Rudy's Delphi Corner

BigIntegers unit

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

BigIntegers

> *The trouble with integers is that we have examined only the very small ones. Maybe all the exciting stuff happens at really big numbers, ones we can't even begin to think about in any very definite way. Our brains have evolved to get us out of the rain, find where the berries are, and keep us from getting killed. Our brains did not evolve to help us grasp really large numbers or to look at things in a hundred thousand dimensions.* — Ronald Graham

On many occasions, I missed a *BigInteger* type in Delphi. I wanted to have a type with which you could do simple things like:

```
var
  A, B, C, D: BigInteger;
begin
  A := 3141592653589793238462643383279;
  B := 1414213562373095 shl 10;
  C := 2718281828459045235360287471352;
  D := A + B * C;
  Writeln(D.ToString);
```

Many programming languages, for instance *Java, C#, Python, Lisp, Ruby, Smalltalk, Scala, Haskell, Kotlin,* etc. have an arbitrary precision integer type. But in Delphi, there is no proper *BigInteger* type that

- is easy to use,
- can handle very long integers,
- is optimized for speed *and* precision,
- can handle different numeric bases (binary, hex, decimal, etc.) and
- can do more than basic arithmetic.

I have been thinking of using the GNU multi-precision library, but that has one big problem: it is GPL-licensed, which makes it unsuitable for my needs. So I decided to write my own, using assembler where appropriate, but also thus that pure Pascal routines would be optimal too. It works well and, as far as I know, faultless.

I initially modelled it after the .NET *System.Numerics.BigInteger* type, but took up influences from other languages with built-in *BigIntegers*, like *Common Lisp*, *Python* and *Java*. And yet I think I managed to make it as "Delphi"-like as possible. It is described below.

In the course of writing this, I had to implement a few other things, which can be useful on their own too. For the most up-to-date version, download from GitHub.

Take a look at my BigDecimal implementation too. It uses BigIntegers for most of the hard work. It can also be found on GitHub.

Note that I wrote this all on my own. I did not use any code from other implementations, but I did implement algorithms described in papers from, for instance, *Burnikel and Ziegler*, *Karatsuba*, etc.

The BigInteger type

Unlike the integral types in Delphi, the *BigInteger* type uses a *sign-magnitude* format, i.e. the *magnitude* (or *absolute value*) is always unsigned, and the sign is an extra bit. A Delphi *Integer* -1000 is stored as 32-bit value of *$FFFFFC18*, which is called *two's complement*. A *BigInteger* −1000 is stored as an **always positive** magnitude (or absolute value) of 1000 (or hex *$000003E8*) and a sign bit. This has some implications for bitwise operations, but more about that later.

The magnitude is stored as a dynamic array of *UInt32*. I, and as it turned out, also others, call such a single large digit a "limb" (a term you will see being used later on too). This is because in his book series *"The Art of Computer Programming" (a.k.a. TAOCP)*, Donald Knuth, who describes a few base case algorithms for multiple precision math in *TAOCP Volume 2*, calls them so. I also like another term I have seen lately, "bigit" (biginteger digit).

BigInteger is a record type. This allows it to be used as a value type, like normal integers, i.e. there is no need to worry about the lifetime, and it also allows the use of operators like *, *div*, *mod*, +, -, *and*, *or*, *xor* (and some others), as well as a big number of methods.

The main thing about a *BigInteger* is that there is a very low chance (virtually nil) that it will overflow, since it will simply grow if more bits are needed. It can hold enormously huge integer values, theoretically — given enough memory is present — in the order of approximately $2^{4,000,000,000}$ or $10^{1,290,000,000}$, in other words, an integer consisting of more than *1 billion digits*. The total number of particles in the known universe is estimated as 10^{80} or even 10^{85}, so a *BigInteger* can theoretically hold a vast multitude of that number.

Initialization

Unfortunately, there is no way to define a large integer literal like:

```
B := 31415926535897932384626433832795;
```

But yet, there are several ways to get values into a BigInteger. You can use constructors, like

```
A := BigInteger.Create(17);
```

or (implicit or explicit) conversion operators like

```
B := '31415926535897932384626433832795'; // string
C := -17;                                 // integral type
E := BigInteger(6e30);                    // floating point type
```

or functions and expressions like

```
D := 6 * BigInteger.Pow(10, 30);          // 6 * 10^30 exactly, unlike the floating point
                                          // value 6e30 which is only an approximation.
```

Constructors

Although *BigInteger* 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 it doesn't matter if you do:

```
A := BigInteger.Create(17);
```

Or just:

```
A.Create(17);
```

Both will do exactly the same and initialize A with the value 17.

The constructors defined are:

`constructor Create(const Limbs: array of TLimb;`
`  Negative: Boolean); overload;`
	Sets up a `BigInteger` using the `Limbs` array as the (unsigned) magnitude and `Negative` as the sign

`constructor Create(const Data: TMagnitude;`
`  Negative: Boolean); overload;`
	Sets up a `BigInteger` using the `Magnitude` array as the (unsigned) magnitude and `Negative` as the sign

`constructor Create(const Int: BigInteger ); overload;`	Sets up a `BigInteger` with the value of the given `BigInteger`
`constructor Create(const Int: Int32); overload;`	Sets up a `BigInteger` with the value of the given signed integer
`constructor Create(const Int: UInt32); overload;`	Sets up a `BigInteger` with the value of the given unsigned integer
`constructor Create(const Int: Int64); overload;`	Sets up a `BigInteger` with the value of the given 64-bit signed integer
`constructor Create(const Int: UInt64); overload;`	Sets up a `BigInteger` with the value of the given 64-bit unsigned integer
`constructor Create(const ADouble: Double); overload;`	Sets up a `BigInteger` with the value of the given `Double`. How this is done depends on the setting of the property `BigInteger.RoundingMode` (see below)

`constructor Create(`
`  const Bytes: array of Byte); overload;`
	Sets up a `BigInteger` using the bytes in `Bytes` as a two's complement value. The opposite of the function `ToByteArray`.

`constructor Create(NumBits: Integer;`
`  const Random: IRandom); overload;`
	Sets up a `BigInteger` with a random value, with a size of NumBits bits and a value determined by the given `IRandom` interface (see below)

Byte array parameter

The bytes in the byte array are considered to be in little-endian two's complement format. That means that if the top bit of the most significant byte is set, then the value in the bytes is interpreted as negative. Say you want to pass a value of 129. Then your array must contain two bytes: *$81, $00*, because a single *$81* is interpreted as -127. Likewise, if you want to pass -129, you must pass *$7F, $FF*, because *$7F* is interpreted as +127.

> *Note that this is like in* C#*, which also accepts a byte array in* little-endian *order, but unlike in* Java*, which requires a* big-endian *byte array.*

RoundingMode

The global RoundingMode property governs how a *Double* is converted to a *BigInteger*. It can have the following values.

`rmTruncate` Any fraction of the `Double` is discarded. This rounds toward 0 and is the default.

`rmSchool` How I learned to round in school: any absolute fraction >= 0.5 rounds “up” (away from 0). So 1.5 rounds to 2 and -1.5 rounds to -2. But 1.49999 rounds to 1.

`rmRound` Rounds the same way as the `System.Round` function. Any (absolute) fraction > 0.5 (this excludes 0.5 itself) is rounded away from 0. So 1.5 rounds to 1, but 1.50001 rounds to 2.

> _Note that, due to the fact that floating point values can only have a limited precision, the results may not always be what you expect. A double value like 6e30 will not result in a BigInteger with an exact value of 6000000000000000000000000000000 (a 6 with 30 zeroes), because the Double is not precise enough to hold such an exact value. The result is actually 5999999999999999556357795610624, which is the conversion of the closest a Double can get to 6e30. If you want exact values, do not use floating point values as input. Use strings or functions like BigInteger.Pow instead (see below). To obtain exactly 6e30, you can use 6 * BigInteger.Pow(10, 30)._

IRandom

I was not sure how to generate a random *BigInteger*. Other languages seem to use separate *Random* classes for this. I wanted to avoid any manual memory management for random numbers, so I declared an interface (called, you guessed it, *IRandom*) with a few suitable methods and two implementations, the first being *TRandom*, using the simple algorithm Knuth proposes and which Java seems to use (using an 48 bit seed) and the second, *TDelphiRandom*, which uses the built-in *System.Random* functions to generate random values. Both classes can be found in unit *RandomNumbers.pas*, included in the download. If you want to create your own, inherit from *TRandom* and override the virtual parameterless function *Next()*.

Implicit conversions and class functions

To get a value into a *BigInteger*, you don’t have to use constructors. There are implicit operators that allow you to assign several integer types directly, for instance:

```
var
  A, B, C: BigInteger;
begin
  A := -1;
  B := 200*1000*1000;           // My usual way to write a literal like 200,000,000
  C := -$8000000;
  D := BigInteger(Pi * 1e20);
```

Another way is to use functions like *Pow()*, e.g.

```
var
  A: BigInteger;
begin
  A := BigInteger.Pow(10, 85); // 10^85
```

The most convenient way, however, to get a well defined big number into a *BigInteger* is to use its string parsing capabilities, which also allow you to assign a string, like so:

```
var
  A, B: BigInteger;
begin
  A := '671998030559713968361666935769';
  B := '-$2940C583C5C79C8A70261FEE080A73B5B23556A5CF802BAB81DB08546F3623D5';
```

Numeric base

Beside parsing simple decimal numbers like the ones in the code snippet above, the parser can do a bit more. But I will first have to explain the numeric base. This is the base used for text input as well as text output of a *BigInteger*. I guess everyone knows decimal (base 10) numbers like 1000 or -17. But in Delphi, we know hexadecimal numbers (base 16) too, like $7F00. Some languages, like *C* or *Java*, also know octal (base 8), in the form 017 which is the same as 1*8 + 7, or decimal 15. The BigIntegers unit knows a bit more. It can have any default base from 2 to 36 (where base 36 has the “digits” 0..9 and A..Z, where A=10 and Z=35). You can set or query the default base by accessing the *BigInteger.Base* class property. That it is a class property means that it is the same for all BigIntegers.

The default numeric base affects input and output. So if you do:

```
var
  A, B: BigInteger;
begin
  BigInteger.Base := 16;
  A := '100';
  BigInteger.Base := 10;
  Writeln('A in decimal is: ', A.ToString);
```

The output is:

```
A in decimal is: 256
```

BigInteger knows a few shortcut methods for the most usual base values, so instead of writing

```
BigInteger.Base := 16;
```

you can write

```
BigInteger.Hex;
```

The convenience methods are:

Name	Base value	
`Binary`	2	Binary, e.g. `'01011010'`
`Octal`	8	Octal, e.g. `'377'`
`Decimal`	10	Decimal
`Hexadecimal`	16	Hexadecimal, e.g. `'12BEEF34'`
`Hex`	16	Short for `Hexadecimal`

Parsing

The methods *Parse*, *TryParse* and the implicit conversion operator (which allows you to assign a string to a *BigInteger*) allow for a few tricks to make it easy to enter numbers.

`class function TryParse(const S: string;`
`  Base: TNumberBase;`
`  out Res: BigInteger): Boolean; overload; static;`
	Tries to parse the specified string into a valid `BigInteger` value in the specified numeric base. Returns `False` if this failed.

`class function TryParse(const S: string;`
`  out Res: BigInteger): Boolean; overload; static;`
	Tries to parse the specified string into a valid `BigInteger` value in the default `BigInteger` numeric base. Returns `False` if this failed.

`class function Parse(const S: string): BigInteger;`
`  static;`
	Parses the specified string into a `BigInteger`, using the default numeric base. Raises an exception if this is not possible.

`class operator Implicit(`
`  const Value: string): BigInteger;`
	Implicitly (i.e. without a cast) converts the specified string to a `BigInteger`. The `BigInteger` is the result of a call to `Parse(Value)`.

There are a few things to know about the strings that can be valid BigIntegers:

- To make it easier to increase the legibility of large numbers, *any '_'* or *' '*, *anywhere* in the numeric string, will completely be ignored, so *'1_000_000_000'*, *'1 000 000 000'* and *'1000000000'* are exactly equivalent.
- The string to be parsed is considered case insensitive, so *'$ABC'* and *'$abc'* represent exactly the same value.
- The number can be prefixed with a sign, either *'+'* or *'-'* with the usual meaning, i.e. a prefix of *'-'* will result in a negative number, the prefix *'+'* in a positive one. If a sign prefix is omitted, *'+'* is assumed.

Usually, the string that is parsed is assumed to be in the base that is set with *BigInteger.Base*. But there are a few overrides that disregard the default base and give the number a specified base. Like in Delphi, this is done with a prefix:

Prefix	Base	
`$`	16	Hex, like in Delphi, e.g. `'-$8000'` for decimal `-32768`
`0x`	16	Hex, like in C and C++, e.g. `'-0x8000'` for decimal `-32768`
`0d`	10	Decimal, e.g. `'0d1234'` for decimal `1234`
`0b`	2	Binary, e.g. `'-0b11001010'` for decimal `-202`
`0o`	8	Octal, e.g. `'0o377'` for decimal `255`
`0k`	8	(Better readable) alternative form for octal, e.g. `'0k377'` for decimal `255`
`%nnR`	nn	*nn* stands for one or two decimal digits and gives the base (or *radix*) to be used. So `'-%36rRudyVelthuis'` is a valid `BigInteger` number in base 36

I made sure that simple numbers starting with a *'0'* are not automatically regarded as octal, like they are in C. Any of the valid prefixes starting with *'0'* has an alphabetic second character.

An example:

```
procedure Test;
var
  A: BigInteger;
begin
```

```
   BigInteger.Decimal;
   A := '-%36r Rudy Velthuis'; // Yes, that's my name
   Writeln(A.ToString);
   BigInteger.Base := 36;
   Writeln(A.ToString);
   BigInteger.Base := 35;
   Writeln(A.ToString);
 end;
```

The output is:

```
-3664889415200015812
-RUDYVELTHUIS
-12XJI7QCQ4S0C
```

Note that the three lines above represent the *exact same* number, but in a different base.

Examples of valid *BigInteger* strings:

String	Decimal value
'0'	0
'1'	1
'01'	1
'0x123'	291
'$0123'	291
'-$17'	−23
'12340'	12340
'%10r12345678901234567890'	12345678901234567890
'0d12345678901234567890'	12345678901234567890
'$12345678901234567890'	85968058271978839505040
'%16R12345678901234567890'	85968058271978839505040
'0x12345678901234567890'	85968058271978839505040
'-17234'	−17234
'+17234'	17234
'0O7771234567'	1071987063
'0k7771234567'	1071987063
'%8R7771234567'	1071987063
'-0b000100100011010001010110011110001001101010111100'	−20015998343868
'0b0001_0010_0011_0100_0101_0110_0111_1000_1001_0000'	78187493520
'$7fffffff9876543289abcdef01234567'	170141183428425841568023956577411351911
'$7fffffff_98765432_89abcdef_01234567'	170141183428425841568023956577411351911
'%16R 7FFFFFFF 98765432 89ABCDEF 01234567'	170141183428425841568023956577411351911
'$1234567898765432FFFFFF80'	5634002667517048507802320768
'%36rRudyVelthuis'	3664889415200015812
'%35rRudyVelthuis'	2690686858144915658
'$DEADBEEF'	3735928559
'%16r_DEADB_EE_F'	3735928559
'$ De ad Be ef'	3735928559
'%26rDead_Beef'	108863310779
'%36rDeadBeef'	1049836114599
'-$cc '	−204
'+0X0'	0
'-0X00000000000000d'	−13
'%36rABCDEFGHIJKLMNOPQRSTUVWXYZ'	8337503854730415241050377135811259267835
'-%36Rabcdefghijklmnopqrstuvwxyz'	−8337503854730415241050377135811259267835

> *Seeing* `'RUDYVELTHUIS'` *as a valid big integer still needs some getting used to. <g>*

Output

There are two simple output routines.

`function ToString: string; overload;`	Returns a string representation of the `BigInteger` in the current default numeric base
`function ToString(Base: Byte): string; overload;`	Returns a string representation of the `BigInteger` in the given numeric base

The second version, *ToString(Base)* accepts only *Base* values in the range 2..36.

I also implemented a few convenience methods for this:

`function ToBinaryString;`	Shortcut for `ToString(2)`
`function ToOctalString;`	Shortcut for `ToString(8)`
`function ToDecimalString;`	Shortcut for `ToString(10)`
`function ToHexString;`	Shortcut for `ToString(16)`

UPDATE (7 Feb 2016)

The classic *ToString(Base)* method (which is still available as *ToStringClassic*) simply did a *DivMod* by 10, turned the remainder into a digit and then repeated this with the quotient until the value of the *BigInteger* was 0. That worked fine, but when I wanted to turn the currently largest known prime, $2^{74,207,281}$ – 1, into a string, I thought my computer had hung up. Well, the result is a 22 million digit string, and (what is now) *ToStringClassic* simply repeatedly divided that huge number by 10, leaving a still huge quotient, with one digit less each time. This was, of course, even with *Burnikel & Ziegler's* division algorithm, dead slow.

To improve things, I came up with a divide-and-conquer algorithm which divides the huge number by a power of ten that more or less splits it in half, and then recursively calls itself for the remainder and the quotient, so on each recursion, only half of the number has to be converted. Very small values are then converted using an optimized version of *ToStringClassic*. This cut the time for *ToString(10)* to form a 22,338,618 digit string down to approx. 150 seconds.

> *I really came up with this algorithm myself, although it turned out I was not the first. :-(*

For what it's worth, I also modified *ToString(Base)* for bases 2, 4 and 16 to take advantage of the fact that these only have to shift limb-wise, in a simple nested loop, to get the digits. The same huge number was converted using *ToString(16)* in – *hold on, hold on...* 133ms, in other words, more than 1,000 times as fast as *ToString(10)*. This shows that base conversions can be quite slow, and that it makes sense to treat different bases differently.

I intend to do something similar for the parser, but note that this is not a priority, at the moment.

Arithmetic operators and functions

The following functions and operators are defined for arithmetic with BigIntegers:

`class operator Add(`
`  const Left, Right: BigInteger): BigInteger;`
	Adds two BigIntegers, operator `+`

`class function Add(`
`  const Left, Right: BigInteger): BigInteger;`
`  static;`
	Adds two BigIntegers

`class operator Subtract(`
`  const Left, Right: BigInteger): BigInteger;`
	Subtracts two BigIntegers, operator `-`

`class function Subtract(`
`  const Left, Right: BigInteger): BigInteger;`
`  overload; static;`
	Subtracts two BigIntegers

`class operator Multiply(`
`  const Left, Right: BigInteger): BigInteger;`
`  overload; static;`
	Multiplies two BigIntegers, operator `*`

`class function Multiply(`
`  const Left, Right: BigInteger): BigInteger;`
`  overload; static;`
	Multiplies two BigIntegers.

`class function MultiplyBaseCase(`
`  const Left, Right: BigInteger): BigInteger;`
`  overload; static;`
	Multiplies two BigIntegers using base case or "schoolbook" multiplication. This may be called internally by `Multiply` and `MultiplyKaratsuba`.

`class function MultiplyKaratsuba(`
`  const Left, Right: BigInteger): BigInteger;`
`  overload; static;`
	Multiplies two BigIntegers using the *Karatsuba* algorithm. This may be called internally by `Multiply` and `MultiplyToomCook3`.

```
class function MultiplyToomCook3(
  const Left, Right: BigInteger): BigInteger;
  overload; static;
```

Multiplies two BigIntegers using the *Toom-Cook 3-way* algorithm. This may be called internally by `Multiply`.

```
class operator Multiply(
  const Left: BigInteger;
  Right: Word): BigInteger;
```

Multiplies a `BigInteger` with a `Word`, operator `*`

```
class operator Multiply(
  Left: Word;
  const Right: BigInteger): BigInteger;
```

Multiplies a `Word` with a `BigInteger`, operator `*`

```
class operator IntDivide(
  const Left, Right: BigInteger): BigInteger;
```

Divides two `BigIntegers`, operator `div`

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

Divides two `BigIntegers`.

```
class operator Modulus(
  const Left, Right: BigInteger): BigInteger;
```

Returns the remainder after the division of the given `BigIntegers`, operator `mod`

```
class function Remainder(
  const Left, Right: BigInteger): BigInteger;
  static;
```

Returns the remainder after the divison of two `BigIntegers`

```
class procedure DivMod(
  const Dividend, Divisor: BigInteger;
  var Quotient, Remainder: BigInteger);
  static;
```

Returns quotient and remainder after the integer division of `Dividend` by `Divisor`. They are the result of the same internal operation. Is called internally by `Divide` and `Modulus`.

```
class function DivModBaseCase(
  const Left, Right: BigInteger): BigInteger;
  static;
```

Divides two `BigIntegers` using simple “schoolbook” division. My be called internally by `DivMod`.

```
class function DivModBurnikelZiegler(
  const Left, Right: BigInteger): BigInteger;
  static;
```

Divides two `BigIntegers` using the *Burnikel-Ziegler* algorithm. May be called internally by `DivMod`.

```
class operator Negative(
  const Int: BigInteger): BigInteger;
```

Returns the negation of a `BigInteger`, unary operator `-`

Bitwise operators

The following bitwise operators are defined for *BigIntegers*. Note that these have two’s complement semantics, so before any bitwise operations are performed, the internal format is converted to *two’s complement*. Aftwerward, the result is converted to *sign-magnitude* again.

```
class operator BitwiseAnd(
  const Left, Right: BigInteger): BigInteger;
```

Returns the result of the bitwise `AND` operation.
Operator `and`.

```
class operator BitwiseOr(
  const Left, Right: BigInteger): BigInteger;
```

Returns the result of the bitwise `OR` operation.
Operator `or`.

```
class operator BitwiseXor(
  const Left, Right: BigInteger): BigInteger;
```

Returns the result of the bitwise `XOR` operation.
Operator `xor`.

```
class operator LogicalNot(
  const Int: BigInteger): BigInteger;
```

Returns the result of the bitwise `NOT` operation.
Operator `not`.

The following shift operators perform, unlike usual in Delphi, *arithmetic* shifts. This means that the sign bit is preserved, so negative numbers remain negative.

```
class operator LeftShift(
  const Value: BigInteger;
  Shift: Integer): BigInteger;
```

Shifts the specified `BigInteger` value the specified number of bits to the left (away from 0). The size of the `BigInteger` is adjusted accordingly.
Operator `shl`.

```
class operator RightShift(
  const Value: BigInteger;
  Shift: Integer): BigInteger;
```

Shifts the specified `BigInteger` value the specified number of bits to the right (toward 0). The size of the `BigInteger` is adjusted accordingly. The sign is preserved, so `-128 shr 8` does not end up as 0, but as -1 instead.
Operator `shr`.

Relational operators and function

The following relational operators and functiond are defined for BigIntegers.

```
class operator Equal(
  const Left, Right: BigInteger): Boolean;
```

operator `=`

```
class operator NotEqual(
  const Left, Right: BigInteger): Boolean;
```

operator `<>`

```
class operator GreaterThan(
  const Left, Right: BigInteger): Boolean;
```

operator `>`

```
class operator GreaterThanOrEqual(
  const Left, Right: BigInteger): Boolean;
```

operator `>=`

```
class operator LessThan(
  const Left, Right: BigInteger): Boolean;
```

operator `<`

```
class operator LessThanOrEqual(
  const Left, Right: BigInteger): Boolean;
```

operator `<=`

```
class function Compare(
  const Left, Right: BigInteger): Integer;
  static;
```

Returns -1 if `Left < Right`; 1 if `Left > Right`; 0 if `Left = Right`

Conversion operators and functions

The following conversion functions perform a conversion to built-in Delphi types. If the value of the *BigInteger* is too large, an *EConvertError* is raised.

```
function AsDouble: Double;
```

Converts the `BigInteger` to a Double if that is possible. Raises an exception if not.

```
function AsInteger: Integer;
```

Converts the `BigInteger` to an Integer if that is possible. Raises an exception if not.

```
function AsCardinal: Cardinal;
```

Converts the `BigInteger` to a Cardinal if that is possible. Raises an exception if not.

```
function AsInt64: Int64;
```

Converts the `BigInteger` to an Int64 if that is possible. Raises an exception if not.

```
function AsUInt64: UInt64;
```

Converts the `BigInteger` to a UInt64 if that is possible. Raises an exception if not.

The following implicit conversion operators (the conversions generally do not require a cast) perform a conversion from built-in Delphi types to a *BigInteger*.

```
class operator Implicit(
  const Int: Integer): BigInteger;
```

Implicitly converts the specified Integer to a `BigInteger`.

```
class operator Implicit(
  const Int: Cardinal): BigInteger;
```

Implicitly converts the specified Cardinal to a `BigInteger`.

```
class operator Implicit(
  const Int: Int64): BigInteger;
```

Implicitly converts the specified Int64 to a `BigInteger`.

```
class operator Implicit(
  const Int: UInt64): BigInteger;
```

Implicitly converts the specified UInt64 to a `BigInteger`.

The following explicit conversion operators, (conversions requiring a cast) from *BigInteger* to built-in Delphi types do not raise exceptions, but truncate or sign extend the values to make them fit, just like Delphi does.

```
class operator Explicit(
  const Int: BigInteger): Integer;
```

Explicitly converts the specified `BigInteger` to an `Integer`.

```
class operator Explicit(
  const Int: BigInteger): Cardinal;
```

Explicitly converts the specified `BigInteger` to a `Cardinal`.

```
class operator Explicit(
  const Int: BigInteger): Int64;
```

Explicitly converts the specified `BigInteger` to an `Int64`.

```
class operator Explicit(
  const Int: BigInteger): UInt64;
```

Explicitly converts the specified `BigInteger` to an `UInt64`.

```
class operator Explicit(
  const Int: BigInteger): Double;
```

Explicitly converts the specified `BigInteger` to a `Double`.

```
class operator Explicit(
  const ADouble: Double): BigInteger;
```

Explicitly converts the specified `Double` to a `BigInteger`, using the constructor for that. More about that can be found under constructors, above.

Conversion to bytes

The individual bytes in the array returned by the following method appear in little-endian order.

```
function ToByteArray: TArray<Byte>;
```

Converts a `BigInteger` value to a byte array. The array is in *little-endian* order, i.e. the least significant byte comes first.

Negative values are written to the array using two's complement representation in the most compact form possible. For example, -1 is represented as a single byte whose value is $FF instead of as an array with multiple elements, such as $FF, $FF or $FF, $FF, $FF, $FF.

Because two's complement representation always interprets the highest-order bit of the last byte in the array (the byte at position *High(Array))* as the sign bit, the method returns a byte array with an extra element whose value is zero to disambiguate positive values that could otherwise be interpreted as having their sign bits set. For example, the value 120 or $78 is represented as a single-byte array: $78. However, 129, or $81, is represented as a two-byte array: $81, $00. Something similar applies to negative values: -179 (or -$B3) must be represented as $4D, $FF.

You can, inversely, convert such a byte array to a *BigInteger* again, using the already mentioned:

```
constructor Create(
  const Bytes: array of Byte); overload;
```

Sets up a `BigInteger` using the bytes in `Bytes` as a two's complement value. The opposite of the function `ToByteArray`.

Mathematical methods

The following mathematical methods are implemented:

```
class function Abs(
  const Int: BigInteger): BigInteger; overload; static;
```

Returns the absolute value of a `BigInteger`

```
function Abs: BigInteger; overload;
```

Returns the absolute value of the current `BigInteger`

```
class function GreatestCommonDivisor(
  const Left, Right: BigInteger): BigInteger;
  static;
```

Returns the (positive) greatest common divisor of the specified `BigInteger` values.

```
class function Ln(
  const Int: BigInteger): Double; overload; static;
```

Returns the natural logarithm of the value in Int.

```
function Ln: Double; overload;
```

Returns the natural logarithm of the current value.

```
class function Log(
  const Int: BigInteger;
  Base: Double): Double; overload; static;
```

Returns the logarithm to the specified base of the value in Int.

```
function Log(Base: Double): Double; overload;
```

Returns the logarithm to the specified base of the current value.

```
class function Log2(
  const Int: BigInteger): Double; overload; static;
```

Returns the logarithm to base 2 of the value in Int.

```
function Log2: Double; overload;
```

Returns the logarithm to base 2 of the current value.

```
class function Log10(
  const Int: BigInteger): Double; overload; static;
```

Returns the logarithm to base 10 of the value in Int.

```
function Log10(
  const Int: BigInteger): Double; overload;
```

Returns the logarithm to base 10 of the current value.

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

Returns the greater of two specified values.

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

Returns the lesser of two specified values.

```
class function ModPow(
  const Value: BigInteger;
  Exponent: Integer;
  const Modulus: BigInteger): BigInteger; static;
```

Returns the specified modulus value of the specified value raised to the specified power.

```
class function Pow(
  const Value: BigInteger;
  Exponent: Integer): BigInteger; static;
```

Returns the specified value raised to the specified power.

```
class function ModInverse(
  const Value, Modulus: BigInteger): BigInteger;
  static;
```

Returns the modular inverse of Value mod Modulus.

Returns an `EInvalidArgument` exception if there is no modular inverse.

```
class function NthRoot(const Radicand: BigInteger;
  Nth: Integer): BigInteger; static;
```

Returns the nth root of the radicand. This means that, for instance, with `Nth = 3`, it returns the cube root. With `Nth = 2`, it returns the same as `Sqrt`.

```
class procedure NthRootRemainder(
  const Radicand: BigInteger; Nth: Integer;
  var Root, Remainder: BigInteger); static;
```

Puts the nth root of the radicand in `Root` and the remainder in `Remainder`.

```
class function Sqrt(
  const Radicand: BigInteger): BigInteger; static;
```

Returns the square root of the radicand.

```
class procedure SqrtRemainder(
  const Radicand: BigInteger;
  var Root, Remainder: BigInteger); static;
```

Puts the square root of the radicand in `Root` and the remainder in `Remainder`.

```
class function Sqr(
  const Value: BigInteger): BigInteger; static;
```

Returns the square of the given `BigInteger`, i.e. `Value * Value`.

Bit fiddling

The following methods to do bit fiddling are implemented. They follow two's complement semantics, so a value of, say, *-$1234000* is regarded as *$EDCC0000* and bits are accessed accordingly. You can access bits past the current size of the *BigInteger*. The *BigInteger* is expanded if that is necessary to contain the new value.

For what it's worth: for *negative* values, these methods can be slow (because of the need to turn the internal sign-magnitude format into two's complement) and thus are not fit for the implementation of bit sets and the like.

```
function TestBit(Index: Integer): Boolean;
```

Returns `True` if the bit at the given position is set.

```
function SetBit(Index: Integer): BigInteger;
```

Returns a new `BigInteger` with the given bit set.

`function ClearBit(Index: Integer): BigInteger;`	Returns a new `BigInteger` with the given bit cleared.
`function FlipBit(Index: Integer): BigInteger;`	Returns a new `BigInteger` with the given bit flipped.

Note that the above does not mean that the size of a BigInteger is always expanded if you access a bit past its size. Say, you have a BigInteger like this:

```
var
  Q1, Q2: BigInteger;
begin
  Q1 := '-$12';
```

The two's complement value of that is $FFFFFFEE, or $FFFFFFFFFFFFFFEE or even more $F's in front, because the value is negative. In other words, in two's complement, any bit beyond the most significant bit of a negative value is regarded as being set (and likewise, any bit beyond the most significant bit of a positive value or 0 is regarded as being clear). So doing:

```
  Q2 := Q1.SetBit(100000);
```

does not return a different value, because, in two's complement, bit 100000 is regarded as being set anyway. The result is still -$12, so in this case, SetBit simply returns Self.

Note that FlipBit will always return a different value.

The following functions also provide information about the bits of a *BigInteger*:

`function BitLength: Integer;`	Returns the bit length, the minimum number of bits needed to represent the value, excluding the sign bit. At the same time, this is the index of the highest set bit.
`function BitCount: Integer;`	Returns the number of set bits (cardinality) in the value.
`function LowestSetBit: Integer;`	Returns the bit index of the lowest bit that is set, which is, at the same time, the number of trailing zero bits.

Properties

The following properties are defined by a *BigInteger*. Apart from the last one, *Base*, they are all instance properties describing the *BigInteger*.

`property Size: Integer read;`	The number of limbs used for the magnitude.
`property Allocated: Integer read;`	The number of limbs actually allocated. If the `RESETSIZE` conditional is defined, the internal `Compact` procedure will occasionally shrink an allocated array that is too large. That can affect performance, so by default, allocations are never shrunk. Of course, if a `BigInteger` goes out of scope, the allocated array will be released completely.
`property Negative: Boolean read;`	`True@ if the BigInteger is negative.`
`property Sign: Integer read write;`	Either `$80000000` if the `BigInteger` is negative, or `$00000000` if it is positive.
`property Magnitude: TMagnitude read;`	The array of unsigned 32-bit integers containing the magnitude of the `BigInteger`.
`class property Base: TNumberBase read write;`	The default numeric base to be used for input and output.

Conditional defines

The following conditional defines can be used. Note that some are off by default. This is reflected in the source file, e.g. the conditional *PUREPASCAL* is easily turned off by turning

```
{$DEFINE PUREPASCAL}
```

into a mere comment:

```
{ $DEFINE PUREPASCAL}
```

`PUREPASCAL`	If this conditional is set, no assembler is used.
`RESETSIZE`	If this conditional is set, the `Compact` internal routine shrinks the internal buffer if the required size of the `BigInteger` gets smaller than half of the allocated size. This can reduce memory usage, bit is can also make code a little slower. By default, this is off, and buffers are never shrunk.
`Experimental`	This conditional is can be used to shield off working code to do experiments. See text box below.

During development, I use the Experimental conditional to shield off existing and working (but perhaps too slow) code from experimental code, by doing something like this:

```
{$IFDEF Experimental}
  // ... the new, experimental code
{$ELSE !Experimental}
  // ... the working original code
{$ENDIF !Experimental}
```

With this setup, it is easy to switch between working, original code and new code, e.g. to debug the values the new code should produce (i.e. the values the original working code already produces), just by turning Experimental off or on.

Internal format

The internal format of a *BigInteger* is quite simple. It consists of two **private** data members (fields). They should only be manipulated indirectly, using the various methods and operators defined for the *BigInteger*. Of course you can access them using a trick, but it is not recommended.

Field	Type	Function	
`FData`	`TArray<TLimb>`	Magnitude	The magnitude of the `BigInteger`
`FSize`	`Integer`	Size and sign bit	The lower 31 bits contain the number of limbs the magnitude has, the top bit contains the sign bit (if the bit is set, the `BigInteger` is negative)

Note that *FSize* is not two's complement either. The lower bits are unsigned, and the sign bit has no relation to them. So negating a *BigInteger* is as simple as flipping the top bit of *FSize*.

In the above, the term "limb" is used a few times. A *TLimb* is one of the elements of the array that forms the magnitude of the *BigInteger*. It is declared as:

```
type
  PLimb = ^TLimb;
  TLimb = type UInt32;
```

Even in 64-bit Windows, I decided to use the same 32-bit limbs. This makes some 64-bit code a lot easier, and probably not slower.

Speed

For Windows, most of the time critical routines are implemented in 32-bit or 64-bit built-in assembler. For other platforms, or if the *PUREPASCAL* conditional is defined, they are completely implemented in plain Object Pascal. I optimized the routines as much as I could, sometimes with the help of the good people on StackOverflow.com (see sources).

Optimizations

Many of the routines use unrolled loops and other small-scale optimizations, not only in assembler, but also in *PUREPASCAL*. Most of the 64 bit assembler processes 64 bit (i.e. 2 limbs) at once. If this made sense, 64 bit variables were used. Just read the source code and see for yourself.

After a lot of reading and testing I was able to implement higher level, divide-and-conquer algorithms for faster operations (see, for example, this pdf and this website as well). As the Wikipedia article says: A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type (divide), until these become simple enough to be solved directly (conquer).

Karatsuba and Toom-Cook 3-way

For faster multiplication, recursive algorithms described by Anatolii Alexeevitch Karatsuba and by Andrei Toom and Stephen Cook (the 3-way algorithm) were implemented.

These are much more complicated than simple base case multiplication, but also asymptotically faster. However, due to the overhead in these functions, they are only faster if the sizes of the *BigIntegers* are above certain thresholds. That is why multiplication only uses *Karatsuba* when both sizes are above *BigInteger.KaratsubaThreshold* and it uses *Toom-Cook 3-way* when the size of one of both operands is above *BigInteger.ToomCook3Threshold*. These thresholds differ for 32-bit and 64-bit, as well as for *PUREPASCAL* or assembler-based code.

Usually, the *Multiply()* function will decide which algorithm is used, but I also made functions *MultiplyBaseCase* (normal "schoolbook" multiplication), *MultiplyKaratsuba* and *MultiplyToomCook3* public, if you have a special reason to use one of these. Note that *MultiplyToomCook* falls back to Karatsuba, and *MultiplyKaratsuba* falls back to base case if the operands do not meet the threshold criteria.

Because of the recursion, both algorithms are **much** faster than normal "schoolbook" multiplication. I measure a speed increase of almost 10 for *BigIntegers* with more than 6,400 limbs (a limb is the basic building block of *BigInteger*, a 32 bit unsigned integer), i.e. more than 61,000 decimal digits.

There are faster, even more complicated algorithms, e.g. the FFT-based *Schönhage-Strassen algorithm* for very large *BigIntegers*. I did not implement any of those yet.

Burnikel-Ziegler

For faster division (and modulus), I implemented the divide-and-conquer algorithm described by Christoph Burnikel and Joachim Ziegler. The speed of this is more or less limited by the speed of multiplication. It mainly consists of two routines, *DivTwoDigitsByOne* and *DivThreeHalvesByTwo*, which call each other recursively until sizes get below a certain threshold. Then normal base case ("Knuth") division is used. The paper linked to explains the algorithm quite nicely, without much ado. It is quite a bit faster than normal "Knuth" division, especially for very large values.

There are faster algorithms, e.g. the *Barrett algorithm* or *Montgomery reduction* for extremely large *BigIntegers*, but I did not implement any of those yet.

> *It took me a while to wrap my head around stuff like Karatsuba (OK, that one was easy), Toom-Cook, Burnikel-Ziegler, Miller-Rabin, Newton-Raphson, Schönhage-Strassen, Barrett, Montgomery, etc. I have done* ***a lot*** *of reading, and while doing that, I learned a lot of other things as well.*

Allocation

As described above, if the *RESETSIZE* conditional is defined, the internal *Compact* procedure will shrink the size of the allocated array if *Size* becomes less than half the allocated size (*Length*) of the *FData* array. Such a reallocation can negatively affect performance, so by default, the conditional is not defined, and *FData* is never shrunk. But this does **not** mean that eventually, you will get an enormous amount of memory leaks. *FData* is a normal dynamic array, so if the last reference to it is lost, it will be freed entirely.

Copy-on-write (COW)

The methods and operators of *BigInteger* follow the COW principle. This means that if you just assign one *BigInteger* to another, only a shallow copy is made. The *FData* pointer is copied and its reference count is updated, but the entire contents of the *FData* are not copied. Because of this, several *BigIntegers* with the same value can share one *FData* array. Only if one of these *BigIntegers* is about to be modified, a new copy of the data for that *BigInteger* is made, while the others can still reference the original *FData*.

An example:

```
var
  A, B, C, D: BigInteger;
begin
  A := 17;      // A gets new FData array with contents 17
  B := A;       // B.FData points to same array as A.FData
  C := B;       // C.FData points to same array as B.FData and A.FData
  B := A + 1;   // B gets new array, but A and C still reference array with value 17.
  ...
```

> *Note that, despite of the above, most of the time, BigIntegers should and can be treated as* immutable*, i.e. most methods or operators operating on BigIntegers return a new one, instead of changing the value of one of the operands. Even a function like SetBit does not modify the BigInteger on which it is called. It returns a new one with the given bit set. Only some self-referential instance methods modify the value of Self and Inc and Dec modify the value of the operand.*

Partial-flags stall problems

This is a problem that caused me some headaches before I could solve it. I noticed that on some CPUs, against all expectations, the *PUREPASCAL* routines were *faster* than the assembler routines. This turned out to be due to a partial-flags stall.

In assembler, on some "older" CPUs, certain loop code combining the opcodes *INC* and/or *DEC* with *ADC* and/or *SBB* or other instructions reading the carry flag, can cause a considerate slowdown (up to 3 times as slow), because of a so called partial-flags stall, which happens when instructions like *INC* or *DEC* write only some of the flags, while other instructions (e.g. *ADC* and *SBB*) read other flags.

I coded routines (using *LEA* and *JECXZ/JRCXZ* instead of *INC/DEC/JNE*) that avoid that problem, but on CPUs that do not have the problem, these modified routines are a bit slower than the "plain" routines. So now, at startup, the unit determines, with a few timing loops, which of the routines it should use: the plain ones or the modified ones. So, on CPUs that are not affected, plain code using *INC/DEC/JNE* is used. That is faster than the modified procedures. But on older, affected CPUs, modified code using *LEA/JECXZ* is used. On these CPUs, the modified code is **much** faster than the plain code. At the moment, only code for addition and subtration is different. The problem does not occur in other routines, probably because in these routines another instruction (e.g. *ADD* or *CMP*) will set the full flags register before it is read.

But every now and then, due to unexpected events happening that mess up the timing, the decision taken by the unit can be wrong, or you may have other reasons to choose one implementation over the other. For that, you can use:

```
BigInteger.AvoidPartialFlagsStall(True); // True to make BigInteger use modified routines,
                                         // False to make it use plain ones.
                                         // Omit routine to keep the setting the unit
                                         // measured.
```

Whether plain or modified routines are used can be queried from the property:

`property StallAvoided: Boolean;`	Returns `True` if modified routines are used, or `False` for plain routines.

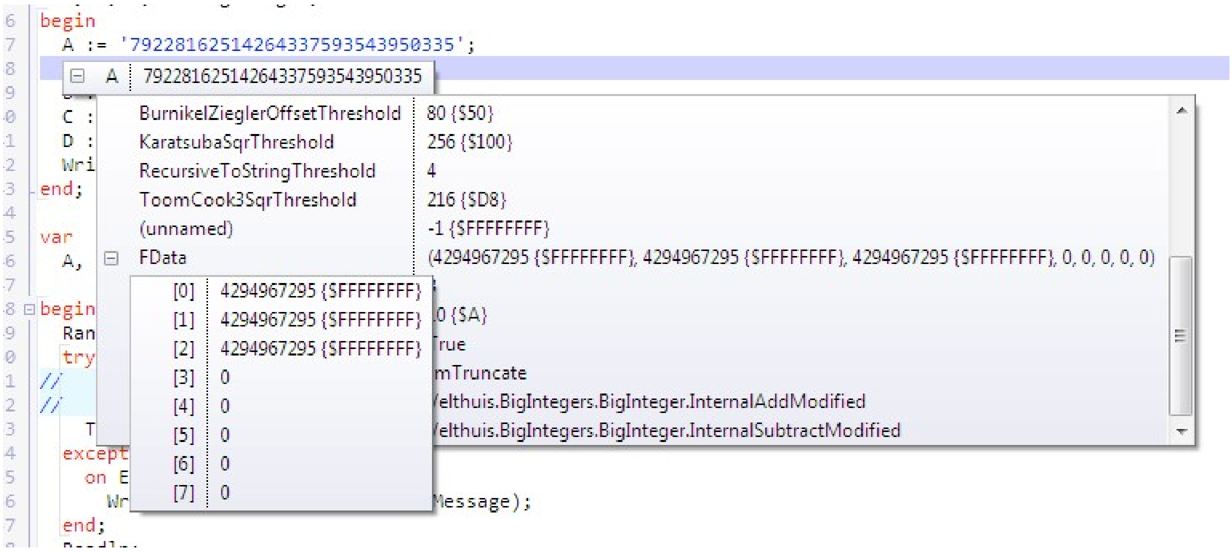
> *If you know a better way to determine the affected CPUs than the simple timing loops I am using, please send me a note.*

Bitwise operators and negative values

As I explained already, bitwise operators like *and*, *or* and *xor* use two's complement semantics. This means that before the operations on the values are performed, they are converted from the usual sign-magnitude to two's complement. For positive values, that means that nothing has to be done, but negative values must be negated completely. I found ways to get around that, partially. People on StackOverflow helped me, as well as the many bitwise tricks mentioned in Hacker's Delight.

Visualizer

All the time when working, to find the numeric value of a *BigInteger*, let's call it *X*, in the debugger, I had to either add a watch entry with *X.ToString*, or I had to run a little piece of code. So I decided to write a *debugger visualizer* for it. Here you can see it in action:



```
begin
  A := '79228162514264337593543950335';
  C :
  D :
  Wri
end;

var
  A,

begin
  Ran
  try
//
//
    T
  except
    on E
      Wr                       Message);
  end;
```

```
A   79228162514264337593543950335
    BurnikelZieglerOffsetThreshold   80 {$50}
    KaratsubaSqrThreshold            256 {$100}
    RecursiveToStringThreshold       4
    ToomCook3SqrThreshold            216 {$D8}
    (unnamed)                        -1 {$FFFFFFFF}
    FData                            (4294967295 {$FFFFFFFF}, 4294967295 {$FFFFFFFF}, 4294967295 {$FFFFFFFF}, 0, 0, 0, 0, 0)
        [0]   4294967295 {$FFFFFFFF}
        [1]   4294967295 {$FFFFFFFF}
        [2]   4294967295 {$FFFFFFFF}
        [3]   0
        [4]   0
        [5]   0
        [6]   0
        [7]   0
                                     0 {$A}
                                     rue
                                     mTruncate
                                     elthuis.BigIntegers.BigInteger.InternalAddModified
                                     elthuis.BigIntegers.BigInteger.InternalSubtractModified
```

Here value *A*, which was initialized with the string above it, is displayed in the tooltip.

I had hoped it would be easy, but it turned out that the IDE only passes you a string like

```
'(FData:(4294967295 {$FFFFFFFF}, 4294967295 {$FFFFFFFF}, 4294967295 {$FFFFFFFF}, 0, 0, 0, 0, 0); FSize:3)'
```

or, in the *Local Variables* pane, a simplified version (for the same *BigInteger*) like:

```
'((4294967295, 4294967295, 4294967295, 0, 0, 0, 0, 0), 3)'
```

or a mix of these styles, which you then have to parse to get the data, turn them into an array and a sign, use these to create a *BigInteger* and then apply *ToString* to get the replacement string. It took me some debugging and quite a few IDE restarts to get the parsing straight for the different formats. Perhaps there is a better way, i.e. one that gives you access to the real data, but I haven't found it yet.

The visualizer package currently comes in source format, which you only have to install in the IDE (click *Install* in the context menu in the *Project Manager*). Make sure that the *BigNumbers.dpk* package was built first and that the visualizer package references its *.dcp* file. I intend to write an installer for this, but please don't hurry me, as this is not something I have done before.

Currently, the code on GitHub represents a DLL expert, which must be added to the known experts in the registry. It is very easy to make this a package-based expert instead, though.

FreePascal

In *{$mode delphi}*, it is possible to compile the units. *FreePascal* doesn't seem to use segmented unit names (yet), so you'll have to rename some of the used units and qualifiers like *System.Math* or *System.SysUtils* to simply *Math* or *SysUtils*. At first, you'll have to *{$DEFINE PUREPASCAL}*, because making the assembler work satisfactorily could be quite a job. It seems to work alright, if you set *{$asmmode intel}*, but it is a little finicky. Stuff like *LEA EAX,[EDX + EDI*ClimbSize]* seems to cause problems, but *[EDX + CLimbSize*EDI]* works. That is a bit of a problem for me, because I most of the time use the former variety. But I guess that, with enough time on your hands, you can make it work.

Some of the tricky routines, e.g. *ToString*, which uses *RecursiveToString*, which in its turn needs an exact alignment of bytes to get a usable string, don't work, but in the meantime — until you solved the problem — you can e.g. use (or wrap or rename) *.ToStringClassic(10)* instead. I guess the tricky routines just need some work and some debugging to make them work under *FreePascal*. I only tried a quick and dirty test, did not do any extensive tests, but it is obviously possible to make it work. Just don't give up immediately.

Types like *TArray* don't seem to work. And there is no way you can define an *array of BigInteger before BigInteger* is defined, so you will have to define a type *BigIntegerArray = array of BigInteger inside* the *BigInteger* record, as a nested type. You will meet other, similar problems. Another was the fact that there is no *DivMod()* function for *UInt64* types, so you will have to either use the one for *Longint* or use *div* and *mod* separately. It could be that the change I made there is one of the causes for the failure of *RecursiveToString*. This needs some investigation.

There will be other, simple problems and it is well possible that there are a few not-so-simple problems. But it looks as if *BigIntegers* can be used in *FreePascal*.

Good luck!

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

These links are being provided as a convenience and for informational purposes only; they do not constitute an endorsement or an approval of any of the products, services or opinions of the corporation or organization or individual. I bear no responsibility for the accuracy, legality or content of the external site or for that of subsequent links. Contact the external site for answers to questions regarding its content.

Disclaimer and Copyright

The coding examples presented here are for illustration purposes only. The author takes no responsibility for end-user use. All content herein is copyrighted by Rudy Velthuis, and may not be reproduced in any form without the author's permission. Source code written by Rudy Velthuis presented as download is subject to the license in the files.

Last update: Jan. 20, 2019

Back to top