Reference version: 0.1
SSTypes version: 1.0.2
Date: March 26, 2016
Reference: https://github.com/dgakh/SSTypes
License: https://github.com/dgakh/SSTypes/blob/master/SSTypesLT/LICENSE.md

SSTypes.SmartInt Structure

General

Represents a 32-bit signed integer. Is similar in almost all usage cases to System.Int32. System.Int32 is underlying type for SmartInt. SmartInt also directly mimics ?int variables representing nullable types.

To view the source code for this type, see the SSTypesLT projects by the link https://github.com/dgakh/SSTypes

SmartInt is an immutable value type that represents signed integers with values that range from negative 2,147,483,647 (which is represented by the SmartInt.MinValue constant) through positive 2,147,483,647 (which is represented by the SmartInt.MaxValue constant.

Value -2,147,483,648 has special purpose and represent SmartInt.BadValue.

A variable of type SmartInt can be declared and initialized like this example:

```
SmartInt si1 = 89234;
SmartInt si2 = -345346;
```

Conversions

There is a predefined implicit conversion from SmartInt to int, long, float, double, or decimal.

For example:

// '89234' is an SmartInt, so an implicit conversion takes place here:
```
float f = si1;
```

There is a predefined implicit conversion from int, sbyte, byte, short, ushort, or char to SmartInt. For example, the following assignment statement will produce a compilation error without a cast:

```
long aLong = 22;
SmartInt si1 = aLong;       // Error: no implicit conversion from long.
SmartInt si2 = (int)aLong;  // OK: explicit conversion.
```

Notice also that there is no implicit conversion from floating-point types to int. For example, the following statement generates a compiler error unless an explicit cast is used:

```
SmartInt x = 3.0;         // Error: no implicit conversion from double.
SmartInt y = (int)3.0;    // OK: explicit conversion.
```

Method Parse()

Parse is the most improved method in SmartInt:

- It shows 4x performance improvement comparing to standard System.Int32.Parse;
- It con parse substring of string without need to splitting it. This ability gives additional improvement of performance;
- There are different overloads of Parse taking different types of data;
- It does not throw exception.

Replacing Nullable

SmartInt can represent the correct range of its integral values, plus an additional null value. Technically null value is represented by SmartInt.BadValue constant. But C# code will look similar.

A nullable type:

```
{
   int? number = null;

  // Is the HasValue property true?
   if (number.HasValue)
      System.Console.WriteLine("number = " + number.Value);
   else
      System.Console.WriteLine("number = Null");

   // Is the number null ?
   if (number == null)
      System.Console.WriteLine("number = " + number.Value);
   else
      System.Console.WriteLine("number = Null");

   // y is set to zero
   int y = number.GetValueOrDefault();

   // number.Value throws an InvalidOperationException if number.HasValue is false
   try
   {
      y = number.Value;
   }
   catch (System.InvalidOperationException e)
   {
      System.Console.WriteLine(e.Message);
   }
}
```

SmartInt type (only one word was changed - int? to SmartInt):

```
{
   SmartInt number = null;

  // Is the HasValue property true?
   if (number.HasValue)
      System.Console.WriteLine("number = " + number.Value);
   else
      System.Console.WriteLine("number = Null");
```

```
    // Is the number null ?
    if (number == null)
        System.Console.WriteLine("number = " + number.Value);
    else
        System.Console.WriteLine("number = Null");

    // y is set to zero
    int y = number.GetValueOrDefault();

    // number.Value throws an InvalidOperationException if number.HasValue is false
    try
    {
        y = number.Value;
    }
    catch (System.InvalidOperationException e)
    {
        System.Console.WriteLine(e.Message);
    }
}
```

Declaration SmartInt? is valid. This increases storage and possible performance cost.

Boxing and unboxing

If HasValue of int? is false, the object reference is assigned to null instead of boxing. But SmartInt boxes in any case.

```
int? x1 = null;
SmartInt x2 = null;

object o1 = x1;
object o2 = x2;

int? xa1 = (int?)o1;
SmartInt xa2 = (SmartInt)o2;
```

Boxed nullable int and SmartInt fully support the functionality of the underlying type or SmartInt. The codes for nullable int and SmartInt are similar here:

For int?:

```
int? i1 = 47;
object iBoxed = i1;
// Access IConvertible interface implemented by int.
IConvertible ic = (IConvertible)iBoxed;
int i = ic.ToInt32(null);
string str = ic.ToString();
```

For SmartInt:

```
SmartInt i1 = 47;
object iBoxed = i1;
// Access IConvertible interface implemented by SmartInt.
IConvertible ic = (IConvertible)iBoxed;
int i = ic.ToInt32(null);
string str = ic.ToString();
```

GetType()

SmartInt.GetType() returns type of System.Int32. The same value returns GetType() for int?.

Size

Simple size test shows that SmartInt uses the same memory size as int uses. Int? uses two time bigger size.

```
public static void ArraySize()
{
   int objects_count = 1000;

   long memory1 = GC.GetTotalMemory(true);
   int[] ai = new int[objects_count];
   long memory2 = GC.GetTotalMemory(true);
   int?[] ani = new int?[objects_count];
   long memory3 = GC.GetTotalMemory(true);
   SmartInt[] asi = new SmartInt[objects_count];
   long memory4 = GC.GetTotalMemory(true);

   // Compiler can optimize and do not allocate arrays if they are not used
   // So we write their lengths
   Console.WriteLine("Array sizes {0}, {1}, {2}", ai.Length, ani.Length, asi.Length);
   Console.WriteLine("Memory for int \t {0}", memory2 - memory1);
   Console.WriteLine("Memory for int? \t {0}", memory3 - memory2);
   Console.WriteLine("Memory for SmartInt \t {0}", memory4 - memory3);
}
```

```
Array sizes 1000, 1000, 1000
Memory for int          4024
Memory for int?         8024
Memory for SmartInt     4024
Press any key to exit.
```

Possible issues

1)
SmartInt has no implicit transformation to string, but has the implicit transformation to int. Putting SmartInts to methods taking int values will lead to transform SmartInt to int and then call method ToString of type int, but not of type SmartInt.

```
SSTypes.SmartInt si2 = 55;
System.Console.WriteLine(si2);              // Will convert si2 to int and call int.ToString()
System.Console.WriteLine(si2.ToString()); // Will call si2.ToString()
```

2)
Value range of SmartInt is from - 2,147,483,647 through 2,147,483,647. Value range of System.Int32 is from -2,147,483,648 through 2,147,483,647. So, SmartInt cannot replace System.Int32 where value -2,147,483,648 is used. But this is a rare case and this issue has a very small meaning.

3)
Default value for HasValue is true. Default value for HasValue for Nullable is false.

4)
Operator ?? cannot be implicitly applied for SmartInt. Explicit conversion must be set.

Compiled:

```
int? x = null;
int y = x ?? -9;
```

Isn't compiled:

```
SmartInt x = null;
int y = x ?? -9;
```

Compiled:

```
SmartInt x = null;
int y = (int?)x ?? -9;
```

5)
If HasValue of int? is false, the object reference is assigned to null instead of boxing. But SmartInt boxes in any case.

```
int? x1 = null;
SmartInt x2 = null;

object o1 = x1;
object o2 = x2;

int? xa1 = (int?)o1;
SmartInt xa2 = (SmartInt)o2;
```

6)
SmartInt.GetType() returns type of System.Int32.