

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

```

./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, value);
60                         return;
61                     case 0:
62                         generator.Emit(OpCodes.Ldc_I4_0, value);
63                         return;
64                     case 1:
65                         generator.Emit(OpCodes.Ldc_I4_1, value);
66                         return;
67                     case 2:
68                         generator.Emit(OpCodes.Ldc_I4_2, value);
69                         return;
70                     case 3:
71                         generator.Emit(OpCodes.Ldc_I4_3, value);
72                         return;

```

```

68         case 4:
69             generator.Emit(OpCodes.Ldc_I4_4, value);
70             return;
71         case 5:
72             generator.Emit(OpCodes.Ldc_I4_5, value);
73             return;
74         case 6:
75             generator.Emit(OpCodes.Ldc_I4_6, value);
76             return;
77         case 7:
78             generator.Emit(OpCodes.Ldc_I4_7, value);
79             return;
80         case 8:
81             generator.Emit(OpCodes.Ldc_I4_8, value);
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, value);
102             return;
103         case 0:
104             generator.Emit(OpCodes.Ldc_I4_0, value);
105             return;
106         case 1:
107             generator.Emit(OpCodes.Ldc_I4_1, value);
108             return;
109         case 2:
110             generator.Emit(OpCodes.Ldc_I4_2, value);
111             return;
112         case 3:
113             generator.Emit(OpCodes.Ldc_I4_3, value);
114             return;
115         case 4:
116             generator.Emit(OpCodes.Ldc_I4_4, value);
117             return;
118         case 5:
119             generator.Emit(OpCodes.Ldc_I4_5, value);
120             return;
121         case 6:
122             generator.Emit(OpCodes.Ldc_I4_6, value);
123             return;
124         case 7:
125             generator.Emit(OpCodes.Ldc_I4_7, value);
126             return;
127         case 8:
128             generator.Emit(OpCodes.Ldc_I4_8, value);
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     generator.ConvertTo<short>();
147 }
148

```

```

149 public static void LoadConstant(this ILGenerator generator, ushort value)
150 {
151     generator.LoadConstant((int)value);
152     generator.ConvertTo<ushort>();
153 }
154
155 public static void LoadConstant(this ILGenerator generator, sbyte value)
156 {
157     generator.LoadConstant((int)value);
158     generator.ConvertTo<sbyte>();
159 }
160
161 public static void LoadConstant(this ILGenerator generator, byte value)
162 {
163     generator.LoadConstant((int)value);
164     generator.ConvertTo<byte>();
165 }
166
167 public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
    ↳ LoadConstantOne(generator, typeof(TConstant));
168
169 public static void LoadConstantOne(this ILGenerator generator, Type constantType)
170 {
171     if (constantType == typeof(float))
172     {
173         generator.LoadConstant(1F);
174     }
175     else if (constantType == typeof(double))
176     {
177         generator.LoadConstant(1D);
178     }
179     else if (constantType == typeof(long))
180     {
181         generator.LoadConstant(1L);
182     }
183     else if (constantType == typeof(ulong))
184     {
185         generator.LoadConstant(1UL);
186     }
187     else if (constantType == typeof(int))
188     {
189         generator.LoadConstant(1);
190     }
191     else if (constantType == typeof(uint))
192     {
193         generator.LoadConstant(1U);
194     }
195     else if (constantType == typeof(short))
196     {
197         generator.LoadConstant((short)1);
198     }
199     else if (constantType == typeof(ushort))
200     {
201         generator.LoadConstant((ushort)1);
202     }
203     else if (constantType == typeof(sbyte))
204     {
205         generator.LoadConstant((sbyte)1);
206     }
207     else if (constantType == typeof(byte))
208     {
209         generator.LoadConstant((byte)1);
210     }
211     else
212     {
213         throw new NotSupportedException();
214     }
215 }
216
217 public static void LoadConstant<TConstant>(this ILGenerator generator, object
    ↳ constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
218
219 public static void LoadConstant(this ILGenerator generator, Type constantType, object
    ↳ constantValue)
220 {
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 }
266
267 public static void Increment<TValue>(this ILGenerator generator) =>
268     ↪ generator.Increment(typeof(TValue));
269
270 public static void Decrement<TValue>(this ILGenerator generator) =>
271     ↪ generator.Decrement(typeof(TValue));
272
273 public static void Increment(this ILGenerator generator, Type valueType)
274 {
275     generator.LoadConstantOne(valueType);
276     generator.Add();
277 }
278
279 public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
280
281 public static void Decrement(this ILGenerator generator, Type valueType)
282 {
283     generator.LoadConstantOne(valueType);
284     generator.Subtract();
285 }
286
287 public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
288
289 public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
290
291 public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
292
293 public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
294
295 public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
296
297 public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
298
299 public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
300
301 public static void LoadArgument(this ILGenerator generator, int argumentIndex)
302 {

```



```

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

```

```

370 {
371     if (isSigned)
372     {
373         generator.BranchIfGreaterOrEqual(label);
374     }
375     else
376     {
377         generator.UnsignedBranchIfGreaterOrEqual(label);
378     }
379 }
380
381 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
382     ↪ generator.Emit(OpCodes.Ble, label);
383
384 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
385     ↪ => generator.Emit(OpCodes.Ble_Un, label);
386
387 public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
388     ↪ label)
389 {
390     if (isSigned)
391     {
392         generator.BranchIfLessOrEqual(label);
393     }
394     else
395     {
396         generator.UnsignedBranchIfLessOrEqual(label);
397     }
398 }
399
400 public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
401
402 public static void Box(this ILGenerator generator, Type boxedType) =>
403     ↪ generator.Emit(OpCodes.Box, boxedType);
404
405 public static void Call(this ILGenerator generator, MethodInfo method) =>
406     ↪ generator.Emit(OpCodes.Call, method);
407
408 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
409
410 public static void Unbox<TUnbox>(this ILGenerator generator) =>
411     ↪ generator.Unbox(typeof(TUnbox));
412
413 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
414     ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
415
416 public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
417     ↪ generator.UnboxValue(typeof(TUnbox));
418
419 public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
420     ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
421
422 public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
423     ↪ generator.DeclareLocal(typeof(T));
424
425 public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
426     ↪ generator.Emit(OpCodes.Ldloc, local);
427
428 public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
429     ↪ generator.Emit(OpCodes.Stloc, local);
430
431 public static void NewObject(this ILGenerator generator, Type type, params Type[]
432     ↪ parameterTypes)
433 {
434     var allConstructors = type.GetConstructors(BindingFlags.Public |
435     ↪ BindingFlags.NonPublic | BindingFlags.Instance
436     | BindingFlags.CreateInstance
437     );
438     var constructor = allConstructors.Where(c => c.GetParameters().Length ==
439     ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
440     ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
441     if (constructor == null)
442     {
443         throw new InvalidOperationException("Type " + type + " must have a constructor
444         ↪ that matches parameters [" + string.Join(", ",
445         ↪ parameterTypes.AsEnumerable()) + "]");
446     }
447 }

```

```

430     }
431     generator.NewObject(constructor);
432 }
433
434 public static void NewObject(this ILGenerator generator, ConstructorInfo constructor)
435 {
436     generator.Emit(OpCodes.Newobj, constructor);
437 }
438
439 public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
440     ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
441
442 public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
443     ↪ false, byte? unaligned = null)
444 {
445     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
446     {
447         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
448     }
449     if (isVolatile)
450     {
451         generator.Emit(OpCodes.Volatile);
452     }
453     if (unaligned.HasValue)
454     {
455         generator.Emit(OpCodes.Unaligned, unaligned.Value);
456     }
457     if (type.IsPointer)
458     {
459         generator.Emit(OpCodes.Ldind_I);
460     }
461     else if (!type.IsValueType)
462     {
463         generator.Emit(OpCodes.Ldind_Ref);
464     }
465     else if (type == typeof(sbyte))
466     {
467         generator.Emit(OpCodes.Ldind_I1);
468     }
469     else if (type == typeof(bool))
470     {
471         generator.Emit(OpCodes.Ldind_I1);
472     }
473     else if (type == typeof(byte))
474     {
475         generator.Emit(OpCodes.Ldind_U1);
476     }
477     else if (type == typeof(short))
478     {
479         generator.Emit(OpCodes.Ldind_I2);
480     }
481     else if (type == typeof(ushort))
482     {
483         generator.Emit(OpCodes.Ldind_U2);
484     }
485     else if (type == typeof(char))
486     {
487         generator.Emit(OpCodes.Ldind_U2);
488     }
489     else if (type == typeof(int))
490     {
491         generator.Emit(OpCodes.Ldind_I4);
492     }
493     else if (type == typeof(uint))
494     {
495         generator.Emit(OpCodes.Ldind_U4);
496     }
497     else if (type == typeof(long) || type == typeof(ulong))
498     {
499         generator.Emit(OpCodes.Ldind_I8);
500     }
501     else if (type == typeof(float))
502     {
503         generator.Emit(OpCodes.Ldind_R4);
504     }
505     else if (type == typeof(double))
506     {
507         generator.Emit(OpCodes.Ldind_R8);
508     }
509 }

```

```

506     }
507     else
508     {
509         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
        ↪      ", LoadObject may be more appropriate");
510     }
511 }
512
513 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
    ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
514
515 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
    ↪ = false, byte? unaligned = null)
516 {
517     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
518     {
519         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
520     }
521     if (isVolatile)
522     {
523         generator.Emit(OpCodes.Volatile);
524     }
525     if (unaligned.HasValue)
526     {
527         generator.Emit(OpCodes.Unaligned, unaligned.Value);
528     }
529     if (type.IsPointer)
530     {
531         generator.Emit(OpCodes.Stind_I);
532     }
533     else if (!type.IsValueType)
534     {
535         generator.Emit(OpCodes.Stind_Ref);
536     }
537     else if (type == typeof(sbyte) || type == typeof(byte))
538     {
539         generator.Emit(OpCodes.Stind_I1);
540     }
541     else if (type == typeof(short) || type == typeof(ushort))
542     {
543         generator.Emit(OpCodes.Stind_I2);
544     }
545     else if (type == typeof(int) || type == typeof(uint))
546     {
547         generator.Emit(OpCodes.Stind_I4);
548     }
549     else if (type == typeof(long) || type == typeof(ulong))
550     {
551         generator.Emit(OpCodes.Stind_I8);
552     }
553     else if (type == typeof(float))
554     {
555         generator.Emit(OpCodes.Stind_R4);
556     }
557     else if (type == typeof(double))
558     {
559         generator.Emit(OpCodes.Stind_R8);
560     }
561     else
562     {
563         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
        ↪      + ", StoreObject may be more appropriate");
564     }
565 }
566 }
567 }

```

./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             }
53             catch { }
54         }
55     }
56 }

```

```

50         MaxValue = maxValue;
51         SignedVersion = signedVersion;
52         UnsignedVersion = unsignedVersion;
53     }
54     catch (Exception exception)
55     {
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
75 ↪ signedVersion, out Type unsignedVersion)
76 {
77     if (isSigned)
78     {
79         signedVersion = type;
80         unsignedVersion = type.GetUnsignedVersionOrNull();
81     }
82     else
83     {
84         signedVersion = type.GetSignedVersionOrNull();
85         unsignedVersion = type;
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) =>
12             ↪ (T)fieldInfo.GetValue(null);
13     }
14 }

```

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

```

```

90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public static Type GetUnsignedVersionOrNull(this Type signedType) =>
95         ↳ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     public static Type GetSignedVersionOrNull(this Type unsignedType) =>
99         ↳ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
100
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
103
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
106
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
109
110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
111     public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
112 }

```

./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>>(nameof(Types<object>.Collection)));
34                     }
35                     else
36                     {
37                         container.Add(element);
38                     }
39                 }
40             }
41         }
42     }

```

./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,
            ↪ T2>().ToReadOnlyCollection();
12         public new static Type[] Array => Collection.ToArray();
13         private Types() { }
14     }
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,
            ↪ T3>().ToReadOnlyCollection();
12         public new static Type[] Array => Collection.ToArray();
13         private Types() { }
14     }
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,
            ↪ T4>().ToReadOnlyCollection();
12         public new static Type[] Array => Collection.ToArray();
13         private Types() { }
14     }
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,
            ↪ T4, T5>().ToReadOnlyCollection();
12         public new static Type[] Array => Collection.ToArray();
13         private Types() { }
14     }
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,
            ↪ T4, T5, T6>().ToReadOnlyCollection();
12         public new static Type[] Array => Collection.ToArray();

```

```

13     private Types() { }
14 }
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     }
16 }

```

./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     }
16 }

```

./Platform.Reflection.Tests/CodeGenerationTests.cs

```

1 using System;
2 using Xunit;
3
4 namespace Platform.Reflection.Tests
5 {
6     public static class CodeGenerationTests
7     {
8         [Fact]
9         public static void EmptyActionCompilationTest()
10         {
11             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
12             {
13                 generator.Return();
14             });
15             compiledAction();
16         }
17
18         [Fact]
19         public static void FailedActionCompilationTest()
20         {
21             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
22             {
23                 throw new NotImplementedException();
24             });
25             Assert.Throws<NotSupportedException>(compiledAction);
26         }
27     }
28 }

```

./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, 18
- ./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, 13
- ./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, 18
- ./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