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, TTarget>();
            }
            else
            {
#if NET471
                if (sourceType == typeof(byte) || sourceType == typeof(ushort))
                {
                    if (targetType == typeof(long))
                    {
                        if (extendSign)
                        {
                            generator.Emit(OpCodes.Conv_I8);
                        }
                        else
                        {
                            generator.Emit(OpCodes.Conv_U8);
                        }
                    }
                }
                if (sourceType == typeof(uint) && targetType == typeof(long) && extendSign)
                {
                    generator.Emit(OpCodes.Conv_I8);
                }
#endif
                if (sourceType == typeof(uint) && targetType == typeof(long) && !extendSign)
                {
                    generator.Emit(OpCodes.Conv_U8);
                }
            }
            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);
            }
        }

        private static void ConvertToInteger<TSource, TTarget>(this ILGenerator generator)
        {
            var targetType = typeof(TTarget);
            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))
            {
                generator.Emit(OpCodes.Conv_I8);
            }
            else if (targetType == typeof(ulong))
            {
                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
            {
                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)]
        public static void LoadConstant(this ILGenerator generator, long value) =>
        ↪  generator.Emit(OpCodes.Ldc_I8, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, uint value)
        {
            switch (value)
```

```
            {
                case uint.MaxValue:
                    generator.Emit(OpCodes.Ldc_I4_M1);
                    return;
                case 0:
                    generator.Emit(OpCodes.Ldc_I4_0);
                    return;
                case 1:
                    generator.Emit(OpCodes.Ldc_I4_1);
                    return;
                case 2:
                    generator.Emit(OpCodes.Ldc_I4_2);
                    return;
                case 3:
                    generator.Emit(OpCodes.Ldc_I4_3);
                    return;
                case 4:
                    generator.Emit(OpCodes.Ldc_I4_4);
                    return;
                case 5:
                    generator.Emit(OpCodes.Ldc_I4_5);
                    return;
                case 6:
                    generator.Emit(OpCodes.Ldc_I4_6);
                    return;
                case 7:
                    generator.Emit(OpCodes.Ldc_I4_7);
                    return;
                case 8:
                    generator.Emit(OpCodes.Ldc_I4_8);
                    return;
                default:
                    if (value <= sbyte.MaxValue)
                    {
                        generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
                    }
                    else
                    {
                        generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
                    }
                    return;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, int value)
        {
            switch (value)
            {
                case -1:
                    generator.Emit(OpCodes.Ldc_I4_M1);
                    return;
                case 0:
                    generator.Emit(OpCodes.Ldc_I4_0);
                    return;
                case 1:
                    generator.Emit(OpCodes.Ldc_I4_1);
                    return;
                case 2:
                    generator.Emit(OpCodes.Ldc_I4_2);
                    return;
                case 3:
                    generator.Emit(OpCodes.Ldc_I4_3);
                    return;
                case 4:
                    generator.Emit(OpCodes.Ldc_I4_4);
                    return;
                case 5:
                    generator.Emit(OpCodes.Ldc_I4_5);
                    return;
                case 6:
                    generator.Emit(OpCodes.Ldc_I4_6);
                    return;
                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)
                    {
```

```
343                         generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
344                     }
345                     else
346                     {
347                         generator.Emit(OpCodes.Ldc_I4, value);
348                     }
349                     return;
350                 }
351             }
352 
353             [MethodImpl(MethodImplOptions.AggressiveInlining)]
354             public static void LoadConstant(this ILGenerator generator, short value) =>
                ↪  generator.LoadConstant((int)value);
355 
356             [MethodImpl(MethodImplOptions.AggressiveInlining)]
357             public static void LoadConstant(this ILGenerator generator, ushort value) =>
                ↪  generator.LoadConstant((int)value);
358 
359             [MethodImpl(MethodImplOptions.AggressiveInlining)]
360             public static void LoadConstant(this ILGenerator generator, sbyte value) =>
                ↪  generator.LoadConstant((int)value);
361 
362             [MethodImpl(MethodImplOptions.AggressiveInlining)]
363             public static void LoadConstant(this ILGenerator generator, byte value) =>
                ↪  generator.LoadConstant((int)value);
364 
365             [MethodImpl(MethodImplOptions.AggressiveInlining)]
366             public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
                ↪  LoadConstantOne(generator, typeof(TConstant));
367 
368             [MethodImpl(MethodImplOptions.AggressiveInlining)]
369             public static void LoadConstantOne(this ILGenerator generator, Type constantType)
370             {
371                 if (constantType == typeof(float))
372                 {
373                     generator.LoadConstant(1F);
374                 }
375                 else if (constantType == typeof(double))
376                 {
377                     generator.LoadConstant(1D);
378                 }
379                 else if (constantType == typeof(long))
380                 {
381                     generator.LoadConstant(1L);
382                 }
383                 else if (constantType == typeof(ulong))
384                 {
385                     generator.LoadConstant(1UL);
386                 }
387                 else if (constantType == typeof(int))
388                 {
389                     generator.LoadConstant(1);
390                 }
391                 else if (constantType == typeof(uint))
392                 {
393                     generator.LoadConstant(1U);
394                 }
395                 else if (constantType == typeof(short))
396                 {
397                     generator.LoadConstant((short)1);
398                 }
399                 else if (constantType == typeof(ushort))
400                 {
401                     generator.LoadConstant((ushort)1);
402                 }
403                 else if (constantType == typeof(sbyte))
404                 {
405                     generator.LoadConstant((sbyte)1);
406                 }
407                 else if (constantType == typeof(byte))
408                 {
409                     generator.LoadConstant((byte)1);
410                 }
411                 else
412                 {
413                     throw new NotSupportedException();
414                 }
415             }
```

```
416
417         [MethodImpl(MethodImplOptions.AggressiveInlining)]
418         public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
            ↪  constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
419
420         [MethodImpl(MethodImplOptions.AggressiveInlining)]
421         public static void LoadConstant(this ILGenerator generator, Type constantType, object
            ↪  constantValue)
422         {
423             constantValue = Convert.ChangeType(constantValue, constantType);
424             if (constantType == typeof(float))
425             {
426                 generator.LoadConstant((float)constantValue);
427             }
428             else if (constantType == typeof(double))
429             {
430                 generator.LoadConstant((double)constantValue);
431             }
432             else if (constantType == typeof(long))
433             {
434                 generator.LoadConstant((long)constantValue);
435             }
436             else if (constantType == typeof(ulong))
437             {
438                 generator.LoadConstant((ulong)constantValue);
439             }
440             else if (constantType == typeof(int))
441             {
442                 generator.LoadConstant((int)constantValue);
443             }
444             else if (constantType == typeof(uint))
445             {
446                 generator.LoadConstant((uint)constantValue);
447             }
448             else if (constantType == typeof(short))
449             {
450                 generator.LoadConstant((short)constantValue);
451             }
452             else if (constantType == typeof(ushort))
453             {
454                 generator.LoadConstant((ushort)constantValue);
455             }
456             else if (constantType == typeof(sbyte))
457             {
458                 generator.LoadConstant((sbyte)constantValue);
459             }
460             else if (constantType == typeof(byte))
461             {
462                 generator.LoadConstant((byte)constantValue);
463             }
464             else
465             {
466                 throw new NotSupportedException();
467             }
468         }
469
470         [MethodImpl(MethodImplOptions.AggressiveInlining)]
471         public static void Increment<TValue>(this ILGenerator generator) =>
            ↪  generator.Increment(typeof(TValue));
472
473         [MethodImpl(MethodImplOptions.AggressiveInlining)]
474         public static void Decrement<TValue>(this ILGenerator generator) =>
            ↪  generator.Decrement(typeof(TValue));
475
476         [MethodImpl(MethodImplOptions.AggressiveInlining)]
477         public static void Increment(this ILGenerator generator, Type valueType)
478         {
479             generator.LoadConstantOne(valueType);
480             generator.Add();
481         }
482
483         [MethodImpl(MethodImplOptions.AggressiveInlining)]
484         public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
485
486         [MethodImpl(MethodImplOptions.AggressiveInlining)]
487         public static void Decrement(this ILGenerator generator, Type valueType)
488         {
489             generator.LoadConstantOne(valueType);
```

```
            generator.Subtract();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadArgument(this ILGenerator generator, int argumentIndex)
        {
            switch (argumentIndex)
            {
                case 0:
                    generator.Emit(OpCodes.Ldarg_0);
                    break;
                case 1:
                    generator.Emit(OpCodes.Ldarg_1);
                    break;
                case 2:
                    generator.Emit(OpCodes.Ldarg_2);
                    break;
                case 3:
                    generator.Emit(OpCodes.Ldarg_3);
                    break;
                default:
                    generator.Emit(OpCodes.Ldarg, argumentIndex);
                    break;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadArguments(this ILGenerator generator, params int[]
        ↪  argumentIndices)
        {
            for (var i = 0; i < argumentIndices.Length; i++)
            {
                generator.LoadArgument(argumentIndices[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
        ↪  generator.Emit(OpCodes.Starg, argumentIndex);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareGreaterThan(this ILGenerator generator) =>
        ↪  generator.Emit(OpCodes.Cgt);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
        ↪  generator.Emit(OpCodes.Cgt_Un);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
        {
            if (isSigned)
            {
                generator.CompareGreaterThan();
            }
            else
            {
                generator.UnsignedCompareGreaterThan();
```

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

```
633
634         [MethodImpl(MethodImplOptions.AggressiveInlining)]
635         public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
636
637         [MethodImpl(MethodImplOptions.AggressiveInlining)]
638         public static void Unbox<TUnbox>(this ILGenerator generator) =>
            ↪  generator.Unbox(typeof(TUnbox));
639
640         [MethodImpl(MethodImplOptions.AggressiveInlining)]
641         public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
            ↪  generator.Emit(OpCodes.Unbox, typeToUnbox);
642
643         [MethodImpl(MethodImplOptions.AggressiveInlining)]
644         public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
            ↪  generator.UnboxValue(typeof(TUnbox));
645
646         [MethodImpl(MethodImplOptions.AggressiveInlining)]
647         public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
            ↪  generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
648
649         [MethodImpl(MethodImplOptions.AggressiveInlining)]
650         public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
            ↪  generator.DeclareLocal(typeof(T));
651
652         [MethodImpl(MethodImplOptions.AggressiveInlining)]
653         public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪  generator.Emit(OpCodes.Ldloc, local);
654
655         [MethodImpl(MethodImplOptions.AggressiveInlining)]
656         public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪  generator.Emit(OpCodes.Stloc, local);
657
658         [MethodImpl(MethodImplOptions.AggressiveInlining)]
659         public static void NewObject(this ILGenerator generator, Type type, params Type[]
            ↪  parameterTypes)
660         {
661             var allConstructors = type.GetConstructors(BindingFlags.Public |
                ↪  BindingFlags.NonPublic | BindingFlags.Instance
662 #if !NETSTANDARD
663                 | BindingFlags.CreateInstance
664 #endif
665                 );
666             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();
667             if (constructor == null)
668             {
669                 throw new InvalidOperationException("Type " + type + " must have a constructor
                    ↪  that matches parameters [" + string.Join(", ",
                    ↪  parameterTypes.AsEnumerable()) + "]");
670             }
671             generator.NewObject(constructor);
672         }
673
674         [MethodImpl(MethodImplOptions.AggressiveInlining)]
675         public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
            ↪  generator.Emit(OpCodes.Newobj, constructor);
676
677         [MethodImpl(MethodImplOptions.AggressiveInlining)]
678         public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪  byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
679
680         [MethodImpl(MethodImplOptions.AggressiveInlining)]
681         public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
            ↪  false, byte? unaligned = null)
682         {
683             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
684             {
685                 throw new ArgumentException("unaligned must be null, 1, 2, or 4");
686             }
687             if (isVolatile)
688             {
689                 generator.Emit(OpCodes.Volatile);
690             }
691             if (unaligned.HasValue)
692             {
693                 generator.Emit(OpCodes.Unaligned, unaligned.Value);
```

```
694             }
695             if (type.IsPointer)
696             {
697                 generator.Emit(OpCodes.Ldind_I);
698             }
699             else if (!type.IsValueType)
700             {
701                 generator.Emit(OpCodes.Ldind_Ref);
702             }
703             else if (type == typeof(sbyte))
704             {
705                 generator.Emit(OpCodes.Ldind_I1);
706             }
707             else if (type == typeof(bool))
708             {
709                 generator.Emit(OpCodes.Ldind_I1);
710             }
711             else if (type == typeof(byte))
712             {
713                 generator.Emit(OpCodes.Ldind_U1);
714             }
715             else if (type == typeof(short))
716             {
717                 generator.Emit(OpCodes.Ldind_I2);
718             }
719             else if (type == typeof(ushort))
720             {
721                 generator.Emit(OpCodes.Ldind_U2);
722             }
723             else if (type == typeof(char))
724             {
725                 generator.Emit(OpCodes.Ldind_U2);
726             }
727             else if (type == typeof(int))
728             {
729                 generator.Emit(OpCodes.Ldind_I4);
730             }
731             else if (type == typeof(uint))
732             {
733                 generator.Emit(OpCodes.Ldind_U4);
734             }
735             else if (type == typeof(long) || type == typeof(ulong))
736             {
737                 generator.Emit(OpCodes.Ldind_I8);
738             }
739             else if (type == typeof(float))
740             {
741                 generator.Emit(OpCodes.Ldind_R4);
742             }
743             else if (type == typeof(double))
744             {
745                 generator.Emit(OpCodes.Ldind_R8);
746             }
747             else
748             {
749                 throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
                    ↪  ", LoadObject may be more appropriate");
750             }
751         }
752 
753         [MethodImpl(MethodImplOptions.AggressiveInlining)]
754         public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪  byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
755 
756         [MethodImpl(MethodImplOptions.AggressiveInlining)]
757         public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
            ↪  = false, byte? unaligned = null)
758         {
759             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
760             {
761                 throw new ArgumentException("unaligned must be null, 1, 2, or 4");
762             }
763             if (isVolatile)
764             {
765                 generator.Emit(OpCodes.Volatile);
766             }
767             if (unaligned.HasValue)
768             {
```

```
                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();
            }
        }

```

```
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
67          {
68              if (type == typeof(bool))
69              {
70                  minValue = (T)(object)false;
71                  maxValue = (T)(object)true;
72              }
73              else
74              {
75                  minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
76                  maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
77              }
78          }
79
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
            ↪  signedVersion, out Type unsignedVersion)
82          {
83              if (isSigned)
84              {
85                  signedVersion = type;
86                  unsignedVersion = type.GetUnsignedVersionOrNull();
87              }
88              else
89              {
90                  signedVersion = type.GetSignedVersionOrNull();
91                  unsignedVersion = type;
92              }
93          }
94      }
95  }
```

1.10 ./Platform.Reflection/PropertyInfoExtensions.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 PropertyInfoExtensions
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public static T GetStaticValue<T>(this PropertyInfo fieldInfo) =>
            ↪  (T)fieldInfo.GetValue(null);
12      }
13  }
```

1.11 ./Platform.Reflection/TypeBuilderExtensions.cs

```
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3   using System;
4   using System.Reflection;
5   using System.Reflection.Emit;
6   using System.Runtime.CompilerServices;
7
8   namespace Platform.Reflection
9   {
10      public static class TypeBuilderExtensions
11      {
12          public static readonly MethodAttributes DefaultStaticMethodAttributes =
            ↪  MethodAttributes.Public | MethodAttributes.Static;
13          public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes =
            ↪  MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
            ↪  MethodAttributes.HideBySig;
14          public static readonly MethodImplAttributes DefaultMethodImplAttributes =
            ↪  MethodImplAttributes.IL | MethodImplAttributes.Managed |
            ↪  MethodImplAttributes.AggressiveInlining;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName,
            ↪  MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes,
            ↪  Action<ILGenerator> emitCode)
18          {
19              typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[]
                ↪  parameterTypes);
20              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;

```

```
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
            ↪ delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();
121
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         public static void GetDelegateCharacteristics(this Type delegateType, out Type
            ↪ returnType, out Type[] parameterTypes)
124         {
125             var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
126             returnType = invoke.ReturnType;
127             parameterTypes = invoke.GetParameterTypes();
128         }
129     }
130 }
```

1.13 ./Platform.Reflection/Types.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Collections.ObjectModel;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8  #pragma warning disable CA1819 // Properties should not return arrays
9
10 namespace Platform.Reflection
11 {
12     public abstract class Types
13     {
14         public static ReadOnlyCollection<Type> Collection { get; } = new
           ↪ ReadOnlyCollection<Type>(System.Array.Empty<Type>());
15         public static Type[] Array => Collection.ToArray();
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected ReadOnlyCollection<Type> ToReadOnlyCollection()
19         {
20             var types = GetType().GetGenericArguments();
21             var result = new List<Type>();
22             AppendTypes(result, types);
23             return new ReadOnlyCollection<Type>(result);
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         private static void AppendTypes(List<Type> container, IList<Type> types)
28         {
29             for (var i = 0; i < types.Count; i++)
30             {
31                 var element = types[i];
32                 if (element != typeof(Types))
33                 {
34                     if (element.IsSubclassOf(typeof(Types)))
35                     {
36                         AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection
                            ↪ <Type>>(nameof(Types<object>.Collection)));
37                     }
38                     else
39                     {
40                         container.Add(element);
41                     }
42                 }
43             }
44         }
45     }
46 }
```

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

```
1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  #pragma warning disable CA1819 // Properties should not return arrays
7
8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
           ↪ T4, T5, T6, T7>().ToReadOnlyCollection();
```

```
13          public new static Type[] Array => Collection.ToArray();
14          private Types() { }
15      }
16  }
```

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

```
1   using System;
2   using System.Collections.ObjectModel;
3   using Platform.Collections.Lists;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6   #pragma warning disable CA1819 // Properties should not return arrays
7
8   namespace Platform.Reflection
9   {
10      public class Types<T1, T2, T3, T4, T5, T6> : Types
11      {
12          public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
            ↪  T4, T5, T6>().ToReadOnlyCollection();
13          public new static Type[] Array => Collection.ToArray();
14          private Types() { }
15      }
16  }
```

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

```
1   using System;
2   using System.Collections.ObjectModel;
3   using Platform.Collections.Lists;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6   #pragma warning disable CA1819 // Properties should not return arrays
7
8   namespace Platform.Reflection
9   {
10      public class Types<T1, T2, T3, T4, T5> : Types
11      {
12          public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
            ↪  T4, T5>().ToReadOnlyCollection();
13          public new static Type[] Array => Collection.ToArray();
14          private Types() { }
15      }
16  }
```

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

```
1   using System;
2   using System.Collections.ObjectModel;
3   using Platform.Collections.Lists;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6   #pragma warning disable CA1819 // Properties should not return arrays
7
8   namespace Platform.Reflection
9   {
10      public class Types<T1, T2, T3, T4> : Types
11      {
12          public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
            ↪  T4>().ToReadOnlyCollection();
13          public new static Type[] Array => Collection.ToArray();
14          private Types() { }
15      }
16  }
```

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

```
1   using System;
2   using System.Collections.ObjectModel;
3   using Platform.Collections.Lists;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6   #pragma warning disable CA1819 // Properties should not return arrays
7
8   namespace Platform.Reflection
9   {
10      public class Types<T1, T2, T3> : Types
11      {
12          public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2,
            ↪  T3>().ToReadOnlyCollection();
13          public new static Type[] Array => Collection.ToArray();
14          private Types() { }
15      }
16  }
```

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 ConversionWithSignExtensionTest()
        {
            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++]);
        }

        private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
        ↪  TTarget>(bool extendSign)
        {
            return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
            {
                generator.LoadArgument(0);
                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