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>.BitsLength > NumericType<TTarget>.BitsLength)
            {
                if (targetType == typeof(short))
                {
                    generator.Emit(OpCodes.Conv_I2);
                }
                else if (targetType == typeof(ushort))
                {
                    generator.Emit(OpCodes.Conv_U2);
                }
                else if (targetType == typeof(sbyte))
                {
                    generator.Emit(OpCodes.Conv_I1);
                }
                else if (targetType == typeof(byte))
                {
                    generator.Emit(OpCodes.Conv_U1);
                }
                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);
                }
            }
            else
            {
                if (!extendSign)
                {
                    if (sourceType == typeof(uint) && targetType == typeof(long))
                    {
                        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);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
        {
            var type = typeof(TTarget);
            if (type == typeof(short))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2);
```

```
110                 }
111                 else
112                 {
113                     generator.Emit(OpCodes.Conv_Ovf_I2_Un);
114                 }
115             }
116             else if (type == typeof(ushort))
117             {
118                 if (NumericType<TSource>.IsSigned)
119                 {
120                     generator.Emit(OpCodes.Conv_Ovf_U2);
121                 }
122                 else
123                 {
124                     generator.Emit(OpCodes.Conv_Ovf_U2_Un);
125                 }
126             }
127             else if (type == typeof(sbyte))
128             {
129                 if (NumericType<TSource>.IsSigned)
130                 {
131                     generator.Emit(OpCodes.Conv_Ovf_I1);
132                 }
133                 else
134                 {
135                     generator.Emit(OpCodes.Conv_Ovf_I1_Un);
136                 }
137             }
138             else if (type == typeof(byte))
139             {
140                 if (NumericType<TSource>.IsSigned)
141                 {
142                     generator.Emit(OpCodes.Conv_Ovf_U1);
143                 }
144                 else
145                 {
146                     generator.Emit(OpCodes.Conv_Ovf_U1_Un);
147                 }
148             }
149             else if (type == typeof(int))
150             {
151                 if (NumericType<TSource>.IsSigned)
152                 {
153                     generator.Emit(OpCodes.Conv_Ovf_I4);
154                 }
155                 else
156                 {
157                     generator.Emit(OpCodes.Conv_Ovf_I4_Un);
158                 }
159             }
160             else if (type == typeof(uint))
161             {
162                 if (NumericType<TSource>.IsSigned)
163                 {
164                     generator.Emit(OpCodes.Conv_Ovf_U4);
165                 }
166                 else
167                 {
168                     generator.Emit(OpCodes.Conv_Ovf_U4_Un);
169                 }
170             }
171             else if (type == typeof(long))
172             {
173                 if (NumericType<TSource>.IsSigned)
174                 {
175                     generator.Emit(OpCodes.Conv_Ovf_I8);
176                 }
177                 else
178                 {
179                     generator.Emit(OpCodes.Conv_Ovf_I8_Un);
180                 }
181             }
182             else if (type == typeof(ulong))
183             {
184                 if (NumericType<TSource>.IsSigned)
185                 {
186                     generator.Emit(OpCodes.Conv_Ovf_U8);
187                 }
```

```
188                 else
189                 {
190                     generator.Emit(OpCodes.Conv_Ovf_U8_Un);
191                 }
192             }
193             else if (type == typeof(float))
194             {
195                 if (NumericType<TSource>.IsSigned)
196                 {
197                     generator.Emit(OpCodes.Conv_R4);
198                 }
199                 else
200                 {
201                     generator.Emit(OpCodes.Conv_R_Un);
202                 }
203             }
204             else if (type == typeof(double))
205             {
206                 generator.Emit(OpCodes.Conv_R8);
207             }
208             else
209             {
210                 throw new NotSupportedException();
211             }
212         }
213 
214         [MethodImpl(MethodImplOptions.AggressiveInlining)]
215         public static void LoadConstant(this ILGenerator generator, bool value) =>
        ↪  generator.LoadConstant(value ? 1 : 0);
216 
217         [MethodImpl(MethodImplOptions.AggressiveInlining)]
218         public static void LoadConstant(this ILGenerator generator, float value) =>
        ↪  generator.Emit(OpCodes.Ldc_R4, value);
219 
220         [MethodImpl(MethodImplOptions.AggressiveInlining)]
221         public static void LoadConstant(this ILGenerator generator, double value) =>
        ↪  generator.Emit(OpCodes.Ldc_R8, value);
222 
223         [MethodImpl(MethodImplOptions.AggressiveInlining)]
224         public static void LoadConstant(this ILGenerator generator, ulong value) =>
        ↪  generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
225 
226         [MethodImpl(MethodImplOptions.AggressiveInlining)]
227         public static void LoadConstant(this ILGenerator generator, long value) =>
        ↪  generator.Emit(OpCodes.Ldc_I8, value);
228 
229         [MethodImpl(MethodImplOptions.AggressiveInlining)]
230         public static void LoadConstant(this ILGenerator generator, uint value)
231         {
232             switch (value)
233             {
234                 case uint.MaxValue:
235                     generator.Emit(OpCodes.Ldc_I4_M1);
236                     return;
237                 case 0:
238                     generator.Emit(OpCodes.Ldc_I4_0);
239                     return;
240                 case 1:
241                     generator.Emit(OpCodes.Ldc_I4_1);
242                     return;
243                 case 2:
244                     generator.Emit(OpCodes.Ldc_I4_2);
245                     return;
246                 case 3:
247                     generator.Emit(OpCodes.Ldc_I4_3);
248                     return;
249                 case 4:
250                     generator.Emit(OpCodes.Ldc_I4_4);
251                     return;
252                 case 5:
253                     generator.Emit(OpCodes.Ldc_I4_5);
254                     return;
255                 case 6:
256                     generator.Emit(OpCodes.Ldc_I4_6);
257                     return;
258                 case 7:
259                     generator.Emit(OpCodes.Ldc_I4_7);
260                     return;
261                 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)
                {
                    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)]
```

```
338         public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
            ↪ LoadConstantOne(generator, typeof(TConstant));
339
340         [MethodImpl(MethodImplOptions.AggressiveInlining)]
341         public static void LoadConstantOne(this ILGenerator generator, Type constantType)
342         {
343             if (constantType == typeof(float))
344             {
345                 generator.LoadConstant(1F);
346             }
347             else if (constantType == typeof(double))
348             {
349                 generator.LoadConstant(1D);
350             }
351             else if (constantType == typeof(long))
352             {
353                 generator.LoadConstant(1L);
354             }
355             else if (constantType == typeof(ulong))
356             {
357                 generator.LoadConstant(1UL);
358             }
359             else if (constantType == typeof(int))
360             {
361                 generator.LoadConstant(1);
362             }
363             else if (constantType == typeof(uint))
364             {
365                 generator.LoadConstant(1U);
366             }
367             else if (constantType == typeof(short))
368             {
369                 generator.LoadConstant((short)1);
370             }
371             else if (constantType == typeof(ushort))
372             {
373                 generator.LoadConstant((ushort)1);
374             }
375             else if (constantType == typeof(sbyte))
376             {
377                 generator.LoadConstant((sbyte)1);
378             }
379             else if (constantType == typeof(byte))
380             {
381                 generator.LoadConstant((byte)1);
382             }
383             else
384             {
385                 throw new NotSupportedException();
386             }
387         }
388
389         [MethodImpl(MethodImplOptions.AggressiveInlining)]
390         public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
            ↪ constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
391
392         [MethodImpl(MethodImplOptions.AggressiveInlining)]
393         public static void LoadConstant(this ILGenerator generator, Type constantType, object
            ↪ constantValue)
394         {
395             constantValue = Convert.ChangeType(constantValue, constantType);
396             if (constantType == typeof(float))
397             {
398                 generator.LoadConstant((float)constantValue);
399             }
400             else if (constantType == typeof(double))
401             {
402                 generator.LoadConstant((double)constantValue);
403             }
404             else if (constantType == typeof(long))
405             {
406                 generator.LoadConstant((long)constantValue);
407             }
408             else if (constantType == typeof(ulong))
409             {
410                 generator.LoadConstant((ulong)constantValue);
411             }
412             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();
        }

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
        ↪  generator.Emit(OpCodes.Bge, label);

```

```
562         [MethodImpl(MethodImplOptions.AggressiveInlining)]
563         public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
            ↪  label) => generator.Emit(OpCodes.Bge_Un, label);
564
565         [MethodImpl(MethodImplOptions.AggressiveInlining)]
566         public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
            ↪  Label label)
567         {
568             if (isSigned)
569             {
570                 generator.BranchIfGreaterOrEqual(label);
571             }
572             else
573             {
574                 generator.UnsignedBranchIfGreaterOrEqual(label);
575             }
576         }
577
578         [MethodImpl(MethodImplOptions.AggressiveInlining)]
579         public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
            ↪  generator.Emit(OpCodes.Ble, label);
580
581         [MethodImpl(MethodImplOptions.AggressiveInlining)]
582         public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
            ↪  => generator.Emit(OpCodes.Ble_Un, label);
583
584         [MethodImpl(MethodImplOptions.AggressiveInlining)]
585         public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
            ↪  label)
586         {
587             if (isSigned)
588             {
589                 generator.BranchIfLessOrEqual(label);
590             }
591             else
592             {
593                 generator.UnsignedBranchIfLessOrEqual(label);
594             }
595         }
596
597         [MethodImpl(MethodImplOptions.AggressiveInlining)]
598         public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
599
600         [MethodImpl(MethodImplOptions.AggressiveInlining)]
601         public static void Box(this ILGenerator generator, Type boxedType) =>
            ↪  generator.Emit(OpCodes.Box, boxedType);
602
603         [MethodImpl(MethodImplOptions.AggressiveInlining)]
604         public static void Call(this ILGenerator generator, MethodInfo method) =>
            ↪  generator.Emit(OpCodes.Call, method);
605
606         [MethodImpl(MethodImplOptions.AggressiveInlining)]
607         public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
608
609         [MethodImpl(MethodImplOptions.AggressiveInlining)]
610         public static void Unbox<TUnbox>(this ILGenerator generator) =>
            ↪  generator.Unbox(typeof(TUnbox));
611
612         [MethodImpl(MethodImplOptions.AggressiveInlining)]
613         public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
            ↪  generator.Emit(OpCodes.Unbox, typeToUnbox);
614
615         [MethodImpl(MethodImplOptions.AggressiveInlining)]
616         public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
            ↪  generator.UnboxValue(typeof(TUnbox));
617
618         [MethodImpl(MethodImplOptions.AggressiveInlining)]
619         public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
            ↪  generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
620
621         [MethodImpl(MethodImplOptions.AggressiveInlining)]
622         public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
            ↪  generator.DeclareLocal(typeof(T));
623
624         [MethodImpl(MethodImplOptions.AggressiveInlining)]
625         public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪  generator.Emit(OpCodes.Ldloc, local);
626
```

```
627         [MethodImpl(MethodImplOptions.AggressiveInlining)]
628         public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪ generator.Emit(OpCodes.Stloc, local);
629
630         [MethodImpl(MethodImplOptions.AggressiveInlining)]
631         public static void NewObject(this ILGenerator generator, Type type, params Type[]
            ↪ parameterTypes)
632         {
633             var allConstructors = type.GetConstructors(BindingFlags.Public |
                ↪ BindingFlags.NonPublic | BindingFlags.Instance
634 #if !NETSTANDARD
635                 | BindingFlags.CreateInstance
636 #endif
637                 );
638             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();
639             if (constructor == null)
640             {
641                 throw new InvalidOperationException("Type " + type + " must have a constructor
                    ↪ that matches parameters [" + string.Join(", ",
                    ↪ parameterTypes.AsEnumerable()) + "]");
642             }
643             generator.NewObject(constructor);
644         }
645
646         [MethodImpl(MethodImplOptions.AggressiveInlining)]
647         public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
            ↪ generator.Emit(OpCodes.Newobj, constructor);
648
649         [MethodImpl(MethodImplOptions.AggressiveInlining)]
650         public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
651
652         [MethodImpl(MethodImplOptions.AggressiveInlining)]
653         public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
            ↪ false, byte? unaligned = null)
654         {
655             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
656             {
657                 throw new ArgumentException("unaligned must be null, 1, 2, or 4");
658             }
659             if (isVolatile)
660             {
661                 generator.Emit(OpCodes.Volatile);
662             }
663             if (unaligned.HasValue)
664             {
665                 generator.Emit(OpCodes.Unaligned, unaligned.Value);
666             }
667             if (type.IsPointer)
668             {
669                 generator.Emit(OpCodes.Ldind_I);
670             }
671             else if (!type.IsValueType)
672             {
673                 generator.Emit(OpCodes.Ldind_Ref);
674             }
675             else if (type == typeof(sbyte))
676             {
677                 generator.Emit(OpCodes.Ldind_I1);
678             }
679             else if (type == typeof(bool))
680             {
681                 generator.Emit(OpCodes.Ldind_I1);
682             }
683             else if (type == typeof(byte))
684             {
685                 generator.Emit(OpCodes.Ldind_U1);
686             }
687             else if (type == typeof(short))
688             {
689                 generator.Emit(OpCodes.Ldind_I2);
690             }
691             else if (type == typeof(ushort))
692             {
693                 generator.Emit(OpCodes.Ldind_U2);
```

```
            }
            else if (type == typeof(char))
            {
                generator.Emit(OpCodes.Ldind_U2);
            }
            else if (type == typeof(int))
            {
                generator.Emit(OpCodes.Ldind_I4);
            }
            else if (type == typeof(uint))
            {
                generator.Emit(OpCodes.Ldind_U4);
            }
            else if (type == typeof(long) || type == typeof(ulong))
            {
                generator.Emit(OpCodes.Ldind_I8);
            }
            else if (type == typeof(float))
            {
                generator.Emit(OpCodes.Ldind_R4);
            }
            else if (type == typeof(double))
            {
                generator.Emit(OpCodes.Ldind_R8);
            }
            else
            {
                throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
                ↪  ", LoadObject may be more appropriate");
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
        ↪  byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
        ↪  = false, byte? unaligned = null)
        {
            if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
            {
                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
{
```

```
13      public static class NumericType<T>
14      {
15          public static readonly Type Type;
16          public static readonly Type UnderlyingType;
17          public static readonly Type SignedVersion;
18          public static readonly Type UnsignedVersion;
19          public static readonly bool IsFloatPoint;
20          public static readonly bool IsNumeric;
21          public static readonly bool IsSigned;
22          public static readonly bool CanBeNumeric;
23          public static readonly bool IsNullable;
24          public static readonly int BitsLength;
25          public static readonly T MinValue;
26          public static readonly T MaxValue;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          static NumericType()
30          {
31              try
32              {
33                  var type = typeof(T);
34                  var isNullable = type.IsNullable();
35                  var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
36                  var canBeNumeric = underlyingType.CanBeNumeric();
37                  var isNumeric = underlyingType.IsNumeric();
38                  var isSigned = underlyingType.IsSigned();
39                  var isFloatPoint = underlyingType.IsFloatPoint();
40                  var bitsLength = Marshal.SizeOf(underlyingType) * 8;
41                  GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
42                  GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
                        ↪  out Type unsignedVersion);
43                  Type = type;
44                  IsNullable = isNullable;
45                  UnderlyingType = underlyingType;
46                  CanBeNumeric = canBeNumeric;
47                  IsNumeric = isNumeric;
48                  IsSigned = isSigned;
49                  IsFloatPoint = isFloatPoint;
50                  BitsLength = bitsLength;
51                  MinValue = minValue;
52                  MaxValue = maxValue;
53                  SignedVersion = signedVersion;
54                  UnsignedVersion = unsignedVersion;
55              }
56              catch (Exception exception)
57              {
58                  exception.Ignore();
59              }
60          }
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
64          {
65              if (type == typeof(bool))
66              {
67                  minValue = (T)(object)false;
68                  maxValue = (T)(object)true;
69              }
70              else
71              {
72                  minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
73                  maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
74              }
75          }
76
77          [MethodImpl(MethodImplOptions.AggressiveInlining)]
78          private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
                ↪  signedVersion, out Type unsignedVersion)
79          {
80              if (isSigned)
81              {
82                  signedVersion = type;
83                  unsignedVersion = type.GetUnsignedVersionOrNull();
84              }
85              else
86              {
87                  signedVersion = type.GetSignedVersionOrNull();
88                  unsignedVersion = type;
89              }
90          }
```

```
    }
}
```

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 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