

# 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)
88             {
89                 throw new NotSupportedException(messageBuilder());
90             }
91         }
92     }
93 }

```

```
{
    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>
        ↳ 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                 {

```

```

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>.BitsSize > NumericType<TTarget>.BitsSize)
40 {
41     generator.ConvertToInteger(targetType);
42 }
43 else
44 {
45     #if NET471
46         if (sourceType == typeof(byte) || sourceType == typeof(ushort))
47         {
48             if (targetType == typeof(long))
49             {
50                 if (extendSign)
51                 {
52                     generator.Emit(OpCodes.Conv_I8);
53                 }
54                 else
55                 {
56                     generator.Emit(OpCodes.Conv_U8);
57                 }
58             }
59             if (sourceType == typeof(uint) && targetType == typeof(long) && extendSign)
60             {
61                 generator.Emit(OpCodes.Conv_I8);
62             }
63             #endif
64             if (sourceType == typeof(uint) && targetType == typeof(long) && !extendSign)
65             {
66                 generator.Emit(OpCodes.Conv_U8);
67             }
68         }
69     }
70     if (targetType == typeof(float))
71     {
72         if (NumericType<TSource>.IsSigned)
73         {
74             generator.Emit(OpCodes.Conv_R4);
75         }
76         else
77         {
78             generator.Emit(OpCodes.Conv_R_Un);
79         }
80     }
81     else if (targetType == typeof(double))
82     {
83         generator.Emit(OpCodes.Conv_R8);
84     }
85     else if (targetType == typeof(bool))
86     {
87         generator.ConvertToBoolean<TSource>();
88     }
89 }
90
91 private static void ConvertToBoolean<TSource>(this ILGenerator generator)
92 {
93     generator.LoadConstant<TSource>(default);
94     var sourceType = typeof(TSource);
95     if (sourceType == typeof(float) || sourceType == typeof(double))
96     {
97         generator.Emit(OpCodes.Ceq);
98         // Inversion of the first Ceq instruction
99         generator.LoadConstant<int>(0);
100        generator.Emit(OpCodes.Ceq);
101    }
102    else
103    {
104        generator.Emit(OpCodes.Cgt_Un);
105    }
106 }
107
108 private static void ConvertToInteger(this ILGenerator generator, Type targetType)
109 {

```

```

110     if (targetType == typeof(sbyte))
111     {
112         generator.Emit(OpCodes.Conv_I1);
113     }
114     else if (targetType == typeof(byte))
115     {
116         generator.Emit(OpCodes.Conv_U1);
117     }
118     else if (targetType == typeof(short))
119     {
120         generator.Emit(OpCodes.Conv_I2);
121     }
122     else if (targetType == typeof(ushort))
123     {
124         generator.Emit(OpCodes.Conv_U2);
125     }
126     else if (targetType == typeof(int))
127     {
128         generator.Emit(OpCodes.Conv_I4);
129     }
130     else if (targetType == typeof(uint))
131     {
132         generator.Emit(OpCodes.Conv_U4);
133     }
134     else if (targetType == typeof(long))
135     {
136         generator.Emit(OpCodes.Conv_I8);
137     }
138     else if (targetType == typeof(ulong))
139     {
140         generator.Emit(OpCodes.Conv_U8);
141     }
142 }
143
144 [MethodImpl(MethodImplOptions.AggressiveInlining)]
145 public static void CheckedConvert(this ILGenerator generator)
146 {
147     var sourceType = typeof(TSource);
148     var targetType = typeof(TTarget);
149     if (sourceType == targetType)
150     {
151         return;
152     }
153     if (targetType == typeof(short))
154     {
155         if (NumericType<TSource>.IsSigned)
156         {
157             generator.Emit(OpCodes.Conv_Ovf_I2);
158         }
159         else
160         {
161             generator.Emit(OpCodes.Conv_Ovf_I2_Un);
162         }
163     }
164     else if (targetType == typeof(ushort))
165     {
166         if (NumericType<TSource>.IsSigned)
167         {
168             generator.Emit(OpCodes.Conv_Ovf_U2);
169         }
170         else
171         {
172             generator.Emit(OpCodes.Conv_Ovf_U2_Un);
173         }
174     }
175     else if (targetType == typeof(sbyte))
176     {
177         if (NumericType<TSource>.IsSigned)
178         {
179             generator.Emit(OpCodes.Conv_Ovf_I1);
180         }
181         else
182         {
183             generator.Emit(OpCodes.Conv_Ovf_I1_Un);
184         }
185     }
186     else if (targetType == typeof(byte))
187     {

```

```

188         if (NumericType<TSource>.IsSigned)
189         {
190             generator.Emit(OpCodes.Conv_Ovf_U1);
191         }
192         else
193         {
194             generator.Emit(OpCodes.Conv_Ovf_U1_Un);
195         }
196     }
197     else if (targetType == typeof(int))
198     {
199         if (NumericType<TSource>.IsSigned)
200         {
201             generator.Emit(OpCodes.Conv_Ovf_I4);
202         }
203         else
204         {
205             generator.Emit(OpCodes.Conv_Ovf_I4_Un);
206         }
207     }
208     else if (targetType == typeof(uint))
209     {
210         if (NumericType<TSource>.IsSigned)
211         {
212             generator.Emit(OpCodes.Conv_Ovf_U4);
213         }
214         else
215         {
216             generator.Emit(OpCodes.Conv_Ovf_U4_Un);
217         }
218     }
219     else if (targetType == typeof(long))
220     {
221         if (NumericType<TSource>.IsSigned)
222         {
223             generator.Emit(OpCodes.Conv_Ovf_I8);
224         }
225         else
226         {
227             generator.Emit(OpCodes.Conv_Ovf_I8_Un);
228         }
229     }
230     else if (targetType == typeof(ulong))
231     {
232         if (NumericType<TSource>.IsSigned)
233         {
234             generator.Emit(OpCodes.Conv_Ovf_U8);
235         }
236         else
237         {
238             generator.Emit(OpCodes.Conv_Ovf_U8_Un);
239         }
240     }
241     else if (targetType == typeof(float))
242     {
243         if (NumericType<TSource>.IsSigned)
244         {
245             generator.Emit(OpCodes.Conv_R4);
246         }
247         else
248         {
249             generator.Emit(OpCodes.Conv_R_Un);
250         }
251     }
252     else if (targetType == typeof(double))
253     {
254         generator.Emit(OpCodes.Conv_R8);
255     }
256     else if (targetType == typeof(bool))
257     {
258         generator.ConvertToBoolean<TSource>();
259     }
260     else
261     {
262         throw new NotSupportedException();
263     }
264 }
265

```



```

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

268
269 [MethodImpl(MethodImplOptions.AggressiveInlining)]
270 public static void LoadConstant(this ILGenerator generator, float value) =>
    ↪ generator.Emit(OpCodes.Ldc_R4, value);

271
272 [MethodImpl(MethodImplOptions.AggressiveInlining)]
273 public static void LoadConstant(this ILGenerator generator, double value) =>
    ↪ generator.Emit(OpCodes.Ldc_R8, value);

274
275 [MethodImpl(MethodImplOptions.AggressiveInlining)]
276 public static void LoadConstant(this ILGenerator generator, ulong value) =>
    ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));

277
278 [MethodImpl(MethodImplOptions.AggressiveInlining)]
279 public static void LoadConstant(this ILGenerator generator, long value) =>
    ↪ generator.Emit(OpCodes.Ldc_I8, value);

280
281 [MethodImpl(MethodImplOptions.AggressiveInlining)]
282 public static void LoadConstant(this ILGenerator generator, uint value)
283 {
284     switch (value)
285     {
286         case uint.MaxValue:
287             generator.Emit(OpCodes.Ldc_I4_M1);
288             return;
289         case 0:
290             generator.Emit(OpCodes.Ldc_I4_0);
291             return;
292         case 1:
293             generator.Emit(OpCodes.Ldc_I4_1);
294             return;
295         case 2:
296             generator.Emit(OpCodes.Ldc_I4_2);
297             return;
298         case 3:
299             generator.Emit(OpCodes.Ldc_I4_3);
300             return;
301         case 4:
302             generator.Emit(OpCodes.Ldc_I4_4);
303             return;
304         case 5:
305             generator.Emit(OpCodes.Ldc_I4_5);
306             return;
307         case 6:
308             generator.Emit(OpCodes.Ldc_I4_6);
309             return;
310         case 7:
311             generator.Emit(OpCodes.Ldc_I4_7);
312             return;
313         case 8:
314             generator.Emit(OpCodes.Ldc_I4_8);
315             return;
316         default:
317             if (value <= sbyte.MaxValue)
318             {
319                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
320             }
321             else
322             {
323                 generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
324             }
325             return;
326     }
327 }

328
329 [MethodImpl(MethodImplOptions.AggressiveInlining)]
330 public static void LoadConstant(this ILGenerator generator, int value)
331 {
332     switch (value)
333     {
334         case -1:
335             generator.Emit(OpCodes.Ldc_I4_M1);
336             return;
337         case 0:
338             generator.Emit(OpCodes.Ldc_I4_0);
339             return;
340         case 1:

```

```

341         generator.Emit(OpCodes.Ldc_I4_1);
342         return;
343     case 2:
344         generator.Emit(OpCodes.Ldc_I4_2);
345         return;
346     case 3:
347         generator.Emit(OpCodes.Ldc_I4_3);
348         return;
349     case 4:
350         generator.Emit(OpCodes.Ldc_I4_4);
351         return;
352     case 5:
353         generator.Emit(OpCodes.Ldc_I4_5);
354         return;
355     case 6:
356         generator.Emit(OpCodes.Ldc_I4_6);
357         return;
358     case 7:
359         generator.Emit(OpCodes.Ldc_I4_7);
360         return;
361     case 8:
362         generator.Emit(OpCodes.Ldc_I4_8);
363         return;
364     default:
365         if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
366         {
367             generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
368         }
369         else
370         {
371             generator.Emit(OpCodes.Ldc_I4, value);
372         }
373         return;
374     }
375 }
376
377 [MethodImpl(MethodImplOptions.AggressiveInlining)]
378 public static void LoadConstant(this ILGenerator generator, short value) =>
379     ↪ generator.LoadConstant((int)value);
380
381 [MethodImpl(MethodImplOptions.AggressiveInlining)]
382 public static void LoadConstant(this ILGenerator generator, ushort value) =>
383     ↪ generator.LoadConstant((int)value);
384
385 [MethodImpl(MethodImplOptions.AggressiveInlining)]
386 public static void LoadConstant(this ILGenerator generator, sbyte value) =>
387     ↪ generator.LoadConstant((int)value);
388
389 [MethodImpl(MethodImplOptions.AggressiveInlining)]
390 public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
391     ↪ LoadConstantOne(generator, typeof(TConstant));
392
393 [MethodImpl(MethodImplOptions.AggressiveInlining)]
394 public static void LoadConstantOne(this ILGenerator generator, Type constantType)
395 {
396     if (constantType == typeof(float))
397     {
398         generator.LoadConstant(1F);
399     }
400     else if (constantType == typeof(double))
401     {
402         generator.LoadConstant(1D);
403     }
404     else if (constantType == typeof(long))
405     {
406         generator.LoadConstant(1L);
407     }
408     else if (constantType == typeof(ulong))
409     {
410         generator.LoadConstant(1UL);
411     }
412     else if (constantType == typeof(int))
413     {
414         generator.LoadConstant(1);
415     }
416 }

```

```

415     else if (constantType == typeof(uint))
416     {
417         generator.LoadConstant(1U);
418     }
419     else if (constantType == typeof(short))
420     {
421         generator.LoadConstant((short)1);
422     }
423     else if (constantType == typeof(ushort))
424     {
425         generator.LoadConstant((ushort)1);
426     }
427     else if (constantType == typeof(sbyte))
428     {
429         generator.LoadConstant((sbyte)1);
430     }
431     else if (constantType == typeof(byte))
432     {
433         generator.LoadConstant((byte)1);
434     }
435     else
436     {
437         throw new NotSupportedException();
438     }
439 }
440
441 [MethodImpl(MethodImplOptions.AggressiveInlining)]
442 public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
↳  constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
443
444 [MethodImpl(MethodImplOptions.AggressiveInlining)]
445 public static void LoadConstant(this ILGenerator generator, Type constantType, object
↳  constantValue)
446 {
447     constantValue = Convert.ChangeType(constantValue, constantType);
448     if (constantType == typeof(float))
449     {
450         generator.LoadConstant((float)constantValue);
451     }
452     else if (constantType == typeof(double))
453     {
454         generator.LoadConstant((double)constantValue);
455     }
456     else if (constantType == typeof(long))
457     {
458         generator.LoadConstant((long)constantValue);
459     }
460     else if (constantType == typeof(ulong))
461     {
462         generator.LoadConstant((ulong)constantValue);
463     }
464     else if (constantType == typeof(int))
465     {
466         generator.LoadConstant((int)constantValue);
467     }
468     else if (constantType == typeof(uint))
469     {
470         generator.LoadConstant((uint)constantValue);
471     }
472     else if (constantType == typeof(short))
473     {
474         generator.LoadConstant((short)constantValue);
475     }
476     else if (constantType == typeof(ushort))
477     {
478         generator.LoadConstant((ushort)constantValue);
479     }
480     else if (constantType == typeof(sbyte))
481     {
482         generator.LoadConstant((sbyte)constantValue);
483     }
484     else if (constantType == typeof(byte))
485     {
486         generator.LoadConstant((byte)constantValue);
487     }
488     else
489     {
490         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]);
    }
}

```

```

567     }
568 }
569
570 [MethodImpl(MethodImplOptions.AggressiveInlining)]
571 public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
572     ↪ generator.Emit(OpCodes.Starg, argumentIndex);
573
574 [MethodImpl(MethodImplOptions.AggressiveInlining)]
575 public static void CompareGreaterThan(this ILGenerator generator) =>
576     ↪ generator.Emit(OpCodes.Cgt);
577
578 [MethodImpl(MethodImplOptions.AggressiveInlining)]
579 public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
580     ↪ generator.Emit(OpCodes.Cgt_Un);
581
582 [MethodImpl(MethodImplOptions.AggressiveInlining)]
583 public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
584 {
585     if (isSigned)
586     {
587         generator.CompareGreaterThan();
588     }
589     else
590     {
591         generator.UnsignedCompareGreaterThan();
592     }
593 }
594
595 [MethodImpl(MethodImplOptions.AggressiveInlining)]
596 public static void CompareLessThan(this ILGenerator generator) =>
597     ↪ generator.Emit(OpCodes.Clt);
598
599 [MethodImpl(MethodImplOptions.AggressiveInlining)]
600 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
601     ↪ generator.Emit(OpCodes.Clt_Un);
602
603 [MethodImpl(MethodImplOptions.AggressiveInlining)]
604 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
605 {
606     if (isSigned)
607     {
608         generator.CompareLessThan();
609     }
610     else
611     {
612         generator.UnsignedCompareLessThan();
613     }
614 }
615
616 [MethodImpl(MethodImplOptions.AggressiveInlining)]
617 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
618     ↪ generator.Emit(OpCodes.Bge, label);
619
620 [MethodImpl(MethodImplOptions.AggressiveInlining)]
621 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
622     ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
623
624 [MethodImpl(MethodImplOptions.AggressiveInlining)]
625 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
626     ↪ Label label)
627 {
628     if (isSigned)
629     {
630         generator.BranchIfGreaterOrEqual(label);
631     }
632     else
633     {
634         generator.UnsignedBranchIfGreaterOrEqual(label);
635     }
636 }
637
638 [MethodImpl(MethodImplOptions.AggressiveInlining)]
639 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
640     ↪ generator.Emit(OpCodes.Ble, label);
641
642 [MethodImpl(MethodImplOptions.AggressiveInlining)]
643 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
644     ↪ => generator.Emit(OpCodes.Ble_Un, label);

```

```

635 [MethodImpl(MethodImplOptions.AggressiveInlining)]
636 public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
637     ↳ label)
638 {
639     if (isSigned)
640     {
641         generator.BranchIfLessOrEqual(label);
642     }
643     else
644     {
645         generator.UnsignedBranchIfLessOrEqual(label);
646     }
647 }
648
649 [MethodImpl(MethodImplOptions.AggressiveInlining)]
650 public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
651
652 [MethodImpl(MethodImplOptions.AggressiveInlining)]
653 public static void Box(this ILGenerator generator, Type boxedType) =>
654     ↳ generator.Emit(OpCodes.Box, boxedType);
655
656 [MethodImpl(MethodImplOptions.AggressiveInlining)]
657 public static void Call(this ILGenerator generator, MethodInfo method) =>
658     ↳ generator.Emit(OpCodes.Call, method);
659
660 [MethodImpl(MethodImplOptions.AggressiveInlining)]
661 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
662
663 [MethodImpl(MethodImplOptions.AggressiveInlining)]
664 public static void Unbox<TUnbox>(this ILGenerator generator) =>
665     ↳ generator.Unbox(typeof(TUnbox));
666
667 [MethodImpl(MethodImplOptions.AggressiveInlining)]
668 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
669     ↳ generator.Emit(OpCodes.Unbox, typeToUnbox);
670
671 [MethodImpl(MethodImplOptions.AggressiveInlining)]
672 public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
673     ↳ generator.UnboxValue(typeof(TUnbox));
674
675 [MethodImpl(MethodImplOptions.AggressiveInlining)]
676 public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
677     ↳ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
678
679 [MethodImpl(MethodImplOptions.AggressiveInlining)]
680 public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
681     ↳ generator.DeclareLocal(typeof(T));
682
683 [MethodImpl(MethodImplOptions.AggressiveInlining)]
684 public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
685     ↳ generator.Emit(OpCodes.Ldloc, local);
686
687 [MethodImpl(MethodImplOptions.AggressiveInlining)]
688 public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
689     ↳ generator.Emit(OpCodes.Stloc, local);
690
691 [MethodImpl(MethodImplOptions.AggressiveInlining)]
692 public static void NewObject(this ILGenerator generator, Type type, params Type[]
693     ↳ parameterTypes)
694 {
695     var allConstructors = type.GetConstructors(BindingFlags.Public |
696     ↳ BindingFlags.NonPublic | BindingFlags.Instance
697     | BindingFlags.CreateInstance
698     );
699     var constructor = allConstructors.Where(c => c.GetParameters().Length ==
700     ↳ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
701     ↳ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
702     if (constructor == null)
703     {
704         throw new InvalidOperationException("Type " + type + " must have a constructor
705         ↳ that matches parameters [" + string.Join(", ",
706         ↳ parameterTypes.AsEnumerable()) + "]");
707     }
708     generator.NewObject(constructor);

```

```

696 }
697
698 [MethodImpl(MethodImplOptions.AggressiveInlining)]
699 public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
    ↪ generator.Emit(OpCodes.Newobj, constructor);
700
701 [MethodImpl(MethodImplOptions.AggressiveInlining)]
702 public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
    ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
703
704 [MethodImpl(MethodImplOptions.AggressiveInlining)]
705 public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
    ↪ false, byte? unaligned = null)
706 {
707     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
708     {
709         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
710     }
711     if (isVolatile)
712     {
713         generator.Emit(OpCodes.Volatile);
714     }
715     if (unaligned.HasValue)
716     {
717         generator.Emit(OpCodes.Unaligned, unaligned.Value);
718     }
719     if (type.IsPointer)
720     {
721         generator.Emit(OpCodes.Ldind_I);
722     }
723     else if (!type.IsValueType)
724     {
725         generator.Emit(OpCodes.Ldind_Ref);
726     }
727     else if (type == typeof(sbyte))
728     {
729         generator.Emit(OpCodes.Ldind_I1);
730     }
731     else if (type == typeof(bool))
732     {
733         generator.Emit(OpCodes.Ldind_I1);
734     }
735     else if (type == typeof(byte))
736     {
737         generator.Emit(OpCodes.Ldind_U1);
738     }
739     else if (type == typeof(short))
740     {
741         generator.Emit(OpCodes.Ldind_I2);
742     }
743     else if (type == typeof(ushort))
744     {
745         generator.Emit(OpCodes.Ldind_U2);
746     }
747     else if (type == typeof(char))
748     {
749         generator.Emit(OpCodes.Ldind_U2);
750     }
751     else if (type == typeof(int))
752     {
753         generator.Emit(OpCodes.Ldind_I4);
754     }
755     else if (type == typeof(uint))
756     {
757         generator.Emit(OpCodes.Ldind_U4);
758     }
759     else if (type == typeof(long) || type == typeof(ulong))
760     {
761         generator.Emit(OpCodes.Ldind_I8);
762     }
763     else if (type == typeof(float))
764     {
765         generator.Emit(OpCodes.Ldind_R4);
766     }
767     else if (type == typeof(double))
768     {
769         generator.Emit(OpCodes.Ldind_R8);
770     }

```

```

771     else
772     {
773         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
774             ↪ ", LoadObject may be more appropriate");
775     }
776
777 [MethodImpl(MethodImplOptions.AggressiveInlining)]
778 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
779     ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
780
781 [MethodImpl(MethodImplOptions.AggressiveInlining)]
782 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
783     ↪ = false, byte? unaligned = null)
784 {
785     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
786     {
787         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
788     }
789     if (isVolatile)
790     {
791         generator.Emit(OpCodes.Volatile);
792     }
793     if (unaligned.HasValue)
794     {
795         generator.Emit(OpCodes.Unaligned, unaligned.Value);
796     }
797     if (type.IsPointer)
798     {
799         generator.Emit(OpCodes.Stind_I);
800     }
801     else if (!type.IsValueType)
802     {
803         generator.Emit(OpCodes.Stind_Ref);
804     }
805     else if (type == typeof(sbyte) || type == typeof(byte))
806     {
807         generator.Emit(OpCodes.Stind_I1);
808     }
809     else if (type == typeof(short) || type == typeof(ushort))
810     {
811         generator.Emit(OpCodes.Stind_I2);
812     }
813     else if (type == typeof(int) || type == typeof(uint))
814     {
815         generator.Emit(OpCodes.Stind_I4);
816     }
817     else if (type == typeof(long) || type == typeof(ulong))
818     {
819         generator.Emit(OpCodes.Stind_I8);
820     }
821     else if (type == typeof(float))
822     {
823         generator.Emit(OpCodes.Stind_R4);
824     }
825     else if (type == typeof(double))
826     {
827         generator.Emit(OpCodes.Stind_R8);
828     }
829     else
830     {
831         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
832             ↪ + ", StoreObject may be more appropriate");
833     }
834 }
835 }

```

## 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 BytesSize;
25         public static readonly int BitsSize;
26         public static readonly T MinValue;
27         public static readonly T MaxValue;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         static NumericType()
31         {
32             try
33             {
34                 var type = typeof(T);
35                 var isNullable = type.IsNullable();
36                 var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
37                 var canBeNumeric = underlyingType.CanBeNumeric();
38                 var isNumeric = underlyingType.IsNumeric();

```

```

39     var isSigned = underlyingType.IsSigned();
40     var isFloatPoint = underlyingType.IsFloatPoint();
41     var bytesSize = Marshal.SizeOf(underlyingType);
42     var bitsSize = bytesSize * 8;
43     GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
44     GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
45     ↪ out Type unsignedVersion);
46     Type = type;
47     IsNullable = isNullable;
48     UnderlyingType = underlyingType;
49     CanBeNumeric = canBeNumeric;
50     IsNumeric = isNumeric;
51     IsSigned = isSigned;
52     IsFloatPoint = isFloatPoint;
53     BytesSize = bytesSize;
54     BitsSize = bitsSize;
55     MinValue = minValue;
56     MaxValue = maxValue;
57     SignedVersion = signedVersion;
58     UnsignedVersion = unsignedVersion;
59 }
60 catch (Exception exception)
61 {
62     exception.Ignore();
63 }
64 }
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
67 {
68     if (type == typeof(bool))
69     {
70         minValue = (T)(object>false;
71         maxValue = (T)(object>true;
72     }
73     else
74     {
75         minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
76         maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
77     }
78 }
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
81 ↪ signedVersion, out Type unsignedVersion)
82 {
83     if (isSigned)
84     {
85         signedVersion = type;
86         unsignedVersion = type.GetUnsignedVersionOrNull();
87     }
88     else
89     {
90         signedVersion = type.GetSignedVersionOrNull();
91         unsignedVersion = type;
92     }
93 }
94 }
95 }

```

## 1.10 ./Platform.Reflection/PropertyInfoExtensions.cs

```

1  using System.Reflection;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Reflection
7  {
8      public static class PropertyInfoExtensions
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T GetStaticValue<T>(this PropertyInfo fieldInfo) =>
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 DefaultFinalVirtualMethodAttributes =
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 EmitFinalVirtualMethod<TDelegate>(this TypeBuilder type, string
51             ↪ methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
52             ↪ DefaultFinalVirtualMethodAttributes, 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)]

```

```

25 static TypeExtensions()
26 {
27     _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
        ↳ typeof(DateTime), typeof(TimeSpan) };
28     _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
        ↳ typeof(ulong) };
29     _canBeNumericTypes.UnionWith(_isNumericTypes);
30     _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
        ↳ typeof(long) };
31     _canBeNumericTypes.UnionWith(_isSignedTypes);
32     _isNumericTypes.UnionWith(_isSignedTypes);
33     _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
        ↳ typeof(float) };
34     _canBeNumericTypes.UnionWith(_isFloatPointTypes);
35     _isNumericTypes.UnionWith(_isFloatPointTypes);
36     _isSignedTypes.UnionWith(_isFloatPointTypes);
37     _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
38     {
39         { typeof(sbyte), typeof(byte) },
40         { typeof(short), typeof(ushort) },
41         { typeof(int), typeof(uint) },
42         { typeof(long), typeof(ulong) },
43     };
44     _signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
45     {
46         { typeof(byte), typeof(sbyte) },
47         { typeof(ushort), typeof(short) },
48         { typeof(uint), typeof(int) },
49         { typeof(ulong), typeof(long) },
50     };
51 }
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 public static T GetStaticFieldValue<T>(this Type type, string name) =>
    ↳ type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 public static T GetStaticPropertyValue<T>(this Type type, string name) =>
    ↳ type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
    ↳ genericParameterTypes, Type[] argumentTypes)
64 {
65     var methods = from m in type.GetMethods()
66                   where m.Name == name
67                       && m.IsGenericMethodDefinition
68                       let typeParams = m.GetGenericArguments()
69                       let normalParams = m.GetParameters().Select(x => x.ParameterType)
70                       where typeParams.SequenceEqual(genericParameterTypes)
71                       && normalParams.SequenceEqual(argumentTypes)
72                       select m;
73     var method = methods.Single();
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) =>
    ↳ type.IsSubclassOf(superClass);
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 public static bool IsValueType(this Type type) => type.IsValueType;
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 public static bool IsGeneric(this Type type) => type.IsGenericType;
91
92 [MethodImpl(MethodImplOptions.AggressiveInlining)]
93 public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
    ↳ type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;

```

```

94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     public static Type GetUnsignedVersionOrNull(this Type signedType) =>
99         ↳ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
100
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     public static Type GetSignedVersionOrNull(this Type unsignedType) =>
103         ↳ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
104
105     [MethodImpl(MethodImplOptions.AggressiveInlining)]
106     public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
110
111     [MethodImpl(MethodImplOptions.AggressiveInlining)]
112     public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
113
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
116
117     [MethodImpl(MethodImplOptions.AggressiveInlining)]
118     public static Type GetDelegateReturnType(this Type delegateType) =>
119         ↳ delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;
120
121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
123         ↳ delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();
124
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]
126     public static void GetDelegateCharacteristics(this Type delegateType, out Type
127         ↳ returnType, out Type[] parameterTypes)
128     {
129         var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
130         returnType = invoke.ReturnType;
131         parameterTypes = invoke.GetParameterTypes();
132     }
133 }

```

### 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
15             ↳ ReadOnlyCollection<Type>(System.Array.Empty<Type>());
16         public static Type[] Array => Collection.ToArray();
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected ReadOnlyCollection<Type> ToReadOnlyCollection()
20         {
21             var types = GetType().GetGenericArguments();
22             var result = new List<Type>();
23             AppendTypes(result, types);
24             return new ReadOnlyCollection<Type>(result);
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         private static void AppendTypes(List<Type> container, IList<Type> types)
29         {
30             for (var i = 0; i < types.Count; i++)
31             {
32                 var element = types[i];
33                 if (element != typeof(Types))
34                 {
35                     if (element.IsSubclassOf(typeof(Types)))

```

```

35         {
36             AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<
                ↳ <Type>>>(nameof(Types<object>.Collection)));
37         }
38     else
39     {
40         container.Add(element);
41     }
42 }
43 }
44 }
45 }
46 }

```

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

```

1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  #pragma warning disable CA1819 // Properties should not return arrays
7
8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
            ↳ T4, T5, T6, T7>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

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

```

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

```

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

```

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

```

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

```

1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  #pragma warning disable CA1819 // Properties should not return arrays

```

```

7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2, T3, T4> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
            ↪ T4>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

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

```

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

```

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

```

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,
            ↪ T2>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

#### 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.Equal(withSignExtension[i], withoutSignExtension[i++]);
82     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
83     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
84     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
85     Assert.Equal(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, 25
- ./Platform.Reflection.Tests/NumericTypeTests.cs, 25
- ./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, 17
- ./Platform.Reflection/NumericType.cs, 17
- ./Platform.Reflection/PropertyInfoExtensions.cs, 18
- ./Platform.Reflection/TypeBuilderExtensions.cs, 18
- ./Platform.Reflection/TypeExtensions.cs, 19
- ./Platform.Reflection/Types.cs, 21
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3, T4].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3].cs, 23
- ./Platform.Reflection/Types[T1, T2].cs, 23
- ./Platform.Reflection/Types[T].cs, 23