

LinksPlatform's Platform.Converters Class Library

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Collections;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Converters
8 {
9     public class CachingConverterDecorator<TSource, TTarget> : IConverter<TSource, TTarget>
10     {
11         private readonly IConverter<TSource, TTarget> _baseConverter;
12         private readonly IDictionary<TSource, TTarget> _cache;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter,
16             ↪ IDictionary<TSource, TTarget> cache) => (_baseConverter, _cache) = (baseConverter,
17             ↪ cache);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter) :
21             ↪ this(baseConverter, new Dictionary<TSource, TTarget>()) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public TTarget Convert(TSource source) => _cache.GetOrAdd(source,
25             ↪ _baseConverter.Convert);
26     }
27 }
```

1.2 ./csharp/Platform.Converters/CheckedConverter.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 CheckedConverter<TSource, TTarget> : ConverterBase<TSource, TTarget>
10     {
11         public static CheckedConverter<TSource, TTarget> Default
12         {
13             [MethodImpl(MethodImplOptions.AggressiveInlining)]
14             get;
15         } = CompileCheckedConverter();
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         private static CheckedConverter<TSource, TTarget> CompileCheckedConverter()
19         {
20             var type = CreateTypeInheritedFrom<CheckedConverter<TSource, TTarget>>();
21             EmitConvertMethod(type, il => il.CheckedConvert<TSource, TTarget>());
22             return (CheckedConverter<TSource,
23                 ↪ TTarget>)Activator.CreateInstance(type.CreateTypeInfo());
24         }
25     }
26 }
```

1.3 ./csharp/Platform.Converters/ConverterBase.cs

```
1 using System;
2 using System.Reflection;
3 using System.Reflection.Emit;
4 using System.Runtime.CompilerServices;
5 using Platform.Reflection;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Converters
10 {
11     public abstract class ConverterBase<TSource, TTarget> : IConverter<TSource, TTarget>
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public abstract TTarget Convert(TSource source);
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected static void ConvertFromObject(ILGenerator il)
18         {
19             var returnDefault = il.DefineLabel();
20             il.Emit(OpCodes.Brfalse_S, returnDefault);
21         }
22     }
23 }
```

```

21         il.LoadArgument(1);
22         il.Emit(OpCodes.Castclass, typeof(IConvertible));
23         il.Emit(OpCodes.Ldnull);
24         il.Emit(OpCodes.Callvirt, GetMethodForConversionToTargetType());
25         il.Return();
26         il.MarkLabel(returnDefault);
27         LoadDefault(il, typeof(TTarget));
28     }
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected static string GetNewName() => Guid.NewGuid().ToString("N");
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected static TypeBuilder CreateTypeInheritedFrom<TBaseClass>()
35     {
36         var assemblyName = new AssemblyName(GetNewName());
37         var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
38             ↳ AssemblyBuilderAccess.Run);
39         var module = assembly.DefineDynamicModule(GetNewName());
40         var type = module.DefineType(GetNewName(), TypeAttributes.Public |
41             ↳ TypeAttributes.Class | TypeAttributes.Sealed, typeof(TBaseClass));
42         return type;
43     }
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected static void EmitConvertMethod(TypeBuilder typeBuilder, Action<ILGenerator>
47         ↳ emitConversion)
48     {
49         typeBuilder.EmitFinalVirtualMethod<Converter<TSource,
50             ↳ TTarget>>(nameof(IConverter<TSource, TTarget>.Convert), il =>
51         {
52             il.LoadArgument(1);
53             if (typeof(TSource) == typeof(object) && typeof(TTarget) != typeof(object))
54             {
55                 ConvertFromObject(il);
56             }
57             else if (typeof(TSource) != typeof(object) && typeof(TTarget) == typeof(object))
58             {
59                 il.Box(typeof(TSource));
60             }
61             else
62             {
63                 emitConversion(il);
64             }
65             il.Return();
66         });
67     }
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected static MethodInfo GetMethodForConversionToTargetType()
71     {
72         var targetType = typeof(TTarget);
73         var convertibleType = typeof(IConvertible);
74         var typeParameters = Types<IFormatProvider>.Array;
75         if (targetType == typeof(bool))
76         {
77             return convertibleType.GetMethod(nameof(IConvertible.ToBoolean), typeParameters);
78         }
79         else if (targetType == typeof(byte))
80         {
81             return convertibleType.GetMethod(nameof(IConvertible.ToByte), typeParameters);
82         }
83         else if (targetType == typeof(char))
84         {
85             return convertibleType.GetMethod(nameof(IConvertible.ToChar), typeParameters);
86         }
87         else if (targetType == typeof(DateTime))
88         {
89             return convertibleType.GetMethod(nameof(IConvertible.ToDateTime),
90                 ↳ typeParameters);
91         }
92         else if (targetType == typeof(decimal))
93         {
94             return convertibleType.GetMethod(nameof(IConvertible.ToDecimal), typeParameters);
95         }
96         else if (targetType == typeof(double))
97         {
98             return convertibleType.GetMethod(nameof(IConvertible.ToDouble), typeParameters);
99         }
100     }

```

```

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),
143             ↪ BindingFlags.Static | BindingFlags.Public));
144     }
145     else if (targetType == typeof(DateTime))
146     {
147         il.Emit(OpCodes.Ldsfld, targetType.GetField(nameof(DateTime.MinValue),
148             ↪ BindingFlags.Static | BindingFlags.Public));
149     }
150     else if (targetType == typeof(decimal))
151     {
152         il.Emit(OpCodes.Ldsfld, targetType.GetField(nameof(decimal.Zero),
153             ↪ BindingFlags.Static | BindingFlags.Public));
154     }
155     else if (targetType == typeof(float))
156     {
157         il.LoadConstant(0.0F);
158     }
159     else if (targetType == typeof(double))
160     {
161         il.LoadConstant(0.0D);
162     }
163     else if (targetType == typeof(long) || targetType == typeof(ulong))
164     {
165         il.LoadConstant(0L);
166     }
167     else
168     {
169         il.LoadConstant(0);
170     }
171 }

```

```
169     }
170 }
```

1.4 ./csharp/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
6     ↪ TTarget).</para>
7     /// </summary>
8     /// <typeparam name="TSource"><para>Source type of conversion.</para><para>Исходный тип
9     ↪ конверсии.</para></typeparam>
10    /// <typeparam name="TTarget"><para>Target type of conversion.</para><para>Целевой тип
11    ↪ конверсии.</para></typeparam>
12    public interface IConverter<in TSource, out TTarget>
13    {
14        /// <summary>
15        /// <para>Converts the value of the source type (TSource) to the value of the target
16        ↪ type.</para>
17        /// <para>Конвертирует значение исходного типа (TSource) в значение целевого типа.</para>
18        /// </summary>
19        /// <param name="source"><para>The source type value (TSource).</para><para>Значение
20        ↪ исходного типа (TSource).</para></param>
21        /// <returns><para>The value is converted to the target type
22        ↪ (TTarget).</para><para>Значение ковертированное в целевой тип
23        ↪ (TTarget).</para></returns>
24        TTarget Convert(TSource source);
25    }
26 }
```

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 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
8     ↪ значения.</para></typeparam>
9     public interface IConverter<T> : IConverter<T, T>
10    {
11    }
12 }
```

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()
19        {
20            var type = CreateTypeInheritedFrom<UncheckedConverter<TSource, TTarget>>();
21            EmitConvertMethod(type, il => il.UncheckedConvert<TSource, TTarget>());
22            return (UncheckedConverter<TSource,
23                ↪ TTarget>)Activator.CreateInstance(type.CreateTypeInfo());
24        }
25    }
26 }
```

1.7 ./csharp/Platform.Converters/UncheckedSignExtendingConverter.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 UncheckedSignExtendingConverter<TSource, TTarget> :
10         ⇨ ConverterBase<TSource, TTarget>
11     {
12         public static UncheckedSignExtendingConverter<TSource, TTarget> Default
13         {
14             [MethodImpl(MethodImplOptions.AggressiveInlining)]
15             get;
16         } = CompileUncheckedConverter();
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         private static UncheckedSignExtendingConverter<TSource, TTarget>
20             ⇨ CompileUncheckedConverter()
21         {
22             var type = CreateTypeInheritedFrom<UncheckedSignExtendingConverter<TSource,
23                 ⇨ TTarget>>();
24             EmitConvertMethod(type, il => il.UncheckedConvert<TSource, TTarget>(extendSign:
25                 ⇨ true));
26             return (UncheckedSignExtendingConverter<TSource,
27                 ⇨ TTarget>)Activator.CreateInstance(type.CreateTypeInfo());
28         }
29     }
30 }

```

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

```

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);

```

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

Index

- ./csharp/Platform.Converters.Tests/ConverterTests.cs, 5
- ./csharp/Platform.Converters/CachingConverterDecorator.cs, 1
- ./csharp/Platform.Converters/CheckedConverter.cs, 1
- ./csharp/Platform.Converters/ConverterBase.cs, 1
- ./csharp/Platform.Converters/IConverter[TSource, TTarget].cs, 4
- ./csharp/Platform.Converters/IConverter[T].cs, 4
- ./csharp/Platform.Converters/UncheckedConverter.cs, 4
- ./csharp/Platform.Converters/UncheckedSignExtendingConverter.cs, 4