Rudy's Delphi Corner

BigDecimals unit

An up-to-date version of this and other files can be found in my BigNumbers project on GitHub.

BigDecimals

> *Ten decimal places of π are sufficient to give the circumference of the earth to a fraction of an inch, and thirty decimal places would give the circumference of the visible universe to a quantity imperceptible to the most powerful microscope.* — Simon Newcomb

Floating point arithmetic can be very useful for non-integer calculations. The fact that they are hardware supported makes them fast, but due to the fact that the currently usual IEEE-754 types Single (*binary32*) and Double (*binary64*) are not only limited in size, they are also limited in precision and range.

Despite the statement in the quote above, sometimes a very high precision is needed. One example are certain financial calculations. The exact calculation of annual rates might require a precision of as much as 2191 digits, as described on this website. The exact calculation of certain physical or mathematical constants like π or *e* might even require a much higher precision.

I already implemented something every Delphi user should have at their disposal, `BigIntegers`, and now I also implemented the logical next step: *BigDecimals*. They are multi-precision, decimal floating point types. A few years ago, I implemented a Delphi version of the .NET-compatible *Decimal* type. *BigDecimal* is the big version of that, with an almost unlimited precision and almost unlimited range.

BigDecimal has a range that can vary between -536,870,912 and 536,870,912, so the tiniest value that can be represented is 1 × $10^{-536,870,912}$ and the largest is at least 999999... × 10536,870,911. The precision is around $10^{5,000,000,000}$.

Usage

BigDecimals are easy to use. They are value types, so you don't have to worry about memory management. They can be used like:

```
var
  A, B, C: BigDecimal;
begin
  A := '1.3456e-17';          // exact
  B := 1234.197;              // floating point approximation
  C := A + B * '3.0';
  Writeln(string(C));
```

While you can use a BigDecimal like a simple floating point type, you should note that their real memory consumption can be much higher. A BigDecimal only consists of a pointer and a few integers, so on your stack it won't take up much memory, but the value it represents is allocated on the heap, and that can be many more bytes, depending on the precision of the number that is represented.

BigDecimals are immutable*, so any expression or* public *function that modifies the value returns a* new *BigDecimal. The original is not modified.*

Initialization

There are several ways to get values into a *BigDecimal*. You can use constructors, like

```
A := BigDecimal.Create(1.79e308);        // floating point value
B := BigDecimal.Create('1.79e308');      // string (exact)
```

or (implicit or explicit) conversion operators like

```
B := '3.14159265358979323846264338327 9'; // string
C := -17;                                 // integral type
E := BigDecimal(6.345E-30);               // floating point type
```

Constructors

Although *BigDecimal* is a record type and record constructors are no real constructors, and the syntax used can deceive you into thinking a **new** instance is allocated somewhere, I think records should have them, for initialization. But note that, if *A* is a *BigDecimal*, it doesn't matter if you do:

```
A := BigDecimal.Create('1.7');
```

Or just:

```
A.Create('1.7');
```

Both will do exactly the same and initialize A with the value 1.7.

The constructors defined are:

`constructor Create(const UnscaledValue: BigInteger; Scale: Integer); overload;`	Creates a `BigDecimal` with given unscaled value and given scale. The sign is determined by the `BigInteger`.
`constructor Create(const E: Extended); overload;`	Creates a `BigDecimal` with the same value as the given `Extended` parameter.
`constructor Create(const D: Double); overload;`	Creates a `BigDecimal` with the same value as the given `Double` parameter.
`constructor Create(const S: Single); overload;`	Creates a `BigDecimal` with the same value as the given `Single` parameter.

For parameters of type `Extended`, `Double` or `Single`, an exception of type `EInvalidArgument` is raised if the parameter contains a NaN or +/- infinity.

`constructor Create(const S: string); overload;`	Creates a `BigDecimal` with the value that results from parsing the given `string` parameter.

For a parameter of type `string`, an exception of type `EConvertError` is raised if the string cannot be parsed to a valid `BigDecimal`.

`constructor Create(const UnscaledValue: BigInteger); overload;`	Creates a `BigDecimal` with the same value as the given `BigInteger` parameter.
`constructor Create(const U: UInt64); overload;`	Creates a `BigDecimal` with the same value as the given unsigned 64 bit integer parameter.
`constructor Create(const I: Int64); overload;`	Creates a `BigDecimal` with the same value as the given signed 64 bit integer parameter.
`constructor Create(U: UInt32); overload;`	Creates a `BigDecimal` with the same value as the given unsigned 32 bit integer parameter.
`constructor Create(I: Int32); overload;`	Creates a `BigDecimal` with the same value as the given signed 32 bit integer parameter.

It may seem convenient to use floating point types to initialize a BigDecimal, but note that this will not always give you what you like. There is a difference between

```
MyBigDecimal := BigDecimal.Create(1.7);
```

and

```
MyBigDecimal := BigDecimal.Create('1.7');
```

The former will create a BigDecimal with the exact value of

1.70000000000000000004336808689942017736029811203479766845703125

This is due to the fact that most floating point numbers are only approximations of the decimal values you give them. The above is the exact value of the closest possible Extended representation of the value 1.7, so that is the value the BigDecimal gets too. This is not the case for the latter piece of code, using a string. It will create a BigDecimal with the exact value parsed from the string, i.e. 1.7 (or, to be exact, 17×10^{-1}*).*

Note that there are values a BigDecimal can not represent exactly either, for instance, you cannot express 1/3 exactly in decimal notation. But while Decimal and BigDecimal are good at representing decimal values exactly, floating point types can only represent multiples of powers of two exactly, so they are fine for 0.5 (or 1/2) and 0.375 (or 3/8), but not for, e.g. 0.1 or 1.7.

Another problem with floating point types is that the range of a floating point type is not equally spaced, i.e. the next representable value above 1.79e300 is not 1.79e300 + 1, it is approx. 1.79e300 + 1e264, and any value inbetween is rounded to one of these. This is not the case for BigDecimal. If you use BigDecimal, 1.79e300 + 1 = 1790000...0001 (middle part omitted for legibility), in other words, exactly 1 more than the original value. There are no gaps in the range of a *BigDecimal.*

But note that while BigDecimals are incredibly accurate, they are also much slower and can require a lot more memory than fixed size, hardware supported floating point types like Single, Double or Extended.

More on floating point types in [my article about them](#).

If you need accurate decimal values, avoid floating point types to initialize BigDecimals. Use strings or predefined values instead.

Implicit conversions and class functions

The following implicit conversion operators are defined. Implicit means that you don’t have to cast, but can, if you want to. So you can either do:

```
MyBigDecimal := BigDecimal('3.141592');
```

or you can do:

```
MyBigDecimal := '3.141592';
```

and the result will be exactly the same.

`class operator Implicit(const E: Extended): BigDecimal;`	Returns a `BigDecimal` with the exact value of the given Extended parameter.
`class operator Implicit(const D: Double): BigDecimal;`	Returns a `BigDecimal` with the exact value of the given Double parameter.
`class operator Implicit(const S: Single): BigDecimal;`	Returns a `BigDecimal` with the exact value of the given Single parameter.
`class operator Implicit(const S: string): BigDecimal;`	Returns a `BigDecimal` with the value parsed from the given string parameter.
`class operator Implicit(` `  const UnscaledValue: BigInteger): BigDecimal;`	Returns a `BigDecimal` with the value of the given BigInteger parameter.
`class operator Implicit(const U: UInt64): BigDecimal;`	Returns a `BigDecimal` with the value of the given unsigned 64 bit integer parameter.
`class operator Implicit(const I: Int64): BigDecimal;`	Returns a `BigDecimal` with the value of the given signed 64 bit integer parameter.

The *string* parameter can have the same contents as a floating point literal, i.e. it consists mainly of decimal digits (*‘0’..‘9’*), it can have at most one decimal separator (*‘.’*), it can have one or more thousand separators (*‘,’*, or, alternatively, *‘ ’*), an exponent part, consisting of an *’e’* or *‘E’*, an optional sign (*‘-’* or *‘+’*), and an exponent consisting of decimal digits. The thousand separators can be anywhere. They are simply ignored.

Examples of valid strings are *‘1’*, *‘1.034’*, *‘1,000,000.345’*, *‘1.79e308’* and *‘3,456,456.4e-13’*. But also *‘1,,,2,,3.4e-99’*, which is simply interpreted as *‘123.4e-99’*.

Note, however, that trailing zeroes matter. So, while *‘1.00’* and *‘1.0000’* will compare as equal, they are different values. One has two, the other has four decimals, and *BigDecimal* remembers that. How this is done is explained later on (but know that the former is stored as 100 × 10^{-2} while the latter is stored as 10,000 × 10^{-4}, so they have different internal representations).

Internals

Before I continue with the other methods and operators, I first want to talk a little about the internals.

A *BigDecimal* is a record type with only a few member fields. One is a *BigInteger*, which contains the significant digits of the *BigDecimal*. This is also called the *unscaled value* (internal field *FValue*). The other important member field is the *scale* (internal field *FScale*), which determines the power of 10 by which the unscaled value must be *divided* to get the nominal value of the *BigDecimal*. This means that a value like 1.79 is stored as an unscaled value of 179 and a scale of 2. In other words, the value is stored as 179 / 10^2 (which can also be seen as 179 × 10^{-2}). As you see, a positive scale means dividing by a power of 10. But, unlike in my *Decimal* type, the scale can be negative too, and then you multiply by a power of 10. So 1.79e+308 is stored as an unscaled value of 179 too, but now with a scale of -306.

> *For what it’s worth, in many programming languages, including* Delphi*, 1.79e30 or 1.79e+30 are the usual* source code *representations of the floating point number 1.79 ×* 10^{30}*. E or e stand for “exponent”. Likewise, 3.45e-8 stands for 3.45 ×* 10^{-8} *(or 0.0000000345).*

The usage of a scale allows *BigDecimals* to have the same nominal values but different precisions. As I said before, 1.00 and 1.0000 compare both as exactly 1, but the former has a precision of 3, while the latter has a precision of 5 digits. The former is stored as 100 / 10^2, while the latter is stored as 10000 / 10^4.

The sign of the *BigDecimal* is simply the sign of the contained *BigInteger*.

Since *BigInteger* already knows how to add, subtract, multiply or divide, these operations are done by the *BigIntegers*. This does not mean that you can simply add two values with different scales. The scale must be adjusted to the larger of the two (by multiplying the *BigInteger* of the *BigDecimal* with the smallest scale by a power of 10 and adjusting the scale accordingly). Multiplication is similar: the *BigIntegers* are multiplied and the scales are added. Division is a little more difficult. You can’t simply divide the *BigIntegers*, because 1 div 3 returns 0, and you want something like 0.3333333…. That is where *precision* comes into play. The dividend is first multiplied by $10^{precision}$, then it is divided by the divisor and the result is then rounded towards the *target scale* as much as possible. The target scale is the difference between *dividend.Scale* and *divisor.Scale*. How the rounding is done depends on the rounding mode.

Rounding and precision

First, let's define *precision* as it is used here. Precision is the number of decimal digits the unscaled value (i.e. the *BigInteger*) represents. This is equivalent to:

```
Precision := System.Math.Ceil(UnscaledValue.BitLength * Ln(2) / Ln(10));
if Precision = 0 then
  Precision := 1;
```

where *BitLength * Ln(2) / Ln(10)* is equivalent to *Log10(N)*. If the *BigInteger* is 0, then the result of the first line is 0 too, but that is seen as a precision of 1 anyway. So the precision of 1.79e+308 is only 3, not 308, because the *BigInteger* is 179 and that has only three digits.

Rounding is cutting off the least significant digits of a value to obtain a number with a lower precision. If the precision you need is higher than the current precison, you simply multiply the unscaled value by the necessary power of 10 and adjust the scale accordingly. No rounding is required. But if the required precision is lower, you must cut off digits at the right and sometimes, you must adjust the unscaled value to do the required rounding.

BigDecimal defines an enumeration type *RoundingMode*, which governs how values are rounded, for instance after a division. It can can have the following values:

`rmUp`	Rounds up, away from zero
`rmDown`	Rounds down towards zero, i.e. it truncates the least significant digits
`fmCeil`	Rounds towards positive infinity
`rmFloor`	Rounds towards negative infinity

The following three modes round to the nearest digit, but if there is a tie, i.e. if the result is exactly halfway two next higher digits, they behave differently:

`rmNearestUp`	Rounds to the nearest higher digit, but on a tie, it rounds to the nearest higher digit that is closer to zero
`rmNearestDown`	Rounds to the nearest higher digit, but on a tie, it rounds to the nearest higher digit that is further away from zero
`rmNearestEven`	Rounds to the nearest higher digit, but on a tie, it rounds towards the nearest higher even digit

The following mode should only be used if you know that no rounding will take place:

`rmUnncessary`	Rounding is not necessary.

Raises an exception of type `ERoundingNecessary` if rounding turned out to be necessary after all.

As you may or may not have noticed, I based most of the interface of BigDecimal on the Java type of the same name. There, rounding and precision are stuck together in a MathContext class, which must be passed in in most circumstances where rounding or a precision are required. So, in Java, to do a simple division, you either hope that the division does not cause a never-ending recurrence of digits, e.g. what happens when you divide 1 by 3, or you pass a MathContext with the required precision and rounding mode. Java does not have operator overloading, so that looks like:

```
MathContext mc = new MathContext(20, RoundingMode.HALF_DOWN); // precision, rounding mode
BigDecimal monthly = total.add(fixed).divide(BigDecimal.valueOf(12), mc);
BigDecimal additional = monthly.multiply(BigDecimal.valueOf("1.19")).add(BigDecimal.valueOf("3.2"));
```

But I wanted operator overloading, and then you can't pass a precision or a rounding mode alongside the operands. That is why I gave BigDecimal two class properties for this, aptly called DefaultPrecision and DefaultRoundingMode. These contain the defaults for all BigDecimal operations. So in Delphi, you do:

```
  BigDecimal.DefaultPrecision := 20;
  BigDecimal.DefaultRoundingMode := rmNearestDown;

  Monthly := (Total + Fixed) / 12;
  Additional := Monthly * 1.19 + 3.2;
```

Back to Usage

Mathematical operations

A type like this is of no use if you can't calculate with it. *BigDecimal* defines the usual mathematical operators +, -, * and /. But it also defines *div* and *mod*. Of those two, the former returns an integral division result, the latter the remainder after that division. These operators have corresponding class functions as well, and the overloaded versions of these take rounding mode and precision parameters too. Here's a table:

```
// Result := Left + Right;
class operator Add(
  const Left, Right: BigDecimal): BigDecimal;
```

Adds two `BigDecimals`. The new scale is `Max(Left.Scale, Right.Scale)`.

An exception of type `EOverflow` is raised if the result would become too big.

```
// Result := Left - Right;
class operator Subtract(
  const Left, Right: BigDecimal): BigDecimal;
```

Subtracts two `BigDecimals`. The new scale is `Max(Left.Scale, Right.Scale)`.

An exception of type `EOverflow` is raised if the result would become too big.

```
// Result := Left * Right;
class operator Multiply(
  const Left, Right: BigDecimal): BigDecimal;
```

Multiplies two `BigDecimals`. The new scale is `Left.Scale + Right.Scale`.

An exception of type `EOverflow` is raised if the result would become too big.

An exception of type `EUnderflow` is raised if the result would become too small.

```
// Result := Left / Right;
class operator Divide(
  const Left, Right: BigDecimal): BigDecimal;
```

Divides two `BigDecimals`.

Uses the default precision and rounding mode to obtain the result.

The target scale is `Left.Scale - Right.Scale`. The result will approach this target scale as much as possible by removing any excessive trailing zeros.

An exception of type `EZeroDivide` is raised if the divisor is zero.

An exception of type `EOverflow` is raised if the result would become too big.

An exception of type `EUnderflow` is raised if the result would become too small.

```
// Result := Left div Right;
class operator IntDivide(
  const Left, Right: BigDecimal): BigDecimal;
```

Divides two `BigDecimals` to obtain an integral result.

An exception of type `EZeroDivide` is raised if the divisor is zero.

An exception of type `EOverflow` is raised if the result would become too big.

An exception of type `EUnderflow` is raised if the result would become too small.

```
// Result := Left mod Right;
class operator Modulus(
  const Left, Right: BigDecimal): BigDecimal;
```

Returns the remainder after `Left` is divided by `Right` to an integral value.

An exception of type `EZeroDivide` is raised if the divisor is zero.

An exception of type `EOverflow` is raised if the result would become too big.

An exception of type `EUnderflow` is raised if the result would become too small.

```
// Result := -Value;
class operator Negative(
  const Value: BigDecimal): BigDecimal;
```

Negates the given `BigDecimal`.

```
// Result := +Value;
class operator Positive(
  const Value: BigDecimal): BigDecimal;
```

Called when a `BigDecimal` is preceded by a unary `+`. Currently a no-op.

```
// Result := Round(Value);
class operator Round(
  const Value: BigDecimal): Int64;
```

Rounds the given `BigDecimal` to an `Int64`.

An exception of type `EConvertError` is raised if the result is too large to fit in an `Int64`.

```
// Result := Trunc(Value);
class operator Trunc(
  const Value: BigDecimal): Int64;
```

Truncates (rounds down towards 0) the given `BigDecimal` to an `Int64`.

An exception of type `EConvertError` is raised if the result is too large to fit in an `Int64`.

```
class function Add(
  const Left, Right: BigDecimal): BigDecimal;
  overload; static;
```

See `operator Add`

```
class function Subtract(
  const Left, Right: BigDecimal): BigDecimal;
  overload; static;
```

See `operator Subtract`

```
class function Multiply(
  const Left, Right: BigDecimal): BigDecimal;
  overload; static;
```

See `operator Multiply`

```
class function Divide(
  const Left, Right: BigDecimal): BigDecimal;
  overload; static;
```

See `operator Divide`

```
class function Divide(
  const Left, Right: BigDecimal; Precision: Integer;
  ARoundingMode: RoundingMode): BigDecimal;
  overload; static;
```

Like `operator Divide`, but uses the given `Precision` and `RoundingMode`

`class function Divide(`
`  const Left, Right: BigDecimal;`
`  Precision: Integer): BigDecimal;`
`  overload; static;`
	Like `operator Divide`, but uses the given `Precision` and the default rounding mode

`class function Divide(`
`  const Left, Right: BigDecimal;`
`  ARoundingMode: RoundingMode): BigDecimal;`
`  overload; static;`
	Like `operator Divide`, but uses the given `RoundingMode` and the default precision

`class function Negate(`
`  const Value: BigDecimal): BigDecimal;`
`  overload; static;`
	See `operator Negative`

`class function Round(`
`  const Value: BigDecimal): Int64;`
`  overload; static;`
	See `operator Round`

`class function Round(`
`  const Value: BigDecimal;`
`  ARoundingMode: RoundingMode): Int64;`
`  overload; static;`
	Like `operator Round`, but uses the given `RoundingMode`

`// Result := Left - (Left div Right) * Right;`
`class function Remainder(`
`  const Left, Right: BigDecimal): BigDecimal;`
`  static;`
	See `operator Modulus`

`class function Abs(`
`  const Value: BigDecimal): BigDecimal;`
`  overload; static;`
	Returns the absolute value of the given `BigDecimal`.

`class function Sqr(`
`  const Value: BigDecimal): BigDecimal;`
`  overload; static;`
	Returns the square (`Value * Value`) of the given `BigDecimal`.

`class function Sqrt(`
`  const Value: BigDecimal;`
`  Precision: Integer): BigDecimal;`
`  overload; static;`
	Returns the square root of the given `BigDecimal`, using the given `Precision`.

`class function Sqrt(`
`  const Value: BigDecimal): BigDecimal;`
`  overload; static;`
	Returns the square root of the given BigDecimal, using the default precision.

`class function IntPower(`
`  const Base: BigDecimal;`
`  Exponent, Precision: Integer): BigDecimal;`
`  overload; static;`
	Returns `Base` raised to the integral power of `Exponent`, in the given `Precision`. This routine is optimized by limiting the precision of intermediate values.
	
	An exception of type `EIntPowerExponent` is raised if the exponent is outside the range -9999999..9999999.

`class function IntPower(`
`  const Base: BigDecimal;`
`  Exponent: Integer): BigDecimal;`
`  overload; static;`
	Returns `Base` raised to the integral power of `Exponent`, in unlimited precision.
	
	An exception of type `EIntPowerExponent` is raised if the exponent is outside the range -9999999..9999999.

`function Abs: BigDecimal; overload;`	Returns the absolute value of the current `BigDecimal`.
`function Int: BigDecimal;`	Returns a `BigDecimal` with any fraction (digits after the decimal point) removed from the current `BigDecimal`.
`function Trunc: Int64;`	Returns a signed 64 bit integer with any fraction (digits after the decimal point) removed from the current `BigDecimal`.
`function Frac: BigDecimal;`	Returns a `BigDecimal` containing only the fractional part (digits after the decimal point) of the current `BigDecimal`.
`function Reciprocal(Precision: Integer): BigDecimal;` `  overload;`	Returns the reciprocal of the current `BigDecimal`, using the given `Precision`. An exception of type `EZeroDivide` is raised if the current `BigDecimal` is zero.
`function Reciprocal: BigDecimal; overload;`	Returns the reciprocal of the current `BigDecimal`, using the default precision. An exception of type `EZeroDivide` is raised if the current `BigDecimal` is zero.
`function Sqrt(Precision: Integer): BigDecimal;` `  overload;`	Returns the square root of the current BigDecimal, with the given precision
`function Sqrt: BigDecimal; overload;`	Returns the square root of the current BigDecimal, with the default precision.
`function Sqr: BigDecimal; overload;`	Returns the square (`Self * Self`) of the current `BigDecimal`.
`function IntPower(` `  Exponent, Precision: Integer): BigDecimal;`	Returns the current `BigDecimal` raised to the integral power of `Exponent`, in the given `Precision`.

```
    overload;
```

An exception of type `EIntPowerExponent` is raised if the exponent is outside the range -9999999..9999999.

```
  function IntPower(
    Exponent: Integer): BigDecimal;
    overload;
```

Returns the current `BigDecimal` raised to the integral power of `Exponent`, in unlimited precision.

An exception of type `EIntPowerExponent` is raised if the exponent is outside the range -9999999..9999999.

Comparison operations

All comparison operations base on the *Compare* function. Here they are:

```
  // Result := (Left <= Right);
  class operator LessThanOrEqual(
    const Left, Right: BigDecimal): Boolean;
```

Returns `True` only if `Left` is mathematically less than or equal to `Right`.

```
  // Result := (Left < Right);
  class operator LessThan(
    const left, Right: BigDecimal): Boolean;
```

Returns `True` only if `Left` is mathematically less than `Right`.

```
  // Result := (Left >= Right);
  class operator GreaterThanOrEqual(
    const Left, Right: BigDecimal): Boolean;
```

Returns `True` only if `Left` is mathematically greater than or equal to `Right`.

```
  // Result := (Left > Right);
  class operator GreaterThan(
    const Left, Right: BigDecimal): Boolean;
```

Returns `True` only if `Left` is mathematically greater than `Right`.

```
  // Result := (Left = Right);
  class operator Equal(
    const Left, Right: BigDecimal): Boolean;
```

Returns `True` only if `Left` is mathematically equal to `Right`.

```
  // Result := (Left <> Right);
  class operator NotEqual(
    const Left, Right: BigDecimal): Boolean;
```

Returns `True` only if `Left` is mathematically not equal to `Right`.

```
  class function Compare(
    const Left, Right: BigDecimal): TValueSign;
    static;
```

Returns 1 if `Left` is mathematically greater than `Right`, 0 if `Left` is mathematically equal to `Right` and -1 if `Left` is mathematically less than `Right`.

```
  class function Max(
    const Left, Right: BigDecimal): BigDecimal;
    static;
```

Returns the maximum of the two given `BigDecimal` values.

```
  class function Min(
    const Left, Right: BigDecimal): BigDecimal;
    static;
```

Returns the minimum of the two given `BigDecimal` values.

Just like in mathematics, *Compare* compares two values with different scale but same nominal value as equal, so

```
X := BigDecimal.Compare('1.10', '1.1000');
```

returns 0, meaning equality. In the same sense,

```
Y := BigDecimal('-1.2300') < BigDecimal('-1.23');
```

returns *False*.

> *In* Java*, if two BigDecimals have the same nominal value, but different scales, the equals() function returns false. And if you use ==, they must even be* identical*, i.e. have the same reference (address). To compare the numerical value of two BigDecimals, let's call them a and b, you must do something like* `areTheyEqual = (a.compareTo(b) == 0);`*. This is not necessary for the Delphi BigDecimals described here. You can simply use = to compare two BigDecimals for numerical equality, even if they have different scales.*

Explicit conversions

Explicit conversions were made to be silent, so if this is possible, they don't raise exceptions. In the following piece of code (without *BigDecimals*):

```
var
  I64: Int64;
  I32: Integer;
begin
  I64 := -10000000000000;
  I32 := Integer(I64);
```

I32 ends up with a value of -1316134912, because that is the value of the low 32 bit part of the *Int64*. The same principle applies to *BigDecimal*, so instead of raising an exception because the value in the *BigDecimal* does not fit in the target type, a silent conversion is performed that makes it fit.

`class operator Explicit(` `  const Value: BigDecimal): Extended;`	Returns an `Extended` with the best possible approximation of the given `BigDecimal` value. The conversion uses the default rounding mode. An exception of type `ERoundingNecessary` is raised if a rounding mode `rmUnnecessary` was specified as default but rounding is necessary after all.
`class operator Explicit(` `  const Value: BigDecimal): Double;`	Returns a `Double` with the best possible approximation of the given `BigDecimal` value. The conversion uses the default rounding mode. An exception of type `ERoundingNecessary` is raised if a rounding mode `rmUnnecessary` was specified as default but rounding is necessary after all.
`class operator Explicit(` `  const Value: BigDecimal): Single;`	Returns a `Single` with the best possible approximation of the given `BigDecimal` value. The conversion uses the default rounding mode. An exception of type `ERoundingNecessary` is raised if a rounding mode `rmUnnecessary` was specified as default but rounding is necessary after all.
`class operator Explicit(` `  const Value: BigDecimal): string;`	Returns a string representation of the given BigDecimal value. This uses `ToString`, which generally returns the shortest possible string representation of the `BigDecimal`.
`class operator Explicit(` `  const Value: BigDecimal): BigInteger;`	Returns a `BigInteger` with the rounded value of the given `BigDecimal`. The conversion uses the rounding mode `rmDown`, i.e. it truncates.
`class operator Explicit(` `  const Value: BigDecimal): UInt64;`	Returns an unsigned 64 bit integer with the rounded value of the given `BigDecimal` value. The conversion uses the rounding mode `rmDown`, i.e. it truncates.
`class operator Explicit(` `  const Value: BigDecimal): Int64;`	Returns a signed 64 bit integer with the rounded value of the given `BigDecimal` value. The conversion uses the rounding mode `rmDown`, i.e. it truncates.

String conversion

There are a few routines for conversion to and from a string. A valid string is like a valid floating point literal:

- an optional sign (*'+'* or *'-'*)
- a significand, consisting of
  - an integral part, consisting of one or more decimal digits (*'0'..'9'*)
  - an optional decimal point
  - an optional fractional part, consisting of one or more decimal digits
  - optional thousands separators or spaces — these are ignored
- an optional exponent, consisting of
  - an exponent delimiter, either *'e'* or *'E'*
  - an optional sign (*'+'* or *'-'*)
  - an exponent value, consisting of one or more decimal digits

Here are a few examples of valid (locale invariant) strings representing a *BigDecimal*:

string	unscaled value	scale	string	unscaled value	scale
`'0'`	0	0	`'0.00'`	0	2
`'179'`	179	0	`'-179'`	-179	0
`'1.79e3'`	179	-1	`'1.79e+3'`	179	-1
`'17.9e+7'`	179	-6	`'17.0'`	170	1
`'17.9'`	179	1	`'0.00179'`	179	5
`'-1.79e-12'`	-179	14	`'1,798.1e-4'`	17981	5
`'0e+7'`	0	-7	`'-0'`	0	0
`'123,456.78'`	12345678	2	`'1 234e+8'`	1234	-8

Parsing

There are four routines for parsing a string into a *BigDecimal*. Two use the invariant format settings, where '.' is the decimal separator and ',' is the thousands separator. The other two use explicit *TFormatSettings* parameters, e.g. for Germany, where I live, I could use *TFormatSettings.Create('de_DE')*.

```
class function TryParse(
  const S: string; const Settings: TFormatSettings;
  out Value: BigDecimal): Boolean;
  overload; static;
```

Tries to parse the given string as a `BigDecimal` into `Value`, using the given format settings.

Returns `True` if the function was successful.

```
class function TryParse(
  const S: string;
  out Value: BigDecimal): Boolean;
  overload; static;
```

Tries to parse the given string as a `BigDecimal` into `Value`, using the system invariant format settings.

Returns `True` if the function was successful.

```
class function Parse(
  const S: string;
  const Settings: TFormatSettings): BigDecimal;
  overload; static;
```

Returns the `BigDecimal` with a value as parsed from the given string, using the given format settings.

An exception of type `EConvertError` is raised if the string cannot be parsed to a valid `BigDecimal`.

```
class function Parse(
  const S: string): BigDecimal;
  overload; static;
```

Returns the `BigDecimal` with a value as parsed from the given string, using the system invariant format settings.

An exception of type `EConvertError` is raised if the string cannot be parsed to a valid `BigDecimal`.

Conversion to string

There are four routines that convert a *BigDecimal* to a string. The routines *ToString* generally return the shortest possible string representation. If necessary, the value is represented in scientific notation, i.e. with an exponent. So 1.79e30 is represented as *'1.79e30'*. The routines *ToPlainString* always use the plain notation and don't use scientific. This means that a value of 1.79e30 is represented as *'1790000000000000000000000000000'*. But note that both representations do not lose precision, so if the precision requires thirty decimals, even *ToString* will represent all of them. An example:

```
var
  D1, D2: BigDecimal;
begin
  D1 := '1.7900000000000000000000e30'; // note: 20 trailing zeros
  D2 := '1.79e30';
  Writeln(Format('D1 = %s, D2 = %s', [D1.ToString, D2.ToString]));
  Writeln(Format('D1 = %s, D2 = %s', [D1.ToPlainString, D2.ToPlainString]));
end;
```

The output is

```
D1 = 1.7900000000000000000000e+30, D2 = 1.79e+30
D1 = 1790000000000000000000000000000, D2 = 1790000000000000000000000000000
```

In other words, *'1.7900000000000000000000e+30'* is the shortest representation *D1.ToString* can generate, because the trailing zeros are part of the precision and they can not be cut off.

> *There is a way to cut off the trailing zeros of a BigDecimal, using RemoveTrailingZeros. This is discussed later on.*

```
function ToString: string; overload;
```

Returns the short notation of the current `BigDecimal` in the system invariant format settings. If necessary, scientific notation is used.

Because this does not use `TFormatSettings`, the result is roundtrip, so it is a valid string that can be parsed using `Parse()` or `TryParse()`.

```
function ToString(
  const Settings: TFormatSettings): string; overload;
```

Returns the short notation of the current `BigDecimal`, using the given `TFormatSettings` to obtain the decimal point `Char`. If necessary, scientific notation is used.

```
function ToPlainString: string; overload;
```

Returns a plain string of the current `BigDecimal`, using the system invariant format settings. This plain notation is sometimes called "decimal notation", and represents the value without the use of exponents.

```
function ToPlainString(
  const Settings: TFormatSettings): string; overload;
```

Returns a plain string of the current `BigDecimal`, using the given `TFormatSettings` to obtain the decimal point `Char`.

Miscellaneous functions

There are a number of functions that return information about a *BigDecimal*, or return modified versions of the original *BigDecimal*.

Rounding and scaling

Rounding and scaling are closely related. Scaling down often requires rounding. The following functions return rounded or truncated versions of the original values.

```
function RoundTo(Digits: Integer;
  ARoundingMode: RoundingMode): BigDecimal;
  overload;
```

Rounds the current `BigDecimal` to a value with at most `Digits` fractional digits, using the given rounding mode. This is more or less equivalent to the `RoundTo` function for floating point types.

An exception of type `ERoundingNecessary` is raised if a rounding mode `rmUnnecessary` was specified but rounding is necessary after all.

The `System.Math.RoundTo` function uses the floating point rounding mode equivalent of `rmNearestEven`, while `System.Math.SimpleRoundTo` uses the equivalent of `rmNearestUp`. This function is more versatile.

This is exactly equivalent to `RoundToScale(-Digits, ARoundingMode)`.

```
function RoundTo(Digits: Integer): BigDecimal;
  overload;
```

Rounds the current `BigDecimal` to a value with at most `Digits` fractional digits, using the default rounding mode.

An exception of type `ERoundingNecessary` is raised if a rounding mode `rmUnnecessary` was specified but rounding is necessary after all.

```
function RoundToScale(NewScale: Integer;
  ARoundingMode: RoundingMode): BigDecimal;
```

Rounds the current `BigDecimal` to a value with the given scale, using the given rounding mode.

An exception of type `ERoundingNecessary` is raised if a rounding mode `rmUnnecessary` was specified but rounding is necessary after all.

```
function RoundToPrecision(
  APrecision: Integer): BigDecimal;
  overload;
```

Rounds the current `BigDecimal` to a certain precision (number of significant digits).

An exception of type `ERoundingNecessary` is raised if a rounding mode `rmUnnecessary` was specified but rounding is necessary after all.

```
function RemoveTrailingZeros(
  TargetScale: Integer): BigDecimal;
```

Returns a new `BigDecimal` with all trailing zeroes (up to the target scale) removed from the current `BigDecimal`. No significant digits will be removed and the numerical value of the result compares as equal to the original value.

`TargetScale` is the scale up to which trailing zeroes can be removed. It is possible that fewer zeroes are removed, but never more than necessary to reach the target scale.

Example: `BigDecimal('1234.5678900000').RemoveTrailingZeros(6)` results in `BigDecimal('1234.567890')`.

Information

The following instance methods return information about the current *BigDecimal*.

`function IsZero: Boolean;`	Returns `True` if the current `BigDecimal`'s value equals `BigDecimal.Zero`.
`function Sign: TValueSign;`	Returns the sign of the current `BigDecimal`: -1 if negative, 0 if zero, 1 if positive.
`function Precision: Integer;`	Returns the number of significant digits of the current `BigDecimal`.
`function ULP: BigDecimal;`	Returns the unit of least precision of the current `BigDecimal`.

Properties

Predefined BigDecimals

The unit defines a few useful constants for often needed *BigDecimal* values. These should be used preferrably to newly created *BigDecimals* with the same values. That avoids duplication of the payload of the contained *BigInteger*.

`class property MinusOne: BigDecimal;`	A `BigDecimal` representing -1.
`class property Zero: BigDecimal;`	A `BigDecimal` representing 0.
`class property One: BigDecimal;`	A `BigDecimal` representing 1.
`class property Two: BigDecimal;`	A `BigDecimal` representing 2.
`class property Ten: BigDecimal;`	A `BigDecimal` representing 10.
`class property Half: BigDecimal;`	A `BigDecimal` representing 0.5.
`class property OneTenth: BigDecimal;`	A `BigDecimal` representing 0.1.

System-wide defaults

Beside the already mentioned defaults for precision and rounding mode, there is also a default for the exponent delimiter used in string output. The default is *'e'* and not, as in most other implementations, *'E'*, because to me, a lower case letter between digits that generally have the height of upper case letters, is more clearly visible than an upper case *'E'*. In other words, I prefer *'1.798765432102345e+30'* over *'1.798765432102345E+30'*, because I find it more readable.

`class property DefaultRoundingMode: RoundingMode read … write …;`	The rounding mode to be used if no specific mode is indicated, e.g. in expressions using overloaded operators.
`class property DefaultPrecision: Integer read … write …;`	The (maximum) precision to be used for e.g. division if the operation would otherwise result in a non-terminating decimal expansion, i.e. if there is no exact representable decimal result, e.g. when dividing `BigDecimal.One / BigDecimal(3)`, resulting in the neverending 0.3333333...
`class property ExponentDelimiter: Char read … write …;`	The string to be used to delimit the exponent part in scientific notation output. Currently, only `'e'` and `'E'` are allowed. Setting any other value will be ignored. The default is `'e'`.

Access to internals

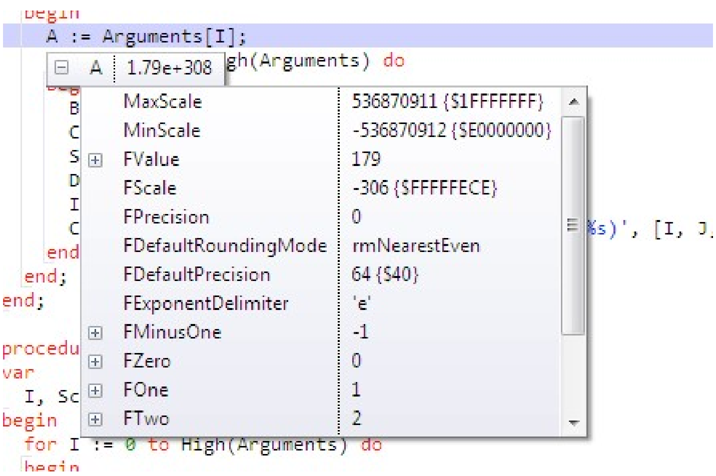
The fields of a *BigDecimal* can be accessed read-only:

`property UnscaledValue: BigInteger read …;`	The unscaled value of the current `BigDecimal`. This is the `BigInteger` that contains the significant digits and the sign of the `BigDecimal`. To obtain the nominal value, it is then scaled (in powers of ten) by `Scale`.
`property Scale: Integer read …;`	The scale of the current `BigDecimal`. This is the power of ten by which the `UnscaledValue` must be *divided* to get the nominal value of the `BigDecimal`. If it is positive, it represents the number of digits after the decimal point. A negative scale value stands for *multiplying* by a power of ten.

A negative scale my be a little hard to understand. Note that 1e+3, which has a precision of 1, will be represented by an *UnscaledValue* of 1, but to make it get the value of 1×10^3, the scale will have to be set to -3. Mathematically, that is the same as 1000, but internally, it isn't. That is because 1e+3 has a precision of 1, while 1000 has a precision of 4.

Visualizer

Since I had to write a simple parser for the debugger visualizer for *BigInteger* anyway, I amended it to parse debug output for *BigDecimal* too, so now there is a debugger visualizer for both *BigInteger* and *BigDecimal*, in the same unit. See here how to get and install it.



Notes

FPrecision

Currently, there is a private *FPrecision* instance field. This is meant to serve as a cache for the *Precision* function. *BigDecimals* are *immutable*, so the value of the record never changes. The idea is that if *FPrecision* is 0, it is uninitialized and the *Precision* function must calculate the precision. But once that is done, it can't change, so it could be stored in *FPrecision* and the function could return this cached value. The problem is, however, that unlike classes, records are not zeroed out on automatic initialization. So if, in the *Precision* function, I read a value that is not 0, I can't be sure if it was calculated before, or if it is garbage resulting from previous use of the memory. All routines that initialize or modify the internals of a new BigDecimal currently initialize *FPrecision* to 0, but the uncertainty remains.

On the other hand, if you use an *Integer* or a *Double* without initializing it, you get garbage and undefined behaviour too. So I could blame undefined values of FPrecision on the same undefined behaviour you get when you don't initialize variables of the built-in types and simply use *FPrecision* as it was intended.

Mathematics

Currently, there are no functions that provide higher mathematical functions (except *Sqrt*) for *BigDecimals*. I intend to write a new unit that provides functions like *Cos*, *ArcTan*, *SinH*, *Ln*, *Exp* or *Pi* with a set precision, but that is still in the planning stage. Such functions will probably be extremely slow, compared to hardware supported floating point, but much more accurate.

Number formatter

I also plan to implement a unit that provides a general number formatter. To this, you pass a record containing a string of digits, a scale and some more information, and it uses a format string like *'#,##000.e+000'*, more or less like Delphi's *FloatToStr* does, to format the output. I intend to make it able to format the built-in floating point types, my *Decimal* type and my *BigDecimal* type by default, but it should also be able to format other types, if you can pass the data in the required record.

Conclusion

I hope this code is useful to you. If you use some of it, please credit me. If you modify or improve the unit, please send me the modifications at [this e-mail address].

I may improve or enhance the unit myself, and I will try to post changes here. But this is not a promise. Please don't request features.

Rudy Velthuis

Standard Disclaimer for External Links

Disclaimer and Copyright

Last update: Dec. 30, 2018