```
            helper.readFromConsole();
        }
}
```

В ответ на запрос можно ввести некоторые данные и получить следующий результат:

```
Введите числовой код:
1001
Введите пароль:
****
Код доступа: 1001
Пароль: pass
```

При вводе значения **code**, не являющегося цифрой, на экран будет выдано сообщение об ошибке при попытке его преобразования в целое число, так как метод **valueOf()** пытается преобразовать строку в целое число, не проверив предварительно, может ли быть выполнено это преобразование.

Класс Scanner

Объект класса **java.util.Scanner** принимает форматированный объект (ввод) и преобразует его в двоичное представление. При вводе могут использоваться данные из консоли, файла, строки или любого другого источника, реализующего интерфейсы **Readable** или **ReadableByteChannel**.

Класс определяет следующие конструкторы:

```
Scanner(File source) throws FileNotFoundException
Scanner(File source, String charset)
   throws FileNotFoundException
Scanner(InputStream source)
Scanner(InputStream source, String charset)
Scanner(Readable source)
Scanner(ReadableByteChannel source)
Scanner(ReadableByteChannel source, String charset)
Scanner(String source),
```

где **source** – источник входных данных, а **charset** – кодировка.

Объект класса **Scanner** читает лексемы из источника, указанного в конструкторе, например из строки или файла. Лексема – это набор данных, выделенный набором разделителей (по умолчанию пробелами). В случае ввода из консоли следует определить объект:

```
Scanner con = new Scanner(System.in);
```

После создания объекта его используют для ввода, например целых чисел, следующим образом:

```
write(con.hasNextInt()) {
   int n = con.nextInt();
}
```

В классе **Scanner** определены группы методов, проверяющих данные заданного типа на доступ для ввода. Для проверки наличия произвольной лексемы используется метод **hasNext()**. Проверка конкретного типа производится с помощью одного из методов **boolean hasNext**Тип**()** или **boolean**

hasNextТип**(int radix)**, где **radix** – основание системы счисления. Например, вызов метода **hasNextInt()** возвращает **true**, только если следующая входящая лексема – целое число. Если данные указанного типа доступны, они считываются с помощью одного из методов Тип **next**Тип**()**. Произвольная лексема считывается методом **String next()**. После извлечения любой лексемы текущий указатель устанавливается перед следующей лексемой.

```
// пример # 10 : разбор файла: ScannerLogic.java : ScannerDemo.java
package chap09;
import java.io.*;
import java.util.Scanner;

class ScannerLogic {
   static String filename = "scan.txt";
   public static void scanFile() {
      try {
         FileReader fr =
            new FileReader(filename);
         Scanner scan = new Scanner(fr);
         while (scan.hasNext()) {//чтение из файла

            if (scan.hasNextInt())
                  System.out.println(
                        scan.nextInt() + ":int");
            else if (scan.hasNextDouble())
                  System.out.println(
                        scan.nextDouble() + ":double");
            else if (scan.hasNextBoolean())
                  System.out.println(
                        scan.nextBoolean() + ":boolean");
            else
                  System.out.println(
                        scan.next() + ":String");
         }
      } catch (FileNotFoundException e) {
         System.err.println(e);
      }
   }
   public static void makeFile() {
      try {
         FileWriter fw =
            new FileWriter(filename);//создание потока для записи
         fw.write("2 Java 1,5 true 1.6 ");//запись данных
         fw.close();
      } catch (IOException e) {
         System.err.println(e);
      }
   }
```

```
}
public class ScannerDemo {
      public static void main(String[] args) {
            ScannerLogic.makeFile();
            ScannerLogic.scanFile();
      }
}
```

В результате выполнения программы будет выведено:

```
2:int
Java:String
1.5:double
true:boolean
1.6:String
```

Процедура проверки типа реализована при с помощью методов **hasNext**Тип**()**. Такой подход предпочтителен из-за отсутствия возможности возникновения исключительной ситуации, так как ее обработка требует на порядок больше ресурсов, чем нормальное течение программы. Для чтения строки из потока ввода применяются методы **next()** или **nextLine()**.

Объект класса **Scanner** определяет границы лексемы, основываясь на наборе разделителей. Можно задавать разделители с помощью метода **useDelimiter(Pattern pattern)** или **useDelimiter(String pattern)**, где **pattern** содержит набор разделителей.

```
/* пример # 11 : применение разделителей: ScannerDelimiterDemo.java*/
package chapt09;
import java.util.Scanner;

public class ScannerDelimiterDemo {
   public static void main(String args[]) {
      double sum = 0.0;

      Scanner scan =
      new Scanner("1,3;2,0;  8,5; 4,8; 9,0; 1; 10");
         scan.useDelimiter(";\\s*");
         while (scan.hasNext()) {
            if (scan.hasNextDouble())
                sum += scan.nextDouble();
            else System.out.println(scan.next());
         }
         System.out.printf("Сумма чисел = " + sum);
      }
   }
}
```

В результате выполнения программы будет выведено:

```
Сумма чисел = 36.6
```

Использование шаблона **"; *"** указывает объекту класса **Scanner**, что ‘;’ и ноль или более пробелов следует рассматривать как разделитель.

Метод **String findInLine(Pattern pattern)** или **String findInLine(String pattern)** ищет заданный шаблон в следующей строке текста. Если шаблон найден, соответствующая ему подстрока извлекается из строки ввода. Если совпадений не найдено, то возвращается **null**.

Методы **String findWithinHorizon(Pattern pattern, int count)** и **String findWithinHorizon(String pattern, int count)** производят поиск заданного шаблона в ближайших **count** символах. Можно пропустить образец с помощью метода **skip(Pattern pattern)**.

Если в строке ввода найдена подстрока, соответствующая образцу **pattern**, метод **skip()** просто перемещается за нее в строке ввода и возвращает ссылку на вызывающий объект. Если подстрока не найдена, метод **skip()** генерирует исключение **NoSuchElementException**.

Архивация

Для хранения классов языка Java и связанных с ними ресурсов в языке Java используются сжатые архивные **jar**-файлы.

Для работы с архивами в спецификации Java существуют два пакета – **java.util.zip** и **java.util.jar** соответственно для архивов **zip** и **jar**. Различие форматов **jar** и **zip** заключается только в расширении архива zip. Пакет **java.util.jar** аналогичен пакету **java.util.zip**, за исключением реализации конструкторов и метода **void putNextEntry(ZipEntry e)** класса **JarOutputStream**. Ниже будет рассмотрен только пакет **java.util.jar**. Чтобы переделать все примеры на использование **zip**-архива, достаточно всюду в коде заменить **Jar** на **Zip**.

Пакет **java.util.jar** позволяет считывать, создавать и изменять файлы форматов **jar**, а также вычислять контрольные суммы входящих потоков данных.

Класс **JarEntry** (подкласс **ZipEntry**) используется для предоставления доступа к записям **jar**-файла. Наиболее важными методами класса являются:

void setMethod(int method) – устанавливает метод сжатия записи;

int getMethod() – возвращает метод сжатия записи;

void setComment(String comment) – устанавливает комментарий записи;

String getComment() – возвращает комментарий записи;

void setSize(long size) – устанавливает размер несжатой записи;

long getSize() – возвращает размер несжатой записи;

long getCompressedSize() – возвращает размер сжатой записи;

У класса **JarOutputStream** существует возможность записи данных в поток вывода в **jar**-формате. Он переопределяет метод **write()** таким образом, чтобы любые данные, записываемые в поток, предварительно сжимались. Основными методами данного класса являются:

void setLevel(int level) – устанавливает уровень сжатия. Чем больше уровень сжатия, тем медленней происходит работа с таким файлом;