

LinksPlatform's Platform.Reflection Class Library

./Platform.Reflection/AssemblyExtensions.cs

```
1 using System;
2 using System.Collections.Concurrent;
3 using System.Reflection;
4 using Platform.Exceptions;
5 using Platform.Collections.Lists;
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 AssemblyExtensions
12     {
13         private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
14             ↳ ConcurrentDictionary<Assembly, Type[]>();
15
16         /// <remarks>
17         /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
18         /// </remarks>
19         public static Type[] GetLoadableTypes(this Assembly assembly)
20         {
21             Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
22             try
23             {
24                 return assembly.GetTypes();
25             }
26             catch (ReflectionTypeLoadException e)
27             {
28                 return e.Types.ToArray(t => t != null);
29             }
30         }
31
32         public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
33             ↳ _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
34     }
35 }
```

./Platform.Reflection/DelegateHelpers.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Reflection.Emit;
5 using Platform.Exceptions;
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 DelegateHelpers
12     {
13         public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode)
14             where TDelegate : Delegate
15         {
16             var @delegate = default(TDelegate);
17             try
18             {
19                 var delegateType = typeof(TDelegate);
20                 var invoke = delegateType.GetMethod("Invoke");
21                 var returnType = invoke.ReturnType;
22                 var parameterTypes = invoke.GetParameters().Select(s =>
23                     ↳ s.ParameterType).ToArray();
24                 var dynamicMethod = new DynamicMethod(Guid.NewGuid().ToString(), returnType,
25                     ↳ parameterTypes);
26                 var generator = dynamicMethod.GetILGenerator();
27                 emitCode(generator);
28                 @delegate = (TDelegate)dynamicMethod.CreateDelegate(delegateType);
29             }
30             catch (Exception exception)
31             {
32                 exception.Ignore();
33             }
34             return @delegate;
35         }
36
37         public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode)
38             where TDelegate : Delegate
39         {
40             var @delegate = CompileOrDefault<TDelegate>(emitCode);
41         }
42     }
43 }
```

```

39         if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
40         {
41             @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
42         }
43         return @delegate;
44     }
45 }
46 }

```

./Platform.Reflection/DynamicExtensions.cs

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Reflection
6  {
7      public static class DynamicExtensions
8      {
9          public static bool HasProperty(this object @object, string propertyName)
10         {
11             var type = @object.GetType();
12             if (type is IDictionary<string, object> dictionary)
13             {
14                 return dictionary.ContainsKey(propertyName);
15             }
16             return type.GetProperty(propertyName) != null;
17         }
18     }
19 }

```

./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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
28                 message)
29             {
30                 string messageBuilder() => message;
31                 IsUnsignedInteger<T>(root, messageBuilder());
32             }
33
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
36                 IsUnsignedInteger<T>(root, (string)null);
37
38             [MethodImpl(MethodImplOptions.AggressiveInlining)]
39             public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
40                 messageBuilder)
41             {
42                 if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
43                     NumericType<T>.IsFloatPoint)
44                 {
45                     throw new NotSupportedException(messageBuilder());
46                 }
47             }
48         }
49     }
50 }

```

```

42     }
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
    ↳ message)
46     {
47         string messageBuilder() => message;
48         IsSignedInteger<T>(root, messageBuilder);
49     }
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
    ↳ IsSignedInteger<T>(root, (string)null);
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
    ↳ messageBuilder)
56     {
57         if (!NumericType<T>.IsSigned)
58         {
59             throw new NotSupportedException(messageBuilder());
60         }
61     }
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
65     {
66         string messageBuilder() => message;
67         IsSigned<T>(root, messageBuilder);
68     }
69
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
    ↳ (string)null);
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
    ↳ messageBuilder)
75     {
76         if (!NumericType<T>.IsNumeric)
77         {
78             throw new NotSupportedException(messageBuilder());
79         }
80     }
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
84     {
85         string messageBuilder() => message;
86         IsNumeric<T>(root, messageBuilder);
87     }
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
    ↳ IsNumeric<T>(root, (string)null);
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
    ↳ messageBuilder)
94     {
95         if (!NumericType<T>.CanBeNumeric)
96         {
97             throw new NotSupportedException(messageBuilder());
98         }
99     }
100
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
103     {
104         string messageBuilder() => message;
105         CanBeNumeric<T>(root, messageBuilder);
106     }
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
    ↳ CanBeNumeric<T>(root, (string)null);
110
111     #endregion

```

```

112 #region OnDebug
113
114 [Conditional("DEBUG")]
115 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
116     ↳ Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
117
118 [Conditional("DEBUG")]
119 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
120     ↳ message) => Ensure.Always.IsUnsignedInteger<T>(message);
121
122 [Conditional("DEBUG")]
123 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
124     ↳ Ensure.Always.IsUnsignedInteger<T>();
125
126 [Conditional("DEBUG")]
127 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
128     ↳ messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
129
130 [Conditional("DEBUG")]
131 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
132     ↳ message) => Ensure.Always.IsSignedInteger<T>(message);
133
134 [Conditional("DEBUG")]
135 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
136     ↳ Ensure.Always.IsSignedInteger<T>();
137
138 [Conditional("DEBUG")]
139 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
140     ↳ messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
141
142 [Conditional("DEBUG")]
143 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
144     ↳ Ensure.Always.IsSigned<T>(message);
145
146 [Conditional("DEBUG")]
147 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
148     ↳ Ensure.Always.IsSigned<T>();
149
150 [Conditional("DEBUG")]
151 public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
152     ↳ messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
153
154 [Conditional("DEBUG")]
155 public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
156     ↳ Ensure.Always.IsNumeric<T>(message);
157
158 [Conditional("DEBUG")]
159 public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
160     ↳ Ensure.Always.IsNumeric<T>();
161
162 [Conditional("DEBUG")]
163 public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
164     ↳ messageBuilder) => Ensure.Always.CanBeNumeric<T>(messageBuilder);
165
166 [Conditional("DEBUG")]
167 public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
168     ↳ => Ensure.Always.CanBeNumeric<T>(message);
169
170 [Conditional("DEBUG")]
171 public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
172     ↳ Ensure.Always.CanBeNumeric<T>();
173
174 #endregion
175 }

```

./Platform.Reflection/FieldInfoExtensions.cs

```

1 using System.Reflection;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Reflection
7 {
8     public static class FieldInfoExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

11         public static T GetStaticValue<T>(this FieldInfo fieldInfo) =>
12             ↪ (T)fieldInfo.GetValue(null);
13     }

```

./Platform.Reflection/ILGeneratorExtensions.cs

```

1  using System;
2  using System.Linq;
3  using System.Reflection;
4  using System.Reflection.Emit;
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 ILGeneratorExtensions
11     {
12         public static void Throw<T>(this ILGenerator generator) =>
13             ↪ generator.ThrowException(typeof(T));
14
15         public static void ConvertTo<T>(this ILGenerator generator)
16         {
17             var type = typeof(T);
18             if (type == typeof(short))
19             {
20                 generator.Emit(OpCodes.Conv_I2);
21             }
22             else if (type == typeof(ushort))
23             {
24                 generator.Emit(OpCodes.Conv_U2);
25             }
26             else if (type == typeof(sbyte))
27             {
28                 generator.Emit(OpCodes.Conv_I1);
29             }
30             else if (type == typeof(byte))
31             {
32                 generator.Emit(OpCodes.Conv_U1);
33             }
34             else
35             {
36                 throw new NotSupportedException();
37             }
38
39         public static void LoadConstant(this ILGenerator generator, bool value) =>
40             ↪ generator.LoadConstant(value ? 1 : 0);
41
42         public static void LoadConstant(this ILGenerator generator, float value) =>
43             ↪ generator.Emit(OpCodes.Ldc_R4, value);
44
45         public static void LoadConstant(this ILGenerator generator, double value) =>
46             ↪ generator.Emit(OpCodes.Ldc_R8, value);
47
48         public static void LoadConstant(this ILGenerator generator, ulong value) =>
49             ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
50
51         public static void LoadConstant(this ILGenerator generator, long value) =>
52             ↪ generator.Emit(OpCodes.Ldc_I8, value);
53
54         public static void LoadConstant(this ILGenerator generator, uint value)
55         {
56             switch (value)
57             {
58                 case uint.MaxValue:
59                     generator.Emit(OpCodes.Ldc_I4_M1);
60                     return;
61                 case 0:
62                     generator.Emit(OpCodes.Ldc_I4_0);
63                     return;
64                 case 1:
65                     generator.Emit(OpCodes.Ldc_I4_1);
66                     return;
67                 case 2:
68                     generator.Emit(OpCodes.Ldc_I4_2);
69                     return;
70                 case 3:
71                     generator.Emit(OpCodes.Ldc_I4_3);
72                     return;

```

```

68         case 4:
69             generator.Emit(OpCodes.Ldc_I4_4);
70             return;
71         case 5:
72             generator.Emit(OpCodes.Ldc_I4_5);
73             return;
74         case 6:
75             generator.Emit(OpCodes.Ldc_I4_6);
76             return;
77         case 7:
78             generator.Emit(OpCodes.Ldc_I4_7);
79             return;
80         case 8:
81             generator.Emit(OpCodes.Ldc_I4_8);
82             return;
83         default:
84             if (value <= sbyte.MaxValue)
85             {
86                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
87             }
88             else
89             {
90                 generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
91             }
92             return;
93     }
94 }
95
96 public static void LoadConstant(this ILGenerator generator, int value)
97 {
98     switch (value)
99     {
100         case -1:
101             generator.Emit(OpCodes.Ldc_I4_M1);
102             return;
103         case 0:
104             generator.Emit(OpCodes.Ldc_I4_0);
105             return;
106         case 1:
107             generator.Emit(OpCodes.Ldc_I4_1);
108             return;
109         case 2:
110             generator.Emit(OpCodes.Ldc_I4_2);
111             return;
112         case 3:
113             generator.Emit(OpCodes.Ldc_I4_3);
114             return;
115         case 4:
116             generator.Emit(OpCodes.Ldc_I4_4);
117             return;
118         case 5:
119             generator.Emit(OpCodes.Ldc_I4_5);
120             return;
121         case 6:
122             generator.Emit(OpCodes.Ldc_I4_6);
123             return;
124         case 7:
125             generator.Emit(OpCodes.Ldc_I4_7);
126             return;
127         case 8:
128             generator.Emit(OpCodes.Ldc_I4_8);
129             return;
130         default:
131             if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
132             {
133                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
134             }
135             else
136             {
137                 generator.Emit(OpCodes.Ldc_I4, value);
138             }
139             return;
140     }
141 }
142
143 public static void LoadConstant(this ILGenerator generator, short value)
144 {
145     generator.LoadConstant((int)value);
146 }
147
148 public static void LoadConstant(this ILGenerator generator, ushort value)

```

```

149 {
150     generator.LoadConstant((int)value);
151 }
152
153 public static void LoadConstant(this ILGenerator generator, sbyte value)
154 {
155     generator.LoadConstant((int)value);
156 }
157
158 public static void LoadConstant(this ILGenerator generator, byte value)
159 {
160     generator.LoadConstant((int)value);
161 }
162
163 public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
164     ↪ LoadConstantOne(generator, typeof(TConstant));
165
166 public static void LoadConstantOne(this ILGenerator generator, Type constantType)
167 {
168     if (constantType == typeof(float))
169     {
170         generator.LoadConstant(1F);
171     }
172     else if (constantType == typeof(double))
173     {
174         generator.LoadConstant(1D);
175     }
176     else if (constantType == typeof(long))
177     {
178         generator.LoadConstant(1L);
179     }
180     else if (constantType == typeof(ulong))
181     {
182         generator.LoadConstant(1UL);
183     }
184     else if (constantType == typeof(int))
185     {
186         generator.LoadConstant(1);
187     }
188     else if (constantType == typeof(uint))
189     {
190         generator.LoadConstant(1U);
191     }
192     else if (constantType == typeof(short))
193     {
194         generator.LoadConstant((short)1);
195     }
196     else if (constantType == typeof(ushort))
197     {
198         generator.LoadConstant((ushort)1);
199     }
200     else if (constantType == typeof(sbyte))
201     {
202         generator.LoadConstant((sbyte)1);
203     }
204     else if (constantType == typeof(byte))
205     {
206         generator.LoadConstant((byte)1);
207     }
208     else
209     {
210         throw new NotSupportedException();
211     }
212 }
213
214 public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
215     ↪ constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
216
217 public static void LoadConstant(this ILGenerator generator, Type constantType, object
218     ↪ constantValue)
219 {
220     constantValue = Convert.ChangeType(constantValue, constantType);
221     if (constantType == typeof(float))
222     {
223         generator.LoadConstant((float)constantValue);
224     }
225     else if (constantType == typeof(double))
226     {
227         generator.LoadConstant((double)constantValue);
228     }
229     else if (constantType == typeof(long))
230     {
231         generator.LoadConstant((long)constantValue);
232     }
233     else if (constantType == typeof(ulong))
234     {
235         generator.LoadConstant((ulong)constantValue);
236     }
237     else if (constantType == typeof(int))
238     {
239         generator.LoadConstant((int)constantValue);
240     }
241     else if (constantType == typeof(uint))
242     {
243         generator.LoadConstant((uint)constantValue);
244     }
245     else if (constantType == typeof(short))
246     {
247         generator.LoadConstant((short)constantValue);
248     }
249     else if (constantType == typeof(ushort))
250     {
251         generator.LoadConstant((ushort)constantValue);
252     }
253     else if (constantType == typeof(sbyte))
254     {
255         generator.LoadConstant((sbyte)constantValue);
256     }
257     else if (constantType == typeof(byte))
258     {
259         generator.LoadConstant((byte)constantValue);
260     }
261     else
262     {
263         throw new NotSupportedException();
264     }
265 }

```

```

224         generator.LoadConstant(((double)constantValue);
225     }
226     else if (constantType == typeof(long))
227     {
228         generator.LoadConstant(((long)constantValue);
229     }
230     else if (constantType == typeof(ulong))
231     {
232         generator.LoadConstant(((ulong)constantValue);
233     }
234     else if (constantType == typeof(int))
235     {
236         generator.LoadConstant(((int)constantValue);
237     }
238     else if (constantType == typeof(uint))
239     {
240         generator.LoadConstant(((uint)constantValue);
241     }
242     else if (constantType == typeof(short))
243     {
244         generator.LoadConstant(((short)constantValue);
245     }
246     else if (constantType == typeof(ushort))
247     {
248         generator.LoadConstant(((ushort)constantValue);
249     }
250     else if (constantType == typeof(sbyte))
251     {
252         generator.LoadConstant(((sbyte)constantValue);
253     }
254     else if (constantType == typeof(byte))
255     {
256         generator.LoadConstant(((byte)constantValue);
257     }
258     else
259     {
260         throw new NotSupportedException();
261     }
262 }
263
264 public static void Increment<TValue>(this ILGenerator generator) =>
265     ↪ generator.Increment(typeof(TValue));
266
267 public static void Decrement<TValue>(this ILGenerator generator) =>
268     ↪ generator.Decrement(typeof(TValue));
269
270 public static void Increment(this ILGenerator generator, Type valueType)
271 {
272     generator.LoadConstantOne(valueType);
273     generator.Add();
274 }
275
276 public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
277
278 public static void Decrement(this ILGenerator generator, Type valueType)
279 {
280     generator.LoadConstantOne(valueType);
281     generator.Subtract();
282 }
283
284 public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
285
286 public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
287
288 public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
289
290 public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
291
292 public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
293
294 public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
295
296 public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
297
298 public static void LoadArgument(this ILGenerator generator, int argumentIndex)
299 {
300     switch (argumentIndex)
301     {
302         case 0:

```



```

301         generator.Emit(OpCodes.Ldarg_0);
302         break;
303     case 1:
304         generator.Emit(OpCodes.Ldarg_1);
305         break;
306     case 2:
307         generator.Emit(OpCodes.Ldarg_2);
308         break;
309     case 3:
310         generator.Emit(OpCodes.Ldarg_3);
311         break;
312     default:
313         generator.Emit(OpCodes.Ldarg, argumentIndex);
314         break;
315     }
316 }
317
318 public static void LoadArguments(this ILGenerator generator, params int[]
    ↪ argumentIndices)
319 {
320     for (var i = 0; i < argumentIndices.Length; i++)
321     {
322         generator.LoadArgument(argumentIndices[i]);
323     }
324 }
325
326 public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
    ↪ generator.Emit(OpCodes.Starg, argumentIndex);
327
328 public static void CompareGreaterThan(this ILGenerator generator) =>
    ↪ generator.Emit(OpCodes.Cgt);
329
330 public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
    ↪ generator.Emit(OpCodes.Cgt_Un);
331
332 public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
333 {
334     if (isSigned)
335     {
336         generator.CompareGreaterThan();
337     }
338     else
339     {
340         generator.UnsignedCompareGreaterThan();
341     }
342 }
343
344 public static void CompareLessThan(this ILGenerator generator) =>
    ↪ generator.Emit(OpCodes.Clt);
345
346 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
    ↪ generator.Emit(OpCodes.Clt_Un);
347
348 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
349 {
350     if (isSigned)
351     {
352         generator.CompareLessThan();
353     }
354     else
355     {
356         generator.UnsignedCompareLessThan();
357     }
358 }
359
360 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
    ↪ generator.Emit(OpCodes.Bge, label);
361
362 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
    ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
363
364 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
    ↪ Label label)
365 {
366     if (isSigned)
367     {
368         generator.BranchIfGreaterOrEqual(label);
369     }
370     else

```

```

371     {
372         generator.UnsignedBranchIfGreaterOrEqual(label);
373     }
374 }
375
376 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
377     ↪ generator.Emit(OpCodes.Ble, label);
378
379 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
380     ↪ => generator.Emit(OpCodes.Ble_Un, label);
381
382 public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
383     ↪ label)
384 {
385     if (isSigned)
386     {
387         generator.BranchIfLessOrEqual(label);
388     }
389     else
390     {
391         generator.UnsignedBranchIfLessOrEqual(label);
392     }
393 }
394
395 public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
396
397 public static void Box(this ILGenerator generator, Type boxedType) =>
398     ↪ generator.Emit(OpCodes.Box, boxedType);
399
400 public static void Call(this ILGenerator generator, MethodInfo method) =>
401     ↪ generator.Emit(OpCodes.Call, method);
402
403 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
404
405 public static void Unbox<TUnbox>(this ILGenerator generator) =>
406     ↪ generator.Unbox(typeof(TUnbox));
407
408 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
409     ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
410
411 public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
412     ↪ generator.UnboxValue(typeof(TUnbox));
413
414 public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
415     ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
416
417 public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
418     ↪ generator.DeclareLocal(typeof(T));
419
420 public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
421     ↪ generator.Emit(OpCodes.Ldloc, local);
422
423 public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
424     ↪ generator.Emit(OpCodes.Stloc, local);
425
426 public static void NewObject(this ILGenerator generator, Type type, params Type[]
427     ↪ parameterTypes)
428 {
429     var allConstructors = type.GetConstructors(BindingFlags.Public |
430         ↪ BindingFlags.NonPublic | BindingFlags.Instance
431         | BindingFlags.CreateInstance
432     );
433     var constructor = allConstructors.Where(c => c.GetParameters().Length ==
434         ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
435         ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
436     if (constructor == null)
437     {
438         throw new InvalidOperationException("Type " + type + " must have a constructor
439             ↪ that matches parameters [" + string.Join(", ",
440             ↪ parameterTypes.AsEnumerable()) + "]");
441     }
442     generator.NewObject(constructor);
443 }
444
445 public static void NewObject(this ILGenerator generator, ConstructorInfo constructor)
446 {

```

```

431     generator.Emit(OpCodes.Newobj, constructor);
432 }
433
434 public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
↳ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
435
436 public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
↳ false, byte? unaligned = null)
437 {
438     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
439     {
440         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
441     }
442     if (isVolatile)
443     {
444         generator.Emit(OpCodes.Volatile);
445     }
446     if (unaligned.HasValue)
447     {
448         generator.Emit(OpCodes.Unaligned, unaligned.Value);
449     }
450     if (type.IsPointer)
451     {
452         generator.Emit(OpCodes.Ldind_I);
453     }
454     else if (!type.IsValueType)
455     {
456         generator.Emit(OpCodes.Ldind_Ref);
457     }
458     else if (type == typeof(sbyte))
459     {
460         generator.Emit(OpCodes.Ldind_I1);
461     }
462     else if (type == typeof(bool))
463     {
464         generator.Emit(OpCodes.Ldind_I1);
465     }
466     else if (type == typeof(byte))
467     {
468         generator.Emit(OpCodes.Ldind_U1);
469     }
470     else if (type == typeof(short))
471     {
472         generator.Emit(OpCodes.Ldind_I2);
473     }
474     else if (type == typeof(ushort))
475     {
476         generator.Emit(OpCodes.Ldind_U2);
477     }
478     else if (type == typeof(char))
479     {
480         generator.Emit(OpCodes.Ldind_U2);
481     }
482     else if (type == typeof(int))
483     {
484         generator.Emit(OpCodes.Ldind_I4);
485     }
486     else if (type == typeof(uint))
487     {
488         generator.Emit(OpCodes.Ldind_U4);
489     }
490     else if (type == typeof(long) || type == typeof(ulong))
491     {
492         generator.Emit(OpCodes.Ldind_I8);
493     }
494     else if (type == typeof(float))
495     {
496         generator.Emit(OpCodes.Ldind_R4);
497     }
498     else if (type == typeof(double))
499     {
500         generator.Emit(OpCodes.Ldind_R8);
501     }
502     else
503     {
504         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
↳ ", LoadObject may be more appropriate");
505     }

```

```

506     }
507
508     public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
    ↪     byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
509
510     public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
    ↪     = false, byte? unaligned = null)
511     {
512         if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
513         {
514             throw new ArgumentException("unaligned must be null, 1, 2, or 4");
515         }
516         if (isVolatile)
517         {
518             generator.Emit(OpCodes.Volatile);
519         }
520         if (unaligned.HasValue)
521         {
522             generator.Emit(OpCodes.Unaligned, unaligned.Value);
523         }
524         if (type.IsPointer)
525         {
526             generator.Emit(OpCodes.Stind_I);
527         }
528         else if (!type.IsValueType)
529         {
530             generator.Emit(OpCodes.Stind_Ref);
531         }
532         else if (type == typeof(sbyte) || type == typeof(byte))
533         {
534             generator.Emit(OpCodes.Stind_I1);
535         }
536         else if (type == typeof(short) || type == typeof(ushort))
537         {
538             generator.Emit(OpCodes.Stind_I2);
539         }
540         else if (type == typeof(int) || type == typeof(uint))
541         {
542             generator.Emit(OpCodes.Stind_I4);
543         }
544         else if (type == typeof(long) || type == typeof(ulong))
545         {
546             generator.Emit(OpCodes.Stind_I8);
547         }
548         else if (type == typeof(float))
549         {
550             generator.Emit(OpCodes.Stind_R4);
551         }
552         else if (type == typeof(double))
553         {
554             generator.Emit(OpCodes.Stind_R8);
555         }
556         else
557         {
558             throw new InvalidOperationException("StoreIndirect cannot be used with " + type
    ↪             + ", StoreObject may be more appropriate");
559         }
560     }
561 }
562 }

```

./Platform.Reflection/MethodInfoExtensions.cs

```

1  using System.Reflection;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Reflection
6  {
7      public static class MethodInfoExtensions
8      {
9          public static byte[] GetILBytes(this MethodInfo methodInfo) =>
    ↪          methodInfo.GetMethodBody().GetILAsByteArray();
10     }
11 }

```

./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs

```

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

```

```

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

```

./Platform.Reflection/NumericType.cs

```

1 using System;
2 using System.Runtime.InteropServices;
3 using Platform.Exceptions;
4
5 // ReSharper disable AssignmentInConditionalExpression
6 // ReSharper disable BuiltInTypeReferenceStyle
7 // ReSharper disable StaticFieldInGenericType
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class NumericType<T>
13     {
14         public static readonly Type Type;
15         public static readonly Type UnderlyingType;
16         public static readonly Type SignedVersion;
17         public static readonly Type UnsignedVersion;
18         public static readonly bool IsFloatPoint;
19         public static readonly bool IsNumeric;
20         public static readonly bool IsSigned;
21         public static readonly bool CanBeNumeric;
22         public static readonly bool IsNullable;
23         public static readonly int BitsLength;
24         public static readonly T MinValue;
25         public static readonly T MaxValue;
26
27         static NumericType()
28         {
29             try
30             {
31                 var type = typeof(T);
32                 var isNullable = type.IsNullable();
33                 var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
34                 var canBeNumeric = underlyingType.CanBeNumeric();
35                 var isNumeric = underlyingType.IsNumeric();
36                 var isSigned = underlyingType.IsSigned();
37                 var isFloatPoint = underlyingType.IsFloatPoint();
38                 var bitsLength = Marshal.SizeOf(underlyingType) * 8;
39                 GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
40                 GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
41                     ↪ out Type unsignedVersion);
42                 Type = type;
43                 IsNullable = isNullable;
44                 UnderlyingType = underlyingType;
45                 CanBeNumeric = canBeNumeric;
46                 IsNumeric = isNumeric;
47                 IsSigned = isSigned;
48                 IsFloatPoint = isFloatPoint;
49                 BitsLength = bitsLength;
50                 MinValue = minValue;
51                 MaxValue = maxValue;
52                 SignedVersion = signedVersion;
53                 UnsignedVersion = unsignedVersion;
54             }
55             catch (Exception exception)
56             {

```

```

56         exception.Ignore();
57     }
58 }
59
60 private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
61 {
62     if (type == typeof(bool))
63     {
64         minValue = (T)(object>false;
65         maxValue = (T)(object>true;
66     }
67     else
68     {
69         minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
70         maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
71     }
72 }
73
74 private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
↪ signedVersion, out Type unsignedVersion)
75 {
76     if (isSigned)
77     {
78         signedVersion = type;
79         unsignedVersion = type.GetUnsignedVersionOrNull();
80     }
81     else
82     {
83         signedVersion = type.GetSignedVersionOrNull();
84         unsignedVersion = type;
85     }
86 }
87 }
88 }

```

./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 }

```

./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 private readonly HashSet<Type> _canBeNumericTypes;
15         static private readonly HashSet<Type> _isNumericTypes;
16         static private readonly HashSet<Type> _isSignedTypes;
17         static private readonly HashSet<Type> _isFloatPointTypes;
18         static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
19         static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;
20
21         static TypeExtensions()
22         {
23             _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
↪             typeof(DateTime), typeof(TimeSpan) };
24             _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
↪             typeof(ulong) };
25             _canBeNumericTypes.UnionWith(_isNumericTypes);

```

```

26     _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
27         ↳ typeof(long) };
28     _canBeNumericTypes.UnionWith(_isSignedTypes);
29     _isNumericTypes.UnionWith(_isSignedTypes);
30     _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
31         ↳ typeof(float) };
32     _canBeNumericTypes.UnionWith(_isFloatPointTypes);
33     _isNumericTypes.UnionWith(_isFloatPointTypes);
34     _isSignedTypes.UnionWith(_isFloatPointTypes);
35     unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
36     {
37         { typeof(sbyte), typeof(byte) },
38         { typeof(short), typeof(ushort) },
39         { typeof(int), typeof(uint) },
40         { typeof(long), typeof(ulong) },
41     };
42     signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
43     {
44         { typeof(byte), typeof(sbyte) },
45         { typeof(ushort), typeof(short) },
46         { typeof(uint), typeof(int) },
47         { typeof(ulong), typeof(long) },
48     };
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     public static T GetStaticFieldValue<T>(this Type type, string name) =>
55         ↳ type.GetTypeInfo().GetField(name, BindingFlags.Public | BindingFlags.NonPublic |
56         ↳ BindingFlags.Static).GetStaticValue<T>();
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     public static T GetStaticPropertyValue<T>(this Type type, string name) =>
60         ↳ type.GetTypeInfo().GetProperty(name, BindingFlags.Public | BindingFlags.NonPublic |
61         ↳ BindingFlags.Static).GetStaticValue<T>();
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
65         ↳ genericParameterTypes, Type[] argumentTypes)
66     {
67         var methods = from m in type.GetMethods()
68             where m.Name == name
69                 && m.IsGenericMethodDefinition
70                 let typeParams = m.GetGenericArguments()
71                 let normalParams = m.GetParameters().Select(x => x.ParameterType)
72                 where typeParams.SequenceEqual(genericParameterTypes)
73                 && normalParams.SequenceEqual(argumentTypes)
74                 select m;
75         var method = methods.Single();
76         return method;
77     }
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     public static Type GetBaseType(this Type type) => type.GetTypeInfo().BaseType;
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     public static Assembly GetAssembly(this Type type) => type.GetTypeInfo().Assembly;
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static bool IsSubclassOf(this Type type, Type superClass) =>
87         ↳ type.GetTypeInfo().IsSubclassOf(superClass);
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     public static bool IsValueType(this Type type) => type.GetTypeInfo().IsValueType;
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public static bool IsGeneric(this Type type) => type.GetTypeInfo().IsGenericType;
94
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
97         ↳ type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
98
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100    public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
101
102    [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

95     public static Type GetUnsignedVersionOrNull(this Type signedType) =>
96         ↪ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
97
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     public static Type GetSignedVersionOrNull(this Type unsignedType) =>
100         ↪ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
101
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
104
105     [MethodImpl(MethodImplOptions.AggressiveInlining)]
106     public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
110
111     [MethodImpl(MethodImplOptions.AggressiveInlining)]
112     public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
113 }

```

./Platform.Reflection/Types.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Collections.ObjectModel;
4  using Platform.Collections.Lists;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Reflection
9  {
10     public abstract class Types
11     {
12         public static ReadOnlyCollection<Type> Collection { get; } = new
13             ↪ ReadOnlyCollection<Type>(new Type[0]);
14         public static Type[] Array => Collection.ToArray();
15
16         protected ReadOnlyCollection<Type> ToReadOnlyCollection()
17         {
18             var types = GetType().GetGenericArguments();
19             var result = new List<Type>();
20             AppendTypes(result, types);
21             return new ReadOnlyCollection<Type>(result);
22         }
23
24         private static void AppendTypes(List<Type> container, IList<Type> types)
25         {
26             for (var i = 0; i < types.Count; i++)
27             {
28                 var element = types[i];
29                 if (element != typeof(Types))
30                 {
31                     if (element.IsSubclassOf(typeof(Types)))
32                     {
33                         AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<Type>>
34                             ↪ (nameof(Types<object>.Collection)));
35                     }
36                     else
37                     {
38                         container.Add(element);
39                     }
40                 }
41             }
42         }
43     }
44 }

```

./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
7  namespace Platform.Reflection
8  {
9     public class Types<T1, T2> : Types
10     {

```



```

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

```

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

```

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

```

./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
7 namespace Platform.Reflection
8 {
9     public class Types<T1, T2, T3, T4, T5> : Types
10     {
11         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
12             ↪ T4, T5>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
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
7 namespace Platform.Reflection
8 {
9     public class Types<T1, T2, T3, T4, T5, T6> : Types
10     {
11         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
12             ↪ T4, T5, T6>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }

```

./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
7 namespace Platform.Reflection
8 {
9     public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
10     {
11         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
12             ↪ T4, T5, T6, T7>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
```

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

./Platform.Reflection.Tests/CodeGenerationTests.cs

```
1 using System;
2 using System.Reflection.Emit;
3 using Xunit;
4
5 namespace Platform.Reflection.Tests
6 {
7     public static class CodeGenerationTests
8     {
9         [Fact]
10         public static void EmptyActionCompilationTest()
11         {
12             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
13             {
14                 generator.Return();
15             });
16             compiledAction();
17         }
18
19         [Fact]
20         public static void FailedActionCompilationTest()
21         {
22             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
23             {
24                 throw new NotImplementedException();
25             });
26             Assert.Throws<NotSupportedException>(compiledAction);
27         }
28
29         [Fact]
30         public static void ConstantLoadingTest()
31         {
32             CheckConstantLoading<byte>(8);
33             CheckConstantLoading<uint>(8);
34             CheckConstantLoading<ushort>(8);
35             CheckConstantLoading<ulong>(8);
36         }
37
38         private static void CheckConstantLoading<T>(T value)
39         {
40             var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
41             {
42                 generator.LoadConstant<T>(value);
43             });
44         }
45     }
```

```

43         generator.Return();
44     });
45     Assert.Equal(value, compiledFunction());
46 }
47 }
48 }

```

./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 }

```

./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, 18
- ./Platform.Reflection.Tests/GetILBytesMethodTests.cs, 19
- ./Platform.Reflection.Tests/NumericTypeTests.cs, 19
- ./Platform.Reflection/AssemblyExtensions.cs, 1
- ./Platform.Reflection/DelegateHelpers.cs, 1
- ./Platform.Reflection/DynamicExtensions.cs, 2
- ./Platform.Reflection/EnsureExtensions.cs, 2
- ./Platform.Reflection/FieldInfoExtensions.cs, 4
- ./Platform.Reflection/ILGeneratorExtensions.cs, 5
- ./Platform.Reflection/MethodInfoExtensions.cs, 12
- ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 12
- ./Platform.Reflection/NumericType.cs, 13
- ./Platform.Reflection/PropertyInfoExtensions.cs, 14
- ./Platform.Reflection/TypeExtensions.cs, 14
- ./Platform.Reflection/Types.cs, 16
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 17
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 17
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 17
- ./Platform.Reflection/Types[T1, T2, T3, T4].cs, 17
- ./Platform.Reflection/Types[T1, T2, T3].cs, 17
- ./Platform.Reflection/Types[T1, T2].cs, 16
- ./Platform.Reflection/Types[T].cs, 18