Decimal Arithmetic for Java

18th June 2000

Mike Cowlishaw

IBM Fellow
IBM UK Laboratories
mfc@uk.ibm.com

Version – 1.08

Table of Contents

Decimal arithmetic for Java

The Java™ runtime environment includes a class (*java.math.BigDecimal*) for decimal arithmetic. While suitable for simple financial calculations, it is missing a number of features that are necessary for general-purpose decimal arithmetic. This document describes what is missing, and proposes a small and upwards compatible enhancement to the BigDecimal class which makes the necessary additions.

Included in this document are:

- an overview of the proposal (this section)
- the concepts behind the design
- the description of the proposed classes
- a detailed definition of the arithmetic
- notes and change history.

The requirements

Java currently provides classes (*java.math.BigDecimal* and *java.math.BigInteger*) for fixed point arithmetic, and also supports native integer and binary floating point arithmetic directly. Binary floating point is usually implemented by hardware, and therefore is widely used for "numerically intensive" work where performance is critical.

However, with the growth in importance of applications where usability is the primary concern, the anomalies of binary floating point arithmetic (such as the inability to represent common values such as 0.10 exactly) are increasingly troublesome.[1] In financial and commercial applications, especially, an arithmetic which can achieve exact decimal results when required is essential. This is the original purpose of the BigDecimal class.

Unfortunately, the current BigDecimal class provides only a limited set of fixed point operations on numbers which are in practice limited to those which can be represented conveniently as "plain" numbers, with no exponent. There are, in addition, a number of problems with conversions to and from other Java types. These, and the other problems with the BigDecimal class, are listed overleaf.

---

[1] See, for example, *Floating point issues*, C. Sweeney, at:
`http://www.truebasic.com/tech08.html`

 Copyright (c) IBM Corporation 2000. All rights reserved.

Problems with the BigDecimal class:

1. The fixed point (integer + scale) arithmetic is suitable for some tasks (such as calculating taxes or balancing a check book), but is inconvenient and awkward for many common applications.

   For example, calculating the total amount repaid on a mortgage over 20 years is difficult, requiring several steps which do not involve exact arithmetic and which may require explicit rounding. For this task (and many others) an arithmetic that allows working to a chosen *precision* is both simpler and more convenient.

2. Several operators commonly used in applications are missing, specifically integer division, remainder, and exponentiation to an integer power (as required for straightforward calculation of the mortgage repayment just described, for example).

3. The constructors for BigDecimal do not accept exponential notation. This means that results from other sources (for example, spreadsheets and calculators, or the Java `Double.toString()` method) are difficult to use.

4. The string form of a BigDecimal is always a plain number. This means that very large or very small numbers are expressed using many digits – this makes them expensive and difficult to handle. For many calculations an exponential or floating point representation is desirable (and is potentially more efficient).

5. The conversions from BigDecimal to Java integer types are dangerous. Specifically, they are treated as a *narrowing primitive conversion*, even though there is a change of base involved. This means that decimal parts of numbers can be dropped without warning, and high order significant bits can also be lost without warning (an error sometimes called "decapitation"). It was exactly this kind of error that caused the loss of the Ariane 5 launcher in 1996.[2]

In the proposal that follows, these deficiencies are addressed by adding floating point arithmetic and exponential notation to the BigDecimal class, in a fully upwards-compatible and seamless manner. In addition, the set of base operators is completed, and new robust conversion methods are added.

The proposal

This proposal answers the primary requirements of the last section by adding support for decimal floating point arithmetic to the BigDecimal class. This is achieved by simply adding a second parameter to the existing operator methods. The augmented class implements the decimal arithmetic defined in the ANSI standard X3.274-1996,[3] which has the following advantages:

- The arithmetic was designed as a full-function decimal floating point arithmetic, directly implementing the rules that people are taught at school.

  For example, number length information is not lost, so trailing zeros can be correctly preserved in most operations: 1.20 times 2 gives 2.40, not 2.4. This behavior is essential both for usability and to maintain compatibility with the existing BigDecimal class.

---

[2] See: `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`

[3] *American National Standard for Information Technology – Programming Language REXX, X3.274-1996,* American National Standards Institute, New York, 1996.

 Copyright (c) IBM Corporation 2000. All rights reserved.

- Being a true decimal arithmetic, exact results are given when expected (for instance, 0.9/10 gives 0.09, not 0.089999996).
- The precision of the arithmetic is freely selectable by the user, not limited to a choice from one or two alternatives; where necessary, calculations may be made using thousands of digits. The operator definitions, and current implementations, impose no upper limit on precision (though in practice, memory or processor constraints will bound some calculations).
- The arithmetic operations are robust; there is no "wrap" of integers at certain sizes, and ill-defined or out-of-range results immediately throw exceptions.
- The concept of a *context* for operations is explicit. This allows application-global rules (such as precision and rounding) to be easily implemented and modified. This aids testing and error analysis, as well as simplifying programming.
- Integers and fixed-scale numbers are a proper subset of all numbers. Conversions to and from a different class are not necessary in order to carry out integer and currency calculations.
- A large range of numbers are supported; by definition, exponents in the range of at least E-999999999 through E+999999999 are supported, with a default precision of nine decimal digits. Both scientific (where one digit is shown before the decimal point) and engineering (where the power of ten is a multiple of three) exponential notations are supported.
- The arithmetic was developed over several years, based directly on user feedback and requirements, and in consultation with professional mathematicians and data processing experts. It has been heavily used for over 16 years without problems, and was recently reviewed in depth and ratified by the X3J18 committee for ANSI.
- Numerous public-domain and commercial implementations of the arithmetic exist. IBM has implementations in C, C++, various Assembler languages, and for Java.

This arithmetic has been further enhanced by supporting a variety of rounding algorithms, as already defined in Java 1.1 for the *java.math.BigDecimal* class.

A prototype of the proposed enhanced BigDecimal class has been specified (see the remainder of this document) and has been fully implemented, including javadoc comments following Java guidelines and an appropriate set of test cases.

It is a small class (at approximately 23,000 bytes, including line number tables, it is smaller than the BigInteger class in Java 1.2), and does not use any native methods. The class is based on code that has been in use since 1996, and which has been packaged as a BigDecimal class since June 1998. It has been available on the IBM alphaWorks site since late 1998.

For reasons explained later, the detail in this document also proposes adding one very small new context class to Java, in addition to enhancing the BigDecimal class. The new class would most logically be added to the *java.math* package in the Java Runtime Environment, and it is suggested that it be called *MathContext*.

The changes to the Java runtime proposed are summarized on the next page.

 Copyright (c) IBM Corporation 2000. All rights reserved.

The changes to the current Java API affect only two classes; BigDecimal (which is enhanced from the current specification) and MathContext (which is new).

BigDecimal

Instantiates a decimal number, and includes:

1. Constructors and methods for creating a BigDecimal number from the primitive Java types, and from strings and BigInteger object. Four constructors and one method have been added.
2. Operator methods, for the usual arithmetic operators, including comparisons. Four new operators have been added, and all operator methods have a second version which specifies a context.
3. Other methods, including standard Java methods (equals, hashCode, *etc.*), and conversions to primitive types and String (intValueExact, floatValue, toString, format, signum, *etc.*). Seven methods have been added, mostly to effect robust (error-detecting) conversions.

This initial proposal does not include transcendental functions.

MathContext

A very small class, used for defining a context for arithmetic, as described in the next section. This comprises four constructors and five methods (four "get" methods and a `toString()` method).

These classes are available in the package *com.ibm.math* (that is, as the classes *com.ibm.math.BigDecimal* and *com.ibm.math.MathContext*). Comments on them and on this draft are welcome. Please send comments to Mike Cowlishaw, `mfc@uk.ibm.com`.

Acknowledgements

Very many people have contributed to the arithmetic described in this document, especially the IBM REXX language committee, the IBM Vienna Compiler group, and the X3 (now NCITS) J18 technical committee. Special thanks for their contributions to the current design are due to Joshua Bloch, Dirk Bosmans, and Brian Marks.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Design concepts

The decimal arithmetic defined here was designed with people in mind, and necessarily has a paramount guiding principle – *computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school.*

Many people are unaware that the algorithms taught for "manual" decimal arithmetic are quite different in different countries, but fortunately (and not surprisingly) the end results differ only in details of presentation.

The arithmetic described here was based on an extensive study of decimal arithmetic and was then evolved over several years (1978-1982) in response to feedback from thousands of users in more than forty countries. Later minor refinements were made during the process of ANSI standardization.

In the past sixteen years the arithmetic has been used successfully for hundreds of thousands of applications covering the entire spectrum of computing; among other fields, that spectrum includes operating system scripting, text processing, commercial data processing, engineering, scientific analysis, and pure mathematics research. From this experience we are confident that the various defaults and other design choices are sound.

Fundamental concepts

When people carry out arithmetic operations, such as adding or multiplying two numbers together, they commonly use decimal arithmetic where the decimal point "floats" as required, and the result that they eventually write down depends on three items:

1. the specific operation carried out
2. the explicit information in the operand or operands themselves
3. the information from the implied context in which the calculation is carried out (the precision required, *etc.*).

The information explicit in the written representation of an operand is more than that conventionally encoded for floating point arithmetic. Specifically, there is:

- an optional *sign* (only significant when negative)
- a numeric part, or *numeric*, which may include a decimal point (which is only significant if followed by any digits)
- an optional *exponent*, which denotes a power of ten by which the numeric is multiplied (significant if both the numeric and exponent are non-zero).

The length of the numeric and original position of the decimal point are not encoded in traditional floating point representations, such as ANSI/IEEE 854-1987,[4] yet they are essential information if the expected result is to be obtained.

[4] ANSI/IEEE 854-1987 – *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1987.

 Copyright (c) IBM Corporation 2000. All rights reserved.

For example, people expect trailing zeros to be indicated properly in a result: the sum `1.57 + 2.03` should result in `3.60`, not `3.6`; however, if the positional information has been lost during the operation it is no longer possible to show the expected result.

Similarly, decimal arithmetic in a scientific or engineering context is based on a floating point model, not a fixed point or fixed scale model (indeed, this is the original basis for the concepts behind binary floating point). Fixed point decimal arithmetic packages such as ADAR[5] or the BigDecimal class in Java 1.1 are therefore only useful for a subset of the problems for which arithmetic is used.

The information contained in the context of a calculation is also important. It usually applies to an entire sequence of operations, rather than to a single operation, and is not associated with individual operands. In practice, sensible defaults can be provided, though provision for user control is necessary for many applications.

The most important contextual information is the desired precision for the calculation. This can range from rather small values (such as six digits) through very large values (hundreds or thousands of digits) for certain problems in Mathematics and Physics. Most decimal arithmetics implemented to date (for example, the decimal arithmetic in the Atari OS,[6] or in the IEEE 854-1987 standard referred to above) offer just one or two alternatives for precision – in some cases, for apparently arbitrary reasons. Again, this does not match the user model of decimal arithmetic; one designed for people to use must provide a wide range of available precisions.

The provision of context for arithmetic operations is therefore a necessary precondition if the desired results are to be achieved, just as a "locale" is needed for operations involving text.

This proposal provides for explicit control over four aspects of the context: the required *precision* – the point at which rounding is applied, the *rounding algorithm* to be used when digits have to be discarded, the preferred *form* of exponential notation to be used for results, and whether *lost digits* checking is to be applied. Other items could be included as future extensions (see page 40).

Embodiment of the concepts

The two kinds of information described (operands and context) are conveniently and naturally represented by two classes for Java: one that represents decimal numbers and implements the operations on those numbers, and one that simply represents the context for decimal arithmetic operations. It is proposed that these classes be called *BigDecimal* and *MathContext* respectively. The BigDecimal class enhances the original class of that name by adding floating point arithmetic.

The defaults for the context have been tuned to satisfy the expectations of the majority of users, and have withstood the test of time well. In the vast majority of cases, therefore, the default MathContext object is all that is required.

---

[5] "Ada Decimal Arithmetic and Representations"
See *An Ada Decimal Arithmetic Capability*, Brosgol et al. 1993.
`http://www.cdrom.com/pub/ada/swcomps/adar/`

[6] See, for example, *The* [*Atari*] *Floating Point Arithmetic Package*, C. Lisowski.
`http://intrepid.mcs.kent.edu/%7Eclisowsk/8bit/atr11.html`

 Copyright (c) IBM Corporation 2000. All rights reserved.

The BigDecimal class

```
public class BigDecimal
extends java.lang.Number
implements java.lang.Serializable, java.lang.Comparable
```

The BigDecimal class provides immutable arbitrary-precision decimal numbers. The methods of the BigDecimal class provide operations for fixed and floating point arithmetic, comparison, format conversions, and hashing.

As the numbers are decimal, there is an exact correspondence between an instance of a BigDecimal object and its String representation; the BigDecimal class provides direct conversions to and from String and character array objects, as well as conversions to and from the Java primitive types (which may not be exact) and BigInteger.

In the descriptions of constructors and methods that follow, the value of a BigDecimal number object is shown as the result of invoking the toString() method on the object. The internal representation of a decimal number is neither defined nor exposed, and is not permitted to affect the result of any operation.

Operations on BigDecimal numbers are controlled by a MathContext object (see page 23), which provides precision and other information. Default settings are used if no MathContext object is provided.

The names of methods in this class follow the conventions established by *java.lang.Number*, *java.math.BigInteger*, and *java.math.BigDecimal* in Java 1.1 and Java 1.2.

Copyright (c) IBM Corporation 2000. All rights reserved.

Fields

The following constant (static and final) fields are provided by the BigDecimal class.

ZERO

A BigDecimal object whose value is exactly `0`.

ONE

A BigDecimal object whose value is exactly `1`.

TEN

A BigDecimal object whose value is exactly `10`.

In addition, the constant fields describing rounding modes (those whose name starts with **ROUND_**) from the Java 1.1 BigDecimal class are preserved. These have the same names and values as the corresponding fields in the MathContext class.[7]

[7] These fields are preserved to maintain upwards compatibility. The MathContext class is their logical home – the constants in that class should be used in preference to those in the BigDecimal class.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Constructors

These constructors create an object of type `BigDecimal` from some other object or primitive value. In all cases except construction from a `double` value (for which exact conversion is often not possible) sufficient digits are used to represent the original value exactly.

BigDecimal(char[])

Constructs a BigDecimal as though the character array had been copied to a String and the **BigDecimal(String)** constructor (see page 10) had then been used. The parameter must not be `null`.

BigDecimal(char[], int, int)

Constructs a BigDecimal from a subarray of characters. The first parameter is the array in which the subarray is to be found, and the other parameters specify its offset and length respectively.

The BigDecimal is constructed as though the subarray had been copied to a String and the **BigDecimal(String)** constructor (see page 10) had then been used. The first parameter must not be `null`, and the subarray must be wholly contained within the array.

BigDecimal(double)

Constructs a BigDecimal which is an exact decimal representation of the 64-bit signed binary floating point parameter. If the parameter is infinite, or is not a number (NaN), a `NumberFormatException` is thrown.

Note: this constructor provides an exact conversion, so does not give the same result as converting the `double` to a `String` using the **Double.toString()** method and then using the **BigDecimal(String)** constructor. For that result, use the static **valueOf(double)** method (see page 22) to construct a BigDecimal from a `double` (or from a `float`).

BigDecimal(int)

Constructs a BigDecimal which is the exact decimal representation of the 32-bit signed binary integer parameter. The BigDecimal will contain only decimal digits, prefixed with a leading minus sign (hyphen) if the parameter is negative. A leading zero will be present only if the parameter is zero.

BigDecimal(java.math.BigDecimal)

Constructs a BigDecimal as though the parameter had been represented as a String (using its **toString** method) and the **BigDecimal(java.lang.String)** constructor (see page 10) (see below) had then been used. The parameter must not be `null`.

(Note: this constructor is provided only in the `com.ibm.math` *version of the BigDecimal class. It would not be present in a* `java.math` *version.)*

 Copyright (c) IBM Corporation 2000. All rights reserved.

BigDecimal(java.math.BigInteger)

Constructs a BigDecimal which is the exact decimal representation of the BigInteger parameter. The parameter must not be `null`.

The BigDecimal will contain only decimal digits, prefixed with a leading minus sign (hyphen) if the BigInteger is negative. A leading zero will be present only if the BigInteger is zero.

BigDecimal(java.math.BigInteger, int)

Constructs a BigDecimal which is the exact decimal representation of the BigInteger, scaled by the second parameter (the *scale*). The value of the BigDecimal is the BigInteger divided by ten to the power of the scale. The BigInteger parameter must not be `null`.

The BigDecimal will contain only decimal digits (with an embedded decimal point followed by *scale* decimal digits if the scale is positive), prefixed with a leading minus sign (hyphen) if the BigInteger is negative. A leading zero will be present only if the BigInteger is zero.

A `NumberFormatException` is thrown if the scale is negative.

BigDecimal(java.lang.String)

Constructs a BigDecimal from the parameter, which must represent a valid *number* (see page 29). The parameter must not be `null`.

In summary, numbers in String form must have at least one digit, may have a leading sign, may have a decimal point, and exponential notation may be used. They follow conventional syntax, and may not contain blanks.

Some valid Strings from which a BigDecimal might be constructed are:

```
  "0"         /* Zero                          */
 "12"         /* A whole number                */
"-76"         /* A signed whole number         */
 "12.70"      /* Some decimal places           */
 "+0.003"     /* A plus sign is allowed, too.  */
 "17."        /* The same as 17                */
   ".5"       /* The same as 0.5               */
 "4E+9"       /* Exponential notation          */
  "0.73e-7"   /* Exponential notation          */
```

(Exponential notation means that the number includes an optional sign and a power of ten following an "`E`" that indicates how the decimal point will be shifted. Thus the `"4E+9"` above is just a short way of writing `4000000000`, and the `"0.73e-7"` is short for `0.000000073`.)

The BigDecimal constructed from the String is in a standard form, as though the **add** method (see page 13) had been used to add zero to the number with unlimited precision.[8]

If the string uses exponential notation (that is, includes an `e` or an `E`), then the BigDecimal number will be expressed in scientific notation (where the power of ten

[8] That is, with a **digits** setting (see page 26) of 0.

 Copyright (c) IBM Corporation 2000. All rights reserved.

is adjusted so there is a single non-zero digit to the left of the decimal point). In this case if the number is zero then it will be expressed as the single digit 0, and if non-zero it will have an exponent unless that exponent would be 0. The exponent must fit in nine digits both before and after it is expressed in scientific notation.

Any digits in the parameter must be decimal; that is:

```
java.lang.Character.digit(c, 10)
```

(where `c` is the character in question) would not return -1.

A `NumberFormatException` is thrown if the parameter is not a valid number or the exponent will not fit in nine digits.

BigDecimal(long)

Constructs a BigDecimal which is the exact decimal representation of the 64-bit signed binary integer parameter. The BigDecimal will contain only decimal digits, prefixed with a leading minus sign (hyphen) if the parameter is negative. A leading zero will be present only if the parameter is zero.

Copyright (c) IBM Corporation 2000. All rights reserved.

Operator methods

These methods implement the standard arithmetic operators for BigDecimal objects.

Each of the methods here[9] takes a MathContext (see page 23) object as a parameter, which must not be `null` but which is optional in that a version of each of the methods is provided which does not require the MathContext parameter. This is indicated below by square brackets in the method prototypes.

The MathContext parameter provides the numeric settings for the operation (precision, and so on). If the parameter is omitted, then the settings used are `"digits=0 form=PLAIN lostDigits=0 roundingMode=ROUND_HALF_UP"`.[10] If `MathContext.DEFAULT` is provided for the parameter then the default values of the settings are used (`"digits=9 form=SCIENTIFIC lostDigits=0 roundingMode=ROUND_HALF_UP"`).

For monadic operators, only the optional MathContext parameter is present; the operation acts upon the current object.

For dyadic operators, a BigDecimal parameter is always present; it must not be `null`. The operation acts with the current object being the left-hand operand and the BigDecimal parameter being the right-hand operand.

For example, adding two BigDecimal objects referred to by the names `award` and `extra` could be written as any of:

```
award.add(extra)
award.add(extra, MathContext.DEFAULT)
award.add(extra, acontext)
```

(where `acontext` is a MathContext object), which would return a BigDecimal object whose value is the result of adding `award` and `extra` under the appropriate context settings.

When a BigDecimal operator method is used, a set of rules define what the result will be (and, by implication, how the result would be represented as a character string). These rules are defined in the Decimal arithmetic section (see page 29), but in summary:

- Results are normally calculated with up to some maximum number of significant digits. For example, if the MathContext parameter for an operation were `MathContext.DEFAULT` then the result would be rounded to 9 digits; the division of 2 by 3 would then result in 0.666666667.

  You can change the default of 9 significant digits by providing the method with a suitable MathContext object. This lets you calculate using as many digits as you need – thousands, if necessary. Fixed point (scaled) arithmetic is indicated by using a **digits** setting of 0 (or omitting the MathContext parameter).

  Similarly, you can change the algorithm used for rounding from the default "classic" algorithm.

---

9 Except for two forms of the **divide** method, which are preserved from the original BigDecimal class.

10 This performs fixed point arithmetic with unlimited precision, as defined for the original BigDecimal class in Java.

 Copyright (c) IBM Corporation 2000. All rights reserved.

- In standard arithmetic (that is, when the **form** setting is not `PLAIN`), a zero result is always expressed as the single digit `'0'` (that is, with no sign, decimal point, or exponent part).
- Except for the division and power operators in standard arithmetic, trailing zeros are preserved (this is in contrast to binary floating point operations and most electronic calculators, which lose the information about trailing zeros in the fractional part of results).

  So, for example:

```
'2.40'.add('2')                          => '4.40'
'2.40'.subtract('2')                     => '0.40'
'2.40'.multiply('2')                     => '4.80'
'2.40'.divide('2', MathContext.DEFAULT) => '1.2'
```

  This preservation of trailing zeros is desirable for most calculations (including financial calculations).

  If necessary, trailing zeros may be easily removed using division by 1.
- In standard arithmetic, exponential form is used for a result depending on its value and the current setting of **digits** (the default is 9 digits). If the number of places needed before the decimal point exceeds the **digits** setting, or the absolute value of the number is less than `0.000001`, then the number will be expressed in exponential notation; thus

```
'1e+6'.multiply('1e+6', MathContext.DEFAULT)
```

  results in "`1E+12`" instead of "`1000000000000`", and

```
'1'.divide('3E+10', MathContext.DEFAULT)
```

  results in "`3.33333333E-11`" instead of "`0.0000000000333333333`".

  The form of the exponential notation (scientific or engineering) is determined by the **form** setting.

The descriptions of the operator methods follow.

abs([MathContext])

Returns the absolute value of the BigDecimal number. If the current object is zero or positive, then the same result as invoking the **plus** method (see page 15) with the same parameter is returned. Otherwise, the same result as invoking the **negate** method (see page 15) with the same parameter is returned.

add(BigDecimal[, MathContext])

Implements the addition (+) operator (see page 32), and returns the result as a BigDecimal object.

compareTo(BigDecimal[, MathContext])

Implements comparison, using *numeric comparison* (see page 36), and returns a result of type `int`. The result will be:

 Copyright (c) IBM Corporation 2000. All rights reserved.

`-1` if the current object is less than the first parameter

`0` if the current object is equal to the first parameter

`1` if the current object is greater than the first parameter.

A **compareTo(Object)** method (see page 16) is also provided.

divide(BigDecimal[, MathContext])

Implements the division (`/`) operator (see page 33), and returns the result as a BigDecimal object.

divide(BigDecimal, int)

Implements the division (`/`) operator (see page 33) using settings "`digits=0 form=PLAIN lostDigits=0 roundingMode=`*`rounding_mode`*", where *`rounding_mode`* is the second parameter, and returns the result as a BigDecimal object.

The length of the decimal part (the scale) of the result will be the same as the scale of the current object, if the latter were formatted without exponential notation.

An `IllegalArgumentException` is thrown if *`rounding_mode`* is not a valid rounding mode.

divide(BigDecimal, int, int)

Implements the division (`/`) operator (see page 33) using settings "`digits=0 form=PLAIN lostDigits=0 roundingMode=`*`rounding_mode`*", where *`rounding_mode`* is the third parameter, and returns the result as a BigDecimal object.

The length of the decimal part (the scale) of the result is specified by the second parameter. An `ArithmeticException` is thrown if this parameter is negative.

An `IllegalArgumentException` is thrown if *`rounding_mode`* is not a valid rounding mode.

divideInteger(BigDecimal[, MathContext])

Implements the integer division operator (see page 34), and returns the result as a BigDecimal object.

max(BigDecimal[, MathContext])

Returns the larger of the current object and the first parameter.

If calling the **compareTo** method (see page 13) with the same parameters would return `1` or `0`, then the result of calling the **plus** method on the current object (using the same MathContext parameter) is returned. Otherwise, the result of calling the **plus** method on the first parameter object (using the same MathContext parameter) is returned.

min(BigDecimal[, MathContext])

Returns the smaller of the current object and the first parameter.

 Copyright (c) IBM Corporation 2000. All rights reserved.

If calling the **compareTo** method (see page 13) with the same parameters would return `-1` or `0`, then the result of calling the **plus** method on the current object (using the same MathContext parameter) is returned. Otherwise, the result of calling the **plus** method on the first parameter object (using the same MathContext parameter) is returned.

multiply(BigDecimal[, MathContext])

Implements the multiply (*) operator (see page 32), and returns the result as a BigDecimal object.

negate([MathContext])

Implements the negation (Prefix –) operator (see page 32), and returns the result as a BigDecimal object.

plus([MathContext])

Implements the plus (Prefix +) operator (see page 32), and returns the result as a BigDecimal object.

Note: This method is provided for symmetry with negate (this page) and also for tracing, *etc.* It is also useful for rounding or otherwise applying a context to a decimal value.

pow(BigDecimal[, MathContext])

Implements the power operator (see page 34), and returns the result as a BigDecimal object.

The first parameter is the power to which the current number will be raised; it must be in the range -999999999 through 999999999, and must have a decimal part of zero. If no MathContext parameter is specified (or its **digits** property is 0) the power must be zero or positive. If any of these conditions is not met, an `ArithmeticException` is thrown.[11]

remainder(BigDecimal[, MathContext])

Implements the remainder operator (see page 35), and returns the result as a BigDecimal object. (This is not the modulo operator – the result may be negative.)

subtract(BigDecimal[, MathContext])

Implements the subtract (–) operator (see page 32), and returns the result as a BigDecimal object.

---

[11] Note that the range and decimal part restrictions may be removed in the future, so should not be relied upon to produce an exception.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Other methods

These methods provide the standard Java methods for the class, along with conversions to primitive types, scaling, and other useful methods.

None of the methods here take a MathContext object (see page 23) as a parameter.

byteValueExact()

Converts the BigDecimal to type `byte`. If the BigDecimal has a non-zero decimal part or is out of the possible range for a `byte` (8-bit signed integer) result then an `ArithmeticException` is thrown.

Note: the inherited method **Number.byteValue()** is also available; if that method is used, then if the BigDecimal is out of the possible range for a `byte` (8-bit signed integer) result then only the low-order 8 bits are used. (That is, the number may be *decapitated*) To avoid unexpected errors when these conditions occur, use the **byteValueExact()** method instead.

compareTo(java.lang.Object)

Compares the BigDecimal with the value of the parameter, and returns a result of type `int`.

If the parameter is `null`, or is not an instance of the BigDecimal type, an exception is thrown. Otherwise, the parameter is cast to type `BigDecimal` and the value of **compareTo(BigDecimal)** using the cast parameter is returned.

The **compareTo(BigDecimal[, MathContext])** method (see page 13) should be used when the type of the parameter is known or when a `MathContext` is needed.

Implements **Comparable.compareTo(java.lang.Object)**.

doubleValue()

Converts the BigDecimal to type `double`.

The double produced is identical to result of expressing the BigDecimal as a String and then converting it using the **Double(String)** constructor; this can result in values of `Double.NEGATIVE INFINITY` or `Double.POSITIVE INFINITY`.

Implements **Number.doubleValue()**.

equals(java.lang.Object)

Compares the BigDecimal with the value of the parameter, and returns a result of type `boolean`.

If the parameter is `null`, or is not an instance of the BigDecimal type, or is not exactly equal to the current BigDecimal object, then *false* is returned. Otherwise, *true* is returned.

"Exactly equal", here, means that the String representations of the BigDecimal numbers are identical (they have the same characters in the same sequence).

 Copyright (c) IBM Corporation 2000. All rights reserved.

The **compareTo(BigDecimal[, MathContext])** method (see page 13) should be used for more general numeric comparisons.

Overrides **Object.equals(java.lang.Object)**.

floatValue()

Converts the BigDecimal to type `float`.

The float produced is identical to result of expressing the BigDecimal as a String and then converting it using the **Float(String)** constructor; this can result in values of `Float.NEGATIVE INFINITY` or `Float.POSITIVE INFINITY`.

Implements **Number.floatValue()**.

format(int, int) *and*

format(int, int, int, int, int, int)

Converts the BigDecimal to type `java.lang.String`, under the control of formatting (layout) parameters. See also the **toString** method (see page 21).

The **format** method is provided as a primitive for use by more sophisticated classes that will apply locale-sensitive editing of the result. The level of formatting that it provides is a necessary part of the BigDecimal class as it is sensitive to and must follow the calculation and rounding rules for BigDecimal arithmetic.

The parameters, all of type `int`, are provided to control the format of the String returned. In the prototypes below, parameters are given names for ease of reference. A value of -1 for a parameter indicates that the default action or value for that parameter should be used.

If an error is detected during the execution of this method, an `ArithmeticException` is thrown.

Most commonly, **format** is called with two parameters:

format(*before***,** *after***)**

The arguments *before* and *after* may be specified to control the number of characters to be used for the integer part and decimal part of the result respectively. If either of these is -1 (which indicates the default action), the number of characters used will be as many as are needed for that part.

before must be a positive number; if it is larger than is needed to contain the integer part, that part is padded on the left with blanks to the requested length. If *before* is not large enough to contain the integer part of the number (including the sign, for negative numbers) an error results.

after must be a non-negative number; if it is not the same size as the decimal part of the number, the number will be rounded (or extended with zeros) to fit. Specifying 0 for *after* will cause the number to be rounded to an integer (that is, it will have no decimal part or decimal point).

 Copyright (c) IBM Corporation 2000. All rights reserved.

Examples: (In this and following examples, simple method notation for the BigDecimal(String) constructor is used for clarity – the "*new*" operator would be added for use from Java.)

```
BigDecimal(" -12.73").format(-1,-1) == "-12.73"
BigDecimal("0.000").format(-1,-1)   == "0.000"
BigDecimal("3").format(4,-1)        == "   3"
BigDecimal("1.73").format(4,0)      == "   2"
BigDecimal("1.73").format(4,3)      == "   1.730"
BigDecimal("-.76").format(4,1)      == "  -0.8"
BigDecimal("3.03").format(4,-1)     == "   3.03"
BigDecimal("3.03").format(4,3)      == "   3.030"
BigDecimal("3.03").format(4,1)      == "   3.0"
BigDecimal(" -12.73").format(-1,4)  == "-12.7300"
```

A further four arguments may be passed to the format method to control the use of exponential notation and rounding. The syntax of the method with these arguments added is then:

format(*before***,** *after***,** *explaces***,** *exdigits***,** *exform***,** *exround***)**

The first two arguments are as already described. The next three (*explaces*, *exdigits*, and *exform*) control the exponent part of the result. As before, the default action for any of these arguments may be selected by using the value -1.

explaces must be a positive number; it sets the number of places (digits after the sign of the exponent) to be used for any exponent part, the default being to use as many as are needed. If *explaces* is specified (is not -1), space is always reserved for an exponent; if one is not needed (for example, if the exponent will be 0) then *explaces*+2 blanks are appended to the result. If *explaces* is specified and is not large enough to contain the exponent, an error results.

exdigits sets the trigger point for use of exponential notation. If, before any rounding, the number of places needed before the decimal point exceeds *exdigits*, or if the absolute value of the result is less than `0.000001`, then exponential form will be used, provided that *exdigits* was specified. When *exdigits* is -1, exponential notation will never be used. If 0 is specified for *exdigits*, exponential notation is always used unless the exponent would be 0.

exform sets the form for exponential notation (if needed). *exform* may be either `SCIENTIFIC` (the default) or `ENGINEERING` from the MathContext (see page 23) class. If the latter, engineering, form is requested, up to three digits (plus sign) may be needed for the integer part of the result (*before*).

Examples:

```
BigDecimal("12345.73").format(-1,-1,2,2,-1,-1)  == "1.234573E+04"
BigDecimal("12345.73").format(-1,3,-1,0,-1,-1)  == "1.235E+4"
BigDecimal("1.234573").format(-1,3,-1,0,-1,-1)  == "1.235"
BigDecimal("123.45").format(-1,3,2,0,-1,-1)     == "1.235E+02"
BigDecimal("1234.5").format(-1,3,2,0,eng,-1)    == "1.235E+03"
BigDecimal("12345").format(-1,3,2,0,eng, -1)    == "12.345E+03"
BigDecimal("1.2345").format(-1,3,2,0,-1,-1)     == "1.235    "
BigDecimal("12345.73").format(-1,-1,3,6,-1,-1)  == "12345.73     "
BigDecimal("12345e+5").format(-1,3,-1,-1,-1,-1) == "1234500000.000"
```

(where eng has the value `MathContext.ENGINEERING`).

 Copyright (c) IBM Corporation 2000. All rights reserved.

Finally, the sixth argument, *exround*, selects the rounding algorithm to be used, and must be one of the values indicated by a public constant in the MathContext class (see page 24) whose name starts with `ROUND_`. The default (`ROUND_HALF_UP`) may also be selected by using the value -1, as before.

Examples:

```
halfdown=MathContext.ROUND_HALF_DOWN
halfeven=MathContext.ROUND_HALF_EVEN
halfup  =MathContext.ROUND_HALF_UP
BigDecimal("0.05").format(-1,1,-1,-1,-1,halfdown)  == "0.0"
BigDecimal("0.05").format(-1,1,-1,-1,-1,halfeven)  == "0.0"
BigDecimal("0.15").format(-1,1,-1,-1,-1,halfeven)  == "0.2"
BigDecimal("0.05").format(-1,1,-1,-1,-1,halfup)    == "0.1"
```

The special value `MathContext.ROUND_UNNECESSARY` may be used to detect whether non-zero digits are discarded – if *exround* has this value than if non-zero digits would be discarded (rounded) during formatting then an `ArithmeticException` is thrown.

hashCode()

Returns a hashcode of type `int` for the object. This hashcode is suitable for use by the `java.util.Hashtable` class.

Overrides **Object.hashCode()**.

intValue()

Converts the BigDecimal to type `int`. If the BigDecimal has a non-zero decimal part it is discarded. If the BigDecimal is out of the possible range for an `int` (32-bit signed integer) result then only the low-order 32 bits are used. (That is, the number may be *decapitated*) To avoid unexpected errors when these conditions occur, use the **intValueExact()** method instead.

Implements **Number.intValue()**.

intValueExact()

Converts the BigDecimal to type `int`. If the BigDecimal has a non-zero decimal part or is out of the possible range for an `int` (32-bit signed integer) result then an `ArithmeticException` is thrown.

longValue()

Converts the BigDecimal to type `long`. If the BigDecimal has a non-zero decimal part it is discarded. If the BigDecimal is out of the possible range for a `long` (64-bit signed integer) result then only the low-order 64 bits are used. (That is, the number may be *decapitated*) To avoid unexpected errors when these conditions occur, use the **longValueExact()** method instead.

Implements **Number.longValue()**.

 Copyright (c) IBM Corporation 2000. All rights reserved.

longValueExact()

Converts the BigDecimal to type `long`. If the BigDecimal has a non-zero decimal part or is out of the possible range for a `long` (64-bit signed integer) result then an `ArithmeticException` is thrown.

movePointLeft(int)

Returns a plain BigDecimal whose decimal point has been moved to the left by the number of positions, *n*, given by the parameter. That is, it returns:

```
this.multiply(TEN.pow(new BigDecimal(–n)))
```

n may be negative, in which case the method returns the same result as **movePointRight(**-*n***)**.

movePointRight(int)

Returns a plain BigDecimal whose decimal point has been moved to the right by the number of positions, *n*, given by the parameter. That is, it returns:

```
this.multiply(TEN.pow(new BigDecimal(n)))
```

n may be negative, in which case the method returns the same result as **movePointLeft(**-*n***)**.

scale()

Returns a non-negative `int` which is the scale of the number. The scale is the number of digits in the decimal part of the number if the number were formatted without exponential notation.

setScale(int[,int])

Returns a plain BigDecimal whose *scale* is given by the first parameter.

If the scale is the same as or greater than the scale of the current BigDecimal then trailing zeros will be added as necessary.

If the scale is less than the scale of the current BigDecimal, then trailing digits will be removed, and the *rounding mode* given by the second parameter is used to determine if the remaining digits are affected by a carry. In this case, an `IllegalArgumentException` is thrown if *rounding mode* is not a valid rounding mode.

The default rounding mode is **ROUND_UNNECESSARY**, which means that an `ArithmeticException` is thrown if any discarded digits are non-zero.

An `ArithmeticException` is thrown if the *scale* is negative.

shortValueExact()

Converts the BigDecimal to type `short`. If the BigDecimal has a non-zero decimal part or is out of the possible range for a `short` (16-bit signed) result then an `ArithmeticException` is thrown.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Note: the inherited method **Number.shortValue()** is also available; if that method is used, then if the BigDecimal is out of the possible range for a `short` (16-bit signed integer) result then only the low-order 16 bits are used. (That is, the number may be *decapitated*) To avoid unexpected errors when these conditions occur, use the **shortValueExact()** method instead.

signum()

Returns an `int` value that represents the sign of the BigDecimal. That is, -1 if the BigDecimal is negative, 0 if it is exactly zero, or 1 if it is positive.

toBigDecimal()

Returns the number as an object of type `java.math.BigDecimal`. This is an exact conversion; the result is the same as if the BigDecimal were formatted without any rounding or exponent and then the **java.math.BigDecimal(java.lang.String)** constructor were used to construct the result.

(Note: this method is provided only in the `com.ibm.math` *version of the BigDecimal class. It would not be present in a* `java.math` *version.)*

toBigInteger()

Returns the number as an object of type `java.math.BigInteger`. Any decimal part is truncated (discarded).

toBigIntegerExact()

Returns the number as an object of type `java.math.BigInteger`. If the BigDecimal has a non-zero decimal part then an `ArithmeticException` is thrown.

toCharArray()

Returns the BigDecimal as a character array of type `char[]`, as though the sequence `toString().toCharArray()` had been used.

toString()

Returns the standard string of type `java.lang.String` that exactly represents the BigDecimal, as described in the BigDecimal arithmetic definition (see page 29).

The **format** method (see page 17) is provided for controlled formatting of BigDecimal numbers.

Overrides **Object.toString()**.

unscaledValue()

Returns the number as a BigInteger after removing the scale. That is, the number is expressed as a plain number, any decimal point is then removed, and the result is then converted to a BigInteger.

 Copyright (c) IBM Corporation 2000. All rights reserved.

valueOf(double)

Returns a BigDecimal whose value is the decimal representation of the 64-bit signed binary floating point parameter.

The BigDecimal is constructed as though the `double` had been converted to a `String` using the **Double.toString()** method and the **BigDecimal(String)** constructor (see page 10) had then been used.

This is a static method. A `NumberFormatException` is thrown if the parameter is infinite, or is not a number (NaN).

valueOf(long[,int])

Returns a plain BigDecimal whose value is the first parameter, *val*, optionally adjusted by a second parameter, *scale*. The default scale is 0.

The result is given by:

```
(new BigDecimal(val)).divide(TEN.pow(new BigDecimal(scale)))
```

This is a static method. A `NumberFormatException` is thrown if the *scale* is negative.

Copyright (c) IBM Corporation 2000. All rights reserved.

The MathContext class

```
public final class MathContext
implements java.lang.Serializable
```

The MathContext immutable class encapsulates the settings understood by the BigDecimal class (see page 7) for the arithmetic operator methods (see page 12). The operator methods are those that effect an operation on a number or a pair of numbers.

At present, the settings comprise the number of digits (precision) to be used for an operation, the form of any exponent that results from the operation, whether checking for lost digits is enabled, and the algorithm to be used for rounding.

When provided, a MathContext object supplies the settings for an operation directly; when `MathContext.DEFAULT` is provided then the default settings are used (`"digits=9 form=SCIENTIFIC lostDigits=0 roundingMode=ROUND_HALF_UP"`).

All methods which accept a MathContext object (or null, implying the defaults) also have a version of the method that will not accept a MathContext object. These versions carry out fixed point arithmetic with unlimited precision (as though the settings were: `"digits=0 form=PLAIN lostDigits=0 roundingMode=ROUND_HALF_UP"`).

Copyright (c) IBM Corporation 2000. All rights reserved.

Fields

These fields contain the context settings or define certain values they may take.

Public constants

DEFAULT

A MathContext object whose settings have their default values for floating point arithmetic.

The default values of the settings are:
`"digits=9 form=SCIENTIFIC lostDigits=0 roundingMode=ROUND_HALF_UP"`

ENGINEERING

A constant of type `int` that signifies that standard floating point notation (with engineering exponential format, where the power of ten is a multiple of 3, if needed) should be used for the result of a BigDecimal operation.

PLAIN

A constant of type `int` that signifies that plain (fixed point) notation, without any exponent, should be used for the result of a BigDecimal operation. A zero result in plain form may have a decimal part of one or more zeros.

SCIENTIFIC

A constant of type `int` that signifies that standard floating point notation (with scientific exponential format, where there is one digit before any decimal point, if needed) should be used for the result of a BigDecimal operation.

The remaining constants indicate rounding algorithms. Rounding is applied when a result needs more digits of precision than are available; in this case the digit to the left of the first discarded digit may be incremented or decremented, depending on the rounding algorithm selected.[12]

ROUND_CEILING

A constant of type `int` that signifies that if any of the discarded digits are non-zero then the result should be rounded towards the next more positive digit.

ROUND_DOWN

A constant of type `int` that signifies that all discarded digits are ignored (truncated). The result is neither incremented nor decremented.

---

[12] These constants have the same values as the constants of the same name in *java.math.BigDecimal*, which have been preserved to maintain compatibility with earlier versions of BigDecimal.

 Copyright (c) IBM Corporation 2000. All rights reserved.

ROUND_FLOOR

A constant of type `int` that signifies that if any of the discarded digits are non-zero then the result should be rounded towards the next more negative digit.

ROUND_HALF_DOWN

A constant of type `int` that signifies that if the discarded digits represent greater than half (0.5) the value of a one in the next position then the result should be rounded up (away from zero). Otherwise the discarded digits are ignored.

ROUND_HALF_EVEN

A constant of type `int` that signifies that if the discarded digits represent greater than half (0.5) the value of a one in the next position then the result should be rounded up (away from zero). If they represent less than half, then the result should be rounded down.

Otherwise (they represent exactly half) the result is rounded down if its rightmost digit is even, or rounded up if its rightmost digit is odd (to make an even digit).

ROUND_HALF_UP

A constant of type `int` that signifies that if the discarded digits represent greater than or equal to half (0.5) the value of a one in the next position then the result should be rounded up (away from zero). Otherwise the discarded digits are ignored.

ROUND_UNNECESSARY

A constant of type `int` that signifies that rounding (potential loss of information) is not permitted. If any of the discarded digits are non-zero then an `ArithmeticException` should be thrown.

ROUND_UP

A constant of type `int` that signifies that if any of the discarded digits are non-zero then the result will be rounded up (away from zero).

Copyright (c) IBM Corporation 2000. All rights reserved.

Shared fields

These fields are shared with the BigDecimal class (that is, only “default access” from the same package is allowed) for efficient access; they are never changed directly from another class.

digits

A value of type `int` that describes the number of digits (precision) to be used for a BigDecimal operation. A value of 0 indicates that unlimited precision (as many digits as are required) fixed-scale arithmetic will be used.

The BigDecimal operator methods (see page 12) use this value to determine the precision of results. Note that leading zeros (in the integer part of a number) are never significant. **digits** will always be non-negative.

form

A value of type `int` that describes the format of results from a BigDecimal operation. The BigDecimal operator methods (see page 12) use this value to determine the formatting of results. **form** will be **ENGINEERING**, **PLAIN**, or **SCIENTIFIC**.

lostDigits

A value of type `boolean` that is *true* if checking for lost digits (see page 37) is enabled, or is *false* otherwise. The BigDecimal operator methods (see page 12) use this value to determine whether checking for lost digits should take place.

roundingMode

A value of type `int` that describes the rounding algorithm to be used for a BigDecimal operation. The BigDecimal operator methods (see page 12) use this value to determine the algorithm to be used when non-zero digits have to be discarded in order to reduce the precision of a result. **roundingMode** will have the value of one of the public constants whose name starts with **ROUND_**.

Copyright (c) IBM Corporation 2000. All rights reserved.

Constructors

These constructors are used to set the initial values of a MathContext object. If any parameter to a constructor has a value that is not in the permitted range for the corresponding shared property then an `IllegalArgumentException` is thrown; the properties of a MathContext object are guaranteed to have valid values.

MathContext(int)

Constructs a MathContext object which has its **digits** property set to the value of the first parameter, its **form** property set to **SCIENTIFIC**, its **lostDigits** property set to **false**, and its **roundingMode** property set to **ROUND_HALF_UP**.

MathContext(int, int)

Constructs a MathContext object which has its **digits** property set to the value of the first parameter, its **form** property set to the value of the second parameter, its **lostDigits** property set to **false**, and its **roundingMode** property set to **ROUND_HALF_UP**.

MathContext(int, int, boolean)

Constructs a MathContext object which has its **digits** property set to the value of the first parameter, its **form** property set to the value of the second parameter, its **lostDigits** property set to the value of the third parameter, and its **roundingMode** property set to **ROUND_HALF_UP**.

MathContext(int, int, boolean, int)

Constructs a MathContext object which has its **digits** property set to the value of the first parameter, its **form** property set to the value of the second parameter, its **lostDigits** property set to the value of the third parameter, and its **roundingMode** property set to the value of the fourth parameter.

Copyright (c) IBM Corporation 2000. All rights reserved.

Methods

The MathContext class has the following public methods for accessing its settings:

getDigits()

Returns a non-negative `int` which is the value of the **digits** setting of the `MathContext` object.

getForm()

Returns an `int` which is the value of the **form** setting of the `MathContext` object. This will be one of **ENGINEERING**, **PLAIN**, or **SCIENTIFIC**.

getLostDigits()

Returns a `boolean` which is the value of the **lostDigits** setting of the `MathContext` object.

getRoundingMode()

Returns an `int` which is the value of the **roundingMode** setting of the `MathContext` object. This will have the value of one of the public constants whose name starts with **ROUND_**.

toString()

Returns a `String` representing the settings of the `MathContext` object as four blank-delimited words separated by a single blank and with no leading or trailing blanks, as follows:

1. The string `digits=`, immediately followed by the value of **digits** as a numeric string.
2. The string `form=`, immediately followed by the value of **form** as a string (one of `SCIENTIFIC`, `PLAIN`, or `ENGINEERING`).
3. The string `lostDigits=`, immediately followed by the value of **lostDigits** (`1` for *true* or `0` for *false*).
4. The string `roundingMode=`, immediately followed by the value of **roundingMode** as a string. This will be the same as the name of the corresponding public constant.

For example:

```
digits=9 form=SCIENTIFIC lostDigits=0 roundingMode=ROUND_HALF_UP
```

Additional words may be appended to the result of **toString** in the future if more properties are added to the MathContext class.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Decimal arithmetic definition

This definition describes the arithmetic implemented for numbers of type *BigDecimal* (see page 7). The arithmetic operations where a precision is specified are identical to those defined in the ANSI standard X3.274-1996,[13] where an algorithmic definition of each operation may be found. When unlimited precision is specified, the operations follow fixed decimal rules, as defined for the Java 1.1 BigDecimal class.

It is important to note that this definition describes results (objects of type BigDecimal) only in terms of their external representation – that is, when viewed with the **toString** method (see page 21). This guarantees that there is no hidden information in the internal representation of the numbers ("what you see is exactly what you've got") and that the intermediate results of any calculation are always defined and inspectable.

Within this constraint, any internal representation of a BigDecimal may be used, provided that the results are identical to those that would result from converting any BigDecimal object to type String and then back to a BigDecimal object at any point in this definition.

Arithmetic operation notation

In this section, a simplified notation is used to illustrate the use of operator methods: a BigDecimal object is shown as the string that would result from calling its **toString** method rather than as the corresponding constructor and Java String. Single quotes are used as a reminder that a BigDecimal rather than String object is implied. Also, the sequence ==> means "results in". Thus,

```
'12'.add('7.00',def) ==> '19.00'
```

means "`new BigDecimal("12").add(new BigDecimal("7.00"))` returns a BigDecimal object exactly equal to `new BigDecimal("19.00")`", or, more formally, it means that the term:

```
new BigDecimal("12").add(new BigDecimal("7.00")).toString().equals(
    new BigDecimal("19.00").toString())
```

evaluates to *true*.

Finally, in this example and in thee examples below, the name `def` is assumed to be a reference to the `MathContext.DEFAULT` object (see page 24).

Numbers from Strings

A *number* accepted by the **BigDecimal(String)** constructor (see page 10) is a character string that includes one or more decimal digits, with an optional decimal point. The decimal point may be embedded in the digits, or may be prefixed or suffixed to them. The group of digits (and optional point) thus constructed may have an optional sign ("+" or "–")

[13] *American National Standard for Information Technology – Programming Language REXX, X3.274-1996,* American National Standards Institute, New York, 1996.

 Copyright (c) IBM Corporation 2000. All rights reserved.

which must come before any digits or decimal point. No blanks or other white space characters are permitted in a number.

Formally:

```
sign    ::=  + | -
digits  ::=  digit [digit]...
numeric ::=  digits . [digits]
             | [.] digits
number  ::=  [sign] numeric
```

where a *digit* is any decimal digit character, such that the value returned by

```
java.lang.Character.digit(c, 10)
```

(where `c` is the character in question) is not –1 and defines the value of the digit (0 through 9). Any decimal digit character outside the range `'0'` through `'9'` is treated as though it were the corresponding character within that range.

Note that a single period alone is not a valid number.

Numbers may also include an exponent, as explained later (see page 36).

Precision

The maximum number of significant digits that can result from an arithmetic operation is controlled by the **digits** setting in a *MathContext object* (see page 23).

If the **digits** setting is greater than 0, it defines the precision (number of significant digits) to which arithmetic calculations will be carried out; results will be rounded to that precision, if necessary. If the **digits** setting is 0, then the precision is not limited to a specific number of digits; the precision used is implied by the operands depending on the specific operation, as described below.

If `MathContext.DEFAULT` is passed as a MathContext parameter, then the context is said to be *default*, and a precision of nine digits is used.

If no MathContext object is provided for an operation,[14] then the **digits** setting used is 0 (that is, unlimited precision).

An implementation-dependent maximum for the **digits** setting (equal to or larger than 1000) may apply: an attempt to create or change a MathContext object so that its **digits** setting exceeds the maximum will cause an `ArithmeticException` to be thrown. Thus if an algorithm is defined to use more than 1000 digits then if the MathContext object can be set up without an exception then the computation will proceed and produce identical results to any other implementation.[15]

Note that **digits** may be set to positive values below the default of nine. Small values, however, should be used with care – the loss of precision and rounding thus requested will affect all computations that use the low-precision MathContext object, including comparisons.

---

[14] That is, when the form of the method that takes no MathContext is used.

[15] The `com.ibm.math.BigDecimal` implementation for Java puts a limit of 999999999 on the **digits** setting.

 Copyright (c) IBM Corporation 2000. All rights reserved.

In the remainder of this section, the notation `digits` refers to the **digits** setting in use, either from a MathContext object or by omission.

Arithmetic operators

The decimal arithmetic is effected by the *operator methods* (see page 12) of the BigDecimal class – that is, the methods that may take a MathContext object as a parameter. These methods either return a BigDecimal object result or, for the **compareTo** method only, an `int` value.

If `digits` is not 0 for an arithmetic operation, the operand or operands (the numbers being operated upon) have leading zeros removed (noting the position of any decimal point, and leaving just one zero if all the digits in the number are zeros) and are then rounded to `digits` significant digits[16] (if necessary) before being used in the computation. The operation is then carried out under up to double that precision, as described under the individual operations below. When the operation is completed, the result is rounded (again, if necessary) to `digits` digits.

If `digits` is 0 for an arithmetic operation, then the precision used for the calculation is not limited by length, and in this case rounding is unnecessary, except for the results of the division and power operators.

By default, all rounding is done in the "classical" manner, in that the extra (guard) digit is inspected and values of 5 through 9 are rounded up, and values of 0 through 4 are rounded down. Alternative rounding algorithms (such as even/odd rounding) may be selected by changing the **round** setting (see page 26) of the MathContext object used for an operation.[17]

In results, a decimal point is only included if it will be followed by one or more decimal digits. A conventional zero is supplied preceding a decimal point if otherwise there would be no digit before it. If the **form** setting is either `ENGINEERING` or `SCIENTIFIC`, then (as required by convention) a result of zero is expressed as the single character `'0'`. Otherwise, trailing zeros are retained for all operations, except as described below (for example, for division when the **form** setting is not `PLAIN`).

The **format** method (see page 17) is defined to allow a number to be represented in a particular format if the standard result provided by the **toString** method (see page 21) does not meet requirements.

---

[16] If rounding of operands is undesirable, then the **lostDigits** setting (see page 37) of the provided MathContext object can be used to cause an exception to be thrown if rounding would be applied.

[17] Or by using the explicit rounding mode parameter on the two **divide** methods from the original BigDecimal class.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Arithmetic operation rules – base operators

The base operators (addition, subtraction, multiplication, and division) operate on BigDecimal numbers as follows:

Addition and subtraction

(The BigDecimal **add**, **subtract**, **plus**, and **negate** methods.)

If either number is zero and the **form** setting is not PLAIN then the other number, rounded to digits digits if necessary, is used as the result (with sign and form adjustment as appropriate).[18] Otherwise, the two numbers are aligned at their units digit and are then extended on the right and left as necessary up to a total maximum of digits+1 digits[19] (or as far as is needed to use all the digits of both numbers, if digits is 0). The numbers are then added or subtracted as appropriate.

For example, the addition

```
'xxxx.xxx'.add('yy.yyyyy', context)
```

(where "x" and "y" are any decimal digits) becomes:

```
  xxxx.xxx00
+ 00yy.yyyyy
  zzzz.zzzzz
```

If digits is not 0 the result is then rounded to digits digits if necessary, taking into account any extra (carry) digit on the left after an addition, but otherwise counting from the position corresponding to the most significant digit of the terms being added or subtracted. Finally, any insignificant leading zeros are removed.

The monadic (**plus** and **negate**) methods are evaluated using the same rules; the terms a.plus() and b.negate() (where a and b refer to any BigDecimal objects) are calculated as BigDecimal.ZERO.add(a) and BigDecimal.ZERO.subtract(b) respectively (using the original MathContext parameter, if any).

Multiplication

(The BigDecimal **multiply** method.)

The numbers are multiplied together ("long multiplication") resulting in a number which may be as long as the sum of the lengths of the two operands. For example:

```
'xxx.xxx'.multiply('yy.yyyyy', context)
```

becomes:

```
'zzzzz.zzzzzzzz'
```

If digits is not 0 the result is then rounded to digits digits if necessary, counting from the first significant digit of the result.

---

[18] If the **form** setting is PLAIN, this short cut cannot be taken as trailing zeros may appear in a zero result.

[19] The number with the smaller absolute value may therefore lose some or all of its digits on the right. In the example, the number 'yy.yyyyy' would have three digits truncated if digits were 5.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Division

(The BigDecimal **divide** methods.)

For the division:

```
'yyy'.divide('xxxxx', context)
```

the following steps are taken: first, the number `'yyy'` is extended with zeros on the right until it is larger than the number `'xxxxx'` (with note being taken of the change in the power of ten that this implies). Thus in this example, `'yyy'` might become `'yyy00'`. Traditional long division then takes place, which can be written:

```
        zzzz
xxxxx ) yyy00
```

The length of the result (`'zzzz'`) is such that the rightmost "z" will be at least as far right as the rightmost digit of the (extended) "y" number in the example. During the division, the "y" number will be extended further as necessary, and the "z" number (which will not include any leading zeros) is also extended as necessary until the division is complete.

If `digits` is not 0, the "z" number may increase up to `digits+1` digits, at which point it is rounded to `digits` digits (taking into account any residue) to form the result.

If `digits` is 0, the "z" number is increased until its scale (digits after the decimal point) is the same as the scale of the original "y" number (or the explicitly specified scale for the **divide** method that takes a scale parameter). An additional digit is then computed (as though the scale were larger by one); this digit is then used, if required, to round the "z" number (taking into account any residue) to form the result.

Finally, if the **form** setting is either `ENGINEERING` or `SCIENTIFIC`, any insignificant trailing zeros are removed.

Examples of the base operators:

```
'12'.add('7.00')       ==>  '19.00'
'1.3'.subtract('1.07') ==>  '0.23'
'1.3'.subtract('2.07') ==>  '-0.77'
'1.20'.multiply('3')   ==>  '3.60'
'7'.multiply('3')      ==>  '21'
'0.9'.multiply('0.8')  ==>  '0.72'
'1'.divide('3',def)    ==>  '0.333333333'
'2'.divide('3',def)    ==>  '0.666666667'
'5'.divide('2',def)    ==>  '2.5'
'1'.divide('10',def)   ==>  '0.1'
'12'.divide('12',def)  ==>  '1'
'8.0'.divide('2',def)  ==>  '4'
```

Note: Except for division under unlimited precision, the position of the decimal point in the terms being operated upon by the base operators is arbitrary. The operations may be carried out as integer operations with the exponent being calculated and applied afterwards. Therefore the significant digits of a result are not in any way dependent on the position of the decimal point in either of the terms involved in the operation.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Arithmetic operation rules – additional operators

The operation rules for the power, integer division, and remainder operators are as follows:

Power

(The BigDecimal **pow** method.)

The power operator raises a number (the left-hand operand) to a whole number power (the right-hand operand).

If `digits` is not 0, the right-hand operand must be a whole number whose integer part has no more digits than `digits` and whose decimal part (if any) is all zeros. It may be positive, negative, or zero; if negative, the absolute value of the power is used, and then the result is inverted (divided into 1).

If `digits` is 0, the right-hand operand must be a non-negative whole number whose decimal part (if any) is all zeros.

For calculating the power, the number is effectively multiplied by itself for the number of times expressed by the power, and finally trailing zeros are removed (as though the result were divided by one).

In practice (see note below for the reasons), the power is calculated by the process of left-to-right binary reduction. For "`x.pow(n)`": "`n`" is converted to binary, and a temporary accumulator is set to 1. If "`n`" has the value 0 then the initial calculation is complete. Otherwise each bit (starting at the first non-zero bit) is inspected from left to right. If the current bit is 1 then the accumulator is multiplied by "`x`". If all bits have now been inspected then the initial calculation is complete, otherwise the accumulator is squared by multiplication and the next bit is inspected. When the initial calculation is complete, the temporary result is divided into 1 if the power was negative.

The multiplications and division are done under the normal arithmetic operation and rounding rules, using the context supplied for the operation, except that if `digits` is not 0, the multiplications (and the division, if needed) are carried out using a precision of `digits+elength+1` digits. Here, `elength` is the length in decimal digits of the integer part of the whole number "`n`" (*i.e.*, excluding any sign, decimal part, decimal point, or insignificant leading zeros, as though the operation `n.abs().divideInteger(BigDecimal.ONE,context)` had been carried out using `digits` digits). The result is then rounded to `digits` digits.

If `digits` is 0, then the context is unchanged; the multiplications are carried out with with `digits` set to 0 (that is, unlimited precision).

Finally, if the **form** setting is either `ENGINEERING` or `SCIENTIFIC`, any insignificant trailing zeros are removed.

Integer division

(The BigDecimal **divideInteger** method.)

The integer divide operator divides two numbers and returns the integer part of the result. The result returned is defined to be that which would result from repeatedly subtracting the divisor from the dividend while the dividend is larger than the divisor. During this subtraction, the absolute values of both the dividend and the

 Copyright (c) IBM Corporation 2000. All rights reserved.

divisor are used: the sign of the final result is the same as that which would result if normal division were used.

The result returned will have no decimal part (that is, no decimal point or zeros following it). If `digits` is not 0 and the result cannot be expressed exactly within `digits` digits, the operation is in error and will fail – that is, the result cannot have more digits than the value of `digits` in effect for the operation. For example, `'10000000000'.divideInteger('3',context)` requires ten digits to express the result exactly (`'3333333333'`) and would therefore fail if `digits` were in the range `1` through `9`.

Remainder

(The BigDecimal **remainder** method.)

The remainder operator will return the remainder from integer division, and is defined as being the residue of the dividend after the operation of calculating integer division as just described, then rounded to `digits` digits (if `digits` is not 0). The sign of the result, if non-zero, is the same as that of the original dividend.

This operation will fail under the same conditions as integer division (that is, if integer division on the same two terms would fail, the remainder cannot be calculated).

Examples of the additional operators:

```
'2'.pow('3',def)            ==>  '8'
'2'.pow('-3',def)           ==>  '0.125'
'1.7'.pow('8',def)          ==>  '69.7575744'
'2'.divideInteger('3',def)  ==>  '0'
'2.1'.remainder('3',def)    ==>  '2.1'
'10'.divideInteger('3',def) ==>  '3'
'10'.remainder('3',def)     ==>  '1'
'-10'.remainder('3',def)    ==>  '-1'
'10.2'.remainder('1',def)   ==>  '0.2'
'10'.remainder('0.3',def)   ==>  '0.1'
'3.6'.remainder('1.3',def)  ==>  '1.0'
```

Notes:

1. A particular algorithm for calculating powers is described, since it is efficient (though not optimal) and considerably reduces the number of actual multiplications performed. It therefore gives better performance than the simpler definition of repeated multiplication. Since results could possibly differ from those of repeated multiplication, the algorithm must be defined here so that different implementations will give identical results for the same operation on the same values. Other algorithms for this (and other) operations may always be used, so long as they give identical results to those described here.

2. The integer divide and remainder operators are defined so that they may be calculated as a by-product of the standard division operation (described above). The division process is ended as soon as the integer result is available; the residue of the dividend is the remainder.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Numeric comparisons

The BigDecimal operator method **compareTo** (see page 13) is used for comparing BigDecimal numbers; it returns an `int` value.

Numeric comparison, using this method, is effected by subtracting the two numbers (calculating the difference, as though by using the **subtract** method with the same parameter or parameters) and then returning the sign of the result (`-1` if the result is negative, `0` if the result is zero, or `1` if the result is positive). For example, the operation:

```
a.compareTo(b,c)
```

returns the same result as:

```
a.subtract(b,c).signum()
```

It is therefore the *difference* between two numbers, when subtracted under the rules for decimal subtraction, that determines their equality.

When the signs of the operands are different the signs of the operands (`-1`, `0`, or `1`) are used for the comparison instead of the operands themselves.[20]

Exponential notation

The definition of numbers above (see page 29) describes "plain" numbers, in the sense that the character strings that describe numbers can be very long.

Examples:

```
'10000000000'.multiply('10000000000') ==> '100000000000000000000'

'0.00000000001'.multiply('0.00000000001') ==> '0.000000000000000000001'
```

For both large and small numbers, however, some form of exponential notation is useful, both to make such long numbers more readable and to make evaluation possible in extreme cases. In addition, exponential notation is normally used whenever the plain form would give misleading information. For example, if `digits` were set to 5 in the `MathContext` object referred to as `context` then:

```
'54321'.multiply('54321',context)
```

would round the multiplication to five digits and hence give the result `'2950800000'` if plain form were to be used. This is misleading, as it appears that the result is an exact multiple of 100000; instead, the result will by default be expressed in exponential notation, in this case as `'2.9508E+9'`.

The definition of `number` (see above) is therefore extended by replacing the description of `numeric` by the following:

```
significand ::=  digits . [digits]
                 | [.] digits
numeric     ::=  significand [E [sign] digits]
```

In other words, the numeric part of a number may be followed by an "`E`" (indicating an exponential part), an optional sign, and an integer following the sign that represents a

[20] This rule removes the possibility of an arithmetic overflow during a numeric comparison.

 Copyright (c) IBM Corporation 2000. All rights reserved.

power of ten that is to be applied. The "E" may be in uppercase or lowercase. Note that no blanks are permitted within this part of a number, but the integer may have leading zeros.

Examples:

The following all result in 0:

```
'12E+11'.compareTo('1200000000000',def)
'12E-5'.compareTo('0.00012',def)
'12e4'.compareTo('120000',def)
```

All valid numbers, expressed as a String, may be used to construct a BigDecimal object.

The results of calculations will be expressed in exponential form depending on their value and the value of `digits` at the time of the calculation: if the number of places needed before the decimal point exceeds `digits` (and `digits` is not 0), or if the absolute value of the result is less than `'0.000001'` (regardless of the value of `digits`), then exponential form will be used.

The exponential form generated by the **toString** method always has a sign following the "E" for readability, and the "E" is in uppercase. If the exponent is 0 then the exponential part is omitted – that is, an exponential part of "E+0" will never be generated.

Different exponential notations for the result of an operation may be selected with the **form** setting in a MathContext object (see page 23). This setting allows the selection of either scientific or engineering notation. *Scientific notation* (the default) adjusts the power of ten so there is a single non-zero digit to the left of the decimal point. *Engineering notation* causes powers of ten to be expressed as a multiple of three – the integer part may therefore range from `1` through `999`.

For example, with scientific notation:

```
'123.45'.multiply('1e11', def) ==> '1.2345E+13'
```

and with engineering notation (where `eng` is a MathContext object with **form** set to `MathContext.ENGINEERING`):

```
'123.45'.multiply('1e11', eng) ==> '12.345E+12'
```

In addition, *plain notation* may be requested. This forces the result to be represented as a plain number, without using exponential notation, regardless of the value of `digits` and the value of the result.

If neither format for a number is satisfactory for a particular application, then the **format** method (see page 17) may be used to control its appearance.

LostDigits checking

In some applications, it is desirable to check that significant information is not being lost by rounding of input data. As it is inefficient to do this explicitly, especially if precisions vary and there are many sources of input data, this definition provides for automatic checking of this condition.

When enabled by the **lostDigits** setting of the provided MathContext object (see page 23) being *true*, then if `digits` is not 0 then the operator methods first check that the number of significant digits in all their BigDecimal operands is less than or equal to the **digits**

 Copyright (c) IBM Corporation 2000. All rights reserved.

setting of the provided MathContext object; if this condition is not met, then an exception is thrown.

Note that trailing zeros are in this case are not significant; if `digits` had the value 5, then none of

```
0.12345
123.45
12345
12345.0000
1234500000
```

would throw an exception (whereas `12345.1` or `1234500001` would).

Exceptions and errors

The following exceptions and errors may be thrown during arithmetic:

- Divide exception

  An `ArithmeticException` will be thrown if division by zero was attempted, or if the integer result of an integer divide or remainder operation had too many digits.

- Overflow/Underflow

  An `ArithmeticException` will be thrown if the exponential part of a result (from an operation that is not an attempt to divide by zero) would exceed the range that can be handled by the implementation when the result is represented according to the current or implied MathContext.

  This document defines a minimum capability for the exponential part, namely exponents whose absolute value is at least as large as the largest number that can be expressed as an exact integer in default precision. Thus, since the default precision is nine, implementations must support exponents in the range `–999999999` through `999999999`.

- Lost digits

  An `ArithmeticException` will be thrown if an operand to an operator method has more significant digits than `digits`, `digits` is not 0, and **lostDigits** checking is enabled (see above).

- Unnecessary rounding

  An `ArithmeticException` will be thrown if the result of an operation requires rounding (that is, non-zero digits are to be discarded) and the current rounding algorithm is `ROUND_UNNECESSARY`, which indicates that no rounding is expected or should be permitted. Similarly, this exception will be thrown by the **format** method in the same circumstances.

- Insufficient storage

  Storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail due to lack of storage. This is considered an operating environment error as usual, rather than an arithmetical exception, and an `OutOfMemoryError` would be thrown.

In addition, other runtime exceptions may be thrown when invalid parameters are passed to a method in the BigDecimal or MathContext classes.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Notes

This section documents open questions and other items relevant to, but not part of, this proposal.

Open questions

As of this draft there are no open questions.

Changes suggested for java.math.BigDecimal

This proposal suggests enhancing the current `java.math.BigDecimal` as defined in this document. This section lists additional changes suggested for such an enhancement.

LostDigits exception

This definition currently uses ArithmeticException to signify the "lost digits" condition. It might be appropriate to define a new LostDigitsException (a subclass of ArithmeticException) for this event.

Internationalization

The BigDecimal class as proposed provides canonical string representations of BigDecimal numbers through the **format** and **toString** methods. These primitives depend on the rules for decimal arithmetic and so are included in the class.

It is assumed that locale-sensitive classes (such as the classes in the *java.text* package) will in due course use these primitives to provide enhanced number presentations: for example, the *DecimalFormat* class might provide formatting for BigDecimal numbers which allows thousands separators or a different decimal indicator. If these are added, and provide a superset of the function of the two **format** methods, then the latter should be removed.

com.ibm.math.BigDecimal conversions

The constructor and **toBigDecimal** method, provided for converting between `com.ibm.math.BigDecimal` and `java.math.BigDecimal`, should be removed.

Deprecated methods

It is suggested that the **byteValue**, **intValue**, **longValue**, and **shortValue** methods be added where necessary and all be deprecated.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Differences from the Java 1.2 BigDecimal class

The BigDecimal class proposed in this document is an upwards compatible enhancement of the original BigDecimal class. The following differences are noted:

1. The string constructors accept a superset of the number representations allowed before. Notably, exponential notation (*e.g.*, `"1.5E+12"`) is now permitted.
2. The **max** and **min** methods now define which object is returned when the numbers compare equal, as per ANSI X3.274-1996. (The current object is returned.)
3. In some cases it appears that the BigDecimal documentation does not fully define the operation of methods (for example, if `null` is passed to methods that take objects as parameters). This definition may therefore differ from the actual BigDecimal operations in these cases (but no program should be relying on them).

Future extensions

The design of the two classes in this proposal allows for future extension. In particular, new properties can be added to the MathContext class with ease. The following list documents some possible future enhancements.

Scaled arithmetic

The concept of scaled fixed point arithmetic, where the final result has a specific scale, could be extended to all operator methods simply by adding a new **form** setting to the MathContext class, indicating that results should be calculated to give an exact result with a scale of **digits** digits.

NaN and Infinity

In everyday arithmetic, undefined and infinite results are considered errors, and the current design reflects this by raising an exception for these circumstances. BigDecimals could be extended to permit the representation of Not-a-Number (NaN) and +/- Infinity, with these values being allowed when enabled by a property in the MathContext object.

Transcendental methods

Transcendental methods could be added to the BigDecimal class, or provided as a separate class (similar to `java.lang.Math`). The most-requested of these would simply extend the existing **pow** method to allow non-whole powers.

Fuzzy comparisons

The ANSI standard from which this proposal is derived includes provision for “fuzzy comparisons” (*numeric fuzz*), where numeric comparisons are carried out at a lower precision than other numeric operations.

This concept could be added, controlled by a property in the MathContext class, but has been omitted from this proposal as it was rarely used (possibly because the need for such comparisons is much reduced in a decimal floating point arithmetic). There is also some evidence that fuzzy comparisons can give confusing results.

 Copyright (c) IBM Corporation 2000. All rights reserved.

Changes

This section documents changes since the first public draft of this specification (0.60, 8 Sep 1997). The classes in the proposal are here called the *number class* and the *context class* because their names have changed over time.

Changes in Draft 0.70 (3 May 1998)

1. The precision of arithmetic (see page 30) required for conforming implementations has been raised from 9 to 100 digits.[21] The current implementation imposes a limit of 999,999,999 digits.
2. The two-valued comparator methods in the number class have been replaced by the three-valued **compareTo** method (see page 13). The number class now implements the **Comparable** interface.
3. The **power** method in the number class has been renamed **pow** (see page 15). This matches the name used in *java.lang.Math* and *java.math.BigInteger*.
4. The context class (see page 23) has been made final and immutable; its **set** methods have been removed. Its **getFormString** method has also been removed and a **toString** method has been added.
5. The **lostdigits** field in the context class has been renamed **lostDigits** (see page 26) to match Java conventions.
6. The **form** property (see page 26) of the context class (and related constants) has been changed from type `byte` to type `int` to match usual Java practice.

Changes in Draft 0.80 (6 June 1998)

1. A new **round** property (see page 26) has been added to the context class, with associated constants and methods. This property is used for selecting rounding algorithms; the constants used for selecting an algorithm have the same values as the constants of the same name in the Java 1.1 *java.math.BigDecimal* class, and the indicated algorithms have the same semantics.

[21] This exceeds the known requirements of current commercial platforms, for example, Oracle Numbers (38 decimal digits), S/390 packed decimals (31 digits), and AS/400 decimals (31 digits).

 Copyright (c) IBM Corporation 2000. All rights reserved.

2. The number class has been enhanced to implement the rounding algorithms that can be described by a context object. The **format** method (see page 17) also supports the same rounding algorithms.

Changes in Draft 0.90 (25 June 1998)

This draft is a major update in which the proposed number class is merged with the original BigDecimal class (and renamed BigDecimal), and the context class is renamed.

1. The following constructors and methods from the BigDecimal class have been added to the number class:

   BigDecimal(BigInteger[,int]), **divide(BigDecimal,int[,int])**, **movePointLeft(int)**, **movePointRight(int)**, **scale()**, **setScale(int[,int])**, **toBigInteger()**, **unscaledValue()**.

   All the other methods in BigDecimal were already in the number class.

2. The constant fields describing rounding modes (those whose name starts with **ROUND_**) have been added to the number class.

3. A **digits** value of 0, indicating the use of unlimited-precision fixed-point arithmetic, is now allowed in the context class. With this setting and using **form**=**PLAIN**, operators have the same semantics as the original BigDecimal class.

4. The context class (**DecimalContext**) has been renamed **MathContext** as it has no base-10 dependencies in its content (other than illustrative commentary).

Changes in Draft 0.92 (12 July 1998)

This draft is a minor refinement of draft 0.90.

1. The conversions to and from the floating point primitives are now defined to match their wrapper classes in *java.lang*. In particular, exponents without a sign will be accepted, and the string form of a BigDecimal constructed from a primitive will be the same as the string constructed using the corresponding wrapper class.

2. The use of a negative power as the argument to the **pow** method when `digits` is 0 is now in error (as the scale used would be 0, leading to a final result of 0 in all cases).

3. The treatment of trailing zeros is now determined by the **form** property of the context class, not the **digits** property.

4. Minor corrections and clarifications.

Changes in Draft 0.93 (28 October 1998)

This draft contains minor clarifications and editorial changes. There are no functional changes.

Changes in Draft 0.94 (14 December 1998)

This draft is a minor refinement of draft 0.93.

1. Two methods have been added (for `com.ibm.math.BigDecimal` only): a constructor from `java.math.BigDecimal`, and a **toBigDecimal** method which converts a number to `java.math.BigDecimal`.

 Copyright (c) IBM Corporation 2000. All rights reserved.

2. The description and implementation of the **pow** (power) method has been changed to conform to ANSI X3.274; the right-hand operand is no longer rounded before use.

3. Minor clarifications in the description of the **format** method (no functional changes).

Changes in Draft 0.95 (4 February 1999)

This draft primarily anticipates an errata to ANSI X3.274-1996.

1. Operands to the operator methods which have more digits than will be used for the calculation are rounded to the specified digits instead of being truncated with a guard digit. (This does not affect the option of causing a **lostDigits** exception to be thrown instead.)

2. The description of the result of the **BigDecimal(String)** constructor when the String includes exponential notation has been clarified.

3. The **format** method (see page 17) has been changed from seven underlying methods (allowing all parameters to be optionally omitted from the right) to the two most common forms.

Changes in Draft 0.96 (8 March 1999)

This draft includes a number of simplifications, changes for alignment with Java 1.2 conventions, and changes suggested by users.

Changes in the context class:

1. The **DEFAULT_** constants have been removed, and a new **DEFAULT** field has been added, to provide a default context for general-purpose arithmetic. The constructor with no arguments is no longer necessary, and has been removed.

2. The **round** setting is now called the **roundingMode** setting, and the corresponding **get** method is now named **getRoundingMode**.

3. The result of the **toString** method has been modified to make it more readable and self-explanatory.

Changes in the number class:

1. It is no longer permitted to specify `null` for a context object, as there is no precedent in the Java core classes for this notation. Instead, the `MathContext.DEFAULT` object (or a reference to it) should be supplied.

2. The set of constructors has been simplified, to follow the style of Java core classes, by dropping the constructors from the `boolean`, `byte`, `char`, `float`, and `short` primitive types.

3. The constructors from `String` and `char[]` no longer permit embedded blanks, following the precedent in other classes, such as **Double(String)**.

4. A new constructor from a `char[]` has been added. This takes an offset and a length, allowing a number embedded in a `char` array to be converted to a `BigDecimal` without constructing an intermediate object.

5. A new **valueOf(double)** static method has been added, which converts a `double` to a `BigDecimal` as though **Double.toString()** had been used to convert the number to a `String`, and the **BigDecimal(String)** constructor had then been used.

 Copyright (c) IBM Corporation 2000. All rights reserved.

6. The **BigDecimal(double)** constructor has been deprecated, as its results (now identical to those of the original **java.math.BigDecimal(double)** constructor) are not consistent with the results of the **Double.toString()** method.
7. The methods **booleanValue()** and **charValue()** have been removed.
8. The methods **byteValue()** (inherited from `java.lang.Number`), **doubleValue()**, **floatValue()**, **intValue()**, **longValue()**, and **shortValue()** (inherited from `java.lang.Number`) all now act as in Java 1.2; no exception is thrown for numbers that do not have an exact conversion.
9. The methods **byteValueExact()**, **intValueExact()**, **longValueExact()**, and **shortValueExact()** have been added; these throw an `ArithmeticException` if the conversion is not exact (that is, if the number has a non-zero decimal part, or its magnitude is too large).

Changes in Draft 0.97 (13 March 1999)

The method **toBigIntegerExact()** has been added; this throws an `ArithmeticException` if the conversion is not exact (that is, if the number has a non-zero decimal part).

Changes in Version 1.04 (10 July 1999)

Minor clarifications; the **BigDecimal(String)** constructor is limited to exponents of no more than nine digits.

Changes in Version 1.08 (18 June 2000)

The **BigDecimal(double)** constructor is no longer deprecated. It provides an exact conversion from double to BigDecimal, which is not available elsewhere.

Copyright (c) IBM Corporation 2000. All rights reserved.

Index

 Copyright (c) IBM Corporation 2000. All rights reserved.

D

E

F

 Copyright (c) IBM Corporation 2000. All rights reserved.

G

H

I

J

L

M

 Copyright (c) IBM Corporation 2000. All rights reserved.

N

O

P

 Copyright (c) IBM Corporation 2000. All rights reserved.

R

S

T

U

V

W

 Copyright (c) IBM Corporation 2000. All rights reserved.

Z

 Copyright (c) IBM Corporation 2000. All rights reserved.