

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>
        ↳ 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 }
86
87 private static void ConvertToInteger(this ILGenerator generator, Type targetType)
88 {
89     if (targetType == typeof(sbyte))
90     {
91         generator.Emit(OpCodes.Conv_I1);
92     }
93     else if (targetType == typeof(byte))
94     {
95         generator.Emit(OpCodes.Conv_U1);
96     }
97     else if (targetType == typeof(short))
98     {
99         generator.Emit(OpCodes.Conv_I2);
100     }
101     else if (targetType == typeof(ushort))
102     {
103         generator.Emit(OpCodes.Conv_U2);
104     }
105     else if (targetType == typeof(int))
106     {
107         generator.Emit(OpCodes.Conv_I4);
108     }
109     else if (targetType == typeof(uint))

```

```

110     {
111         generator.Emit(OpCodes.Conv_U4);
112     }
113     else if (targetType == typeof(long))
114     {
115         generator.Emit(OpCodes.Conv_I8);
116     }
117     else if (targetType == typeof(ulong))
118     {
119         generator.Emit(OpCodes.Conv_U8);
120     }
121 }
122
123 [MethodImpl(MethodImplOptions.AggressiveInlining)]
124 public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
125 {
126     var sourceType = typeof(TSource);
127     var targetType = typeof(TTarget);
128     if (sourceType == targetType)
129     {
130         return;
131     }
132     if (targetType == typeof(short))
133     {
134         if (NumericType<TSource>.IsSigned)
135         {
136             generator.Emit(OpCodes.Conv_Ovf_I2);
137         }
138         else
139         {
140             generator.Emit(OpCodes.Conv_Ovf_I2_Un);
141         }
142     }
143     else if (targetType == typeof(ushort))
144     {
145         if (NumericType<TSource>.IsSigned)
146         {
147             generator.Emit(OpCodes.Conv_Ovf_U2);
148         }
149         else
150         {
151             generator.Emit(OpCodes.Conv_Ovf_U2_Un);
152         }
153     }
154     else if (targetType == typeof(sbyte))
155     {
156         if (NumericType<TSource>.IsSigned)
157         {
158             generator.Emit(OpCodes.Conv_Ovf_I1);
159         }
160         else
161         {
162             generator.Emit(OpCodes.Conv_Ovf_I1_Un);
163         }
164     }
165     else if (targetType == typeof(byte))
166     {
167         if (NumericType<TSource>.IsSigned)
168         {
169             generator.Emit(OpCodes.Conv_Ovf_U1);
170         }
171         else
172         {
173             generator.Emit(OpCodes.Conv_Ovf_U1_Un);
174         }
175     }
176     else if (targetType == typeof(int))
177     {
178         if (NumericType<TSource>.IsSigned)
179         {
180             generator.Emit(OpCodes.Conv_Ovf_I4);
181         }
182         else
183         {
184             generator.Emit(OpCodes.Conv_Ovf_I4_Un);
185         }
186     }
187     else if (targetType == typeof(uint))

```

```

188     {
189         if (NumericType<TSource>.IsSigned)
190         {
191             generator.Emit(OpCodes.Conv_Ovf_U4);
192         }
193         else
194         {
195             generator.Emit(OpCodes.Conv_Ovf_U4_Un);
196         }
197     }
198     else if (targetType == typeof(long))
199     {
200         if (NumericType<TSource>.IsSigned)
201         {
202             generator.Emit(OpCodes.Conv_Ovf_I8);
203         }
204         else
205         {
206             generator.Emit(OpCodes.Conv_Ovf_I8_Un);
207         }
208     }
209     else if (targetType == typeof(ulong))
210     {
211         if (NumericType<TSource>.IsSigned)
212         {
213             generator.Emit(OpCodes.Conv_Ovf_U8);
214         }
215         else
216         {
217             generator.Emit(OpCodes.Conv_Ovf_U8_Un);
218         }
219     }
220     else if (targetType == typeof(float))
221     {
222         if (NumericType<TSource>.IsSigned)
223         {
224             generator.Emit(OpCodes.Conv_R4);
225         }
226         else
227         {
228             generator.Emit(OpCodes.Conv_R_Un);
229         }
230     }
231     else if (targetType == typeof(double))
232     {
233         generator.Emit(OpCodes.Conv_R8);
234     }
235     else
236     {
237         throw new NotSupportedException();
238     }
239 }
240
241 [MethodImpl(MethodImplOptions.AggressiveInlining)]
242 public static void LoadConstant(this ILGenerator generator, bool value) =>
243     ↪ generator.LoadConstant(value ? 1 : 0);
244
245 [MethodImpl(MethodImplOptions.AggressiveInlining)]
246 public static void LoadConstant(this ILGenerator generator, float value) =>
247     ↪ generator.Emit(OpCodes.Ldc_R4, value);
248
249 [MethodImpl(MethodImplOptions.AggressiveInlining)]
250 public static void LoadConstant(this ILGenerator generator, double value) =>
251     ↪ generator.Emit(OpCodes.Ldc_R8, value);
252
253 [MethodImpl(MethodImplOptions.AggressiveInlining)]
254 public static void LoadConstant(this ILGenerator generator, ulong value) =>
255     ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
256
257 [MethodImpl(MethodImplOptions.AggressiveInlining)]
258 public static void LoadConstant(this ILGenerator generator, long value) =>
259     ↪ generator.Emit(OpCodes.Ldc_I8, value);
260
261 [MethodImpl(MethodImplOptions.AggressiveInlining)]
262 public static void LoadConstant(this ILGenerator generator, uint value)
263 {
264     switch (value)
265     {

```



```

261         case uint.MaxValue:
262             generator.Emit(OpCodes.Ldc_I4_M1);
263             return;
264         case 0:
265             generator.Emit(OpCodes.Ldc_I4_0);
266             return;
267         case 1:
268             generator.Emit(OpCodes.Ldc_I4_1);
269             return;
270         case 2:
271             generator.Emit(OpCodes.Ldc_I4_2);
272             return;
273         case 3:
274             generator.Emit(OpCodes.Ldc_I4_3);
275             return;
276         case 4:
277             generator.Emit(OpCodes.Ldc_I4_4);
278             return;
279         case 5:
280             generator.Emit(OpCodes.Ldc_I4_5);
281             return;
282         case 6:
283             generator.Emit(OpCodes.Ldc_I4_6);
284             return;
285         case 7:
286             generator.Emit(OpCodes.Ldc_I4_7);
287             return;
288         case 8:
289             generator.Emit(OpCodes.Ldc_I4_8);
290             return;
291         default:
292             if (value <= sbyte.MaxValue)
293             {
294                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
295             }
296             else
297             {
298                 generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
299             }
300             return;
301     }
302 }
303
304 [MethodImpl(MethodImplOptions.AggressiveInlining)]
305 public static void LoadConstant(this ILGenerator generator, int value)
306 {
307     switch (value)
308     {
309         case -1:
310             generator.Emit(OpCodes.Ldc_I4_M1);
311             return;
312         case 0:
313             generator.Emit(OpCodes.Ldc_I4_0);
314             return;
315         case 1:
316             generator.Emit(OpCodes.Ldc_I4_1);
317             return;
318         case 2:
319             generator.Emit(OpCodes.Ldc_I4_2);
320             return;
321         case 3:
322             generator.Emit(OpCodes.Ldc_I4_3);
323             return;
324         case 4:
325             generator.Emit(OpCodes.Ldc_I4_4);
326             return;
327         case 5:
328             generator.Emit(OpCodes.Ldc_I4_5);
329             return;
330         case 6:
331             generator.Emit(OpCodes.Ldc_I4_6);
332             return;
333         case 7:
334             generator.Emit(OpCodes.Ldc_I4_7);
335             return;
336         case 8:
337             generator.Emit(OpCodes.Ldc_I4_8);
338             return;
339         default:
340             if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
341             {

```

```

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

```

```

415 [MethodImpl(MethodImplOptions.AggressiveInlining)]
416 public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
417     ↪ constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
418
419 [MethodImpl(MethodImplOptions.AggressiveInlining)]
420 public static void LoadConstant(this ILGenerator generator, Type constantType, object
421     ↪ constantValue)
422 {
423     constantValue = Convert.ChangeType(constantValue, constantType);
424     if (constantType == typeof(float))
425     {
426         generator.LoadConstant((float)constantValue);
427     }
428     else if (constantType == typeof(double))
429     {
430         generator.LoadConstant((double)constantValue);
431     }
432     else if (constantType == typeof(long))
433     {
434         generator.LoadConstant((long)constantValue);
435     }
436     else if (constantType == typeof(ulong))
437     {
438         generator.LoadConstant((ulong)constantValue);
439     }
440     else if (constantType == typeof(int))
441     {
442         generator.LoadConstant((int)constantValue);
443     }
444     else if (constantType == typeof(uint))
445     {
446         generator.LoadConstant((uint)constantValue);
447     }
448     else if (constantType == typeof(short))
449     {
450         generator.LoadConstant((short)constantValue);
451     }
452     else if (constantType == typeof(ushort))
453     {
454         generator.LoadConstant((ushort)constantValue);
455     }
456     else if (constantType == typeof(sbyte))
457     {
458         generator.LoadConstant((sbyte)constantValue);
459     }
460     else if (constantType == typeof(byte))
461     {
462         generator.LoadConstant((byte)constantValue);
463     }
464     else
465     {
466         throw new NotSupportedException();
467     }
468 }
469
470 [MethodImpl(MethodImplOptions.AggressiveInlining)]
471 public static void Increment<TValue>(this ILGenerator generator) =>
472     ↪ generator.Increment(typeof(TValue));
473
474 [MethodImpl(MethodImplOptions.AggressiveInlining)]
475 public static void Decrement<TValue>(this ILGenerator generator) =>
476     ↪ generator.Decrement(typeof(TValue));
477
478 [MethodImpl(MethodImplOptions.AggressiveInlining)]
479 public static void Increment(this ILGenerator generator, Type valueType)
480 {
481     generator.LoadConstantOne(valueType);
482     generator.Add();
483 }
484
485 [MethodImpl(MethodImplOptions.AggressiveInlining)]
486 public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 public static void Decrement(this ILGenerator generator, Type valueType)
490 {
491     generator.LoadConstantOne(valueType);
492     generator.Sub();
493 }

```

```

489     generator.Subtract();
490 }
491
492 [MethodImpl(MethodImplOptions.AggressiveInlining)]
493 public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
494
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
497
498 [MethodImpl(MethodImplOptions.AggressiveInlining)]
499 public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
500
501 [MethodImpl(MethodImplOptions.AggressiveInlining)]
502 public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
503
504 [MethodImpl(MethodImplOptions.AggressiveInlining)]
505 public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
506
507 [MethodImpl(MethodImplOptions.AggressiveInlining)]
508 public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
509
510 [MethodImpl(MethodImplOptions.AggressiveInlining)]
511 public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
512
513 [MethodImpl(MethodImplOptions.AggressiveInlining)]
514 public static void LoadArgument(this ILGenerator generator, int argumentIndex)
515 {
516     switch (argumentIndex)
517     {
518         case 0:
519             generator.Emit(OpCodes.Ldarg_0);
520             break;
521         case 1:
522             generator.Emit(OpCodes.Ldarg_1);
523             break;
524         case 2:
525             generator.Emit(OpCodes.Ldarg_2);
526             break;
527         case 3:
528             generator.Emit(OpCodes.Ldarg_3);
529             break;
530         default:
531             generator.Emit(OpCodes.Ldarg, argumentIndex);
532             break;
533     }
534 }
535
536 [MethodImpl(MethodImplOptions.AggressiveInlining)]
537 public static void LoadArguments(this ILGenerator generator, params int[]
538     ↪ argumentIndices)
539 {
540     for (var i = 0; i < argumentIndices.Length; i++)
541     {
542         generator.LoadArgument(argumentIndices[i]);
543     }
544 }
545
546 [MethodImpl(MethodImplOptions.AggressiveInlining)]
547 public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
548     ↪ generator.Emit(OpCodes.Starg, argumentIndex);
549
550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
551 public static void CompareGreaterThan(this ILGenerator generator) =>
552     ↪ generator.Emit(OpCodes.Cgt);
553
554 [MethodImpl(MethodImplOptions.AggressiveInlining)]
555 public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
556     ↪ generator.Emit(OpCodes.Cgt_Un);
557
558 [MethodImpl(MethodImplOptions.AggressiveInlining)]
559 public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
560 {
561     if (isSigned)
562     {
563         generator.CompareGreaterThan();
564     }
565     else
566     {
567         generator.UnsignedCompareGreaterThan();
568     }
569 }

```

```

564     }
565 }
566
567 [MethodImpl(MethodImplOptions.AggressiveInlining)]
568 public static void CompareLessThan(this ILGenerator generator) =>
569     ↪ generator.Emit(OpCodes.Clt);
570
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
573     ↪ generator.Emit(OpCodes.Clt_Un);
574
575 [MethodImpl(MethodImplOptions.AggressiveInlining)]
576 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
577 {
578     if (isSigned)
579     {
580         generator.CompareLessThan();
581     }
582     else
583     {
584         generator.UnsignedCompareLessThan();
585     }
586 }
587
588 [MethodImpl(MethodImplOptions.AggressiveInlining)]
589 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
590     ↪ generator.Emit(OpCodes.Bge, label);
591
592 [MethodImpl(MethodImplOptions.AggressiveInlining)]
593 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
594     ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
595
596 [MethodImpl(MethodImplOptions.AggressiveInlining)]
597 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
598     ↪ Label label)
599 {
600     if (isSigned)
601     {
602         generator.BranchIfGreaterOrEqual(label);
603     }
604     else
605     {
606         generator.UnsignedBranchIfGreaterOrEqual(label);
607     }
608 }
609
610 [MethodImpl(MethodImplOptions.AggressiveInlining)]
611 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
612     ↪ generator.Emit(OpCodes.Ble, label);
613
614 [MethodImpl(MethodImplOptions.AggressiveInlining)]
615 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
616     ↪ => generator.Emit(OpCodes.Ble_Un, label);
617
618 [MethodImpl(MethodImplOptions.AggressiveInlining)]
619 public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
620     ↪ label)
621 {
622     if (isSigned)
623     {
624         generator.BranchIfLessOrEqual(label);
625     }
626     else
627     {
628         generator.UnsignedBranchIfLessOrEqual(label);
629     }
630 }
631
632 [MethodImpl(MethodImplOptions.AggressiveInlining)]
633 public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
634
635 [MethodImpl(MethodImplOptions.AggressiveInlining)]
636 public static void Box(this ILGenerator generator, Type boxedType) =>
637     ↪ generator.Emit(OpCodes.Box, boxedType);
638
639 [MethodImpl(MethodImplOptions.AggressiveInlining)]
640 public static void Call(this ILGenerator generator, MethodInfo method) =>
641     ↪ generator.Emit(OpCodes.Call, method);

```

```

632 [MethodImpl(MethodImplOptions.AggressiveInlining)]
633 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
634
635 [MethodImpl(MethodImplOptions.AggressiveInlining)]
636 public static void Unbox<TUnbox>(this ILGenerator generator) =>
637     ↪ generator.Unbox(typeof(TUnbox));
638
639 [MethodImpl(MethodImplOptions.AggressiveInlining)]
640 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
641     ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
642
643 [MethodImpl(MethodImplOptions.AggressiveInlining)]
644 public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
645     ↪ generator.UnboxValue(typeof(TUnbox));
646
647 [MethodImpl(MethodImplOptions.AggressiveInlining)]
648 public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
649     ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
650
651 [MethodImpl(MethodImplOptions.AggressiveInlining)]
652 public static void DeclareLocal<T>(this ILGenerator generator) =>
653     ↪ generator.DeclareLocal(typeof(T));
654
655 [MethodImpl(MethodImplOptions.AggressiveInlining)]
656 public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
657     ↪ generator.Emit(OpCodes.Ldloc, local);
658
659 [MethodImpl(MethodImplOptions.AggressiveInlining)]
660 public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
661     ↪ generator.Emit(OpCodes.Stloc, local);
662
663 [MethodImpl(MethodImplOptions.AggressiveInlining)]
664 public static void NewObject(this ILGenerator generator, Type type, params Type[]
665     ↪ parameterTypes)
666 {
667     var allConstructors = type.GetConstructors(BindingFlags.Public |
668     ↪ BindingFlags.NonPublic | BindingFlags.Instance
669     | BindingFlags.CreateInstance
670     );
671     var constructor = allConstructors.Where(c => c.GetParameters().Length ==
672     ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
673     ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
674     if (constructor == null)
675     {
676         throw new InvalidOperationException("Type " + type + " must have a constructor
677         ↪ that matches parameters [" + string.Join(", ",
678         ↪ parameterTypes.AsEnumerable()) + "]");
679     }
680     generator.NewObject(constructor);
681 }
682
683 [MethodImpl(MethodImplOptions.AggressiveInlining)]
684 public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
685     ↪ generator.Emit(OpCodes.Newobj, constructor);
686
687 [MethodImpl(MethodImplOptions.AggressiveInlining)]
688 public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
689     ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
690
691 [MethodImpl(MethodImplOptions.AggressiveInlining)]
692 public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
693     ↪ false, byte? unaligned = null)
694 {
695     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
696     {
697         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
698     }
699     if (isVolatile)
700     {
701         generator.Emit(OpCodes.Volatile);
702     }
703     if (unaligned.HasValue)
704     {
705         generator.Emit(OpCodes.Unaligned, unaligned.Value);
706     }
707 }

```

```

693     }
694     if (type.IsPointer)
695     {
696         generator.Emit(OpCodes.Ldind_I);
697     }
698     else if (!type.IsValueType)
699     {
700         generator.Emit(OpCodes.Ldind_Ref);
701     }
702     else if (type == typeof(sbyte))
703     {
704         generator.Emit(OpCodes.Ldind_I1);
705     }
706     else if (type == typeof(bool))
707     {
708         generator.Emit(OpCodes.Ldind_I1);
709     }
710     else if (type == typeof(byte))
711     {
712         generator.Emit(OpCodes.Ldind_U1);
713     }
714     else if (type == typeof(short))
715     {
716         generator.Emit(OpCodes.Ldind_I2);
717     }
718     else if (type == typeof(ushort))
719     {
720         generator.Emit(OpCodes.Ldind_U2);
721     }
722     else if (type == typeof(char))
723     {
724         generator.Emit(OpCodes.Ldind_U2);
725     }
726     else if (type == typeof(int))
727     {
728         generator.Emit(OpCodes.Ldind_I4);
729     }
730     else if (type == typeof(uint))
731     {
732         generator.Emit(OpCodes.Ldind_U4);
733     }
734     else if (type == typeof(long) || type == typeof(ulong))
735     {
736         generator.Emit(OpCodes.Ldind_I8);
737     }
738     else if (type == typeof(float))
739     {
740         generator.Emit(OpCodes.Ldind_R4);
741     }
742     else if (type == typeof(double))
743     {
744         generator.Emit(OpCodes.Ldind_R8);
745     }
746     else
747     {
748         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
749             ↪ ", LoadObject may be more appropriate");
750     }
751 }
752 [MethodImpl(MethodImplOptions.AggressiveInlining)]
753 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
754     ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
755 [MethodImpl(MethodImplOptions.AggressiveInlining)]
756 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
757     ↪ = false, byte? unaligned = null)
758 {
759     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
760     {
761         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
762     }
763     if (isVolatile)
764     {
765         generator.Emit(OpCodes.Volatile);
766     }
767     if (unaligned.HasValue)
768     {

```

```

768         generator.Emit(OpCodes.Unaligned, unaligned.Value);
769     }
770     if (type.IsPointer)
771     {
772         generator.Emit(OpCodes.Stind_I);
773     }
774     else if (!type.IsValueType)
775     {
776         generator.Emit(OpCodes.Stind_Ref);
777     }
778     else if (type == typeof(sbyte) || type == typeof(byte))
779     {
780         generator.Emit(OpCodes.Stind_I1);
781     }
782     else if (type == typeof(short) || type == typeof(ushort))
783     {
784         generator.Emit(OpCodes.Stind_I2);
785     }
786     else if (type == typeof(int) || type == typeof(uint))
787     {
788         generator.Emit(OpCodes.Stind_I4);
789     }
790     else if (type == typeof(long) || type == typeof(ulong))
791     {
792         generator.Emit(OpCodes.Stind_I8);
793     }
794     else if (type == typeof(float))
795     {
796         generator.Emit(OpCodes.Stind_R4);
797     }
798     else if (type == typeof(double))
799     {
800         generator.Emit(OpCodes.Stind_R8);
801     }
802     else
803     {
804         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
            ↪ + ", StoreObject may be more appropriate");
805     }
806 }
807 }
808 }

```

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) =>
            ↪ methodInfo.GetMethodBody().GetILAsByteArray();
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
            ↪ methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
17     }
18 }

```

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     }
66 }

```

```

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

```

1.10 ./Platform.Reflection/PropertyInfoExtensions.cs

```

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

```

1.11 ./Platform.Reflection/TypeBuilderExtensions.cs

```

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

```

```

21     }
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
        ↪ methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[]
        ↪ parameterTypes, Action<ILGenerator> emitCode)
25     {
26         MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
        ↪ parameterTypes);
27         method.SetImplementationFlags(methodImplAttributes);
28         var generator = method.GetILGenerator();
29         emitCode(generator);
30     }
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName,
        ↪ Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
        ↪ DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public static void EmitFinalVirtualMethod<TDelegate>(this TypeBuilder type, string
        ↪ methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
        ↪ DefaultFinalVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
37 }
38 }

```

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 |
        ↪ BindingFlags.NonPublic | BindingFlags.Static;
15         static public readonly string DefaultDelegateMethodName = "Invoke";
16
17         static private readonly HashSet<Type> _canBeNumericTypes;
18         static private readonly HashSet<Type> _isNumericTypes;
19         static private readonly HashSet<Type> _isSignedTypes;
20         static private readonly HashSet<Type> _isFloatPointTypes;
21         static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
22         static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;
23
24         [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) =>
58     ↪ type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public static T GetStaticPropertyValue<T>(this Type type, string name) =>
62     ↪ type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
66     ↪ genericParameterTypes, Type[] argumentTypes)
67 {
68     var methods = from m in type.GetMethods()
69                     where m.Name == name
70                         && m.IsGenericMethodDefinition
71                         let typeParams = m.GetGenericArguments()
72                         let normalParams = m.GetParameters().Select(x => x.ParameterType)
73                         where typeParams.SequenceEqual(genericParameterTypes)
74                             && normalParams.SequenceEqual(argumentTypes)
75                     select m;
76     var method = methods.Single();
77     return method;
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public static Type GetBaseType(this Type type) => type.BaseType;
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 public static Assembly GetAssembly(this Type type) => type.Assembly;
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 public static bool IsSubclassOf(this Type type, Type superClass) =>
88     ↪ type.IsSubclassOf(superClass);
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public static bool IsValueType(this Type type) => type.IsValueType;
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 public static bool IsGeneric(this Type type) => type.IsGenericType;
95
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
98     ↪ type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public static Type GetUnsignedVersionOrNull(this Type signedType) =>
105     ↪ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 public static Type GetSignedVersionOrNull(this Type unsignedType) =>
109     ↪ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
110
111 [MethodImpl(MethodImplOptions.AggressiveInlining)]
112 public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
119
120 [MethodImpl(MethodImplOptions.AggressiveInlining)]
121 public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
122
123 [MethodImpl(MethodImplOptions.AggressiveInlining)]
124 public static Type GetDelegateReturnType(this Type delegateType) =>
125     ↪ delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;

```

```

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

```

1.13 ./Platform.Reflection/Types.cs

```

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

```

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

```

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

```

```

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,
13             ↪ T4, T5, T6>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

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

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.Return(value);
42             });
43             Assert.Equal(value, compiledFunction());
44         }
45     }
46 }
```

```

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,
90     ↪ TTarget>(bool extendSign)
91 {
92     return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
93     {
94         generator.LoadArgument(0);
95         generator.UncheckedConvert<TSource, TTarget>(extendSign);
96         generator.Return();
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 }

```



```

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, 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, 16
- ./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, 21
- ./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, 22
- ./Platform.Reflection/Types[T1, T2].cs, 23
- ./Platform.Reflection/Types[T].cs, 23