Въведение в Java

Първоначално искаме да отбележим, че статиите в категория “Java” са предназначени за хора, които са запознати с програмиране на C и C++. Няма да се спираме детайлно върху основите на езика – всичката теория, която се припокрива с програмиране на C и C++ ще приемаме, че е позната. Така например няма да уточняваме специално какво означават различните типове данни и какви стойности приемат, а просто ще ги изброим набързо.

Java е език за програмиране създаден през 1991г. (първоначално под името Oak, а след 1995г. Java). Основите на езика идват именно от споменатите езици C и C++. Синтаксисът на езика е много близък до C, а обектния модел е изключително близък до C++. Поради тази причина е изключително лесен преходът между тези езици.

Първоначално Java е измислен за писане на софтуер за обслужване на различни електронни устройства и неговата реализация за персонални компютри е била на заден план. Основният проблем който производителите от Sun са си поставили е платформената независимост. Широко популярният език за програмиране C++ на който те са стъпили като основа има един съществен недостатък – всяка една програма трябва да бъде компилирана до изпълним код. Това създава изключително много проблеми при писането на софтуер, който трябва да се изпълнява на различни устройства, защото те често имат различни типове процесори и архитектура. За да напише една програма на C++, която да бъде изпълнима както на телефони Motorola, така и на Ericsson е било нужно тя да бъде компилирана два пъти с два различни компилатора. Това е сериозна пречка за крайните програмисти.

Идеята стояща в Java е програмите да не се компилират до изпълним код, а до междинен, т.нар. “байткод”. За да бъдат изпълнени програмите, те се обработват от специална програма наречена

“виртуална машина”. Така един и същи байткод може да бъде изпълняван на различни платформи по един и същи начин – единствено е нужно съответната платформа да има инсталиран собствен вариант на виртуална машина за Java. Така програмистите няма нужда да пишат отделни версии за своите програми за различните платформи. Освен предимството на платформената независимост, този подход носи и допълнителни преимущества относно сигурността. Виртуалните машини изпълняват байткода в защитена среда и така програми писани на Java трудно могат да навредят на друг софтуер от операционната система. Един от малкото недостатъци на този подход пък е, че нуждата от “докомпилиране” на байткода от виртуалната машина до изпълним код довежда до малка загуба на производителност при стартиране на програмите.

С развитието на интернет от корпорацията Sun са забелязали, че там също започва да се наблюдава нужда от платформена независимост. Така те насочват езика Java към създаване на интернет приложения и за известно време успяват да се похвалят с лидерство в сферата на интернет програмирането. По-късно други езици като PHP и технологията .Net на Microsoft започват успешно да конкурират Java, но въпреки това и до ден днешен не може да се отрече една от лидерските позиции в тази сфера.

Както споменахме в началото Java е обектно-ориентиран език за програмиране. Това е модерният подход за програмиране от високо ниво, където логиката на програмите е фокусирана върху данните, а не върху програмния код. Както всеки език за ООП, така и Java се основава на трите най-важни принципа:

Капсулиране – дефиниране на връзката между данните и програмния код;

Полиморфизъм – възможност за извършване на различни действия с един обект според ситуация и нужди;

Наследяване – процес, при който един обект наследява и разширява свойствата на друг.

В следващите статии ще разгледаме набързо основите на синтаксиса на езика и ще преминем по-подробно към същината на обектното програмиране. В началото много от статиите ще се припокриват с материала от статиите за C и C++, затова просто ще правим аналогии.

Преди да започнем ще е нужно да се сдобиете с “компилаторът”, който превръща код на Java до байткод. Системата за разработка на Java програми се нарича Java Development Kit (JDK). Ще използваме Java Standart Edition, която може да бъде изтеглена от следния адрес:

http://java.sun.com/javase/downloads/index.jsp

За изпълнение на програмите на вашия компютър (тоест споменатото “докомпилиране” на байт код до изпълним) е нужно да инсталирате и Java Runtime Environment (JRE), която е налична на същия адрес.

Колкото до среда за разработка на програмите – налични са множество безплатни продукти. Най-популярният от тях е Netbeans и е предназначен за професионални разработки. За учебни цели обаче ние ще използваме една много “по-лека” среда за разработка наречена DrJava. Кодът, който ще пишем обаче тъй или иначе се компилира от компилаторът на JDK, а не от средата – затова средата за разработка е просто едно удобство и написаният код ще бъде валиден независимо от средата, която е използвана. Ентусиастите могат дори да използват и обикновен текстов редактор.

Типове данни, масиви и оператори в Java

Вече споменахме, че обектите са в основана на езика Java. В езика е залегнала и йерархична структура за групиране на обектите. Всички обекти със сходни свойства се обединяват в класове. Класовете от своя

страна се обединяват в пакети. Засега обаче ще се фокусираме към синтаксис на езика и ще се спрем по-подробно на класовете и модификаторите за достъп в по-следваща статия.

“Сорс-кодът” на програмите на Java се записва във файлове с разширение .java. Когато бъдат компилирани до байткод те получават разширение .class. Именно този .class файл е преносимият код, който може да бъде компилиран на различни машини.

Както всяка стандартна програма, така и Java програмите имат своя “входна точка”, т.е. място от което се стартират. Обикновено е прието в .java файла да дефинираме клас със същото име като файла и той да бъде публичен, т.е. “достъпен за външния свят”. В този клас се дефинира и основен метод (в Java е прието функциите и процедурите да се наричат методи) с име “main”. Ето как ще изглежда нашата първа Java програма. Създайте текстов файл с име myfirstprogram.java и запишете в него следната информация:

```
public class MyFirstProgram{
  public static void main(String[] args) {
    System.out.format("%s", "Hello World\n");
  }
}
```

Ако компилирате този файл се досещате, че ще бъде изписан надписът “Hello World”. По конвенция методът main трябва да бъде публичен и статичен (както казахме по-късно ще се спрем над дефиниция на тези термини – засега приемете това наготово). Подадения му входен параметър args е масив от входни параметри от тип String, който може да бъде подаден чрез командния ред. Методът System.out.format() е стандартен форматиран изход познат от C като printf() и се използва за отпечатване на информация в конзолата.

Няма да се спираме над дефиниция на основните типове данни – те са добре познати от програмирането на C:

- boolean – булев тип приемащ за стойност литерали true или false;
- char – символ в кодировка unicode;
- byte – 8 битово цяло число;
- short – 16 битово цяло число;
- int – 32 битово цяло число;
- long – 64 битово цяло число;
- float – 32 битово число с плаваща запетая;
- double – 64 битово число с плаваща запетая.

При тип char специалните символи са:

- \n – нов ред;
- \r – връщане в началото на ред;
- \t – табулация;
- \b – връщане на символ назад (backspace);
- \’ – апостроф;
- \” – кавички;
- \\ – знак наклонена черта.

За разлика от C и C++ стандартно в езика Java (без допълнителна библиотека) стандартно е наличен и низов тип String. Той се състои от поредица от 16 битови unicode символи (т.е. естествена алтернатива на масив от тип char). За разлика от основните типове данни обаче, той е обект. Ще се спрем на обектите по-късно.

Масиви в Java могат да бъдат задавани по два еквивалентни начина:

int[] array1;

int array2[];

Първият запис е нововъведение в Java спрямо C и C++. Вторият начин на записване е по-удобен за програмистите свикнали да пишат на C. Между тези два записа няма никаква разлика. Въпреки това не се

окуражава използването на втория синтаксис – от Sun препоръчват първия, тъй като той е типичен за езика Java.

Масивите в Java са обекти. Това означава, че те се съхраняват в динамичната памет (heap) за разлика от примитивните типове, които се записват в стек (stack). За разлика от C в Java не е предвидена възможност за създаване на масив с стек.

Показания пример по-горе се нарича “дефиниция на масив”, т.е. създадена е променлива, която сочи към обект от тип масив, но самият той все още не е “създаден”. Тъй като масивите са обекти, то те се създават чрез оператор new:

array1 = new int[10];

array2 = new int[5];

Както виждате размерността на масива се посочва при неговото създаване, а не при неговата дефиниция. Следващата стъпка е “инициализация” на масива. Тя става по стандартния познат начин:

array1[0] = 4;

array1[1] = 5;

Искаме да отбележим като особеност спрямо езика C, че в момента на създаване на масив той се инициализира автоматично със стандартни начални стойности. При целочислените типове данни това е числото 0 (с L накрая ако е long), при типовете в плаваща запетая е 0.0 (с f накрая ако е float), при char е символът \u0000, а при тип boolean е false. Това означава, че ако искаме да отпечатаме съдържанието на така записания масив array1, то първите два елемента ще са числата 4 и 5, а останалите до края ще са 0.

Двумерните и “n”-мерните масиви се дефинират аналогично:

int[][] array3 = new int[2][3];

array3[0][2] = 5;

Операторите в Java са следните:

1. Унарни:

a) ++ – инкрементиране с 1;

b) — – декрементиране с 1;

c) + – положително число, например +3;

d) – - отрицателно число, например -3;

e) ~ – побитово инвертиране, например 00001111 става 11110000;

f) ! – логическо отрицание;

g) (<тип>) – преобразуване на тип данни, например ако x е от тип int, то можем да направим double d = (double)x.

2. Аритметични:

a) + – събиране (използва се и за конкатенация на обекти от тип String);

b) – - изваждане;

c) * – умножение;

d) / – деление;

e) % – остатък от деление при целочислените типове.

3. Сравнение:

a) < – по-малко;

b) <= – по-малко или равно;

c) > – по-голямо;

d) >= – по-голямо или равно;

e) == – равно;

f) != – не равно.

4. Логически:

a) && – логическо “и”;

b) || – логическо “или”.

5. Побитови:

a) & – побитово “и”;

b) | – побитово “или”;

c) ^ – побитово “изключващо или”;

d) << – изместване вляво;

e) >> – изместване вдясно;

f) >>> – изместване вдясно без знак (нов оператор, който не присъства в C/C++, но на който няма да се спираме).

Оператори if-else, switch, цикли

При операторите if-else и switch, както и при циклите for, while и do-while няма никакви изненади спрямо езика за програмиране C. Синтаксисът при Java е абсолютно аналогичен. Нов е само последният цикъл (т.8), на когото ще обърнем повече внимание. Ще демонстрираме директно с примери без да обясняваме детайлно функционалността на операторите и циклите.

1. Конструкция if-else: решава уравнението ax+b=0:

```
int a=2, b=3;
if (a!=0) System.out.format("x = %d\n",(-b/a));
else{
  if (b == 0) System.out.format("%s", "Any x is a solution");
  else System.out.format("%s", "There is no solution");
}
```

2. Оператор switch: показва ден от месеца спрямо подадено число:

```
int month = 3;
switch (month) {
  case 1:  System.out.format("%s", "January\n");
           break;
  case 2:  System.out.format("%s", "February\n");
           break;
  case 3:  System.out.format("%s", "March\n");
           break;
  case 4:  System.out.format("%s", "April\n");
           break;
  case 5:  System.out.format("%s", "May\n");
```

```
            break;
    case 6:  System.out.format("%s", "June\n");
            break;
    case 7:  System.out.format("%s", "July\n");
            break;
    case 8:  System.out.format("%s", "August\n");
            break;
    case 9:  System.out.format("%s", "September\n");
            break;
    case 10: System.out.format("%s", "October\n");
            break;
    case 11: System.out.format("%s", "November\n");
            break;
    case 12: System.out.format("%s", "December\n");
            break;
    default: System.out.format("%s", "Invalid month.\n");
  }
```

3. Цикъл while: намира НОД по алгоритъма на Евклид:

```
  int A = 28;
  int B = 48;
  System.out.format("NOD(%d,%d) = ", A,B);
  while (A != B){
    if (A > B)  A = (A-B);
    else  B = (B-A);
```

```
    }
    System.out.format("%d",A);
  }
```

4. Цикъл do-while: изчислява n!:

```
    int n = 6;
    System.out.format("n = %d =>",n);
    long factorial = 1L;
    do {
      factorial *= (long)n;
      n--;
    }while (n > 0);
    System.out.format("n! = %d\n",factorial);
```

5. Цикъл for: извежда четните числа от масив:

```
    int[] array = {2,5,12,4,3,13,18,-5,-4,8};
    for (int i=0; i<array.length; i++){
      if (array[i]%2 == 0) System.out.format("%d ",array[i]);
    }
    System.out.format("\n");
```

6. Оператор break: търси дали в масив е налично поне веднъж числото 13:

```
    int[] array = {2,5,12,4,3,13,18,-5,-4,8};
    boolean found = false;
```

```
int search = 13;
System.out.format("Searching for %d",search);
for (int i=0; i<array.length; i++){
  if (array[i] == search){
    found = true;
    break;
  }
  System.out.format("%d... ", array[i]);
}
if (found) System.out.format("FOUND >>> %d <<< FOUND!",search);
else System.out.format("sorry, no %d in the array",search);
```

7. Оператор continue: сумира всички нечетни числа на масив:

```
int[] array = {2,5,12,4,3,13,18,-5,-4,8};
long sum = 0L;
for (int i=0; i<array.length; i++){
  if (array[i]%2 == 0) continue;

  sum += (long)array[i];
}
System.out.format("Sum of odd numbers = %d\n",sum);
```

8. Цикъл for-each: това е друг вид запис на цикъл for. Синтаксисът е следния:

```
<тип>[] array = {стойности};
```

```
for (<тип> var : array) {
    операции;
}
```

Този запис е абсолютно аналогичен на следния:

```
<тип>[] array = {стойности};
for (int i=0; i<array.length; i++) {
   <тип> var = array[i]
   операции;
}
```

Този запис е изключително удобен когато обхождаме масиви. Ето пример за цикъл, който отпечатва елементите на масив на екрана:

```
int[] array = {2,5,12,4,3,13,18,-5,-4,8};
for (int i : array){
  System.out.format("%d ",i);
}
System.out.format("\n");
```

Имайте в предвид, че на променливата i се предава копие на елемент от масива, а тя не е указател към елемента. Това означава, че ако промените i в тялото на цикъла оригиналния масив НЕ се променя.

Възможностите на for-each са доста по-ограничени от тези на пълния запис на for. Въпреки това този запис прави кода доста по-компактен и по-лесно четим. Насърчаваме използването на този цикъл когато е нужно съставянето на кратки алгоритми при четене на масив.

9. Вложени цикли и контрол над изпълнението: когато правим вложени цикли, то в Java ни е предоставена една допълнителна възможност за оператор break. Можем да прекъсваме не само текущия цикъл (в тялото на който се изпълнява оператор break), но и външен цикъл. Нека разгледаме следния пример: имаме двумерен масив с числа. Търсим числото 13 вътре в масива:

```
int[][] array = {
                {2,5,12,4},
                {3,13,18,-5},
                {12,8,-4,8}
               };
boolean found = false;
int search = 13;
System.out.format("Searching for %d\n",search);
int i,j=0;
for (i=0; i<array.length; i++){
  for (j=0; j<array[i].length; j++){
    if (array[i][j] == search){
      System.out.format("Found! >>> %d <<< Found!",search);
      found = true;
      break;
    }
    System.out.format("%d... ",array[i][j]);
  }
  System.out.format("\n");
}
```

```
if (!found) System.out.format("%d not found", search);
```

Резултат:

Searching for 13

2... 5... 12... 4...

3... Found! >>> 13 <<< Found!

12... 8... -4... 8...

Виждаме, че програмата не се прекрати при намиране на числото 13. Прекрати се само вътрешният цикъл, т.е. пропуснахме две колони. Ако искаме да прекратим търсенето въобще, т.е. да спрем изпълнението както на външния цикъл, така и на вътрешния цикъл, то използваме следния “трик”:

```
int[][] array = {
                 {2,5,12,4},
                 {3,13,18,-5},
                 {12,8,-4,8}
                };
boolean found = false;
int search = 13;
System.out.format("Searching for %d\n",search);
int i,j=0;
outerfor:
for (i=0; i<array.length; i++){
  innerfor:
  for (j=0; j<array[i].length; j++){
```

```
    if (array[i][j] == search){
      System.out.format("Found! >>> %d <<< Found!\n",search);
      found = true;
      break outerfor;
    }
    System.out.format("%d... ",array[i][j]);
  }
  System.out.format("\n");
}
if (!found) System.out.format("%d not found\n", search);
```

Ако искаме да направим двата примера еквивалентни, то бихме сменили “break outerfor” с “break innerfor” или просто “break”. Така дефинирани “outerfor” и “innerfor” наричаме “етикети“. Аналогично етикети могат да се използват при оператор continue.

Стандартен вход и изход

Любителите на Linux много харесват Java заради концепцията на езика да работи с входни и изходни потоци. Това означава една абстракция на входните и изходните данни, които използва и връща програмата. Така чрез дребна настройка може една програма вместо да изкарва данни на екрана – да ги записва във файл, да ги изпрати до принтер, порт на компютъра, друга програма, да ги изпрати към сървър в интернет и т.н.; и обратно – да ги получава от най-различни източници. Независимо с какви данни работят потоците, ние можем да ги разглеждаме просто като последователни парчета данни. Входни са потоците, които програмата получава, а изходни са потоците, които изпраща. Засега обаче ще разгледаме само няколко примера за методи, които работят със стандартния вход и изход (клавиатура и монитор) и само ще загатнем как се пренасочват. В следващата статия ще

разгледаме как се чете и пише във файл и как можем да пренасочваме стандартните потоци към файлове.

1. Стандартен изход: най-просто казано стандартния изход е отпечатване на информацията на екрана на компютъра (това впрочем може да бъде променено). Ние вече показахме няколко примера с функцията System.out.format(). По-популярни обаче са System.out.print() и System.out.println(). При тях можем да конкатенираме различни типове данни като символен низ. За всеки от тях поотделно се извиква метод “toString()” (преобразува данните в тип String), след което получените низове се конкатенират и изпращат към стандартния изход (монитора) във вид на низ. Ето един пример:

```
int i=2;
double d=3;
System.out.print("i = "+i+", and d = "+d+"\n");
```

Виждате, че миксирахме различни типове данни без проблем. Методът println() действа по същия начин. Разликата с print е само една – при println() се слага автоматично символ за нов ред \n в края на низа. Следващия пример е напълно еквивалентен на първия:

```
int i=2;
double d=3;
System.out.println("i = "+i+", and d = "+d);
```

Тук е мястото да отбележим, че знакът “+” може да предизвика объркване. Следните два реда например връщат коренно различен резултат:

```
System.out.println("Result: "+2+3);
System.out.println("Result: "+(2+3));
```

Първото ще върне резултат “Result: 23″ (тоест ще отпечати първото число и после второто), а второто ще върне резултат “Result 5″, т.е. първо е изчислило изразът в скобите и след това го е отпечатило. Затова бъдете внимателни. Трябва да отбележим, че методът “format” има изключително богати възможности, като отличителни са операциите с дата и час. Няма обаче да се спираме подробно на тях , но ви насърчаваме да ги прочетете от документацията на Java.

2. Стандартен изход за грешки: абсолютно същите методи както в System.out ги има и в System.err. Това се нарича “стандарен изход за грешки”. Идеята за такова дублиране е да има разделение между стандартната информация и тази, която се изписва при непредвидени ситуации. По подразбиране стандартния изход за грешки отпечатва на екрана. Понякога обаче се използва за правенето на логове (т.нар. dump файлове). Ще използваме System.err когато по-късно изучим “обработка на изключения”.

3. Стандартен вход: “обратните” на System.out методи се дефинират в System.in. Стандартния вход за всеки компютър по подразбиране е клавиатурата. Още в началото искаме да споменем, че за разлика от System.out което има функционалност за работа със символи, при System.in винаги се работи с байтове. Методът, който се използва за четене на информация байт по байт е System.in.read(). Въпреки, че резултатът е byte, ние трябва да го присвояваме в променлива от тип int. Това е така, защото при грешка (или прекъсване) се връща -1 (допълнителна стойност, за която трябва да има отделено място). Следният пример чете информация от клавиатурата байт по байт до натискане на Enter:

```
int input;
System.out.print("Enter text: ");
do{
  try{
    input = System.in.read();
    System.out.print((char)input);
```

```
    }
    catch(java.io.IOException e){
      System.err.println("ERROR READING");
      break;
    }
  }while(input!=(int)'\n');
```

Засега приемете изписаните try-catch блокове като даденост – по-късно ще ги разгледаме подробно. Тук се сблъскваме с първата “странност” в поведението на System.in.read(). Резултатът от изпълнението на програмата ще бъде следното:

Enter text: <текст, който вие пишете>

<текстът, който сте написали>

тоест ако сте написали от клавиатурата “Misho” то на екрана ще се появи следното:

Enter text: Misho

Misho

Защо се получи така? Точно след изпълнението на System.in.read() ние правихме System.out.print(…). Би било редно след прочитане на байт той да се изпише моментално?!? Полученият резултат е нормален поради спецификата на методът read(). На практика когато бъде извикан метода, програмата става в режим на изчакване да въведете информация. Да, но по дефиниция информация се въвежда след натискане на бутон Enter. Тоест още при първото извикване на read() ние можем да въведем повече от един символ – от примера с “Misho” това са 5 символа. Чак тогава натискаме бутон Enter и изпълнението на програмата преминава към операторът System.out.print(…). За да го демонстрираме по-добре нека преправим предишния код така, че вместо при \n да се спира при символ s:

```
int input;
System.out.print("Enter text: ");
do{
  try{
    input = System.in.read();
    System.out.print((char)input);
  }
  catch(java.io.IOException e){
    System.err.println("ERROR READING");
    break;
  }
}while(input!=(int)'s');
```

Резултатът при написването на “Misho” ще бъде следния:

Enter text: Misho

Mis

Виждате, че “h” и “o”, както и символът за нов ред не се отпечатаха! Тогава къде се “изгубиха”? Отговорът е, че те останаха в буферът на клавиатурата. Получи се следното:

- При първото “завъртане” на цикъла в буфера на клавиатурата записахме последователността от символи (във вид на байтове) “Misho\n”. На променливата input подадохме първата буква, която след това се отпечата на екрана;

- При второто “завъртане” на цикъла ние имаме в буферът символите “isho\n”. Така вече не се извиква поле искащо входни данни, а веднага се взима буквата ‘i’, която после се отпечатва на екрана;

- При третото “завъртане” на цикъла се взима буквата ‘s’ и се отпечатва. Тук заради условието на цикъла той прекъсва. Така на екрана остават отпечатани само буквите “Mis”. Тук веднага трябва да ви направи впечатление нещо много важно. В буферът на клавиатурата са останали символите “ho\n”! Това би могло да доведе до значителни проблеми по-нататък в програмата, защото ако отново ви се наложи да четете от клавиатурата, то вие ще вземете директно оставащите букви от буфера на клавиатурата! Ето как можем да покажем това:

```
int input;
System.out.print("Enter text: ");
do{
  try{
    input = System.in.read();
    System.out.print((char)input);
  }
  catch(java.io.IOException e){
    System.err.println("ERROR READING");
    break;
  }
}while(input!=(int)'s');

try{
   System.out.println((char)System.in.read());
}
```

```
catch(java.io.IOException e){
  System.err.println("ERROR READING");
}
```

При изписване на думата “Misho” резултатът ще е следния:

Enter text: Misho

Mish

Виждаме, че System.out.println((char)System.in.read()); отпечата символът “h”, т.е. се случи точно това, което очаквахме – в буферът има информация! Освен това дори след това четене в буферът са останали символите “o\n”. За да се справим с подобни ситуации ние трябва да “почистваме” буфера от излишната информация. Най-лесно това става по следния начин:

```
try{
  while(System.in.available() > 0)
System.in.skip(System.in.available());
}
catch(java.io.IOException e){
  System.err.println("ERROR READING");
}
```

System.in.available() връща int с броя байтове, които са налични в буфера. System.in.skip() пък приема число от тип long, с което се пропускат определен брой байтове (т.нар. flush). Именно заради разликата в типовете се налага да използваме цикъл, тъй като буферът може да има повече от 32 бита (един int) и така методът available() да не връща точна информация. Всъщност съвсем възможно е да имаме огромни количества данни чакащи в този буфер.

4. Форматиран вход: след Java 5 се наложи един доста полезен стандартен обект – Scanner. Чрез “скенерът” ние можем да се възползваме от форматиране на входните данни. При него четенето от входния поток става не байт по байт, а чрез типове данни. Следният пример чете всичко до достигане на интервал (space) и го отпечатва на екрана:

```
java.util.Scanner s = null;
String input;
s = new java.util.Scanner(System.in);
System.out.print("Enter text: ");
input = s.next();
System.out.print(input);
```

В обект Scanner обаче има много по-мощни методи. Погледнете следния пример:

```
String input = "Hello 123 World";
java.util.Scanner s = new java.util.Scanner(input);
String s1 = s.next();
int i = s.nextInt();
String s2 = s.next();
System.out.println(s1+s2+i);
s.close();
```

Първо забележете последния ред – тук затваряме скенера. В предишния пример не го направихме, понеже той работеше със System.in – не бихме изкали да си затворим буфера на клавиатурата! Тук обаче това е добре да бъде направено. Виждате, че можем да “очакваме” четене на целочислен тип. Аналогични са методите

nextBoolean(), nextByte(), nextShort(), nextLong(), nextFloat() и nextDouble(). За изчистване на буфера има вече познатия skip(). За четене на цял ред (тоест “оставащото до края на текущия ред”) има много често използвана функция nextLine() връщаща String. Също така много удобни са функциите hasNextInt(), hasNextDouble() и т.н. – те връщат true или false за това дали има следващ елемент от такъв тип. Пример: Намира първите намерени числа от подаден String и ги отпечатва на екрана:

```
String input = "9,-12,4,a,32";
java.util.Scanner s = new java.util.Scanner(input);
s.useDelimiter(",");
while (s.hasNextInt()) System.out.print(s.nextInt()+" ");
s.close();
```

Резултат: 9 -12 4 Видяхте и още една функция – useDelimiter(). Тя променя стандартният разделител от интервал в зададения String. С тези примери впрочем демонстрирахме и възможността да четем от различни източници (в случая от String вместо от клавиатурата). В следващата статия ще покажем как се чете и пише във файл и как се пренасочва стандартния изход. Внимание: При подаване на грешни данни в Scanner може да се очаква InputMismatchException, тоест “несъвпадащи данни”. В такъв случай програмата ще се прекрати с това съобщение за грешка.

Вход и изход – работа с файлове

Вече се запознахме със стандартните вход и изход като байтови потоци. Те всъщност са наследници на абстрактните класове InputStream и OutputStream. Двата класа FileInputStream и FileOutputStream също ги наследяват. Нека разгледаме един пример за програма, която чете информация от един файл и я записва в друг:

```
java.io.FileInputStream in = null;
java.io.FileOutputStream out = null;
try {
  in = new java.io.FileInputStream("input.txt");
  out = new java.io.FileOutputStream("output.txt");
  int readbyte;

  while ((readbyte = in.read()) != -1) {
    out.write(readbyte);
  }
}
catch(java.io.IOException e){
  System.err.println("Error reading file");
}
finally{
  try{
    if (in != null) in.close();
```

```
    if (out != null) out.close();
  }
  catch(java.io.IOException e){}
}
```

FileInputStream и FileOutputStream може да се използват за четене и запис на всякакви видове файлове. Въпреки това по конвенция е добре да ги използваме само за бинарни файлове. За четене и запис на символни файлове използваме FileReader и FileWriter (наследници на абстрактните Reader и Writer), които работят със символи. Що се отнася до ASCII файлове – те работят по един и същи начин. Ето горния пример реализиран чрез FileReader и FileWriter:

```
java.io.FileReader in = null;
java.io.FileWriter out = null;
try {
  in = new java.io.FileReader("input.txt");
  out = new java.io.FileWriter("output.txt");
  int readbyte;

  while ((readbyte = in.read()) != -1) {
    out.write(readbyte);
  }
}
catch(java.io.IOException e){
  System.err.println("Error reading file");
}
finally{
```

```
  try{
    if (in != null) in.close();
    if (out != null) out.close();
  }
  catch(java.io.IOException e){}
}
```

По конвенция когато работим с текстови файлове ще използваме FileReader и FileWriter. На практика символните потоци използват байтови и един вид ги “надграждат”. Основна разлика между тях е, че InputStream и OutputStream четат/пишат по 8 бита (числа от -128 до 127), а Reader и Writer четат/пишат char (до 16 бита). Това не означава, че Reader и Writer работят с по два байта едновременно. Всичко зависи от кодировката на файла.

Има смесени типове потоци. Това са InputStreamReader() и OutputStreamWriter(). Те се използват за специални случаи като например текстови файлове с нестандартна кодировка. Също така се използват предимно при правене на мрежови връзки, които пренасят смесена информация. Засега няма да ги разглеждаме подробно.

Примерите от по-горе се наричат “небуферирани” потоци. Това означава, че използват директен достъп до информацията така, както е поднесена от операционната система. Това е изключително неефективно особено при честа работа с информацията, защото предизвиква много дискови операции. Затова по-често ще използваме т.нар. “буферирани” потоци. При тях се заделя място в паметта, в което се записват големи части от информацията, която четем. По този начин ние намаляваме обръщенията към диска. Ето горния пример със символните потоци написан чрез буфериран поток:

```
java.io.BufferedReader in = null;
java.io.BufferedWriter out = null;
```

```
    try {
      in = new java.io.BufferedReader(new
java.io.FileReader("input.txt"));
      out = new java.io.BufferedWriter(new
java.io.FileWriter("output.txt"));
      int readbyte;


      while ((readbyte = in.read()) != -1) {
        out.write(readbyte);
      }
    }
    catch(java.io.IOException e){
      System.err.println("Error reading file");
    }
    finally{
      try{
        if (in != null) in.close();
        if (out != null) out.close();
      }
      catch(java.io.IOException e){
        System.err.println("System error occured");
        return;
      }
    }
```

Аналогично за байтови потоци можете да използвате BufferedInputStream() и BufferedOutputStream(). Няма да даваме същия пример с тях, защото е напълно аналогичен.

Накрая за файлови потоци ще посочим т.нар. “потоци от данни” или “Data Streams”. Това са бинарни потоци от данни, които се използват за записване на различни типове информация. Изключително удобни са когато искаме да четем и пишем комбинирана информация от числа и символи. Ето един пример как можем да запишем един int, последван от double и текст:

```
java.io.DataOutputStream out = null;
try{
  out = new java.io.DataOutputStream(
                  new java.io.BufferedOutputStream(
                    new java.io.FileOutputStream("output.txt")
                  )
                );
  int i = 5;
  double d = 5.5;
  String str = "Hello";

  out.writeInt(i);
  out.writeDouble(d);
  out.writeUTF(str);
}
catch(java.io.IOException e){
  System.err.print("Error writing to file");
```

```
}
finally{
  try{
     if (out != null) out.close();
  }
  catch(java.io.IOException e){}
}
```

Ако искаме да прочетем обратно тази информация, то използваме “обратния обект” за четене:

```
java.io.DataInputStream in = null;
try{
  in = new java.io.DataInputStream(
                  new java.io.BufferedInputStream(
                     new java.io.FileInputStream("output.txt")
                  )
                );
  System.out.print(in.readInt());
  System.out.print(in.readDouble());
  System.out.print(in.readUTF());
}
catch(java.io.IOException e){
  System.err.print("Error reading file");
}
finally{
```

```
    try{
      if (in != null) in.close();
    }
    catch(java.io.IOException e){}
  }
```

Специално внимание трябва да обърнем на още един обект за писане във файл. Това е PrintStream(). Чрез него можем да използваме функциите print() и println(), които имат абсолютно същата функционалност както при System.out:

```
  java.io.PrintStream out = null;
  try {
    out = new java.io.PrintStream(new
java.io.FileOutputStream("output.txt"));
    int i = 123;
    out.print("Hello");
    out.println(" world "+i);
  }
  catch (java.io.FileNotFoundException e) {
        System.err.println("FileNotFound");
        return;
  }
  finally {
    if (out!=null) out.close();
  }
```

Вече сме готови да стигнем до същината на идеята на абстракциите на потоци в Java и да покажем как можем да предефинираме стандартния вход/изход. Ето един пример – пренасочваме System.out.* вместо да изкарва информация на екрана да я записва във файл:

```
try {
  System.setOut(new java.io.PrintStream(
                new java.io.FileOutputStream("output.txt")
            ));
}
catch (java.io.FileNotFoundException e) {
      System.err.println("FileNotFound");
      return;
}
System.out.println("Tozi text shte se zapishe vav file");
System.out.close();
```

Редно е да се спомене, че System.out и System.err всъщност са обекти от тип PrintStream. Именно поради тази причина можем да ги заменим по показания начин. Същото можем да направим и за стандартния вход. Помните ли какво говорихме за “буфера на клавиатурата” в миналата статия? Е, след като вече знаете какво е буфериран поток, то можете да се досетите, че System.in може да бъде пренасочен към BufferedInputStream():

```
try {
  System.setIn(new java.io.BufferedInputStream(
                new java.io.FileInputStream("input.txt")
            ));
  int c = System.in.read();
```

```
  while (c!=-1){
    System.out.print((char)c);
    c = System.in.read();
  }
}
catch (java.io.IOException e) {
        System.err.println("FileNotFound");
        return;
}
finally{
  try{
    if (System.in != null) System.in.close();
  }
  catch (java.io.IOException e){}
}
```

Виждате, че на практика четенето и писането в конзолата не е по-различно от това във файлове. По-нататък ще разгледаме и мрежови връзки (sockets) – те също не се различават съществено. Методиката на работа е една и съща.