TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

ACTA TECHNICA NAPOCENSIS

Series: Applied Mathematics, Mechanics, and Engineering
Vol. 65, Issue I, March, 2022

SOME ISSUES RELATED TO THE DOMAIN AND ACCURACY OF THE NUMERICAL PRIMITIVE DATA TYPES IN JAVA THAT CAN BE AVOIDED USING OBJECTS BASED ON THE BigInteger AND BigDecimal CLASSES

Tiberiu Alexandru ANTAL

Abstract: *The paper aims to present the possibilities of the Java language to give up the primitive types, existing in most imperative programming languages and replace them with a group of predefined analog classes that work in the case of integer numbers with arbitrary precision (BigInteger) or in the case of real numbers with arbitrary precision decimal numbers (BigDecimal). Specific examples are presented to describe how to work with these special classes compared to equivalent primitive types and the results returned by them.*

Key words: *arbitrary precision, BigInteger, BigDecimal, integer, java, primitive, real.*

1. INTRODUCTION

In computer science a data model is a mathematical formalism for the description of the data structures and the operators for the validation and manipulation of the data. One category of data models is called strict because it provides predefined categories called “types” that must be used to describe the data. In the modeling process it is mandatory that any data be forced to be part of a certain type otherwise that data will not be able to be represented in the strict model. The Java language uses the strict data model and provides the programmer with data “types” through which he can model the problem to be solved. Technically the "type" determines how many bits are used for that particular data, and how the bits are to be interpreted.

1.1 A brief description of Java numerical primitive types

It is not the purpose of this article to describe in depth how data types are represented in Java, however it should be remembered that Java has two distinct categories of data types:

- primitive - are predefined in the Java programming language and named by their corresponding keyword; the numeric types are the integer types and the floating point types;
- reference - Java has four kinds of reference types - class type, interface type, type variables, and array type.

The integer types are `byte`, `short`, `int`, and `long` which are stored on 8, 16, 32 and 64 bits, using signed two’s complement. By default integer arithmetic is carried out using 32-bit precision and the result is of type int. If an integer operator has at least one long type operand, then the operation is carried out using 64-bit precision and the result of the operation is of type long. The integer operators do not indicate overflow and underflow which is why some results may come as a surprise (although they are correct).The floating point types are `float`, and `double` are stored on 32 and 64 bits using the ANSI/IEEE 754 -1985 standard for floating point number representation. The `float` floating point type (sometimes called single precision float) use 32 total bits and 24 bits for digits (and 8 exponent bits), yielding about 7 decimal digits of precision and a range from

Received: 08.01.22; **Similarities**: 09.02.22: **Reviewed**: 11.02./11.02.22: **Accepted**:17.03.22.

about 10^{-38} to 10^{38}. The `double` floating point type (often just called `double`) use 64 total bits and 53 bits for digits (and 11 exponent bits), yielding about 15/16 decimal digits of precision and a range from about 10^{-308} to 10^{308}. The standard's default rounding mode is round to nearest. A floating point operation that overflows produces a signed infinity. A floating point operation that underflow produces a signed zero. A floating point operation that is not mathematically defined produces NaN result.

2. USING BIG NUMBERS IN JAVA

As already mentioned, the primitive numeric types in Java are limited when it comes to precision. There are several categories of technical problems in which we want the accuracy to be as high as possible [5] - [8]. If the precision of the integers or the floating point numbers are not sufficient the `java.math` package contains two special classes, `BigInteger` and `BigDecimal`, which solve the problem of precision with the price of the calculation time and the writing method of the operators. Both classes implement arbitrary precision arithmetic for numbers (integer and floating point) and use methods for the familiar mathematical operations. The internal representation of the big numbers is different from the numeric primitive types, for this reason big classes have the a static `valueOf()` method to convert primitive type numbers to big numbers:

```
BigInteger ib = BigInteger.valueOf(1);
BigDecimal fb = BigDecimal.valueOf(1.);
```

Unlike C++, Java does not allow operator overloading. This means that there no way the `BigInteger` and `BigDecimal` classes to redefine operators like: +, -, *, / and % to form mathematical expressions. Instead, methods such as `add()`, subtract(), `multiply()`, `divide()` and `mod()` must be called to perform the corresponding mathematical operations.

```
//a = b + c; - if a,b,c are int
BigInteger a = b.add(c); //b, c are BigInteger
//z=x*y/(x+y); - x,y,z are int
BigInteger z = x.multiply(y).divide(x.add(y)); //
x and y are BigInteger
```

To compare two big numbers the `int compareTo(BigInteger arg)` method is provided; it returns 0 if `BigInteger` equals `arg`, a negative `int` result if `BigInteger` is less than `arg`, and a positive result otherwise. When working with `BigDecimal` the division is defined as `BigDecimal divide(BigDecimal arg, RoundingMode mode)`. To compute the quotient the *rounding mode* must be provided. The `RoundingMode.HALF_UP` is the way taught in primary school (digit 0, …, 4 are rounded down, digits 5, …, 9 are rounded up). Some division operations may produce an infinite number of decimals and for this case exceptions ([2], [3]) will be thrown and the division operation needs an integer called scale that limits the number of decimals to be produces in the result (see [4], 300 in the 2.2 example).

2.1 Using BigInteger to compute the factorial

A short Java program that can be used to compute the factorial ($n!=1\cdot2\cdot3\cdot\ldots\cdot n$) is:

```
import java.util.Scanner;
public class Factorial {
 public static void main(String[] args) {
  Scanner in =  new Scanner(System.in);
  System.out.print("n: ");
  long n = in.nextLong();
  long fact = 1L;
  for(long i = 1L; i<=n ; ++i)
   fact*=i;
  System.out.println(n+"! = "+fact);
 }}
```

Some of the obtained results are:

```
4!   =                  24
5!   =                 120
20!  = 2432902008176640000
21!  = -4249290049419214848
100! =                   0
```

The code is using the `long` integer type as the 64-bit precision is the highest that can be used in Java (the upper bound of the domain is computed as $+2^{63}-1$ = +9223372036854775807). As we can see 20! is the last "good value" as 21! can't be negative and 100! can't be zero. The `BigInteger` corresponding code is:

```
import java.math.BigInteger;
import java.util.*;
public class FactorialBI {
 public static void main(String[] args) {
  Scanner in =  new Scanner(System.in);
  System.out.print("n: ");
  long n = in.nextInt();
  BigInteger fact = BigInteger.valueOf(1);
  for(long i=1; i<=n ; ++i)
```

```
   fact = fact.multiply(BigInteger.valueOf(i));
  System.out.println(n + "! = " + fact);
 }}
```

```
20! =  2432902008176640000
21! = 51090942171709440000
100! =
933262154439441526816992388562667004907159682643
816214685929638952175999932299156089414639761565
182862536979208272237582511852109168640000000000
00000000000000
```

2.2 Using BigDecimal to reach a desired precision in the bisection method

The bisection method is a well-known numerical algorithm [1] that can be used to find the roots of the f(x) = 0 equation on a given interval [a, b]. Specific to this method is that the length (|a-b|) of the initial interval, in which the solution is found, will be reduced by modifying the initial [(a, a_1, a_2,... a_k), (b, b_1, b_2, b_m)] margins of the interval. The algorithm is iterative, and the condition of leaving the solution search loop is about finding a new interval [a_k, b_m], containing the solution, and having the length (|a_k-b_m|) under a very small given number (`eps`):

```
public class BisectionOO {
 private double a, b, x, eps;
 protected int max_iter;

 public double f(double x) {
  return 4. * Math.exp(-x) - x + 1.; }
 public BisectionOO(double st, double dr, double
prec) {
  a = st;  b = dr;   eps = prec;
  max_iter = 100;
  solve(); }
 public void solve() {
  int n = 0;
  while  ((Math.abs(b  -  a)  >  eps)  &&  (n++  <
max_iter)) {
   x = (a + b) / 2.;
   if (f(x) == 0) {
    System.out.println("Exact root: " + x);
    System.exit(0);   }
   if (Math.signum(f(a)) * Math.signum(f(x)) > 0)
    a = x;
   else
    b = x;   }
   if (Math.abs(b - a) > eps)
    System.out.println("Precision     was     not
reached in " + (n - 1) + " iterations");
   else
    System.out.printf("The  aprox.  root  is  %-
25.16g\n", x);
 }
 public static void main(String[] args) {
  BisectionOO ics = new BisectionOO(1., 4., 1.e-
15);
 }}
```

Getting a more precise result may be obtained by manipulating the number of the printed decimals in the format specifier: `%-25.16`, `%-25.17`, `%-25.19`, `%-25.20`. As shown in the following results this will not help us as the language will fill with zeros de decimal position where the result beyond the maximum precision of the `double` data type. Although the language documentation states that the precision of the representation of real numbers in `double` is limited to 16 decimal places it seems that we can even reach 17 decimal precision. However, this is not true, as the result printed by the `printf()` method is rounded before printing (the total number of digits in the resulting magnitude is rounded before printing):

```
The aprox. root is 1.717824512494594     %-25.16
The aprox. root is 1.7178245124945943    %-25.17
The aprox. root is 1.7178245124945943000 %-25.19
The aprox. root is 1.7178245124945943000 %-25.20
```

Another way of obtaining a better precision would involve setting the length of the distance between the left and the right margin to a lower value. In other to increase the precision this value can be set from 1.e-15 to 1.e-16 in the line `BisectionOO ics = new BisectionOO(1., 4., 1.e-16);` However, the margins result from computations, and there is no magic way for dealing with accumulation of errors and loss of precision so this approach will produce no more answers as the required precision can't be reached in the fixed given maximum number of iterations (in the code `max_iter = 100`). A solution of this new problem would be to increase the maximum number of iterations (`max_iter will get the values {1000,10000}`). However, the corresponding answers will be:

```
Precision was not reached in 100 iterations
Precision was not reached in 10000 iterations
Precision was not reached in 100000 iterations
```

The results show that the solutions can no longer be found, because the algorithm is entering an infinite loop as the required precision can no longer be reached. The margins are obtained by computation; the length is obtained by arithmetic from the computed margins, the termination condition of the loop is beyond the maximum precision of the language arithmetic of the `double` primitive type. The equivalent `BigDecimal` code is:

```
import java.math.BigDecimal;
import java.math.RoundingMode;
public class BisectionOOBDBI {
 private BigDecimal a, b, x, eps;
 protected int max_iter;
 static BigDecimal elax(int n, BigDecimal X) {
  BigDecimal exp_sum = BigDecimal.valueOf(1.);
  for (int i = n - 1; i > 0; --i)
```

```
   exp_sum =
BigDecimal.valueOf(1.).add(X.multiply(exp_sum.di
vide(BigDecimal.valueOf(i),                300,
RoundingMode.CEILING)));
  return exp_sum; }
 public static BigDecimal f(BigDecimal X) {
  return                              elax(300,
X.multiply(BigDecimal.valueOf(-
1.))).multiply(BigDecimal.valueOf(4.))
     .subtract(X).add(BigDecimal.valueOf(1.)); }
 public  BisectionOOBDBI(double  st,  double  dr,
double prec) {
  a = BigDecimal.valueOf(st);
  b = BigDecimal.valueOf(dr);
  eps = BigDecimal.valueOf(prec);
  max_iter = 10000; solve();
 }
 public void solve() {
  int n = 0;
  while   ((b.subtract(a).abs().compareTo(   eps)
==1) && (n++ < max_iter)) {
   x  =  a.add(b).divide(BigDecimal.valueOf(2.),
300, RoundingMode.CEILING);
   if (f(x).compareTo(BigDecimal.valueOf(0.)) ==
0) {
    System.out.println("Exact root: " + x);
    System.exit(0);
   }
   if (f(a).signum()*(f(x).signum()) ==1)
    a = x;
   else
    b = x;
   }
   if (b.subtract(a).abs().compareTo(eps) ==1)
    System.out.println("Precision     was     not
reached in " + (n - 1) + " iterations");
   else
    System.out.println("The  aprox.  root  is  " +
x);
 }
 public static void main(String[] args) {
  BisectionOOBDBI ics = new BisectionOOBDBI(1.,
4., 1.e-300);
 }}
```

```
The aprox. root is 1.71782451249459490159607232
81636610385167713133422468877196928944901956435 7
80205448834448320379392147734898433738444257223 0
20162849019964381488970257721874455727223502391 8
```

```
52997343042234891165460412749987333443538030479 0
16100627881447392026915566684442082907413341090 0
79128462604780944404898129144357 39
```

3. REFERENCES

[1] Antal, Tiberiu Alexandru. *Visual BASIC pentru ingineri.* Risoprint, 2003, p. 244, ISBN 973-656-514-9

[2] ANTAL, T. A., *Elemente de Java cu JDeveloper - îndrumător de laborator*, Editura UTPRES, 2013, p.150, ISBN: 978-973-662-827-6.

[3] ANTAL, T. A., *Java - Inițiere - îndrumător de laborator*, Editura UTPRES, 2013, p. 246, ISBN: 978-973-662-832-0.

[4] https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/math/BigDecimal.html#divide(java.math.BigDecimal,int,int)

[5] Husty, M., Birlescu, I., Tucan, P., Vaida, C., Pisla, D.: *An algebraic parameterization approach for parallel robots analysis*, Mechanism and Machine Theory, vol. 140, pp. 245-257, 2019.

[6] Tarnita, D., Pisla, D,. Geonea, I., Vaida, C., Catana, M., Tarnita D.N.: *Static and Dynamic Analysis of Osteoarthritic and Orthotic Human Knee*, Journal of Bionic Engineering, vol. 16(3), pp. 514-525, 2019

[7] Pisla, D., Plitea, N., Vaida, C. (c.a.), Hesselbach, J., Raatz, A., Vlad, L., Graur, F., PARAMIS Parallel Robot for Laparoscopic Surgery, (2010), Chirurgia 105(5), pp. 677-683

[8] Tucan, P., Vaida, C., Plitea, N., Pisla, A., Carbone, G., Pisla, D.: Risk-Based Assessment Engineering of a Parallel Robot Used in Post-Stroke Upper Limb Rehabilitation, Sustainability, vol. 11(10), 2893, 2019

Câteva probleme legate de domeniul şi precizia tipurilor de date numerice primitive în Java care pot fi evitate cu obiectele din clasele BigInteger şi BigDecimal

Lucrarea doreşte să prezinte posibilităţile limbajului Java de a evita tipurile primitive, existente în majoritatea limbajelor de programare imperative şi înlocuirea acestora un grup de clase analoage predefinite care lucrează cu precizie arbitrară. Exemple specifice sunt prezentate pentru descrie modul de lucru cu aceste clase speciale în comparație cu tipurile primitive echivalente precum şi rezultatele întoarse de către acestea.

ANTAL Tiberiu Alexandru, Professor, dr. eng., Technical University of Cluj-Napoca, Department of Mechanical System Engineering, antaljr@bavaria.utcluj.ro, 0264-401667, B-dul Muncii, Nr. 103-105, Cluj-Napoca, ROMANIA.