LinksPlatform's Platform.Converters Class Library

1.1 ./csharp/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 ./csharp/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 ./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
{
    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);
```

```
            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);
```

```
94              }
95              else if (targetType == typeof(short))
96              {
97                  return convertibleType.GetMethod(nameof(IConvertible.ToInt16), typeParameters);
98              }
99              else if (targetType == typeof(int))
100             {
101                 return convertibleType.GetMethod(nameof(IConvertible.ToInt32), typeParameters);
102             }
103             else if (targetType == typeof(long))
104             {
105                 return convertibleType.GetMethod(nameof(IConvertible.ToInt64), typeParameters);
106             }
107             else if (targetType == typeof(sbyte))
108             {
109                 return convertibleType.GetMethod(nameof(IConvertible.ToSByte), typeParameters);
110             }
111             else if (targetType == typeof(float))
112             {
113                 return convertibleType.GetMethod(nameof(IConvertible.ToSingle), typeParameters);
114             }
115             else if (targetType == typeof(string))
116             {
117                 return convertibleType.GetMethod(nameof(IConvertible.ToString), typeParameters);
118             }
119             else if (targetType == typeof(ushort))
120             {
121                 return convertibleType.GetMethod(nameof(IConvertible.ToUInt16), typeParameters);
122             }
123             else if (targetType == typeof(uint))
124             {
125                 return convertibleType.GetMethod(nameof(IConvertible.ToUInt32), typeParameters);
126             }
127             else if (targetType == typeof(ulong))
128             {
129                 return convertibleType.GetMethod(nameof(IConvertible.ToUInt64), typeParameters);
130             }
131             else
132             {
133                 throw new NotSupportedException();
134             }
135         }
136
137         [MethodImpl(MethodImplOptions.AggressiveInlining)]
138         protected static void LoadDefault(ILGenerator il, Type targetType)
139         {
140             if (targetType == typeof(string))
141             {
142                 il.Emit(OpCodes.Ldsfld, targetType.GetField(nameof(string.Empty),
                    ↪  BindingFlags.Static | BindingFlags.Public));
143             }
144             else if (targetType == typeof(DateTime))
145             {
146                 il.Emit(OpCodes.Ldsfld, targetType.GetField(nameof(DateTime.MinValue),
                    ↪  BindingFlags.Static | BindingFlags.Public));
147             }
148             else if (targetType == typeof(decimal))
149             {
150                 il.Emit(OpCodes.Ldsfld, targetType.GetField(nameof(decimal.Zero),
                    ↪  BindingFlags.Static | BindingFlags.Public));
151             }
152             else if (targetType == typeof(float))
153             {
154                 il.LoadConstant(0.0F);
155             }
156             else if (targetType == typeof(double))
157             {
158                 il.LoadConstant(0.0D);
159             }
160             else if (targetType == typeof(long) || targetType == typeof(ulong))
161             {
162                 il.LoadConstant(0L);
163             }
164             else
165             {
166                 il.LoadConstant(0);
167             }
168         }
```

```
    }
}
```

1.4 ./csharp/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 ./csharp/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 ./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
{
    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());
        }
    }
}
```

1.7 ./csharp/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());
        }
    }
}
```

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

```
using System;
using Xunit;

namespace Platform.Converters.Tests
{
    public static class ConverterTests
    {
        [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);
        }

        [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);
        }

        [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);
        }

        [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);
```

```
51              TestObjectConversion((sbyte)1);
52              TestObjectConversion(true);
53          }
54
55          private static void TestObjectConversion<T>(T value) => Assert.Equal(value,
            ↪  UncheckedConverter<object, T>.Default.Convert(value));
56      }
57  }
```

Index