

# LinksPlatform's Platform.Reflection Class Library

## 1.1 ./Platform.Reflection/AssemblyExtensions.cs

```
1 using System;
2 using System.Collections.Concurrent;
3 using System.Reflection;
4 using System.Runtime.CompilerServices;
5 using Platform.Exceptions;
6 using Platform.Collections.Lists;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class AssemblyExtensions
13     {
14         private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
15             ↳ ConcurrentDictionary<Assembly, Type[]>();
16
17         /// <remarks>
18         /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
19         /// </remarks>
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public static Type[] GetLoadableTypes(this Assembly assembly)
22         {
23             Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
24             try
25             {
26                 return assembly.GetTypes();
27             }
28             catch (ReflectionTypeLoadException e)
29             {
30                 return e.Types.ToArray(t => t != null);
31             }
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
36             ↳ _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
37     }
38 }
```

## 1.2 ./Platform.Reflection/DelegateHelpers.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Reflection;
4 using System.Reflection.Emit;
5 using System.Runtime.CompilerServices;
6 using Platform.Exceptions;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class DelegateHelpers
13     {
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode, bool
16             ↳ typeMemberMethod)
17             where TDelegate : Delegate
18         {
19             var @delegate = default(TDelegate);
20             try
21             {
22                 @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
23                     ↳ CompileDynamicMethod<TDelegate>(emitCode);
24             }
25             catch (Exception exception)
26             {
27                 exception.Ignore();
28             }
29             return @delegate;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
34             ↳ TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
38             ↳ typeMemberMethod)
39         {
40             var @delegate = default(TDelegate);
41             try
42             {
43                 @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
44                     ↳ CompileDynamicMethod<TDelegate>(emitCode);
45             }
46             catch (Exception exception)
47             {
48                 exception.Ignore();
49             }
50             return @delegate;
51         }
52     }
53 }
```

```

35     where TDelegate : Delegate
36 {
37     var @delegate = CompileOrDefault<TDelegate>(emitCode, typeMemberMethod);
38     if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
39     {
40         @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
41     }
42     return @delegate;
43 }
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
    ↳ : Delegate => Compile<TDelegate>(emitCode, false);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 public static TDelegate CompileDynamicMethod<TDelegate>(Action<ILGenerator> emitCode)
50 {
51     var delegateType = typeof(TDelegate);
52     delegateType.GetDelegateCharacteristics(out Type returnType, out Type[]
        ↳ parameterTypes);
53     var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
54     emitCode(dynamicMethod.GetILGenerator());
55     return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public static TDelegate CompileTypeMemberMethod<TDelegate>(Action<ILGenerator> emitCode)
60 {
61     AssemblyName assemblyName = new AssemblyName(GetNewName());
62     var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
        ↳ AssemblyBuilderAccess.Run);
63     var module = assembly.DefineDynamicModule(GetNewName());
64     var type = module.DefineType(GetNewName());
65     var methodName = GetNewName();
66     type.EmitStaticMethod<TDelegate>(methodName, emitCode);
67     var typeInfo = type.CreateTypeInfo();
68     return (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(typeof(TDele
        ↳ gate));
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 private static string GetNewName() => Guid.NewGuid().ToString("N");
73 }
74 }

```

### 1.3 ./Platform.Reflection/DynamicExtensions.cs

```

1 using System.Collections.Generic;
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 DynamicExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static bool HasProperty(this object @object, string propertyName)
12         {
13             var type = @object.GetType();
14             if (type is IDictionary<string, object> dictionary)
15             {
16                 return dictionary.ContainsKey(propertyName);
17             }
18             return type.GetProperty(propertyName) != null;
19         }
20     }
21 }

```

### 1.4 ./Platform.Reflection/EnsureExtensions.cs

```

1 using System;
2 using System.Diagnostics;
3 using System.Runtime.CompilerServices;
4 using Platform.Exceptions;
5 using Platform.Exceptions.ExtensionRoots;
6
7 #pragma warning disable IDE0060 // Remove unused parameter
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9

```

```

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,
18         ↪ Func<string> messageBuilder)
19         {
20             if (!NumericType<T>.IsNumeric || NumericType<T>.IsSigned ||
21             ↪ NumericType<T>.IsFloatPoint)
22             {
23                 throw new NotSupportedException(messageBuilder());
24             }
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
29         ↪ message)
30         {
31             string messageBuilder() => message;
32             IsUnsignedInteger<T>(root, messageBuilder());
33         }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
37         ↪ IsUnsignedInteger<T>(root, (string)null);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
41         ↪ messageBuilder)
42         {
43             if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
44             ↪ NumericType<T>.IsFloatPoint)
45             {
46                 throw new NotSupportedException(messageBuilder());
47             }
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
52         ↪ message)
53         {
54             string messageBuilder() => message;
55             IsSignedInteger<T>(root, messageBuilder());
56         }
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
60         ↪ IsSignedInteger<T>(root, (string)null);
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
64         ↪ messageBuilder)
65         {
66             if (!NumericType<T>.IsSigned)
67             {
68                 throw new NotSupportedException(messageBuilder());
69             }
70         }
71
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
74         {
75             string messageBuilder() => message;
76             IsSigned<T>(root, messageBuilder());
77         }
78
79         [MethodImpl(MethodImplOptions.AggressiveInlining)]
80         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
81         ↪ (string)null);
82
83         [MethodImpl(MethodImplOptions.AggressiveInlining)]
84         public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
85         ↪ messageBuilder)
86         {
87             if (!NumericType<T>.IsNumeric)

```

```

    {
        throw new NotSupportedException(messageBuilder());
    }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
{
    string messageBuilder() => message;
    IsNumeric<T>(root, messageBuilder());
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
    ↪ IsNumeric<T>(root, (string)null);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
    ↪ messageBuilder)
{
    if (!NumericType<T>.CanBeNumeric)
    {
        throw new NotSupportedException(messageBuilder());
    }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
{
    string messageBuilder() => message;
    CanBeNumeric<T>(root, messageBuilder());
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
    ↪ CanBeNumeric<T>(root, (string)null);

#endregion

#region OnDebug

[Conditional("DEBUG")]
public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
    ↪ Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);

[Conditional("DEBUG")]
public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
    ↪ message) => Ensure.Always.IsUnsignedInteger<T>(message);

[Conditional("DEBUG")]
public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
    ↪ Ensure.Always.IsUnsignedInteger<T>();

[Conditional("DEBUG")]
public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
    ↪ messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);

[Conditional("DEBUG")]
public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
    ↪ message) => Ensure.Always.IsSignedInteger<T>(message);

[Conditional("DEBUG")]
public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
    ↪ Ensure.Always.IsSignedInteger<T>();

[Conditional("DEBUG")]
public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
    ↪ messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);

[Conditional("DEBUG")]
public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
    ↪ Ensure.Always.IsSigned<T>(message);

[Conditional("DEBUG")]
public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
    ↪ Ensure.Always.IsSigned<T>();

[Conditional("DEBUG")]

```

```

143     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
144         ↳ messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
145
146     [Conditional("DEBUG")]
147     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
148         ↳ Ensure.Always.IsNumeric<T>(message);
149
150     [Conditional("DEBUG")]
151     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
152         ↳ Ensure.Always.IsNumeric<T>();
153
154     [Conditional("DEBUG")]
155     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
156         ↳ messageBuilder) => Ensure.Always.CanBeNumeric<T>(messageBuilder);
157
158     [Conditional("DEBUG")]
159     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
160         ↳ => Ensure.Always.CanBeNumeric<T>(message);
161
162     [Conditional("DEBUG")]
163     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
164         ↳ Ensure.Always.CanBeNumeric<T>();
165
166     #endregion
167 }
168 }

```

### 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) =>
12             ↳ (T)fieldInfo.GetValue(null);
13     }
14 }

```

### 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) =>
15             ↳ generator.ThrowException(typeof(T));
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator) =>
19             ↳ UncheckedConvert<TSource, TTarget>(generator, extendSign: false);
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator, bool
23             ↳ extendSign)
24         {
25             var sourceType = typeof(TSource);
26             var targetType = typeof(TTarget);
27             if (sourceType == targetType)
28             {
29                 return;
30             }
31             if (extendSign)
32             {
33                 if (sourceType == typeof(byte))
34                 {

```

```

32         generator.Emit(OpCodes.Conv_I1);
33     }
34     if (sourceType == typeof(ushort))
35     {
36         generator.Emit(OpCodes.Conv_I2);
37     }
38 }
39 if (NumericType<TSource>.BitsLength > NumericType<TTarget>.BitsLength)
40 {
41     if (targetType == typeof(short))
42     {
43         generator.Emit(OpCodes.Conv_I2);
44     }
45     else if (targetType == typeof(ushort))
46     {
47         generator.Emit(OpCodes.Conv_U2);
48     }
49     else if (targetType == typeof(sbyte))
50     {
51         generator.Emit(OpCodes.Conv_I1);
52     }
53     else if (targetType == typeof(byte))
54     {
55         generator.Emit(OpCodes.Conv_U1);
56     }
57     else if (targetType == typeof(int))
58     {
59         generator.Emit(OpCodes.Conv_I4);
60     }
61     else if (targetType == typeof(uint))
62     {
63         generator.Emit(OpCodes.Conv_U4);
64     }
65     else if (targetType == typeof(long))
66     {
67         generator.Emit(OpCodes.Conv_I8);
68     }
69     else if (targetType == typeof(ulong))
70     {
71         generator.Emit(OpCodes.Conv_U8);
72     }
73 }
74 else
75 {
76     if (!extendSign)
77     {
78         if (sourceType == typeof(uint) && targetType == typeof(long))
79         {
80             generator.Emit(OpCodes.Conv_U8);
81         }
82     }
83 }
84 if (targetType == typeof(float))
85 {
86     if (NumericType<TSource>.IsSigned)
87     {
88         generator.Emit(OpCodes.Conv_R4);
89     }
90     else
91     {
92         generator.Emit(OpCodes.Conv_R_Un);
93     }
94 }
95 else if (targetType == typeof(double))
96 {
97     generator.Emit(OpCodes.Conv_R8);
98 }
99 }
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
103 {
104     var type = typeof(TTarget);
105     if (type == typeof(short))
106     {
107         if (NumericType<TSource>.IsSigned)
108         {
109             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) =>
216     ↪ generator.LoadConstant(value ? 1 : 0);
217
218 [MethodImpl(MethodImplOptions.AggressiveInlining)]
219 public static void LoadConstant(this ILGenerator generator, float value) =>
220     ↪ generator.Emit(OpCodes.Ldc_R4, value);
221
222 [MethodImpl(MethodImplOptions.AggressiveInlining)]
223 public static void LoadConstant(this ILGenerator generator, double value) =>
224     ↪ generator.Emit(OpCodes.Ldc_R8, value);
225
226 [MethodImpl(MethodImplOptions.AggressiveInlining)]
227 public static void LoadConstant(this ILGenerator generator, ulong value) =>
228     ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
229
230 [MethodImpl(MethodImplOptions.AggressiveInlining)]
231 public static void LoadConstant(this ILGenerator generator, long value) =>
232     ↪ generator.Emit(OpCodes.Ldc_I8, value);
233
234 [MethodImpl(MethodImplOptions.AggressiveInlining)]
235 public static void LoadConstant(this ILGenerator generator, uint value)
236 {
237     switch (value)
238     {
239         case uint.MaxValue:
240             generator.Emit(OpCodes.Ldc_I4_M1);
241             return;
242         case 0:
243             generator.Emit(OpCodes.Ldc_I4_0);
244             return;
245         case 1:
246             generator.Emit(OpCodes.Ldc_I4_1);
247             return;
248         case 2:
249             generator.Emit(OpCodes.Ldc_I4_2);
250             return;
251         case 3:
252             generator.Emit(OpCodes.Ldc_I4_3);
253             return;
254         case 4:
255             generator.Emit(OpCodes.Ldc_I4_4);
256             return;
257         case 5:
258             generator.Emit(OpCodes.Ldc_I4_5);
259             return;
260         case 6:
261             generator.Emit(OpCodes.Ldc_I4_6);
262             return;
263         case 7:
264             generator.Emit(OpCodes.Ldc_I4_7);
265             return;
266         case 8:

```



```

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

```

```

338 public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
339     ↳ LoadConstantOne(generator, typeof(TConstant));
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 public static void LoadConstantOne(this ILGenerator generator, Type constantType)
343 {
344     if (constantType == typeof(float))
345     {
346         generator.LoadConstant(1F);
347     }
348     else if (constantType == typeof(double))
349     {
350         generator.LoadConstant(1D);
351     }
352     else if (constantType == typeof(long))
353     {
354         generator.LoadConstant(1L);
355     }
356     else if (constantType == typeof(ulong))
357     {
358         generator.LoadConstant(1UL);
359     }
360     else if (constantType == typeof(int))
361     {
362         generator.LoadConstant(1);
363     }
364     else if (constantType == typeof(uint))
365     {
366         generator.LoadConstant(1U);
367     }
368     else if (constantType == typeof(short))
369     {
370         generator.LoadConstant((short)1);
371     }
372     else if (constantType == typeof(ushort))
373     {
374         generator.LoadConstant((ushort)1);
375     }
376     else if (constantType == typeof(sbyte))
377     {
378         generator.LoadConstant((sbyte)1);
379     }
380     else if (constantType == typeof(byte))
381     {
382         generator.LoadConstant((byte)1);
383     }
384     else
385     {
386         throw new NotSupportedException();
387     }
388 }
389
390 [MethodImpl(MethodImplOptions.AggressiveInlining)]
391 public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
392     ↳ constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
393
394 [MethodImpl(MethodImplOptions.AggressiveInlining)]
395 public static void LoadConstant(this ILGenerator generator, Type constantType, object
396     ↳ constantValue)
397 {
398     constantValue = Convert.ChangeType(constantValue, constantType);
399     if (constantType == typeof(float))
400     {
401         generator.LoadConstant((float)constantValue);
402     }
403     else if (constantType == typeof(double))
404     {
405         generator.LoadConstant((double)constantValue);
406     }
407     else if (constantType == typeof(long))
408     {
409         generator.LoadConstant((long)constantValue);
410     }
411     else if (constantType == typeof(ulong))
412     {
413         generator.LoadConstant((ulong)constantValue);
414     }
415     else if (constantType == typeof(int))
416     {
417         generator.LoadConstant((int)constantValue);
418     }
419     else if (constantType == typeof(uint))
420     {
421         generator.LoadConstant((uint)constantValue);
422     }
423     else if (constantType == typeof(short))
424     {
425         generator.LoadConstant((short)constantValue);
426     }
427     else if (constantType == typeof(ushort))
428     {
429         generator.LoadConstant((ushort)constantValue);
430     }
431     else if (constantType == typeof(sbyte))
432     {
433         generator.LoadConstant((sbyte)constantValue);
434     }
435     else if (constantType == typeof(byte))
436     {
437         generator.LoadConstant((byte)constantValue);
438     }
439     else
440     {
441         throw new NotSupportedException();
442     }
443 }

```

```

413     {
414         generator.LoadConstant((int) constantValue);
415     }
416     else if (constantType == typeof(uint))
417     {
418         generator.LoadConstant((uint) constantValue);
419     }
420     else if (constantType == typeof(short))
421     {
422         generator.LoadConstant((short) constantValue);
423     }
424     else if (constantType == typeof(ushort))
425     {
426         generator.LoadConstant((ushort) constantValue);
427     }
428     else if (constantType == typeof(sbyte))
429     {
430         generator.LoadConstant((sbyte) constantValue);
431     }
432     else if (constantType == typeof(byte))
433     {
434         generator.LoadConstant((byte) constantValue);
435     }
436     else
437     {
438         throw new NotSupportedException();
439     }
440 }
441
442 [MethodImpl(MethodImplOptions.AggressiveInlining)]
443 public static void Increment<TValue>(this ILGenerator generator) =>
444     ↪ generator.Increment(typeof(TValue));
445
446 [MethodImpl(MethodImplOptions.AggressiveInlining)]
447 public static void Decrement<TValue>(this ILGenerator generator) =>
448     ↪ generator.Decrement(typeof(TValue));
449
450 [MethodImpl(MethodImplOptions.AggressiveInlining)]
451 public static void Increment(this ILGenerator generator, Type valueType)
452 {
453     generator.LoadConstantOne(valueType);
454     generator.Add();
455 }
456
457 [MethodImpl(MethodImplOptions.AggressiveInlining)]
458 public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
459
460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
461 public static void Decrement(this ILGenerator generator, Type valueType)
462 {
463     generator.LoadConstantOne(valueType);
464     generator.Subtract();
465 }
466
467 [MethodImpl(MethodImplOptions.AggressiveInlining)]
468 public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
469
470 [MethodImpl(MethodImplOptions.AggressiveInlining)]
471 public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
472
473 [MethodImpl(MethodImplOptions.AggressiveInlining)]
474 public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
475
476 [MethodImpl(MethodImplOptions.AggressiveInlining)]
477 public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
478
479 [MethodImpl(MethodImplOptions.AggressiveInlining)]
480 public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
481
482 [MethodImpl(MethodImplOptions.AggressiveInlining)]
483 public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
484
485 [MethodImpl(MethodImplOptions.AggressiveInlining)]
486 public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 public static void LoadArgument(this ILGenerator generator, int argumentIndex)
490 {
491     switch (argumentIndex)

```

```

490     {
491         case 0:
492             generator.Emit(OpCodes.Ldarg_0);
493             break;
494         case 1:
495             generator.Emit(OpCodes.Ldarg_1);
496             break;
497         case 2:
498             generator.Emit(OpCodes.Ldarg_2);
499             break;
500         case 3:
501             generator.Emit(OpCodes.Ldarg_3);
502             break;
503         default:
504             generator.Emit(OpCodes.Ldarg, argumentIndex);
505             break;
506     }
507 }
508
509 [MethodImpl(MethodImplOptions.AggressiveInlining)]
510 public static void LoadArguments(this ILGenerator generator, params int[]
    → argumentIndices)
511 {
512     for (var i = 0; i < argumentIndices.Length; i++)
513     {
514         generator.LoadArgument(argumentIndices[i]);
515     }
516 }
517
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
    → generator.Emit(OpCodes.Starg, argumentIndex);
520
521 [MethodImpl(MethodImplOptions.AggressiveInlining)]
522 public static void CompareGreaterThan(this ILGenerator generator) =>
    → generator.Emit(OpCodes.Cgt);
523
524 [MethodImpl(MethodImplOptions.AggressiveInlining)]
525 public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
    → generator.Emit(OpCodes.Cgt_Un);
526
527 [MethodImpl(MethodImplOptions.AggressiveInlining)]
528 public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
529 {
530     if (isSigned)
531     {
532         generator.CompareGreaterThan();
533     }
534     else
535     {
536         generator.UnsignedCompareGreaterThan();
537     }
538 }
539
540 [MethodImpl(MethodImplOptions.AggressiveInlining)]
541 public static void CompareLessThan(this ILGenerator generator) =>
    → generator.Emit(OpCodes.Clt);
542
543 [MethodImpl(MethodImplOptions.AggressiveInlining)]
544 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
    → generator.Emit(OpCodes.Clt_Un);
545
546 [MethodImpl(MethodImplOptions.AggressiveInlining)]
547 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
548 {
549     if (isSigned)
550     {
551         generator.CompareLessThan();
552     }
553     else
554     {
555         generator.UnsignedCompareLessThan();
556     }
557 }
558
559 [MethodImpl(MethodImplOptions.AggressiveInlining)]
560 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
    → generator.Emit(OpCodes.Bge, label);
561

```

```

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
    ↪ BindingFlags.CreateInstance
635 #endif
    );
636
637     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();
638
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);

```

```

694     }
695     else if (type == typeof(char))
696     {
697         generator.Emit(OpCodes.Ldind_U2);
698     }
699     else if (type == typeof(int))
700     {
701         generator.Emit(OpCodes.Ldind_I4);
702     }
703     else if (type == typeof(uint))
704     {
705         generator.Emit(OpCodes.Ldind_U4);
706     }
707     else if (type == typeof(long) || type == typeof(ulong))
708     {
709         generator.Emit(OpCodes.Ldind_I8);
710     }
711     else if (type == typeof(float))
712     {
713         generator.Emit(OpCodes.Ldind_R4);
714     }
715     else if (type == typeof(double))
716     {
717         generator.Emit(OpCodes.Ldind_R8);
718     }
719     else
720     {
721         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
722             ↪ ", LoadObject may be more appropriate");
723     }
724 }
725
726 [MethodImpl(MethodImplOptions.AggressiveInlining)]
727 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
728     ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
729
730 [MethodImpl(MethodImplOptions.AggressiveInlining)]
731 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
732     ↪ = false, byte? unaligned = null)
733 {
734     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
735     {
736         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
737     }
738     if (isVolatile)
739     {
740         generator.Emit(OpCodes.Volatile);
741     }
742     if (unaligned.HasValue)
743     {
744         generator.Emit(OpCodes.Unaligned, unaligned.Value);
745     }
746     if (type.IsPointer)
747     {
748         generator.Emit(OpCodes.Stind_I);
749     }
750     else if (!type.IsValueType)
751     {
752         generator.Emit(OpCodes.Stind_Ref);
753     }
754     else if (type == typeof(sbyte) || type == typeof(byte))
755     {
756         generator.Emit(OpCodes.Stind_I1);
757     }
758     else if (type == typeof(short) || type == typeof(ushort))
759     {
760         generator.Emit(OpCodes.Stind_I2);
761     }
762     else if (type == typeof(int) || type == typeof(uint))
763     {
764         generator.Emit(OpCodes.Stind_I4);
765     }
766     else if (type == typeof(long) || type == typeof(ulong))
767     {
768         generator.Emit(OpCodes.Stind_I8);
769     }
770     else if (type == typeof(float))
771     {
772         generator.Emit(OpCodes.Stind_R4);
773     }
774     else if (type == typeof(double))
775     {
776         generator.Emit(OpCodes.Stind_R8);
777     }
778     else
779     {
780         throw new InvalidOperationException("StoreIndirect cannot be used with " + type +
781             ↪ ", StoreObject may be more appropriate");
782     }
783 }

```

```

769         generator.Emit(OpCodes.Stind_R4);
770     }
771     else if (type == typeof(double))
772     {
773         generator.Emit(OpCodes.Stind_R8);
774     }
775     else
776     {
777         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
778             ↪ + ", StoreObject may be more appropriate");
779     }
780 }
781 }

```

### 1.7 ./Platform.Reflection/MethodInfoExtensions.cs

```

1  using System;
2  using System.Linq;
3  using System.Reflection;
4  using System.Runtime.CompilerServices;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Reflection
9  {
10     public static class MethodInfoExtensions
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static byte[] GetILBytes(this MethodInfo methodInfo) =>
14             ↪ methodInfo.GetMethodBody().GetILAsByteArray();
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
18             ↪ methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
19     }
20 }

```

### 1.8 ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Reflection
9  {
10     public class NotSupportedExceptionDelegateFactory<TDelegate> : IFactory<TDelegate>
11         where TDelegate : Delegate
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public TDelegate Create()
15         {
16             var @delegate = DelegateHelpers.CompileOrDefault<TDelegate>(generator =>
17             {
18                 generator.Throw<NotSupportedException>();
19             });
20             if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
21             {
22                 throw new InvalidOperationException("Unable to compile stub delegate.");
23             }
24             return @delegate;
25         }
26     }
27 }

```

### 1.9 ./Platform.Reflection/NumericType.cs

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using System.Runtime.InteropServices;
4  using Platform.Exceptions;
5
6  // ReSharper disable AssignmentInConditionalExpression
7  // ReSharper disable BuiltInTypeReferenceStyle
8  // ReSharper disable StaticFieldInGenericType
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11  namespace Platform.Reflection
12  {

```



```

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,
43                 ↪ out Type unsignedVersion);
44             Type = type;
45             IsNullable = isNullable;
46             UnderlyingType = underlyingType;
47             CanBeNumeric = canBeNumeric;
48             IsNumeric = isNumeric;
49             IsSigned = isSigned;
50             IsFloatPoint = isFloatPoint;
51             BitsLength = bitsLength;
52             MinValue = minValue;
53             MaxValue = maxValue;
54             SignedVersion = signedVersion;
55             UnsignedVersion = unsignedVersion;
56         }
57         catch (Exception exception)
58         {
59             exception.Ignore();
60         }
61     }
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
65     {
66         if (type == typeof(bool))
67         {
68             minValue = (T)(object>false;
69             maxValue = (T)(object>true;
70         }
71         else
72         {
73             minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
74             maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
75         }
76     }
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
80         ↪ signedVersion, out Type unsignedVersion)
81     {
82         if (isSigned)
83         {
84             signedVersion = type;
85             unsignedVersion = type.GetUnsignedVersionOrNull();
86         }
87         else
88         {
89             signedVersion = type.GetSignedVersionOrNull();
90             unsignedVersion = type;
91         }
92     }

```

```

91     }
92 }

```

#### 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) =>
12             ↪ (T)fieldInfo.GetValue(null);
13     }
14 }

```

#### 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 =
13             ↪ MethodAttributes.Public | MethodAttributes.Static;
14         public static readonly MethodAttributes DefaultVirtualMethodAttributes =
15             ↪ MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
16             ↪ MethodAttributes.HideBySig;
17         public static readonly MethodImplAttributes DefaultMethodImplAttributes =
18             ↪ MethodImplAttributes.IL | MethodImplAttributes.Managed |
19             ↪ MethodImplAttributes.AggressiveInlining;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName,
23             ↪ MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes,
24             ↪ Action<ILGenerator> emitCode)
25         {
26             typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[]
27                 ↪ parameterTypes);
28             EmitMethod(type, methodName, methodAttributes, methodImplAttributes, returnType,
29                 ↪ parameterTypes, emitCode);
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
34             ↪ methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[]
35             ↪ parameterTypes, Action<ILGenerator> emitCode)
36         {
37             MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
38                 ↪ parameterTypes);
39             method.SetImplementationFlags(methodImplAttributes);
40             var generator = method.GetILGenerator();
41             emitCode(generator);
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName,
46             ↪ Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
47             ↪ DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static void EmitVirtualMethod<TDelegate>(this TypeBuilder type, string
51             ↪ methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
52             ↪ DefaultVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
53     }
54 }

```

#### 1.12 ./Platform.Reflection/TypeExtensions.cs

```

1  using System;
2  using System.Collections.Generic;

```

```

3 using System.Linq;
4 using System.Reflection;
5 using System.Runtime.CompilerServices;
6 using Platform.Collections;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class TypeExtensions
13     {
14         static public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
15             ↳ BindingFlags.NonPublic | BindingFlags.Static;
16         static public readonly string DefaultDelegateMethodName = "Invoke";
17
18         static private readonly HashSet<Type> _canBeNumericTypes;
19         static private readonly HashSet<Type> _isNumericTypes;
20         static private readonly HashSet<Type> _isSignedTypes;
21         static private readonly HashSet<Type> _isFloatPointTypes;
22         static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
23         static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         static TypeExtensions()
27         {
28             _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
29                 ↳ typeof(DateTime), typeof(TimeSpan) };
30             _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
31                 ↳ typeof(ulong) };
32             _canBeNumericTypes.UnionWith(_isNumericTypes);
33             _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
34                 ↳ typeof(long) };
35             _canBeNumericTypes.UnionWith(_isSignedTypes);
36             _isNumericTypes.UnionWith(_isSignedTypes);
37             _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
38                 ↳ typeof(float) };
39             _canBeNumericTypes.UnionWith(_isFloatPointTypes);
40             _isNumericTypes.UnionWith(_isFloatPointTypes);
41             _isSignedTypes.UnionWith(_isFloatPointTypes);
42             _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
43             {
44                 { typeof(sbyte), typeof(byte) },
45                 { typeof(short), typeof(ushort) },
46                 { typeof(int), typeof(uint) },
47                 { typeof(long), typeof(ulong) },
48             };
49             _signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
50             {
51                 { typeof(byte), typeof(sbyte) },
52                 { typeof(ushort), typeof(short) },
53                 { typeof(uint), typeof(int) },
54                 { typeof(ulong), typeof(long) },
55             };
56         }
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         public static T GetStaticFieldValue<T>(this Type type, string name) =>
63             ↳ type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public static T GetStaticPropertyValue<T>(this Type type, string name) =>
67             ↳ type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
71             ↳ genericParameterTypes, Type[] argumentTypes)
72         {
73             var methods = from m in type.GetMethods()
74                 where m.Name == name
75                     && m.IsGenericMethodDefinition
76                 let typeParams = m.GetGenericArguments()
77                 let normalParams = m.GetParameters().Select(x => x.ParameterType)
78                 where typeParams.SequenceEqual(genericParameterTypes)
79                     && normalParams.SequenceEqual(argumentTypes)
80                 select m;
81             var method = methods.Single();
82         }
83     }
84 }

```

```

74         return method;
75     }
76
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     public static Type GetBaseType(this Type type) => type.BaseType;
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     public static Assembly GetAssembly(this Type type) => type.Assembly;
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     public static bool IsSubclassOf(this Type type, Type superClass) =>
85         type.IsSubclassOf(superClass);
86
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     public static bool IsValueType(this Type type) => type.IsValueType;
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public static bool IsGeneric(this Type type) => type.IsGenericType;
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
95         type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
99
100     [MethodImpl(MethodImplOptions.AggressiveInlining)]
101     public static Type GetUnsignedVersionOrNull(this Type signedType) =>
102         _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
103
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     public static Type GetSignedVersionOrNull(this Type unsignedType) =>
106         _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
110
111     [MethodImpl(MethodImplOptions.AggressiveInlining)]
112     public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
113
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
116
117     [MethodImpl(MethodImplOptions.AggressiveInlining)]
118     public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
119
120     [MethodImpl(MethodImplOptions.AggressiveInlining)]
121     public static Type GetDelegateReturnType(this Type delegateType) =>
122         delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;
123
124     [MethodImpl(MethodImplOptions.AggressiveInlining)]
125     public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
126         delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();
127
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     public static void GetDelegateCharacteristics(this Type delegateType, out Type
130         returnType, out Type[] parameterTypes)
131     {
132         var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
133         returnType = invoke.ReturnType;
134         parameterTypes = invoke.GetParameterTypes();
135     }
136 }

```

### 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,
13             ↪ T4, T5>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

#### 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,
13             ↪ T4>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

#### 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,
13             ↪ T3>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

#### 1.19 ./Platform.Reflection/Types[T1, T2].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> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1,
13             ↪ T2>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

#### 1.20 ./Platform.Reflection/Types[T].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<T> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new
            ↳ Types<T>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

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

```

1 using System;
2 using Xunit;
3
4 namespace Platform.Reflection.Tests
5 {
6     public class CodeGenerationTests
7     {
8         [Fact]
9         public void EmptyActionCompilationTest()
10        {
11            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
12            {
13                generator.Return();
14            });
15            compiledAction();
16        }
17
18        [Fact]
19        public void FailedActionCompilationTest()
20        {
21            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
22            {
23                throw new NotImplementedException();
24            });
25            Assert.Throws<NotSupportedException>(compiledAction);
26        }
27
28        [Fact]
29        public void ConstantLoadingTest()
30        {
31            CheckConstantLoading<byte>(8);
32            CheckConstantLoading<uint>(8);
33            CheckConstantLoading<ushort>(8);
34            CheckConstantLoading<ulong>(8);
35        }
36
37        private void CheckConstantLoading<T>(T value)
38        {
39            var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
40            {
41                generator.LoadConstant(value);
42                generator.Return();
43            });
44            Assert.Equal(value, compiledFunction());
45        }
46
47        [Fact]
48        public void ConversionWithSignExtensionTest()
49        {
50            object[] withSignExtension = new object[]
51            {
52                CompileUncheckedConverter<byte, sbyte>(extendSign: true)(128),
53                CompileUncheckedConverter<byte, short>(extendSign: true)(128),
54                CompileUncheckedConverter<ushort, short>(extendSign: true)(32768),
55                CompileUncheckedConverter<byte, int>(extendSign: true)(128),
56                CompileUncheckedConverter<ushort, int>(extendSign: true)(32768),
57                CompileUncheckedConverter<uint, int>(extendSign: true)(2147483648),
58                CompileUncheckedConverter<byte, long>(extendSign: true)(128),
59                CompileUncheckedConverter<ushort, long>(extendSign: true)(32768),
60                CompileUncheckedConverter<uint, long>(extendSign: true)(2147483648),
61                CompileUncheckedConverter<ulong, long>(extendSign: true)(9223372036854775808)
62            };
63            object[] withoutSignExtension = new object[]
64            {
65                CompileUncheckedConverter<byte, sbyte>(extendSign: false)(128),
66                CompileUncheckedConverter<byte, short>(extendSign: false)(128),

```

```

67         CompileUncheckedConverter<ushort, short>(extendSign: false)(32768),
68         CompileUncheckedConverter<byte, int>(extendSign: false)(128),
69         CompileUncheckedConverter<ushort, int>(extendSign: false)(32768),
70         CompileUncheckedConverter<uint, int>(extendSign: false)(2147483648),
71         CompileUncheckedConverter<byte, long>(extendSign: false)(128),
72         CompileUncheckedConverter<ushort, long>(extendSign: false)(32768),
73         CompileUncheckedConverter<uint, long>(extendSign: false)(2147483648),
74         CompileUncheckedConverter<ulong, long>(extendSign: false)(9223372036854775808)
75     };
76     var i = 0;
77     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
78     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
79     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
80     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
81     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
82     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
83     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
84     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
85     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
86     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
87 }
88
89 private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
    ↪ TTarget>(bool extendSign)
90 {
91     return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
92     {
93         generator.LoadArgument(0);
94         generator.UncheckedConvert<TSource, TTarget>(extendSign);
95         generator.Return();
96     });
97 }
98 }
99 }

```

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

```

1  using System;
2  using System.Reflection;
3  using Xunit;
4  using Platform.Collections;
5  using Platform.Collections.Lists;
6
7  namespace Platform.Reflection.Tests
8  {
9      public static class GetILBytesMethodTests
10     {
11         [Fact]
12         public static void ILBytesForDelegateAreAvailableTest()
13         {
14             var function = new Func<object, int>(argument => 0);
15             var bytes = function.GetMethodInfo().GetILBytes();
16             Assert.False(bytes.IsNullOrEmpty());
17         }
18
19         [Fact]
20         public static void ILBytesForDifferentDelegatesAreTheSameTest()
21         {
22             var firstFunction = new Func<object, int>(argument => 0);
23             var secondFunction = new Func<object, int>(argument => 0);
24             Assert.False(firstFunction == secondFunction);
25             var firstFunctionBytes = firstFunction.GetMethodInfo().GetILBytes();
26             Assert.False(firstFunctionBytes.IsNullOrEmpty());
27             var secondFunctionBytes = secondFunction.GetMethodInfo().GetILBytes();
28             Assert.False(secondFunctionBytes.IsNullOrEmpty());
29             Assert.True(firstFunctionBytes.EqualTo(secondFunctionBytes));
30         }
31     }
32 }

```

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

```

1  using Xunit;
2
3  namespace Platform.Reflection.Tests
4  {
5      public class NumericTypeTests
6      {
7          [Fact]
8          public void UInt64IsNumericTest()
9          {

```



```
10         Assert.True(NumericType<ulong>.IsNumeric);
11     }
12 }
13 }
```

## Index

- ./Platform.Reflection.Tests/CodeGenerationTests.cs, 23
- ./Platform.Reflection.Tests/GetILBytesMethodTests.cs, 24
- ./Platform.Reflection.Tests/NumericTypeTests.cs, 24
- ./Platform.Reflection/AssemblyExtensions.cs, 1
- ./Platform.Reflection/DelegateHelpers.cs, 1
- ./Platform.Reflection/DynamicExtensions.cs, 2
- ./Platform.Reflection/EnsureExtensions.cs, 2
- ./Platform.Reflection/FieldInfoExtensions.cs, 5
- ./Platform.Reflection/ILGeneratorExtensions.cs, 5
- ./Platform.Reflection/MethodInfoExtensions.cs, 16
- ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 16
- ./Platform.Reflection/NumericType.cs, 16
- ./Platform.Reflection/PropertyInfoExtensions.cs, 18
- ./Platform.Reflection/TypeBuilderExtensions.cs, 18
- ./Platform.Reflection/TypeExtensions.cs, 18
- ./Platform.Reflection/Types.cs, 20
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 21
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 21
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 21
- ./Platform.Reflection/Types[T1, T2, T3, T4].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3].cs, 22
- ./Platform.Reflection/Types[T1, T2].cs, 22
- ./Platform.Reflection/Types[T].cs, 22