LinksPlatform's Platform.Reflection Class Library

1.1 ./Platform.Reflection/AssemblyExtensions.cs

```
using System;
using System.Collections.Concurrent;
using System.Reflection;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class AssemblyExtensions
    {
        private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
        ↪  ConcurrentDictionary<Assembly, Type[]>();

        /// <remarks>
        /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetLoadableTypes(this Assembly assembly)
        {
            Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
            try
            {
                return assembly.GetTypes();
            }
            catch (ReflectionTypeLoadException e)
            {
                return e.Types.ToArray(t => t != null);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
        ↪  _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
    }
}
```

1.2 ./Platform.Reflection/DelegateHelpers.cs

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Exceptions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class DelegateHelpers
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode, bool
        ↪  typeMemberMethod)
            where TDelegate : Delegate
        {
            var @delegate = default(TDelegate);
            try
            {
                @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
                ↪  CompileDynamicMethod<TDelegate>(emitCode);
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
            return @delegate;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
        ↪  TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
        ↪  typeMemberMethod)
```

```
            where TDelegate : Delegate
        {
            var @delegate = CompileOrDefault<TDelegate>(emitCode, typeMemberMethod);
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
                @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
            }
            return @delegate;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
        ↪  : Delegate => Compile<TDelegate>(emitCode, false);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileDynamicMethod<TDelegate>(Action<ILGenerator> emitCode)
        {
            var delegateType = typeof(TDelegate);
            delegateType.GetDelegateCharacteristics(out Type returnType, out Type[]
            ↪  parameterTypes);
            var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
            emitCode(dynamicMethod.GetILGenerator());
            return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileTypeMemberMethod<TDelegate>(Action<ILGenerator> emitCode)
        {
            AssemblyName assemblyName = new AssemblyName(GetNewName());
            var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
            ↪  AssemblyBuilderAccess.Run);
            var module = assembly.DefineDynamicModule(GetNewName());
            var type = module.DefineType(GetNewName());
            var methodName = GetNewName();
            type.EmitStaticMethod<TDelegate>(methodName, emitCode);
            var typeInfo = type.CreateTypeInfo();
            return (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(typeof(TDele
            ↪  gate));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string GetNewName() => Guid.NewGuid().ToString("N");
    }
}
```

1.3 ./Platform.Reflection/DynamicExtensions.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class DynamicExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool HasProperty(this object @object, string propertyName)
        {
            var type = @object.GetType();
            if (type is IDictionary<string, object> dictionary)
            {
                return dictionary.ContainsKey(propertyName);
            }
            return type.GetProperty(propertyName) != null;
        }
    }
}
```

1.4 ./Platform.Reflection/EnsureExtensions.cs

```
using System;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Exceptions.ExtensionRoots;

#pragma warning disable IDE0060 // Remove unused parameter
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

```
10  namespace Platform.Reflection
11  {
12      public static class EnsureExtensions
13      {
14          #region Always
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root,
            ↪  Func<string> messageBuilder)
18          {
19              if (!NumericType<T>.IsNumeric || NumericType<T>.IsSigned ||
                ↪  NumericType<T>.IsFloatPoint)
20              {
21                  throw new NotSupportedException(messageBuilder());
22              }
23          }
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
            ↪  message)
27          {
28              string messageBuilder() => message;
29              IsUnsignedInteger<T>(root, messageBuilder);
30          }
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪  IsUnsignedInteger<T>(root, (string)null);
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
            ↪  messageBuilder)
37          {
38              if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
                ↪  NumericType<T>.IsFloatPoint)
39              {
40                  throw new NotSupportedException(messageBuilder());
41              }
42          }
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
            ↪  message)
46          {
47              string messageBuilder() => message;
48              IsSignedInteger<T>(root, messageBuilder);
49          }
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪  IsSignedInteger<T>(root, (string)null);
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
            ↪  messageBuilder)
56          {
57              if (!NumericType<T>.IsSigned)
58              {
59                  throw new NotSupportedException(messageBuilder());
60              }
61          }
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
65          {
66              string messageBuilder() => message;
67              IsSigned<T>(root, messageBuilder);
68          }
69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
            ↪  (string)null);
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
            ↪  messageBuilder)
75          {
76              if (!NumericType<T>.IsNumeric)
```

```
77              {
78                  throw new NotSupportedException(messageBuilder());
79              }
80          }
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
84          {
85              string messageBuilder() => message;
86              IsNumeric<T>(root, messageBuilder);
87          }
88
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪  IsNumeric<T>(root, (string)null);
91
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
            ↪  messageBuilder)
94          {
95              if (!NumericType<T>.CanBeNumeric)
96              {
97                  throw new NotSupportedException(messageBuilder());
98              }
99          }
100
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
103         {
104             string messageBuilder() => message;
105             CanBeNumeric<T>(root, messageBuilder);
106         }
107
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪  CanBeNumeric<T>(root, (string)null);
110
111         #endregion
112
113         #region OnDebug
114
115         [Conditional("DEBUG")]
116         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
            ↪  Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
117
118         [Conditional("DEBUG")]
119         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
            ↪  message) => Ensure.Always.IsUnsignedInteger<T>(message);
120
121         [Conditional("DEBUG")]
122         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsUnsignedInteger<T>();
123
124         [Conditional("DEBUG")]
125         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
126
127         [Conditional("DEBUG")]
128         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
            ↪  message) => Ensure.Always.IsSignedInteger<T>(message);
129
130         [Conditional("DEBUG")]
131         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsSignedInteger<T>();
132
133         [Conditional("DEBUG")]
134         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
135
136         [Conditional("DEBUG")]
137         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
            ↪  Ensure.Always.IsSigned<T>(message);
138
139         [Conditional("DEBUG")]
140         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsSigned<T>();
141
142         [Conditional("DEBUG")]
```

```
143         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
144
145         [Conditional("DEBUG")]
146         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
            ↪  Ensure.Always.IsNumeric<T>(message);
147
148         [Conditional("DEBUG")]
149         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsNumeric<T>();
150
151         [Conditional("DEBUG")]
152         public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.CanBeNumeric<T>(messageBuilder);
153
154         [Conditional("DEBUG")]
155         public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
            ↪  => Ensure.Always.CanBeNumeric<T>(message);
156
157         [Conditional("DEBUG")]
158         public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.CanBeNumeric<T>();
159
160         #endregion
161     }
162 }
```

1.5 ./Platform.Reflection/FieldInfoExtensions.cs

```
1  using System.Reflection;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Reflection
7  {
8      public static class FieldInfoExtensions
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T GetStaticValue<T>(this FieldInfo fieldInfo) =>
           ↪  (T)fieldInfo.GetValue(null);
12     }
13 }
```

1.6 ./Platform.Reflection/ILGeneratorExtensions.cs

```
1  using System;
2  using System.Linq;
3  using System.Reflection;
4  using System.Reflection.Emit;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Reflection
10 {
11     public static class ILGeneratorExtensions
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void Throw<T>(this ILGenerator generator) =>
           ↪  generator.ThrowException(typeof(T));
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator) =>
           ↪  UncheckedConvert<TSource, TTarget>(generator, extendSign: false);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator, bool
           ↪  extendSign)
21         {
22             var sourceType = typeof(TSource);
23             var targetType = typeof(TTarget);
24             if (sourceType == targetType)
25             {
26                 return;
27             }
28             if (extendSign)
29             {
30                 if (sourceType == typeof(byte))
31                 {
```

```
                    generator.Emit(OpCodes.Conv_I1);
                }
                if (sourceType == typeof(ushort))
                {
                    generator.Emit(OpCodes.Conv_I2);
                }
            }
            if (NumericType<TSource>.BitsSize > NumericType<TTarget>.BitsSize)
            {
                generator.ConvertToInteger<TSource>(targetType, extendSign: false);
            }
            else
            {
                generator.ConvertToInteger<TSource>(targetType, extendSign);
            }
            if (targetType == typeof(float))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_R4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_R_Un);
                }
            }
            else if (targetType == typeof(double))
            {
                generator.Emit(OpCodes.Conv_R8);
            }
            else if (targetType == typeof(bool))
            {
                generator.ConvertToBoolean<TSource>();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void ConvertToBoolean<TSource>(this ILGenerator generator)
        {
            generator.LoadConstant<TSource>(default);
            var sourceType = typeof(TSource);
            if (sourceType == typeof(float) || sourceType == typeof(double))
            {
                generator.Emit(OpCodes.Ceq);
                // Inversion of the first Ceq instruction
                generator.LoadConstant<int>(0);
                generator.Emit(OpCodes.Ceq);
            }
            else
            {
                generator.Emit(OpCodes.Cgt_Un);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void ConvertToInteger<TSource>(this ILGenerator generator, Type
        ↪  targetType, bool extendSign)
        {
            if (targetType == typeof(sbyte))
            {
                generator.Emit(OpCodes.Conv_I1);
            }
            else if (targetType == typeof(byte))
            {
                generator.Emit(OpCodes.Conv_U1);
            }
            else if (targetType == typeof(short))
            {
                generator.Emit(OpCodes.Conv_I2);
            }
            else if (targetType == typeof(ushort))
            {
                generator.Emit(OpCodes.Conv_U2);
            }
            else if (targetType == typeof(int))
            {
                generator.Emit(OpCodes.Conv_I4);
            }
```

```
            else if (targetType == typeof(uint))
            {
                generator.Emit(OpCodes.Conv_U4);
            }
            else if (targetType == typeof(long) || targetType == typeof(ulong))
            {
                if (NumericType<TSource>.IsSigned || extendSign)
                {
                    generator.Emit(OpCodes.Conv_I8);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_U8);
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
        {
            var sourceType = typeof(TSource);
            var targetType = typeof(TTarget);
            if (sourceType == targetType)
            {
                return;
            }
            if (targetType == typeof(short))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2_Un);
                }
            }
            else if (targetType == typeof(ushort))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U2);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U2_Un);
                }
            }
            else if (targetType == typeof(sbyte))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I1);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I1_Un);
                }
            }
            else if (targetType == typeof(byte))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U1);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U1_Un);
                }
            }
            else if (targetType == typeof(int))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I4);
                }
                else
                {
```

```
                    generator.Emit(OpCodes.Conv_Ovf_I4_Un);
                }
            }
            else if (targetType == typeof(uint))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U4_Un);
                }
            }
            else if (targetType == typeof(long))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I8);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I8_Un);
                }
            }
            else if (targetType == typeof(ulong))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U8);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U8_Un);
                }
            }
            else if (targetType == typeof(float))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_R4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_R_Un);
                }
            }
            else if (targetType == typeof(double))
            {
                generator.Emit(OpCodes.Conv_R8);
            }
            else if (targetType == typeof(bool))
            {
                generator.ConvertToBoolean<TSource>();
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, bool value) =>
        ↪  generator.LoadConstant(value ? 1 : 0);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, float value) =>
        ↪  generator.Emit(OpCodes.Ldc_R4, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, double value) =>
        ↪  generator.Emit(OpCodes.Ldc_R8, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, ulong value) =>
        ↪  generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
261         public static void LoadConstant(this ILGenerator generator, long value) =>
            ↪ generator.Emit(OpCodes.Ldc_I8, value);
262
263         [MethodImpl(MethodImplOptions.AggressiveInlining)]
264         public static void LoadConstant(this ILGenerator generator, uint value)
265         {
266             switch (value)
267             {
268                 case uint.MaxValue:
269                     generator.Emit(OpCodes.Ldc_I4_M1);
270                     return;
271                 case 0:
272                     generator.Emit(OpCodes.Ldc_I4_0);
273                     return;
274                 case 1:
275                     generator.Emit(OpCodes.Ldc_I4_1);
276                     return;
277                 case 2:
278                     generator.Emit(OpCodes.Ldc_I4_2);
279                     return;
280                 case 3:
281                     generator.Emit(OpCodes.Ldc_I4_3);
282                     return;
283                 case 4:
284                     generator.Emit(OpCodes.Ldc_I4_4);
285                     return;
286                 case 5:
287                     generator.Emit(OpCodes.Ldc_I4_5);
288                     return;
289                 case 6:
290                     generator.Emit(OpCodes.Ldc_I4_6);
291                     return;
292                 case 7:
293                     generator.Emit(OpCodes.Ldc_I4_7);
294                     return;
295                 case 8:
296                     generator.Emit(OpCodes.Ldc_I4_8);
297                     return;
298                 default:
299                     if (value <= sbyte.MaxValue)
300                     {
301                         generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
302                     }
303                     else
304                     {
305                         generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
306                     }
307                     return;
308             }
309         }
310
311         [MethodImpl(MethodImplOptions.AggressiveInlining)]
312         public static void LoadConstant(this ILGenerator generator, int value)
313         {
314             switch (value)
315             {
316                 case -1:
317                     generator.Emit(OpCodes.Ldc_I4_M1);
318                     return;
319                 case 0:
320                     generator.Emit(OpCodes.Ldc_I4_0);
321                     return;
322                 case 1:
323                     generator.Emit(OpCodes.Ldc_I4_1);
324                     return;
325                 case 2:
326                     generator.Emit(OpCodes.Ldc_I4_2);
327                     return;
328                 case 3:
329                     generator.Emit(OpCodes.Ldc_I4_3);
330                     return;
331                 case 4:
332                     generator.Emit(OpCodes.Ldc_I4_4);
333                     return;
334                 case 5:
335                     generator.Emit(OpCodes.Ldc_I4_5);
336                     return;
337                 case 6:
338                     generator.Emit(OpCodes.Ldc_I4_6);
339                     return;
340                 case 7:
```

```
                generator.Emit(OpCodes.Ldc_I4_7);
                return;
            case 8:
                generator.Emit(OpCodes.Ldc_I4_8);
                return;
            default:
                if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
                {
                    generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
                }
                else
                {
                    generator.Emit(OpCodes.Ldc_I4, value);
                }
                return;
        }
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void LoadConstant(this ILGenerator generator, short value) =>
    ↪  generator.LoadConstant((int)value);

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void LoadConstant(this ILGenerator generator, ushort value) =>
    ↪  generator.LoadConstant((int)value);

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void LoadConstant(this ILGenerator generator, sbyte value) =>
    ↪  generator.LoadConstant((int)value);

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void LoadConstant(this ILGenerator generator, byte value) =>
    ↪  generator.LoadConstant((int)value);

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
    ↪  LoadConstantOne(generator, typeof(TConstant));

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void LoadConstantOne(this ILGenerator generator, Type constantType)
    {
        if (constantType == typeof(float))
        {
            generator.LoadConstant(1F);
        }
        else if (constantType == typeof(double))
        {
            generator.LoadConstant(1D);
        }
        else if (constantType == typeof(long))
        {
            generator.LoadConstant(1L);
        }
        else if (constantType == typeof(ulong))
        {
            generator.LoadConstant(1UL);
        }
        else if (constantType == typeof(int))
        {
            generator.LoadConstant(1);
        }
        else if (constantType == typeof(uint))
        {
            generator.LoadConstant(1U);
        }
        else if (constantType == typeof(short))
        {
            generator.LoadConstant((short)1);
        }
        else if (constantType == typeof(ushort))
        {
            generator.LoadConstant((ushort)1);
        }
        else if (constantType == typeof(sbyte))
        {
            generator.LoadConstant((sbyte)1);
        }
        else if (constantType == typeof(byte))
```

```
            {
                generator.LoadConstant((byte)1);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
        ↪  constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, Type constantType, object
        ↪  constantValue)
        {
            constantValue = Convert.ChangeType(constantValue, constantType);
            if (constantType == typeof(float))
            {
                generator.LoadConstant((float)constantValue);
            }
            else if (constantType == typeof(double))
            {
                generator.LoadConstant((double)constantValue);
            }
            else if (constantType == typeof(long))
            {
                generator.LoadConstant((long)constantValue);
            }
            else if (constantType == typeof(ulong))
            {
                generator.LoadConstant((ulong)constantValue);
            }
            else if (constantType == typeof(int))
            {
                generator.LoadConstant((int)constantValue);
            }
            else if (constantType == typeof(uint))
            {
                generator.LoadConstant((uint)constantValue);
            }
            else if (constantType == typeof(short))
            {
                generator.LoadConstant((short)constantValue);
            }
            else if (constantType == typeof(ushort))
            {
                generator.LoadConstant((ushort)constantValue);
            }
            else if (constantType == typeof(sbyte))
            {
                generator.LoadConstant((sbyte)constantValue);
            }
            else if (constantType == typeof(byte))
            {
                generator.LoadConstant((byte)constantValue);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Increment<TValue>(this ILGenerator generator) =>
        ↪  generator.Increment(typeof(TValue));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Decrement<TValue>(this ILGenerator generator) =>
        ↪  generator.Decrement(typeof(TValue));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Increment(this ILGenerator generator, Type valueType)
        {
            generator.LoadConstantOne(valueType);
            generator.Add();
        }
```

```
488
489         [MethodImpl(MethodImplOptions.AggressiveInlining)]
490         public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
491
492         [MethodImpl(MethodImplOptions.AggressiveInlining)]
493         public static void Decrement(this ILGenerator generator, Type valueType)
494         {
495             generator.LoadConstantOne(valueType);
496             generator.Subtract();
497         }
498
499         [MethodImpl(MethodImplOptions.AggressiveInlining)]
500         public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
501
502         [MethodImpl(MethodImplOptions.AggressiveInlining)]
503         public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
504
505         [MethodImpl(MethodImplOptions.AggressiveInlining)]
506         public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
507
508         [MethodImpl(MethodImplOptions.AggressiveInlining)]
509         public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
510
511         [MethodImpl(MethodImplOptions.AggressiveInlining)]
512         public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
513
514         [MethodImpl(MethodImplOptions.AggressiveInlining)]
515         public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
516
517         [MethodImpl(MethodImplOptions.AggressiveInlining)]
518         public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
519
520         [MethodImpl(MethodImplOptions.AggressiveInlining)]
521         public static void LoadArgument(this ILGenerator generator, int argumentIndex)
522         {
523             switch (argumentIndex)
524             {
525                 case 0:
526                     generator.Emit(OpCodes.Ldarg_0);
527                     break;
528                 case 1:
529                     generator.Emit(OpCodes.Ldarg_1);
530                     break;
531                 case 2:
532                     generator.Emit(OpCodes.Ldarg_2);
533                     break;
534                 case 3:
535                     generator.Emit(OpCodes.Ldarg_3);
536                     break;
537                 default:
538                     generator.Emit(OpCodes.Ldarg, argumentIndex);
539                     break;
540             }
541         }
542
543         [MethodImpl(MethodImplOptions.AggressiveInlining)]
544         public static void LoadArguments(this ILGenerator generator, params int[]
            ↪ argumentIndices)
545         {
546             for (var i = 0; i < argumentIndices.Length; i++)
547             {
548                 generator.LoadArgument(argumentIndices[i]);
549             }
550         }
551
552         [MethodImpl(MethodImplOptions.AggressiveInlining)]
553         public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
            ↪ generator.Emit(OpCodes.Starg, argumentIndex);
554
555         [MethodImpl(MethodImplOptions.AggressiveInlining)]
556         public static void CompareGreaterThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Cgt);
557
558         [MethodImpl(MethodImplOptions.AggressiveInlining)]
559         public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Cgt_Un);
560
561         [MethodImpl(MethodImplOptions.AggressiveInlining)]
562         public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
```

```
563         {
564             if (isSigned)
565             {
566                 generator.CompareGreaterThan();
567             }
568             else
569             {
570                 generator.UnsignedCompareGreaterThan();
571             }
572         }
573 
574         [MethodImpl(MethodImplOptions.AggressiveInlining)]
575         public static void CompareLessThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Clt);
576 
577         [MethodImpl(MethodImplOptions.AggressiveInlining)]
578         public static void UnsignedCompareLessThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Clt_Un);
579 
580         [MethodImpl(MethodImplOptions.AggressiveInlining)]
581         public static void CompareLessThan(this ILGenerator generator, bool isSigned)
582         {
583             if (isSigned)
584             {
585                 generator.CompareLessThan();
586             }
587             else
588             {
589                 generator.UnsignedCompareLessThan();
590             }
591         }
592 
593         [MethodImpl(MethodImplOptions.AggressiveInlining)]
594         public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
            ↪ generator.Emit(OpCodes.Bge, label);
595 
596         [MethodImpl(MethodImplOptions.AggressiveInlining)]
597         public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
            ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
598 
599         [MethodImpl(MethodImplOptions.AggressiveInlining)]
600         public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
            ↪ Label label)
601         {
602             if (isSigned)
603             {
604                 generator.BranchIfGreaterOrEqual(label);
605             }
606             else
607             {
608                 generator.UnsignedBranchIfGreaterOrEqual(label);
609             }
610         }
611 
612         [MethodImpl(MethodImplOptions.AggressiveInlining)]
613         public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
            ↪ generator.Emit(OpCodes.Ble, label);
614 
615         [MethodImpl(MethodImplOptions.AggressiveInlining)]
616         public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
            ↪ => generator.Emit(OpCodes.Ble_Un, label);
617 
618         [MethodImpl(MethodImplOptions.AggressiveInlining)]
619         public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
            ↪ label)
620         {
621             if (isSigned)
622             {
623                 generator.BranchIfLessOrEqual(label);
624             }
625             else
626             {
627                 generator.UnsignedBranchIfLessOrEqual(label);
628             }
629         }
630 
631         [MethodImpl(MethodImplOptions.AggressiveInlining)]
632         public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
```

```
633
634         [MethodImpl(MethodImplOptions.AggressiveInlining)]
635         public static void Box(this ILGenerator generator, Type boxedType) =>
            ↪ generator.Emit(OpCodes.Box, boxedType);
636
637         [MethodImpl(MethodImplOptions.AggressiveInlining)]
638         public static void Call(this ILGenerator generator, MethodInfo method) =>
            ↪ generator.Emit(OpCodes.Call, method);
639
640         [MethodImpl(MethodImplOptions.AggressiveInlining)]
641         public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
642
643         [MethodImpl(MethodImplOptions.AggressiveInlining)]
644         public static void Unbox<TUnbox>(this ILGenerator generator) =>
            ↪ generator.Unbox(typeof(TUnbox));
645
646         [MethodImpl(MethodImplOptions.AggressiveInlining)]
647         public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
            ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
648
649         [MethodImpl(MethodImplOptions.AggressiveInlining)]
650         public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
            ↪ generator.UnboxValue(typeof(TUnbox));
651
652         [MethodImpl(MethodImplOptions.AggressiveInlining)]
653         public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
            ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
654
655         [MethodImpl(MethodImplOptions.AggressiveInlining)]
656         public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
            ↪ generator.DeclareLocal(typeof(T));
657
658         [MethodImpl(MethodImplOptions.AggressiveInlining)]
659         public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪ generator.Emit(OpCodes.Ldloc, local);
660
661         [MethodImpl(MethodImplOptions.AggressiveInlining)]
662         public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪ generator.Emit(OpCodes.Stloc, local);
663
664         [MethodImpl(MethodImplOptions.AggressiveInlining)]
665         public static void NewObject(this ILGenerator generator, Type type, params Type[]
            ↪ parameterTypes)
666         {
667             var allConstructors = type.GetConstructors(BindingFlags.Public |
                ↪ BindingFlags.NonPublic | BindingFlags.Instance
668 #if !NETSTANDARD
669                 | BindingFlags.CreateInstance
670 #endif
671                 );
672             var constructor = allConstructors.Where(c => c.GetParameters().Length ==
                ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
                ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
673             if (constructor == null)
674             {
675                 throw new InvalidOperationException("Type " + type + " must have a constructor
                    ↪ that matches parameters [" + string.Join(", ",
                    ↪ parameterTypes.AsEnumerable()) + "]");
676             }
677             generator.NewObject(constructor);
678         }
679
680         [MethodImpl(MethodImplOptions.AggressiveInlining)]
681         public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
            ↪ generator.Emit(OpCodes.Newobj, constructor);
682
683         [MethodImpl(MethodImplOptions.AggressiveInlining)]
684         public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
685
686         [MethodImpl(MethodImplOptions.AggressiveInlining)]
687         public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
            ↪ false, byte? unaligned = null)
688         {
689             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
690             {
691                 throw new ArgumentException("unaligned must be null, 1, 2, or 4");
```

```
692             }
693             if (isVolatile)
694             {
695                 generator.Emit(OpCodes.Volatile);
696             }
697             if (unaligned.HasValue)
698             {
699                 generator.Emit(OpCodes.Unaligned, unaligned.Value);
700             }
701             if (type.IsPointer)
702             {
703                 generator.Emit(OpCodes.Ldind_I);
704             }
705             else if (!type.IsValueType)
706             {
707                 generator.Emit(OpCodes.Ldind_Ref);
708             }
709             else if (type == typeof(sbyte))
710             {
711                 generator.Emit(OpCodes.Ldind_I1);
712             }
713             else if (type == typeof(bool))
714             {
715                 generator.Emit(OpCodes.Ldind_I1);
716             }
717             else if (type == typeof(byte))
718             {
719                 generator.Emit(OpCodes.Ldind_U1);
720             }
721             else if (type == typeof(short))
722             {
723                 generator.Emit(OpCodes.Ldind_I2);
724             }
725             else if (type == typeof(ushort))
726             {
727                 generator.Emit(OpCodes.Ldind_U2);
728             }
729             else if (type == typeof(char))
730             {
731                 generator.Emit(OpCodes.Ldind_U2);
732             }
733             else if (type == typeof(int))
734             {
735                 generator.Emit(OpCodes.Ldind_I4);
736             }
737             else if (type == typeof(uint))
738             {
739                 generator.Emit(OpCodes.Ldind_U4);
740             }
741             else if (type == typeof(long) || type == typeof(ulong))
742             {
743                 generator.Emit(OpCodes.Ldind_I8);
744             }
745             else if (type == typeof(float))
746             {
747                 generator.Emit(OpCodes.Ldind_R4);
748             }
749             else if (type == typeof(double))
750             {
751                 generator.Emit(OpCodes.Ldind_R8);
752             }
753             else
754             {
755                 throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
                    ↪  ", LoadObject may be more appropriate");
756             }
757         }
758
759         [MethodImpl(MethodImplOptions.AggressiveInlining)]
760         public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪  byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
761
762         [MethodImpl(MethodImplOptions.AggressiveInlining)]
763         public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
            ↪  = false, byte? unaligned = null)
764         {
765             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
766             {
```

```
                throw new ArgumentException("unaligned must be null, 1, 2, or 4");
            }
            if (isVolatile)
            {
                generator.Emit(OpCodes.Volatile);
            }
            if (unaligned.HasValue)
            {
                generator.Emit(OpCodes.Unaligned, unaligned.Value);
            }
            if (type.IsPointer)
            {
                generator.Emit(OpCodes.Stind_I);
            }
            else if (!type.IsValueType)
            {
                generator.Emit(OpCodes.Stind_Ref);
            }
            else if (type == typeof(sbyte) || type == typeof(byte))
            {
                generator.Emit(OpCodes.Stind_I1);
            }
            else if (type == typeof(short) || type == typeof(ushort))
            {
                generator.Emit(OpCodes.Stind_I2);
            }
            else if (type == typeof(int) || type == typeof(uint))
            {
                generator.Emit(OpCodes.Stind_I4);
            }
            else if (type == typeof(long) || type == typeof(ulong))
            {
                generator.Emit(OpCodes.Stind_I8);
            }
            else if (type == typeof(float))
            {
                generator.Emit(OpCodes.Stind_R4);
            }
            else if (type == typeof(double))
            {
                generator.Emit(OpCodes.Stind_R8);
            }
            else
            {
                throw new InvalidOperationException("StoreIndirect cannot be used with " + type
                ↪  + ", StoreObject may be more appropriate");
            }
        }
    }
}
```

1.7 ./Platform.Reflection/MethodInfoExtensions.cs

```
using System;
using System.Linq;
using System.Reflection;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class MethodInfoExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static byte[] GetILBytes(this MethodInfo methodInfo) =>
        ↪  methodInfo.GetMethodBody().GetILAsByteArray();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
        ↪  methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
    }
}
```

1.8 ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
```

```

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public class NotSupportedExceptionDelegateFactory<TDelegate> : IFactory<TDelegate>
        where TDelegate : Delegate
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TDelegate Create()
        {
            var @delegate = DelegateHelpers.CompileOrDefault<TDelegate>(generator =>
            {
                generator.Throw<NotSupportedException>();
            });
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
                throw new InvalidOperationException("Unable to compile stub delegate.");
            }
            return @delegate;
        }
    }
}
```

1.9 ./Platform.Reflection/NumericType.cs

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using Platform.Exceptions;

// ReSharper disable AssignmentInConditionalExpression
// ReSharper disable BuiltInTypeReferenceStyle
// ReSharper disable StaticFieldInGenericType
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class NumericType<T>
    {
        public static readonly Type Type;
        public static readonly Type UnderlyingType;
        public static readonly Type SignedVersion;
        public static readonly Type UnsignedVersion;
        public static readonly bool IsFloatPoint;
        public static readonly bool IsNumeric;
        public static readonly bool IsSigned;
        public static readonly bool CanBeNumeric;
        public static readonly bool IsNullable;
        public static readonly int BytesSize;
        public static readonly int BitsSize;
        public static readonly T MinValue;
        public static readonly T MaxValue;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static NumericType()
        {
            try
            {
                var type = typeof(T);
                var isNullable = type.IsNullable();
                var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
                var canBeNumeric = underlyingType.CanBeNumeric();
                var isNumeric = underlyingType.IsNumeric();
                var isSigned = underlyingType.IsSigned();
                var isFloatPoint = underlyingType.IsFloatPoint();
                var bytesSize = Marshal.SizeOf(underlyingType);
                var bitsSize = bytesSize * 8;
                GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
                GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
                ↪  out Type unsignedVersion);
                Type = type;
                IsNullable = isNullable;
                UnderlyingType = underlyingType;
                CanBeNumeric = canBeNumeric;
                IsNumeric = isNumeric;
                IsSigned = isSigned;
                IsFloatPoint = isFloatPoint;
                BytesSize = bytesSize;
                BitsSize = bitsSize;
                MinValue = minValue;
                MaxValue = maxValue;
```

```
                SignedVersion = signedVersion;
                UnsignedVersion = unsignedVersion;
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
        {
            if (type == typeof(bool))
            {
                minValue = (T)(object)false;
                maxValue = (T)(object)true;
            }
            else
            {
                minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
                maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
        ↪  signedVersion, out Type unsignedVersion)
        {
            if (isSigned)
            {
                signedVersion = type;
                unsignedVersion = type.GetUnsignedVersionOrNull();
            }
            else
            {
                signedVersion = type.GetSignedVersionOrNull();
                unsignedVersion = type;
            }
        }
    }
}
```

1.10 ./Platform.Reflection/PropertyInfoExtensions.cs

```
using System.Reflection;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class PropertyInfoExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticValue<T>(this PropertyInfo fieldInfo) =>
        ↪  (T)fieldInfo.GetValue(null);
    }
}
```

1.11 ./Platform.Reflection/TypeBuilderExtensions.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;

namespace Platform.Reflection
{
    public static class TypeBuilderExtensions
    {
        public static readonly MethodAttributes DefaultStaticMethodAttributes =
        ↪  MethodAttributes.Public | MethodAttributes.Static;
        public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes =
        ↪  MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
        ↪  MethodAttributes.HideBySig;
        public static readonly MethodImplAttributes DefaultMethodImplAttributes =
        ↪  MethodImplAttributes.IL | MethodImplAttributes.Managed |
        ↪  MethodImplAttributes.AggressiveInlining;

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName,
        ↪  MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes,
        ↪  Action<ILGenerator> emitCode)
        {
            typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[]
            ↪  parameterTypes);
            EmitMethod(type, methodName, methodAttributes, methodImplAttributes, returnType,
            ↪  parameterTypes, emitCode);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
        ↪  methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[]
        ↪  parameterTypes, Action<ILGenerator> emitCode)
        {
            MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
            ↪  parameterTypes);
            method.SetImplementationFlags(methodImplAttributes);
            var generator = method.GetILGenerator();
            emitCode(generator);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName,
        ↪  Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
        ↪  DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitFinalVirtualMethod<TDelegate>(this TypeBuilder type, string
        ↪  methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
        ↪  DefaultFinalVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
    }
}
```

1.12 ./Platform.Reflection/TypeExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Runtime.CompilerServices;
using Platform.Collections;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class TypeExtensions
    {
        static public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
        ↪  BindingFlags.NonPublic | BindingFlags.Static;
        static public readonly string DefaultDelegateMethodName = "Invoke";

        static private readonly HashSet<Type> _canBeNumericTypes;
        static private readonly HashSet<Type> _isNumericTypes;
        static private readonly HashSet<Type> _isSignedTypes;
        static private readonly HashSet<Type> _isFloatPointTypes;
        static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
        static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static TypeExtensions()
        {
            _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
            ↪  typeof(DateTime), typeof(TimeSpan) };
            _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
            ↪  typeof(ulong) };
            _canBeNumericTypes.UnionWith(_isNumericTypes);
            _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
            ↪  typeof(long) };
            _canBeNumericTypes.UnionWith(_isSignedTypes);
            _isNumericTypes.UnionWith(_isSignedTypes);
            _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
            ↪  typeof(float) };
            _canBeNumericTypes.UnionWith(_isFloatPointTypes);
            _isNumericTypes.UnionWith(_isFloatPointTypes);
            _isSignedTypes.UnionWith(_isFloatPointTypes);
            _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
```

```
            {
                { typeof(sbyte), typeof(byte) },
                { typeof(short), typeof(ushort) },
                { typeof(int), typeof(uint) },
                { typeof(long), typeof(ulong) },
            };
            _signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
            {
                { typeof(byte), typeof(sbyte)},
                { typeof(ushort), typeof(short) },
                { typeof(uint), typeof(int) },
                { typeof(ulong), typeof(long) },
            };
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticFieldValue<T>(this Type type, string name) =>
        ↪  type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticPropertyValue<T>(this Type type, string name) =>
        ↪  type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
        ↪  genericParameterTypes, Type[] argumentTypes)
        {
            var methods = from m in type.GetMethods()
                          where m.Name == name
                             && m.IsGenericMethodDefinition
                          let typeParams = m.GetGenericArguments()
                          let normalParams = m.GetParameters().Select(x => x.ParameterType)
                          where typeParams.SequenceEqual(genericParameterTypes)
                             && normalParams.SequenceEqual(argumentTypes)
                          select m;
            var method = methods.Single();
            return method;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetBaseType(this Type type) => type.BaseType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Assembly GetAssembly(this Type type) => type.Assembly;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsSubclassOf(this Type type, Type superClass) =>
        ↪  type.IsSubclassOf(superClass);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsValueType(this Type type) => type.IsValueType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsGeneric(this Type type) => type.IsGenericType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
        ↪  type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetUnsignedVersionOrNull(this Type signedType) =>
        ↪  _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetSignedVersionOrNull(this Type unsignedType) =>
        ↪  _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetDelegateReturnType(this Type delegateType) =>
        ↪ delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
        ↪ delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetDelegateCharacteristics(this Type delegateType, out Type
        ↪ returnType, out Type[] parameterTypes)
        {
            var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
            returnType = invoke.ReturnType;
            parameterTypes = invoke.GetParameterTypes();
        }
    }
}
```

1.13 ./Platform.Reflection/Types.cs

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Runtime.CompilerServices;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public abstract class Types
    {
        public static ReadOnlyCollection<Type> Collection { get; } = new
        ↪ ReadOnlyCollection<Type>(System.Array.Empty<Type>());
        public static Type[] Array => Collection.ToArray();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected ReadOnlyCollection<Type> ToReadOnlyCollection()
        {
            var types = GetType().GetGenericArguments();
            var result = new List<Type>();
            AppendTypes(result, types);
            return new ReadOnlyCollection<Type>(result);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void AppendTypes(List<Type> container, IList<Type> types)
        {
            for (var i = 0; i < types.Count; i++)
            {
                var element = types[i];
                if (element != typeof(Types))
                {
                    if (element.IsSubclassOf(typeof(Types)))
                    {
                        AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection
                        ↪ <Type>>(nameof(Types<object>.Collection)));
                    }
                    else
                    {
                        container.Add(element);
                    }
                }
            }
        }
    }
}
```

1.14 ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs

```
using System;
using System.Collections.ObjectModel;
```

```
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4, T5, T6, T7>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.15 ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2, T3, T4, T5, T6> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4, T5, T6>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.16 ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2, T3, T4, T5> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4, T5>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.17 ./Platform.Reflection/Types[T1, T2, T3, T4].cs

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2, T3, T4> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.18 ./Platform.Reflection/Types[T1, T2, T3].cs

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
```

```
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2, T3> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2,
        ↪  T3>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.19 ./Platform.Reflection/Types[T1, T2].cs

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1,
        ↪  T2>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.20 ./Platform.Reflection/Types[T].cs

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new
        ↪  Types<T>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.21 ./Platform.Reflection.Tests/CodeGenerationTests.cs

```
using System;
using Xunit;

namespace Platform.Reflection.Tests
{
    public class CodeGenerationTests
    {
        [Fact]
        public void EmptyActionCompilationTest()
        {
            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                generator.Return();
            });
            compiledAction();
        }

        [Fact]
        public void FailedActionCompilationTest()
        {
            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                throw new NotImplementedException();
            });
            Assert.Throws<NotSupportedException>(compiledAction);
        }

```

```
        [Fact]
        public void ConstantLoadingTest()
        {
            CheckConstantLoading<byte>(8);
            CheckConstantLoading<uint>(8);
            CheckConstantLoading<ushort>(8);
            CheckConstantLoading<ulong>(8);
        }

        private void CheckConstantLoading<T>(T value)
        {
            var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
            {
                generator.LoadConstant(value);
                generator.Return();
            });
            Assert.Equal(value, compiledFunction());
        }

        [Fact]
        public void UnsignedIntegersConversionWithSignExtensionTest()
        {
            object[] withSignExtension = new object[]
            {
                CompileUncheckedConverter<byte, sbyte>(extendSign: true)(128),
                CompileUncheckedConverter<byte, short>(extendSign: true)(128),
                CompileUncheckedConverter<ushort, short>(extendSign: true)(32768),
                CompileUncheckedConverter<byte, int>(extendSign: true)(128),
                CompileUncheckedConverter<ushort, int>(extendSign: true)(32768),
                CompileUncheckedConverter<uint, int>(extendSign: true)(2147483648),
                CompileUncheckedConverter<byte, long>(extendSign: true)(128),
                CompileUncheckedConverter<ushort, long>(extendSign: true)(32768),
                CompileUncheckedConverter<uint, long>(extendSign: true)(2147483648),
                CompileUncheckedConverter<ulong, long>(extendSign: true)(9223372036854775808)
            };
            object[] withoutSignExtension = new object[]
            {
                CompileUncheckedConverter<byte, sbyte>(extendSign: false)(128),
                CompileUncheckedConverter<byte, short>(extendSign: false)(128),
                CompileUncheckedConverter<ushort, short>(extendSign: false)(32768),
                CompileUncheckedConverter<byte, int>(extendSign: false)(128),
                CompileUncheckedConverter<ushort, int>(extendSign: false)(32768),
                CompileUncheckedConverter<uint, int>(extendSign: false)(2147483648),
                CompileUncheckedConverter<byte, long>(extendSign: false)(128),
                CompileUncheckedConverter<ushort, long>(extendSign: false)(32768),
                CompileUncheckedConverter<uint, long>(extendSign: false)(2147483648),
                CompileUncheckedConverter<ulong, long>(extendSign: false)(9223372036854775808)
            };
            var i = 0;
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
        }

        [Fact]
        public void SignedIntegersConversionOfMinusOneWithSignExtensionTest()
        {
            object[] withSignExtension = new object[]
            {
                CompileUncheckedConverter<sbyte, byte>(extendSign: true)(-1),
                CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(-1),
                CompileUncheckedConverter<short, ushort>(extendSign: true)(-1),
                CompileUncheckedConverter<sbyte, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<short, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<int, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<short, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<int, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<long, ulong>(extendSign: true)(-1)
            };
            object[] withoutSignExtension = new object[]
```

```
106             {
107                 CompileUncheckedConverter<sbyte, byte>(extendSign: false)(-1),
108                 CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(-1),
109                 CompileUncheckedConverter<short, ushort>(extendSign: false)(-1),
110                 CompileUncheckedConverter<sbyte, uint>(extendSign: false)(-1),
111                 CompileUncheckedConverter<short, uint>(extendSign: false)(-1),
112                 CompileUncheckedConverter<int, uint>(extendSign: false)(-1),
113                 CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(-1),
114                 CompileUncheckedConverter<short, ulong>(extendSign: false)(-1),
115                 CompileUncheckedConverter<int, ulong>(extendSign: false)(-1),
116                 CompileUncheckedConverter<long, ulong>(extendSign: false)(-1)
117             };
118             var i = 0;
119             Assert.Equal((byte)255, (byte)withSignExtension[i]);
120             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
121             Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
122             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
123             Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
124             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
125             Assert.Equal(4294967295, withSignExtension[i]);
126             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
127             Assert.Equal(4294967295, withSignExtension[i]);
128             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
129             Assert.Equal(4294967295, withSignExtension[i]);
130             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
131             Assert.Equal(18446744073709551615, withSignExtension[i]);
132             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
133             Assert.Equal(18446744073709551615, withSignExtension[i]);
134             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
135             Assert.Equal(18446744073709551615, withSignExtension[i]);
136             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
137             Assert.Equal(18446744073709551615, withSignExtension[i]);
138             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
139         }
140 
141         [Fact]
142         public void SignedIntegersConversionOfTwoWithSignExtensionTest()
143         {
144             object[] withSignExtension = new object[]
145             {
146                 CompileUncheckedConverter<sbyte, byte>(extendSign: true)(2),
147                 CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(2),
148                 CompileUncheckedConverter<short, ushort>(extendSign: true)(2),
149                 CompileUncheckedConverter<sbyte, uint>(extendSign: true)(2),
150                 CompileUncheckedConverter<short, uint>(extendSign: true)(2),
151                 CompileUncheckedConverter<int, uint>(extendSign: true)(2),
152                 CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(2),
153                 CompileUncheckedConverter<short, ulong>(extendSign: true)(2),
154                 CompileUncheckedConverter<int, ulong>(extendSign: true)(2),
155                 CompileUncheckedConverter<long, ulong>(extendSign: true)(2)
156             };
157             object[] withoutSignExtension = new object[]
158             {
159                 CompileUncheckedConverter<sbyte, byte>(extendSign: false)(2),
160                 CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(2),
161                 CompileUncheckedConverter<short, ushort>(extendSign: false)(2),
162                 CompileUncheckedConverter<sbyte, uint>(extendSign: false)(2),
163                 CompileUncheckedConverter<short, uint>(extendSign: false)(2),
164                 CompileUncheckedConverter<int, uint>(extendSign: false)(2),
165                 CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(2),
166                 CompileUncheckedConverter<short, ulong>(extendSign: false)(2),
167                 CompileUncheckedConverter<int, ulong>(extendSign: false)(2),
168                 CompileUncheckedConverter<long, ulong>(extendSign: false)(2)
169             };
170             for (var i = 0; i < withSignExtension.Length; i++)
171             {
172                 Assert.Equal(2UL, Convert.ToUInt64(withSignExtension[i]));
173                 Assert.Equal(withSignExtension[i], withoutSignExtension[i]);
174             }
175         }
176 
177         private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
            ↪ TTarget>(bool extendSign)
178         {
179             return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
180             {
181                 generator.LoadArgument(0);
182                 generator.UncheckedConvert<TSource, TTarget>(extendSign);
```

```
                generator.Return();
            });
        }
    }
}
```

1.22 ./Platform.Reflection.Tests/GetILBytesMethodTests.cs

```
using System;
using System.Reflection;
using Xunit;
using Platform.Collections;
using Platform.Collections.Lists;

namespace Platform.Reflection.Tests
{
    public static class GetILBytesMethodTests
    {
        [Fact]
        public static void ILBytesForDelegateAreAvailableTest()
        {
            var function = new Func<object, int>(argument => 0);
            var bytes = function.GetMethodInfo().GetILBytes();
            Assert.False(bytes.IsNullOrEmpty());
        }

        [Fact]
        public static void ILBytesForDifferentDelegatesAreTheSameTest()
        {
            var firstFunction = new Func<object, int>(argument => 0);
            var secondFunction = new Func<object, int>(argument => 0);
            Assert.False(firstFunction == secondFunction);
            var firstFunctionBytes = firstFunction.GetMethodInfo().GetILBytes();
            Assert.False(firstFunctionBytes.IsNullOrEmpty());
            var secondFunctionBytes = secondFunction.GetMethodInfo().GetILBytes();
            Assert.False(secondFunctionBytes.IsNullOrEmpty());
            Assert.True(firstFunctionBytes.EqualTo(secondFunctionBytes));
        }
    }
}
```

1.23 ./Platform.Reflection.Tests/NumericTypeTests.cs

```
using Xunit;

namespace Platform.Reflection.Tests
{
    public class NumericTypeTests
    {
        [Fact]
        public void UInt64IsNumericTest()
        {
            Assert.True(NumericType<ulong>.IsNumeric);
        }
    }
}
```

Index