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
{
    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 ./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
{
    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 ./csharp/Platform.Converters/ConverterBase.cs

```csharp
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>
```

```csharp
    /// <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="source"><para>The source type value (TSource).</para><para>Значение
        ↪   исходного типа (TSource).</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>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 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;
        }
```

```csharp
        /// <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))
```

```csharp
                    {
                        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);
                }
```

```
207            else if (targetType == typeof(long) || targetType == typeof(ulong))
208            {
209                il.LoadConstant(0L);
210            }
211            else
212            {
213                il.LoadConstant(0);
214            }
215        }
216    }
217 }
```

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

```
1  namespace Platform.Converters
2  {
3      /// <summary>
4      /// <para>Defines a value converter from the <typeparamref name="TSource"/> type to the
       ↪  <typeparamref name="TTarget"/> type.</para>
5      /// <para>Определяет конвертер значений из типа <typeparamref name="TSource"/> в тип
       ↪  <typeparamref name="TTarget"/>.</para>
6      /// </summary>
7      /// <typeparam name="TSource"><para>Source type of conversion.</para><para>Исходный тип
       ↪  конверсии.</para></typeparam>
8      /// <typeparam name="TTarget"><para>Target type of conversion.</para><para>Целевой тип
       ↪  конверсии.</para></typeparam>
9      public interface IConverter<in TSource, out TTarget>
10     {
11         /// <summary>
12         /// <para>Converts the value of the <typeparamref name="TSource"/> type to the value of
          ↪  the <typeparamref name="TTarget"/> type.</para>
13         /// <para>Конвертирует значение типа <typeparamref name="TSource"/> в значение типа
          ↪  <typeparamref name="TTarget"/>.</para>
14         /// </summary>
15         /// <param name="source"><para>The <typeparamref name=="TSource"/> type
          ↪  value.</para><para>Значение типа <typeparamref name="TSource"/>.</para></param>
16         /// <returns><para>The converted value of the <typeparamref name="TTarget"/>
          ↪  type.</para><para>Значение конвертированное в тип <typeparamref
          ↪  name="TTarget"/>.</para></returns>
17         TTarget Convert(TSource source);
18     }
19 }
```

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

```
1  namespace Platform.Converters
2  {
3      /// <summary>
4      /// <para>Defines a converter between two values of the same <typeparamref name="T"/>
       ↪  type.</para>
5      /// <para>Определяет конвертер между двумя значениями одного типа <typeparamref
       ↪  name="T"/>.</para>
6      /// </summary>
7      /// <typeparam name="T"><para>The type of value to convert.</para><para>Тип преобразуемого
       ↪  значения.</para></typeparam>
8      public interface IConverter<T> : IConverter<T, T>
9      {
10     }
11 }
```

1.6 ./csharp/Platform.Converters/UncheckedConverter.cs

```
1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Reflection;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Converters
8  {
9      public abstract class UncheckedConverter<TSource, TTarget> : ConverterBase<TSource, TTarget>
10     {
11         public static UncheckedConverter<TSource, TTarget> Default
12         {
13             [MethodImpl(MethodImplOptions.AggressiveInlining)]
14             get;
15         } = CompileUncheckedConverter();
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         private static UncheckedConverter<TSource, TTarget> CompileUncheckedConverter()
```

```csharp
19          {
20              var type = CreateTypeInheritedFrom<UncheckedConverter<TSource, TTarget>>();
21              EmitConvertMethod(type, il => il.UncheckedConvert<TSource, TTarget>());
22              return (UncheckedConverter<TSource,
                →   TTarget>)Activator.CreateInstance(type.CreateTypeInfo());
23          }
24      }
25  }
```

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

```csharp
1   using System;
2   using System.Runtime.CompilerServices;
3   using Platform.Reflection;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Converters
8   {
9       public abstract class UncheckedSignExtendingConverter<TSource, TTarget> :
        →   ConverterBase<TSource, TTarget>
10      {
11          public static UncheckedSignExtendingConverter<TSource, TTarget> Default
12          {
13              [MethodImpl(MethodImplOptions.AggressiveInlining)]
14              get;
15          } = CompileUncheckedConverter();
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          private static UncheckedSignExtendingConverter<TSource, TTarget>
            →   CompileUncheckedConverter()
19          {
20              var type = CreateTypeInheritedFrom<UncheckedSignExtendingConverter<TSource,
                →   TTarget>>();
21              EmitConvertMethod(type, il => il.UncheckedConvert<TSource, TTarget>(extendSign:
                →   true));
22              return (UncheckedSignExtendingConverter<TSource,
                →   TTarget>)Activator.CreateInstance(type.CreateTypeInfo());
23          }
24      }
25  }
```

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

```csharp
1   using System;
2   using Xunit;
3
4   namespace Platform.Converters.Tests
5   {
6       public static class ConverterTests
7       {
8           [Fact]
9           public static void SameTypeTest()
10          {
11              var result = UncheckedConverter<ulong, ulong>.Default.Convert(2UL);
12              Assert.Equal(2UL, result);
13              result = CheckedConverter<ulong, ulong>.Default.Convert(2UL);
14              Assert.Equal(2UL, result);
15          }
16
17          [Fact]
18          public static void Int32ToUInt64Test()
19          {
20              var result = UncheckedConverter<int, ulong>.Default.Convert(2);
21              Assert.Equal(2UL, result);
22              result = CheckedConverter<int, ulong>.Default.Convert(2);
23              Assert.Equal(2UL, result);
24          }
25
26          [Fact]
27          public static void SignExtensionTest()
28          {
29              var result = UncheckedSignExtendingConverter<byte, long>.Default.Convert(128);
30              Assert.Equal(-128L, result);
31              result = UncheckedConverter<byte, long>.Default.Convert(128);
32              Assert.Equal(128L, result);
33          }
34
35          [Fact]
36          public static void ObjectTest()
37          {
```

```csharp
            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