LinksPlatform's Platform.Converters Class Library

1.1 ./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 ./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 ./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
{
    public abstract class ConverterBase<TSource, TTarget> : IConverter<TSource, TTarget>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TTarget Convert(TSource source);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected static void ConvertFromObject(ILGenerator il)
        {
            var returnDefault = il.DefineLabel();
            il.Emit(OpCodes.Brfalse_S, returnDefault);
```

```csharp
                    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));
        }

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

        [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);
```

```csharp
            }
            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();
            }
        }

        [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);
            }
        }
```

```
169          }
170      }
```

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

```
1   namespace Platform.Converters
2   {
3       /// <summary>
4       /// <para>Defines a converter between two types (TSource and TTarget).</para>
5       /// <para>Определяет конвертер между двумя типами (исходным TSource и целевым
        ↪  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 source type (TSource) to the value of the target
            ↪  type.</para>
13          /// <para>Конвертирует значение исходного типа (TSource) в значение целевого типа.</para>
14          /// </summary>
15          /// <param name="source"><para>The source type value (TSource).</para><para>Значение
            ↪  исходного типа (TSource).</para></param>
16          /// <returns><para>The value is converted to the target type
            ↪  (TTarget).</para><para>Значение ковертированное в целевой тип
            ↪  (TTarget).</para></returns>
17          TTarget Convert(TSource source);
18      }
19  }
```

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

```
1   namespace Platform.Converters
2   {
3       /// <summary>
4       /// <para>Defines a converter between two values of the same type.</para>
5       /// <para>Определяет конвертер между двумя значениями одного типа.</para>
6       /// </summary>
7       /// <typeparam name="T"><para>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 ./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()
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 ./Platform.Converters/UncheckedSignExtendingConverter.cs

```
1   using System;
2   using System.Runtime.CompilerServices;
3   using Platform.Reflection;
```

```csharp
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 ./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          {
38              TestObjectConversion("1");
39              TestObjectConversion(DateTime.UtcNow);
40              TestObjectConversion(1.0F);
41              TestObjectConversion(1.0D);
42              TestObjectConversion(1.0M);
43              TestObjectConversion(1UL);
44              TestObjectConversion(1L);
45              TestObjectConversion(1U);
46              TestObjectConversion(1);
47              TestObjectConversion((char)1);
48              TestObjectConversion((ushort)1);
49              TestObjectConversion((short)1);
50              TestObjectConversion((byte)1);
```

```csharp
            TestObjectConversion((sbyte)1);
            TestObjectConversion(true);
        }

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

Index