LinksPlatform's Platform.Converters Class Library

./CachingConverterDecorator.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Converters
{
    public class CachingConverterDecorator<TSource, TTarget> : IConverter<TSource, TTarget>
    {
        private readonly IConverter<TSource, TTarget> _baseConverter;
        private readonly IDictionary<TSource, TTarget> _cache;

        public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter,
        ↪  IDictionary<TSource, TTarget> cache) => (_baseConverter, _cache) = (baseConverter,
        ↪  cache);

        public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter) :
        ↪  this(baseConverter, new Dictionary<TSource, TTarget>()) { }

        public TTarget Convert(TSource source)
        {
            if (!_cache.TryGetValue(source, out TTarget value))
            {
                value = _baseConverter.Convert(source);
                _cache.Add(source, value);
            }
            return value;
        }
    }
}
```

./Converter.cs

```
using System;
using System.Linq;
using System.Reflection;
using System.Reflection.Emit;
using Platform.Reflection;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Converters
{
    public static class Converter<TSource, TTarget>
    {
        public static IConverter<TSource, TTarget> Default { get; set; }

        static Converter()
        {
            AssemblyName assemblyName = new AssemblyName(GetNewName());
            var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
            ↪  AssemblyBuilderAccess.Run);
            var module = assembly.DefineDynamicModule(GetNewName());
            var type = module.DefineType(GetNewName(), TypeAttributes.Public |
            ↪  TypeAttributes.Class, null, Types<IConverter<TSource, TTarget>>.Array);

            EmitMethod<System.Converter<TSource, TTarget>>(type, "Convert", (il) =>
            {
                if (typeof(TSource) == typeof(TTarget))
                {
                    il.Return();
                }
                else
                {
                    throw new NotSupportedException();
                }
            });

            var typeInfo = type.CreateTypeInfo();

            Default = (IConverter<TSource, TTarget>)Activator.CreateInstance(typeInfo);
        }

        private static void EmitMethod<TDelegate>(TypeBuilder type, string methodName,
        ↪  Action<ILGenerator> emitCode)
        {
            var delegateType = typeof(TDelegate);
            var invoke = delegateType.GetMethod("Invoke");
            var returnType = invoke.ReturnType;
```

```
            var parameterTypes = invoke.GetParameters().Select(s => s.ParameterType).ToArray();
            MethodBuilder method = type.DefineMethod(methodName, MethodAttributes.Public | MethodAttributes.Static, returnType, parameterTypes);
            method.SetImplementationFlags(MethodImplAttributes.IL | MethodImplAttributes.Managed | MethodImplAttributes.AggressiveInlining);
            var generator = method.GetILGenerator();
            emitCode(generator);
        }

        private static string GetNewName() => Guid.NewGuid().ToString("N");
    }
}
```

./IConverter[T].cs

```
namespace Platform.Converters
{
    /// <summary>
    /// <para>Defines a converter between two values of the same type.</para>
    /// <para>Определяет конвертер между двумя значениями одного типа.</para>
    /// </summary>
    /// <typeparam name="T"><para>Type of value to convert.</para><para>Тип преобразуемого значения.</para></typeparam>
    public interface IConverter<T> : IConverter<T, T>
    {
    }
}
```

./IConverter[TSource, TTarget].cs

```
namespace Platform.Converters
{
    /// <summary>
    /// <para>Defines a converter between two types (TSource and TTarget).</para>
    /// <para>Определяет конвертер между двумя типами (исходным TSource и целевым TTarget).</para>
    /// </summary>
    /// <typeparam name="TSource"><para>Source type of conversion.</para><para>Исходный тип конверсии.</para></typeparam>
    /// <typeparam name="TTarget"><para>Target type of conversion.</para><para>Целевой тип конверсии.</para></typeparam>
    public interface IConverter<in TSource, out TTarget>
    {
        /// <summary>
        /// <para>Converts the value of the source type (TSource) to the value of the target type.</para>
        /// <para>Конвертирует значение исходного типа (TSource) в значение целевого типа.</para>
        /// </summary>
        /// <param name="source"><para>The source type value (TSource).</para><para>Значение исходного типа (TSource).</para></param>
        /// <returns><para>The value is converted to the target type (TTarget).</para><para>Значение ковертированное в целевой тип (TTarget).</para></returns>
        TTarget Convert(TSource source);
    }
}
```

./To.cs

```
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Converters
{
    public static class To
    {
        public static readonly char UnknownCharacter = '';

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(ulong value) => value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long Int64(ulong value) => unchecked(value > long.MaxValue ? long.MaxValue : (long)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static uint UInt32(ulong value) => unchecked(value > uint.MaxValue ? uint.MaxValue : (uint)value);
```

```
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          public static int Int32(ulong value) => unchecked(value > int.MaxValue ? int.MaxValue :
            ↪  (int)value);
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public static ushort UInt16(ulong value) => unchecked(value > ushort.MaxValue ?
            ↪  ushort.MaxValue : (ushort)value);
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          public static short Int16(ulong value) => unchecked(value > (ulong)short.MaxValue ?
            ↪  short.MaxValue : (short)value);
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public static byte Byte(ulong value) => unchecked(value > byte.MaxValue ? byte.MaxValue
            ↪  : (byte)value);
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public static sbyte SByte(ulong value) => unchecked(value > (ulong)sbyte.MaxValue ?
            ↪  sbyte.MaxValue : (sbyte)value);
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public static bool Boolean(ulong value) => value > 0UL;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public static char Char(ulong value) => unchecked(value > char.MaxValue ?
            ↪  UnknownCharacter : (char)value);
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          public static DateTime DateTime(ulong value) => unchecked(value > long.MaxValue ?
            ↪  System.DateTime.MaxValue : new DateTime((long)value));
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public static TimeSpan TimeSpan(ulong value) => unchecked(value > long.MaxValue ?
            ↪  System.TimeSpan.MaxValue : new TimeSpan((long)value));
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          public static ulong UInt64(long value) => unchecked(value < (long)ulong.MinValue ?
            ↪  ulong.MinValue : (ulong)value);
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          public static ulong UInt64(int value) => unchecked(value < (int)ulong.MinValue ?
            ↪  ulong.MinValue : (ulong)value);
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public static ulong UInt64(short value) => unchecked(value < (short)ulong.MinValue ?
            ↪  ulong.MinValue : (ulong)value);
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          public static ulong UInt64(sbyte value) => unchecked(value < (sbyte)ulong.MinValue ?
            ↪  ulong.MinValue : (ulong)value);
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          public static ulong UInt64(bool value) => value ? 1UL : 0UL;
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          public static ulong UInt64(char value) => value;
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public static long Signed(ulong value) => unchecked((long)value);
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          public static int Signed(uint value) => unchecked((int)value);
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          public static short Signed(ushort value) => unchecked((short)value);
74
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          public static sbyte Signed(byte value) => unchecked((sbyte)value);
77
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          public static object Signed<T>(T value) => To<T>.Signed(value);
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          public static ulong Unsigned(long value) => unchecked((ulong)value);
83
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          public static uint Unsigned(int value) => unchecked((uint)value);
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ushort Unsigned(short value) => unchecked((ushort)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static byte Unsigned(sbyte value) => unchecked((byte)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static object Unsigned<T>(T value) => To<T>.Unsigned(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T UnsignedAs<T>(object value) => To<T>.UnsignedAs(value);
    }
}
```

./To[T].cs

```
using System;
using Platform.Exceptions;
using Platform.Reflection;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Converters
{
    public static class To<T>
    {
        public static readonly Func<T, object> Signed;
        public static readonly Func<T, object> Unsigned;
        public static readonly Func<object, T> UnsignedAs;

        static To()
        {
            Signed = CompileSignedDelegate();
            Unsigned = CompileUnsignedDelegate();
            UnsignedAs = CompileUnsignedAsDelegate();
        }

        static private Func<T, object> CompileSignedDelegate()
        {
            return DelegateHelpers.Compile<Func<T, object>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<T>();
                emiter.LoadArgument(0);
                var method = typeof(To).GetMethod("Signed", Types<T>.Array);
                emiter.Call(method);
                emiter.Box(method.ReturnType);
                emiter.Return();
            });
        }

        static private Func<T, object> CompileUnsignedDelegate()
        {
            return DelegateHelpers.Compile<Func<T, object>>(emiter =>
            {
                Ensure.Always.IsSignedInteger<T>();
                emiter.LoadArgument(0);
                var method = typeof(To).GetMethod("Unsigned", Types<T>.Array);
                emiter.Call(method);
                emiter.Box(method.ReturnType);
                emiter.Return();
            });
        }

        static private Func<object, T> CompileUnsignedAsDelegate()
        {
            return DelegateHelpers.Compile<Func<object, T>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<T>();
                emiter.LoadArgument(0);
                var signedVersion = NumericType<T>.SignedVersion;
                emiter.UnboxValue(signedVersion);
                var method = typeof(To).GetMethod("Unsigned", new[] { signedVersion });
                emiter.Call(method);
                emiter.Return();
            });
        }
    }
}
```

Index