Java Cookbook

Matthias Baitsch

May 2009

Introduction

This document provides strategies how to solve common Java programming tasks. In detail, the following topics are covered:

1. using elementary Java classes for arrays and strings,
2. reading from a file and writing to a file,
3. some Swing classes,
4. using a table to render and manipulate structured data,
5. using linear algebra classes and solving linear systems,
6. plotting functions,
7. defining a mathematical function,
8. creating two-dimensional drawings using coordinate transformation,
9. managing a graphical application including animation.

For each problem, some background information and a straightforward application example that can be adapted to your actual problem is provided.

Since the examples in this document are based on various external libraries, the “ICEB MPCE” plug-in has to be installed for eclipse. Please consider the Internet pages for the “Modern Programming Concepts in Engineering” course for details.

1 Elementary Java classes

1.1 Working with arrays

Java ships with a class `java.util.Arrays` which provides handy and efficient methods for common array tasks. Furthermore, the use of so called vararg-arguments can simplify code significantly.

Printing In order to quickly print an array to the console, the `toString` and `deep-ToString` methods can be used:

```
double[] a1 = { 6, 3, 9.99, 1 };
double[][] a2 = { { 3, 6, 2, 99 }, { 22, -1 } };
System.out.println("a1 = " + Arrays.toString(a1));
System.out.println("a2 = " + Arrays.deepToString(a2));
```

where the output is

```
a1 = [6.0, 3.0, 9.99, 1.0]
a2 = [[3.0, 6.0, 2.0, 99.0], [22.0, -1.0]]
```

Note that the `deepToString` method applies the `toString` method recursively to arrays of arrays (which might again contain arrays).

Sorting In order to sort an array, the `sort` method can be applied

```
double[] a3 = { 66, -99.4, 6, 3, 9.99, 1 };
Arrays.sort(a3);
System.out.println("a3 = " + Arrays.toString(a3));
```

Note that in the output, the array is sorted:

```
a3 = [-99.4, 1.0, 3.0, 6.0, 9.99, 66.0]
```

The `sort` method uses a variant of the quicksort algorithm which offers $n \log n$ performance on many data sets that cause other quicksorts to degrade to quadratic performance.

Searching In order to find the index of an entry in a *sorted* array, the `binarySearch` method can be used:

```
double[] a4 = { -99.4, 1.0, 3.0, 6.0, 9.99, 66.0 };
System.out.println("Index of 6 in a4: " + Arrays.binarySearch(a4, 6));
System.out.println("Index of 7 in a4: " + Arrays.binarySearch(a4, 7));
```

The method returns the index of the specified element, if the element is present in the array. Otherwise, a negative number n is returned such that $-n-1$ is the index of the first element larger than the specified number. Consequently, the output of the above code is

```
Index of 6 in a3: 3
Index of 7 in a3: -5
```

Note that the result of `binarySearch` is unspecified if the input array is not sorted in ascending order. As the name of the method implies, a binary search algorithm is employed such that the time complexity is $\log n$.

Varargs parameters Passing an arbitrary number of items as parameters to a method requires array type parameters. Without the use of vararg parameters, the array has to be created before invoking the method (see first two lines of `main` method). Using varargs, an arbitrary number of parameters can be passed to a method (see lines three and four of `main`). In many situations, this can save a lot of typing and makes the code much more readable. In order to use varargs in a method, you just use `type...` instead of `type[]`, e.g. `int...` instead of `int[]`. The only limitation is that you can have only one vararg type parameter and that it must be the last parameter in the parameter list.

```
public class Varargs {

   public static void main(String[] args) {
      double[] a = { 1, 2, 3 };
      print("a1", a);
      print("a2", 1, 2, 3);
      print("a3", 1, 2, 3, 4);
   }

   private static void print(String l, double... x) {
      System.out.print(l + " = ");
      for (int i = 0; i < x.length; i++) {
         System.out.print(x[i] + " ");
      }
      System.out.println();
   }
}
```

1.2 Strings – conversion, formatting and splitting

Converting Strings into elementary data types. When reading working with files or graphical user interfaces, data is provided as an object of type **`String`** where an elementary data type (such as `int` or `double`) is required for further processing. This problem can be solved using the classes accomplishing elementary types:

```
  int b = Integer.parseInt("187462897");
  double c = Double.parseDouble("33.3");
  long d = Long.parseLong("84987627854923874");
```

Note that it must be possible to convert the parameter passed to the parse-method into the corresponding type; otherwise an exception will be raised. For instance, the statement

```
  double e = Double.parseDouble("abc");
```

results in a `NumberFormatException` to be thrown since there is no common convention how to convert “abc” into a number. Serious applications have to handle such a situation adequately.

Formatting numbers Outputting numbers in a nicely formatted fashion is in Java not straightforward. By default, the full number of digits is included when converting a number into a string, such that

```
double f = 100000.0 / 3.0;
System.out.println("f=" + f);
```

gives

```
f=33333.333333333336
```

as output. Often, this large number number of displayed digits is not acceptable. To solve this problem, there basically exist two approaches: using a dedicated class or using a special print method.

The first solution is to use an object of type `java.text.NumberFormat`. Basic usage is

```
double f = 100000.0 / 3.0;
NumberFormat nf1 = DecimalFormat.getNumberInstance();
NumberFormat nf2 = new DecimalFormat("0.000000E00#");
System.out.println("Format 1: " + nf1.format(f));
System.out.println("Format 2: " + nf2.format(f));
```

where the corresponding output to the console is

```
Format 1: 33.333,333
Format 2: 3.3333333E04
```

The first number format object uses the default settings while for the second object, the formatting rules are specified explicitly. Note that the actual output might also depend on the language settings of your computer system.

The second option is to use the `printf` method defined in the `PrintStream` class. For example,

```
double f = 100000.0 / 3.0;
System.out.printf("f=%15.2f\n", f);
```

results in

```
f=       33333.33
```

The `printf` methods takes as parameters one string and an arbitrary number of further parameters. The first string describes how the remaining parameters are formatted. In the above example, we format the second parameter as floating point number reserving space for 15 characters while printing 2 decimal digits. This options is often suitable for printing numbers in tables but using it is not trivial and requires some reading of the documentation.