LinksPlatform's Platform.Converters Class Library

1.1 ./Platform.Converters/CachingConverterDecorator.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections;

#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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter,
        ↪  IDictionary<TSource, TTarget> cache) => (_baseConverter, _cache) = (baseConverter,
        ↪  cache);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter) :
        ↪  this(baseConverter, new Dictionary<TSource, TTarget>()) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TTarget Convert(TSource source) => _cache.GetOrAdd(source,
        ↪  _baseConverter.Convert);
    }
}
```

1.2 ./Platform.Converters/CheckedConverter.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Reflection;

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

namespace Platform.Converters
{
    public abstract class CheckedConverter<TSource, TTarget> : ConverterBase<TSource, TTarget>
    {
        public static CheckedConverter<TSource, TTarget> Default
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        } = CompileCheckedConverter();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static CheckedConverter<TSource, TTarget> CompileCheckedConverter()
        {
            var type = CreateTypeInheritedFrom<CheckedConverter<TSource, TTarget>>();
            EmitConvertMethod(type, il => il.CheckedConvert<TSource, TTarget>());
            return (CheckedConverter<TSource,
            ↪  TTarget>)Activator.CreateInstance(type.CreateTypeInfo());
        }
    }
}
```

1.3 ./Platform.Converters/ConverterBase.cs

```
using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Reflection;

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

namespace Platform.Converters
{
    public abstract class ConverterBase<TSource, TTarget> : IConverter<TSource, TTarget>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TTarget Convert(TSource source);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static void ConvertAndUnbox(ILGenerator il)
        {
            var typeContainer =
            ↪  typeof(NumericType<TTarget>).GetField(nameof(NumericType<TTarget>.Type),
            ↪  BindingFlags.Static | BindingFlags.Public);
```

```
            il.Emit(OpCodes.Ldsfld, typeContainer);
            il.Call(typeof(Convert).GetMethod(nameof(System.Convert.ChangeType), Types<object, Type>.Array));
            il.UnboxValue(typeof(TTarget));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static string GetNewName() => Guid.NewGuid().ToString("N");

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static TypeBuilder CreateTypeInheritedFrom<TBaseClass>()
        {
            var 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 | TypeAttributes.Sealed, typeof(TBaseClass));
            return type;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static void EmitConvertMethod(TypeBuilder typeBuilder, Action<ILGenerator> emitConversion)
        {
            typeBuilder.EmitFinalVirtualMethod<Converter<TSource, TTarget>>(nameof(IConverter<TSource, TTarget>.Convert), il =>
            {
                il.LoadArgument(1);
                if (typeof(TSource) == typeof(object) && typeof(TTarget) != typeof(object))
                {
                    ConvertAndUnbox(il);
                }
                else if (typeof(TSource) != typeof(object) && typeof(TTarget) == typeof(object))
                {
                    il.Box(typeof(TSource));
                }
                else
                {
                    emitConversion(il);
                }
                il.Return();
            });
        }
    }
}
```

1.4 ./Platform.Converters/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);
    }
}
```

1.5 ./Platform.Converters/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>
    {
    }
}
```

1.6 ./Platform.Converters/To.cs

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

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

namespace Platform.Converters
{
    [Obsolete]
    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);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int Int32(ulong value) => unchecked(value > int.MaxValue ? int.MaxValue :
        ↪ (int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ushort UInt16(ulong value) => unchecked(value > ushort.MaxValue ?
        ↪ ushort.MaxValue : (ushort)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static short Int16(ulong value) => unchecked(value > (ulong)short.MaxValue ?
        ↪ short.MaxValue : (short)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static byte Byte(ulong value) => unchecked(value > byte.MaxValue ? byte.MaxValue
        ↪ : (byte)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static sbyte SByte(ulong value) => unchecked(value > (ulong)sbyte.MaxValue ?
        ↪ sbyte.MaxValue : (sbyte)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Boolean(ulong value) => value > 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static char Char(ulong value) => unchecked(value > char.MaxValue ?
        ↪ UnknownCharacter : (char)value);

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(long value) => unchecked(value < (long)ulong.MinValue ?
        ↪ ulong.MinValue : (ulong)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(int value) => unchecked(value < (int)ulong.MinValue ?
        ↪ ulong.MinValue : (ulong)value);

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(short value) => unchecked(value < (short)ulong.MinValue ?
        ↪  ulong.MinValue : (ulong)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(sbyte value) => unchecked(value < (sbyte)ulong.MinValue ?
        ↪  ulong.MinValue : (ulong)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(bool value) => value ? 1UL : 0UL;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long Signed(ulong value) => unchecked((long)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int Signed(uint value) => unchecked((int)value);

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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);
    }
}
```

1.7 ./Platform.Converters/To[T].cs

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

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

namespace Platform.Converters
{
    [Obsolete]
    public static class To<T>
    {
        public static readonly Func<T, object> Signed = CompileSignedDelegate();
        public static readonly Func<T, object> Unsigned = CompileUnsignedDelegate();
        public static readonly Func<object, T> UnsignedAs = CompileUnsignedAsDelegate();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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();
            });
        }

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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();
            });
        }
    }
}
```

1.8 ./Platform.Converters/UncheckedConverter.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Reflection;

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

namespace Platform.Converters
{
    public abstract class UncheckedConverter<TSource, TTarget> : ConverterBase<TSource, TTarget>
    {
        public static UncheckedConverter<TSource, TTarget> Default
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        } = CompileUncheckedConverter();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static UncheckedConverter<TSource, TTarget> CompileUncheckedConverter()
        {
            var type = CreateTypeInheritedFrom<UncheckedConverter<TSource, TTarget>>();
            EmitConvertMethod(type, il => il.UncheckedConvert<TSource, TTarget>());
            return (UncheckedConverter<TSource,
             ↪  TTarget>)Activator.CreateInstance(type.CreateTypeInfo());
        }
    }
}
```

1.9 ./Platform.Converters/UncheckedSignExtendingConverter.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Reflection;

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

namespace Platform.Converters
{
    public abstract class UncheckedSignExtendingConverter<TSource, TTarget> :
     ↪  ConverterBase<TSource, TTarget>
    {
        public static UncheckedSignExtendingConverter<TSource, TTarget> Default
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        } = CompileUncheckedConverter();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static UncheckedSignExtendingConverter<TSource, TTarget>
         ↪  CompileUncheckedConverter()
```

```
        {
            var type = CreateTypeInheritedFrom<UncheckedSignExtendingConverter<TSource,
            ↪  TTarget>>();
            EmitConvertMethod(type, il => il.UncheckedConvert<TSource, TTarget>(extendSign:
            ↪  true));
            return (UncheckedSignExtendingConverter<TSource,
            ↪  TTarget>)Activator.CreateInstance(type.CreateTypeInfo());
        }
    }
}
```

1.10 ./Platform.Converters.Tests/ConverterTests.cs

```
using Xunit;

namespace Platform.Converters.Tests
{
    public class ConverterTests
    {
        [Fact]
        public void SameTypeTest()
        {
            var result = UncheckedConverter<ulong, ulong>.Default.Convert(2UL);
            Assert.Equal(2UL, result);
            result = CheckedConverter<ulong, ulong>.Default.Convert(2UL);
            Assert.Equal(2UL, result);
        }

        [Fact]
        public void Int32ToUInt64Test()
        {
            var result = UncheckedConverter<int, ulong>.Default.Convert(2);
            Assert.Equal(2UL, result);
            result = CheckedConverter<int, ulong>.Default.Convert(2);
            Assert.Equal(2UL, result);
        }

        [Fact]
        public void SignExtensionTest()
        {
            var result = UncheckedSignExtendingConverter<byte, long>.Default.Convert(128);
            Assert.Equal(-128L, result);
            result = UncheckedConverter<byte, long>.Default.Convert(128);
            Assert.Equal(128L, result);
        }
    }
}
```

Index