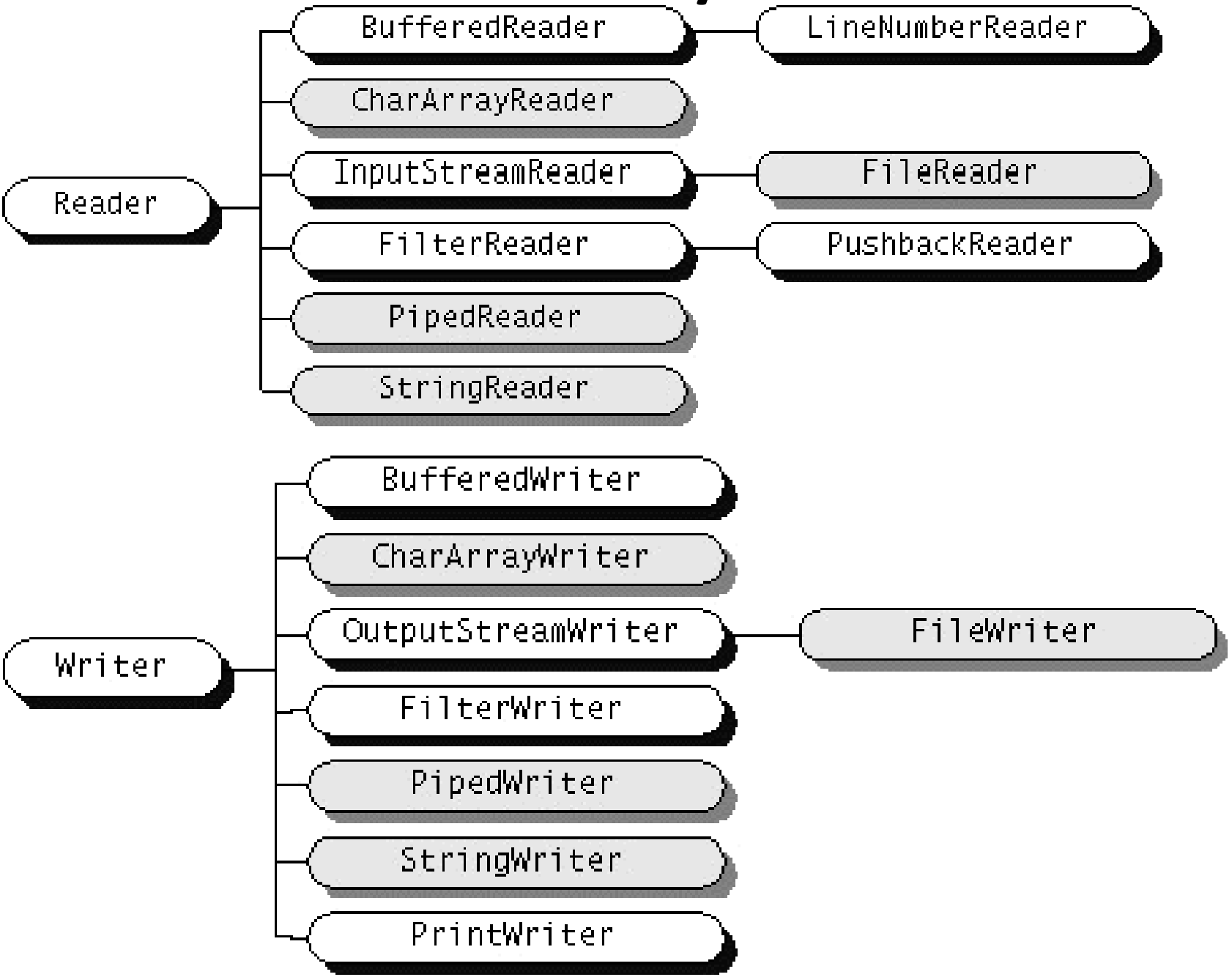
Reader/Writer

Reader и **Writer** обрабатывают потоки символов Unicode. В Java существуют несколько конкретных подклассов каждого из них.
Абстрактные классы **Reader** и **Writer** определяют несколько ключевых методов, которые реализуются другими поточными классами. Два самых важных метода — **read()** и **write()**, которые читают и записывают символы данных, соответственно. Они переопределяются производными поточными классами.

Закрытие потоков

По окончании чтения или записи поток нужно закрывать методом close().При этом происходит сбрасывание всех данных из буферов, используемых этим потоком. Также освобождаются ресурсы системы. В частности, есть вероятность, что если не закрыть файл, то последний пакет байтов не будет доставлен получателям.

Reader/Writer



Буферизованное чтение из файла

Чтобы открыть файл для посимвольного чтения, используется класс ***FileInputReader***; имя файла задается в виде строки (***String***) или объекта ***File***. Ускорить процесс чтения помогает буферизация ввода, для этого полученная ссылка передается в конструктор класса ***BufferedReader***. Так как в интерфейсе класса- имеется метод ***readLine()***, все необходимое для чтения имеется в вашем распоряжении. При достижении конца файла метод ***readLine()*** возвращает ссылку ***null***.

```
public static String read(String filename) throws
IOException{
   BufferedReader in = new BufferedReader( new
            FileReader(filename));
   String s; StringBuilder sb = new StringBuilder();
      while((s = in.readLine())!= null)
        sb.append(s + "\n"); in.close();
   return sb.toString();
}
```

Вывод в файл

Объект ***FileWriter*** записывает данные в файл. При вводе/выводе практически всегда применяется буферизация поэтому присоединяется надстройку ***BufferedWriter***. После этого подключается ***PrintWriter***, чтобы выполнять форматированный вывод. Файл данных, созданный такой конфигурацией ввода/вывода, можно прочитать как обычный текстовый файл.

```
BufferedReader in = new BufferedReader( new StringReader(
      BufferedInputFile.read("BasicFileOutput.java")));
PrintWriter out = new PrintWriter( new BufferedWriter(new
      FileWriter(file)));
int lineCount = 1; String s;
while((s = in.readLine()) != null )
out.println(lineCount++ + ": " + s);
out.close();
```

Стандартный ввод/вывод

Следуя модели стандартного ввода/вывода, *Java* определяет необходимые потоки для стандартного ввода, вывода и ошибок: ***System.in, System.out*** и ***System.err***.

Обычно чтение осуществляется построчно, методом ***readLine()***, поэтому имеет смысл буферизовать стандартный ввод ***System.in*** посредством ***BufferedReader***. Чтобы сделать это, предварительно следует конвертировать поток ***System.in*** в считывающее устройство ***Reader*** посредством класса-преобразователя ***InputStreamReader***. Следующий пример просто отображает на экране последнюю строку, введенную пользователем

```
BufferedReader stdin = new BufferedReader( new
InputStreamReader(System.in));
String s;
while((s = stdin.readLine()) != null &&
          s.length()!= 0)
System.out.println(s);
```

Перенаправление стандартного ввода/вывода

Класс ***System*** позволяет вам перенаправить стандартный ввод, вывод и поток ошибок. Для этого предусмотрены простые статические методы: ***setIn(InputStream)***; ***setOut(PrintStream)***; ***setErr(PrintStream)***. Перенаправление стандартного вывода особенно полезно тогда, когда ваша программа выдает слишком много сообщений сразу и вы попросту не успеваете читать их, поскольку они заменяются новыми сообщениями.

```
PrintStream console = System.out;
BufferedInputStream in = new BufferedInputStream( new
      FileInputStream("Redirecting.java"));
PrintStream out = new PrintStream( new
BufferedOutputStream( new FileOutputStream("test.out")));
System.setIn(in); System.setOut(out); System.setErr(out);
BufferedReader br = new BufferedReader( new
InputStreamReader(System.in));
String s;
while((s = br.readLine()) != null) System.out.println(s);
out.close(); System.setOut(console);
```

Литература

1. Брюс Эккель Философия Java. 4-е издание
2. Хорстманн К. С., Корнелл Г. -- Java 2. Том 1. Основы
3. Habrahabr.ru
4. Sql.ru
5. http://grepcode.com/project/repository.grepcode.com/java/root/jdk/openjdk/
6. !!! Google.com