LinksPlatform's Platform.Converters Class Library

1.1 ./csharp/Platform.Converters/CachingConverterDecorator.cs

```csharp
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
{
    /// <summary>
    /// <para>
    /// Represents the caching converter decorator.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IConverter{TSource, TTarget}"/>
    public class CachingConverterDecorator<TSource, TTarget> : IConverter<TSource, TTarget>
    {
        private readonly IConverter<TSource, TTarget> _baseConverter;
        private readonly IDictionary<TSource, TTarget> _cache;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CachingConverterDecorator"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="baseConverter">
        /// <para>A base converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="cache">
        /// <para>A cache.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter,
         ↪  IDictionary<TSource, TTarget> cache) => (_baseConverter, _cache) = (baseConverter,
         ↪  cache);

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CachingConverterDecorator"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="baseConverter">
        /// <para>A base converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter) :
         ↪  this(baseConverter, new Dictionary<TSource, TTarget>()) { }

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The target</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TTarget Convert(TSource source) => _cache.GetOrAdd(source,
         ↪  _baseConverter.Convert);
    }
}
```

1.2 ./csharp/Platform.Converters/CheckedConverter.cs

```csharp
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
{
    /// <summary>
    /// <para>
    /// Represents the checked converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ConverterBase{TSource, TTarget}"/>
    public abstract class CheckedConverter<TSource, TTarget> : ConverterBase<TSource, TTarget>
    {
        /// <summary>
        /// <para>
        /// Gets the default value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static CheckedConverter<TSource, TTarget> Default
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        } = CompileCheckedConverter();

        /// <summary>
        /// <para>
        /// Compiles the checked converter.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A checked converter of t source and t target</para>
        /// <para></para>
        /// </returns>
        [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 ./csharp/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
{
    /// <summary>
    /// <para>Represents a base implementation for IConverter interface with the basic logic
    ↪  necessary for value converter from the <typeparamref name="TSource"/> type to the
    ↪  <typeparamref name="TTarget"/> type.</para>
    /// <para>Представляет базовую реализацию для интерфейса IConverter с основной логикой
    ↪  необходимой для конвертера значений из типа <typeparamref name="TSource"/> в тип
    ↪  <typeparamref name="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 abstract class ConverterBase<TSource, TTarget> : IConverter<TSource, TTarget>
    {
        /// <summary>
        /// <para>Converts the value of the <typeparamref name="TSource"/> type to the value of
        ↪  the <typeparamref name="TTarget"/> type.</para>
        /// <para>Конвертирует значение типа <typeparamref name="TSource"/> в значение типа
        ↪  <typeparamref name="TTarget"/>.</para>
        /// </summary>
        /// <param name="source"><para>The <typeparamref name=="TSource"/> type
        ↪  value.</para><para>Значение типа <typeparamref name="TSource"/>.</para></param>
        /// <returns><para>The converted value of the <typeparamref name="TTarget"/>
        ↪  type.</para><para>Значение конвертированное в тип <typeparamref
        ↪  name="TTarget"/>.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TTarget Convert(TSource source);

        /// <summary>
        /// <para>Generates a sequence of instructions using <see cref="ILGenerator"/> that
        ↪  converts a value of type <see cref="System.Object"/> to a value of type
        ↪  <typeparamref name="TTarget"/>.</para>
        /// <para>Генерирует последовательность инструкций при помощи <see cref="ILGenerator"/>
        ↪  выполняющую преобразование значения типа <see cref="System.Object"/> к значению типа
        ↪  <typeparamref name="TTarget"/>.</para>
        /// </summary>
        /// <param name="il"><para>An <see cref="ILGenerator"/> instance.</para><para>Экземпляр
        ↪  <see cref="ILGenerator"/>.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static void ConvertFromObject(ILGenerator il)
        {
            var returnDefault = il.DefineLabel();
            il.Emit(OpCodes.Brfalse_S, returnDefault);
            il.LoadArgument(1);
            il.Emit(OpCodes.Castclass, typeof(IConvertible));
            il.Emit(OpCodes.Ldnull);
            il.Emit(OpCodes.Callvirt, GetMethodForConversionToTargetType());
            il.Return();
            il.MarkLabel(returnDefault);
            LoadDefault(il, typeof(TTarget));
        }

        /// <summary>
        /// <para>Gets a new unique name of an assembly.</para>
        /// <para>Возвращает новое уникальное имя сборки.</para>
        /// </summary>
        /// <returns><para>A new unique name of an assembly.</para><para>Новое уникальное имя
        ↪  сборки.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static string GetNewName() => Guid.NewGuid().ToString("N");

        /// <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>
        [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;
        }

        /// <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>
        [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))
                {
                    ConvertFromObject(il);
                }
                else if (typeof(TSource) != typeof(object) && typeof(TTarget) == typeof(object))
                {
                    il.Box(typeof(TSource));
                }
                else
                {
                    emitConversion(il);
                }
                il.Return();
            });
        }

        /// <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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static MethodInfo GetMethodForConversionToTargetType()
        {
            var targetType = typeof(TTarget);
            var convertibleType = typeof(IConvertible);
            var typeParameters = Types<IFormatProvider>.Array;
            if (targetType == typeof(bool))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToBoolean), typeParameters);
            }
            else if (targetType == typeof(byte))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToByte), typeParameters);
            }
            else if (targetType == typeof(char))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToChar), typeParameters);
            }
            else if (targetType == typeof(DateTime))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToDateTime),
                 ↪  typeParameters);
            }
            else if (targetType == typeof(decimal))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToDecimal), typeParameters);
            }
            else if (targetType == typeof(double))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToDouble), typeParameters);
            }
            else if (targetType == typeof(short))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToInt16), typeParameters);
            }
            else if (targetType == typeof(int))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToInt32), typeParameters);
            }
            else if (targetType == typeof(long))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToInt64), typeParameters);
            }
            else if (targetType == typeof(sbyte))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToSByte), typeParameters);
            }
            else if (targetType == typeof(float))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToSingle), typeParameters);
            }
            else if (targetType == typeof(string))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToString), typeParameters);
            }
            else if (targetType == typeof(ushort))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToUInt16), typeParameters);
            }
            else if (targetType == typeof(uint))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToUInt32), typeParameters);
            }
            else if (targetType == typeof(ulong))
            {
                return convertibleType.GetMethod(nameof(IConvertible.ToUInt64), typeParameters);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        /// <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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static void LoadDefault(ILGenerator il, Type targetType)
        {
            if (targetType == typeof(string))
            {
                il.Emit(OpCodes.Ldsfld, targetType.GetField(nameof(string.Empty),
                ↪  BindingFlags.Static | BindingFlags.Public));
            }
            else if (targetType == typeof(DateTime))
            {
                il.Emit(OpCodes.Ldsfld, targetType.GetField(nameof(DateTime.MinValue),
                ↪  BindingFlags.Static | BindingFlags.Public));
            }
            else if (targetType == typeof(decimal))
            {
                il.Emit(OpCodes.Ldsfld, targetType.GetField(nameof(decimal.Zero),
                ↪  BindingFlags.Static | BindingFlags.Public));
            }
            else if (targetType == typeof(float))
            {
                il.LoadConstant(0.0F);
            }
            else if (targetType == typeof(double))
            {
                il.LoadConstant(0.0D);
            }
            else if (targetType == typeof(long) || targetType == typeof(ulong))
            {
                il.LoadConstant(0L);
            }
            else
            {
                il.LoadConstant(0);
            }
        }
    }
}
```

1.4 ./csharp/Platform.Converters/IConverter[TSource, TTarget].cs

```
namespace Platform.Converters
{
    /// <summary>
    /// <para>Defines a value converter from the <typeparamref name="TSource"/> type to the
    ↪  <typeparamref name="TTarget"/> type.</para>
    /// <para>Определяет конвертер значений из типа <typeparamref name="TSource"/> в тип
    ↪  <typeparamref name="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 <typeparamref name="TSource"/> type to the value of
        ↪  the <typeparamref name="TTarget"/> type.</para>
        /// <para>Конвертирует значение типа <typeparamref name="TSource"/> в значение типа
        ↪  <typeparamref name="TTarget"/>.</para>
        /// </summary>
        /// <param name="source"><para>The <typeparamref name=="TSource"/> type
        ↪  value.</para><para>Значение типа <typeparamref name="TSource"/>.</para></param>
        /// <returns><para>The converted value of the <typeparamref name="TTarget"/>
        ↪  type.</para><para>Значение конвертированное в тип <typeparamref
        ↪  name="TTarget"/>.</para></returns>
        TTarget Convert(TSource source);
    }
}
```

1.5 ./csharp/Platform.Converters/IConverter[T].cs

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

1.6 ./csharp/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
{
    /// <summary>
    /// <para>
    /// Represents the unchecked converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ConverterBase{TSource, TTarget}"/>
    public abstract class UncheckedConverter<TSource, TTarget> : ConverterBase<TSource, TTarget>
    {
        /// <summary>
        /// <para>
        /// Gets the default value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static UncheckedConverter<TSource, TTarget> Default
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        } = CompileUncheckedConverter();

        /// <summary>
```

```csharp
        /// <para>
        /// Compiles the unchecked converter.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>An unchecked converter of t source and t target</para>
        /// <para></para>
        /// </returns>
        [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());
        }
    }
}
```

1.7 ./csharp/Platform.Converters/UncheckedSignExtendingConverter.cs

```csharp
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
{
    /// <summary>
    /// <para>
    /// Represents the unchecked sign extending converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ConverterBase{TSource, TTarget}"/>
    public abstract class UncheckedSignExtendingConverter<TSource, TTarget> :
     ↪  ConverterBase<TSource, TTarget>
    {
        /// <summary>
        /// <para>
        /// Gets the default value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static UncheckedSignExtendingConverter<TSource, TTarget> Default
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        } = CompileUncheckedConverter();

        /// <summary>
        /// <para>
        /// Compiles the unchecked converter.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>An unchecked sign extending converter of t source and t target</para>
        /// <para></para>
        /// </returns>
        [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());
        }
    }
}
```

1.8 ./csharp/Platform.Converters.Tests/ConverterTests.cs

```csharp
using System;
using Xunit;
```

```csharp

namespace Platform.Converters.Tests
{
    /// <summary>
    /// <para>
    /// Represents the converter tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class ConverterTests
    {
        /// <summary>
        /// <para>
        /// Tests that same type test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static 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);
        }

        /// <summary>
        /// <para>
        /// Tests that int 32 to u int 64 test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static 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);
        }

        /// <summary>
        /// <para>
        /// Tests that sign extension test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static 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);
        }

        /// <summary>
        /// <para>
        /// Tests that object test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void ObjectTest()
        {
            TestObjectConversion("1");
            TestObjectConversion(DateTime.UtcNow);
            TestObjectConversion(1.0F);
            TestObjectConversion(1.0D);
            TestObjectConversion(1.0M);
            TestObjectConversion(1UL);
            TestObjectConversion(1L);
            TestObjectConversion(1U);
            TestObjectConversion(1);
            TestObjectConversion((char)1);
            TestObjectConversion((ushort)1);
            TestObjectConversion((short)1);
            TestObjectConversion((byte)1);
            TestObjectConversion((sbyte)1);
            TestObjectConversion(true);
        }

        private static void TestObjectConversion<T>(T value) => Assert.Equal(value, UncheckedConverter<object, T>.Default.Convert(value));
    }
}
```

Index