

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: generator.Emit(OpCodes.Ldc_I4_M1, value); return;
59                 case 0: generator.Emit(OpCodes.Ldc_I4_0, value); return;
60                 case 1: generator.Emit(OpCodes.Ldc_I4_1, value); return;
61                 case 2: generator.Emit(OpCodes.Ldc_I4_2, value); return;
62                 case 3: generator.Emit(OpCodes.Ldc_I4_3, value); return;
63                 case 4: generator.Emit(OpCodes.Ldc_I4_4, value); return;
64                 case 5: generator.Emit(OpCodes.Ldc_I4_5, value); return;
65                 case 6: generator.Emit(OpCodes.Ldc_I4_6, value); return;
66                 case 7: generator.Emit(OpCodes.Ldc_I4_7, value); return;
67                 case 8: generator.Emit(OpCodes.Ldc_I4_8, value); return;
68             }
69             if (value <= sbyte.MaxValue)
70             {
71                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
72             }
73         }
74     }
75 }

```

```

67     }
68     else
69     {
70         generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
71     }
72 }
73
74 public static void LoadConstant(this ILGenerator generator, int value)
75 {
76     switch (value)
77     {
78         case -1: generator.Emit(OpCodes.Ldc_I4_M1, value); return;
79         case 0: generator.Emit(OpCodes.Ldc_I4_0, value); return;
80         case 1: generator.Emit(OpCodes.Ldc_I4_1, value); return;
81         case 2: generator.Emit(OpCodes.Ldc_I4_2, value); return;
82         case 3: generator.Emit(OpCodes.Ldc_I4_3, value); return;
83         case 4: generator.Emit(OpCodes.Ldc_I4_4, value); return;
84         case 5: generator.Emit(OpCodes.Ldc_I4_5, value); return;
85         case 6: generator.Emit(OpCodes.Ldc_I4_6, value); return;
86         case 7: generator.Emit(OpCodes.Ldc_I4_7, value); return;
87         case 8: generator.Emit(OpCodes.Ldc_I4_8, value); return;
88     }
89     if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
90     {
91         generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
92     }
93     else
94     {
95         generator.Emit(OpCodes.Ldc_I4, value);
96     }
97 }
98
99 public static void LoadConstant(this ILGenerator generator, short value)
100 {
101     generator.LoadConstant((int)value);
102     generator.ConvertTo<short>();
103 }
104
105 public static void LoadConstant(this ILGenerator generator, ushort value)
106 {
107     generator.LoadConstant((int)value);
108     generator.ConvertTo<ushort>();
109 }
110
111 public static void LoadConstant(this ILGenerator generator, sbyte value)
112 {
113     generator.LoadConstant((int)value);
114     generator.ConvertTo<sbyte>();
115 }
116
117 public static void LoadConstant(this ILGenerator generator, byte value)
118 {
119     generator.LoadConstant((int)value);
120     generator.ConvertTo<byte>();
121 }
122
123 public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
124     ↪ LoadConstantOne(generator, typeof(TConstant));
125
126 public static void LoadConstantOne(this ILGenerator generator, Type constantType)
127 {
128     if (constantType == typeof(float))
129     {
130         generator.LoadConstant(1F);
131     }
132     else if (constantType == typeof(double))
133     {
134         generator.LoadConstant(1D);
135     }
136     else if (constantType == typeof(long))
137     {
138         generator.LoadConstant(1L);
139     }
140     else if (constantType == typeof(ulong))
141     {
142         generator.LoadConstant(1UL);
143     }
144     else if (constantType == typeof(int))

```

```

144     {
145         generator.LoadConstant(1);
146     }
147     else if (constantType == typeof(uint))
148     {
149         generator.LoadConstant(1U);
150     }
151     else if (constantType == typeof(short))
152     {
153         generator.LoadConstant((short)1);
154     }
155     else if (constantType == typeof(ushort))
156     {
157         generator.LoadConstant((ushort)1);
158     }
159     else if (constantType == typeof(sbyte))
160     {
161         generator.LoadConstant((sbyte)1);
162     }
163     else if (constantType == typeof(byte))
164     {
165         generator.LoadConstant((byte)1);
166     }
167     else
168     {
169         throw new NotSupportedException();
170     }
171 }
172
173 public static void LoadConstant<TConstant>(this ILGenerator generator, object
↪  constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
174
175 public static void LoadConstant(this ILGenerator generator, Type constantType, object
↪  constantValue)
176 {
177     if (constantType == typeof(float))
178     {
179         generator.LoadConstant((float)constantValue);
180     }
181     else if (constantType == typeof(double))
182     {
183         generator.LoadConstant((double)constantValue);
184     }
185     else if (constantType == typeof(long))
186     {
187         generator.LoadConstant((long)constantValue);
188     }
189     else if (constantType == typeof(ulong))
190     {
191         generator.LoadConstant((ulong)constantValue);
192     }
193     else if (constantType == typeof(int))
194     {
195         generator.LoadConstant((int)constantValue);
196     }
197     else if (constantType == typeof(uint))
198     {
199         generator.LoadConstant((uint)constantValue);
200     }
201     else if (constantType == typeof(short))
202     {
203         generator.LoadConstant((short)constantValue);
204     }
205     else if (constantType == typeof(ushort))
206     {
207         generator.LoadConstant((ushort)constantValue);
208     }
209     else if (constantType == typeof(sbyte))
210     {
211         generator.LoadConstant((sbyte)constantValue);
212     }
213     else if (constantType == typeof(byte))
214     {
215         generator.LoadConstant((byte)constantValue);
216     }
217     else
218     {
219         throw new NotSupportedException();

```

```

220     }
221 }
222
223 public static void Increment<TValue>(this ILGenerator generator) =>
224     ↪ generator.Increment(typeof(TValue));
225
226 public static void Decrement<TValue>(this ILGenerator generator) =>
227     ↪ generator.Decrement(typeof(TValue));
228
229 public static void Increment(this ILGenerator generator, Type valueType)
230 {
231     generator.LoadConstantOne(valueType);
232     generator.Add();
233 }
234
235 public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
236
237 public static void Decrement(this ILGenerator generator, Type valueType)
238 {
239     generator.LoadConstantOne(valueType);
240     generator.Subtract();
241 }
242
243 public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
244
245 public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
246
247 public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
248
249 public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
250
251 public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
252
253 public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
254
255 public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
256
257 public static void LoadArgument(this ILGenerator generator, int argumentIndex)
258 {
259     if (argumentIndex == 0)
260     {
261         generator.Emit(OpCodes.Ldarg_0);
262     }
263     else if (argumentIndex == 1)
264     {
265         generator.Emit(OpCodes.Ldarg_1);
266     }
267     else if (argumentIndex == 2)
268     {
269         generator.Emit(OpCodes.Ldarg_2);
270     }
271     else if (argumentIndex == 3)
272     {
273         generator.Emit(OpCodes.Ldarg_3);
274     }
275     else
276     {
277         generator.Emit(OpCodes.Ldarg, argumentIndex);
278     }
279 }
280
281 public static void LoadArguments(this ILGenerator generator, params int[]
282     ↪ argumentIndices)
283 {
284     for (var i = 0; i < argumentIndices.Length; i++)
285     {
286         generator.LoadArgument(argumentIndices[i]);
287     }
288 }
289
290 public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
291     ↪ generator.Emit(OpCodes.Starg, argumentIndex);
292
293 public static void CompareGreaterThan(this ILGenerator generator) =>
294     ↪ generator.Emit(OpCodes.Cgt);
295
296 public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
297     ↪ generator.Emit(OpCodes.Cgt_Un);

```



```

293 public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
294 {
295     if (isSigned)
296     {
297         generator.CompareGreaterThan();
298     }
299     else
300     {
301         generator.UnsignedCompareGreaterThan();
302     }
303 }
304
305 public static void CompareLessThan(this ILGenerator generator) =>
306     ↪ generator.Emit(OpCodes.Clt);
307
308 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
309     ↪ generator.Emit(OpCodes.Clt_Un);
310
311 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
312 {
313     if (isSigned)
314     {
315         generator.CompareLessThan();
316     }
317     else
318     {
319         generator.UnsignedCompareLessThan();
320     }
321 }
322
323 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
324     ↪ generator.Emit(OpCodes.Bge, label);
325
326 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
327     ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
328
329 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
330     ↪ Label label)
331 {
332     if (isSigned)
333     {
334         generator.BranchIfGreaterOrEqual(label);
335     }
336     else
337     {
338         generator.UnsignedBranchIfGreaterOrEqual(label);
339     }
340 }
341
342 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
343     ↪ generator.Emit(OpCodes.Ble, label);
344
345 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
346     ↪ => generator.Emit(OpCodes.Ble_Un, label);
347
348 public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
349     ↪ label)
350 {
351     if (isSigned)
352     {
353         generator.BranchIfLessOrEqual(label);
354     }
355     else
356     {
357         generator.UnsignedBranchIfLessOrEqual(label);
358     }
359 }
360
361 public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
362
363 public static void Box(this ILGenerator generator, Type boxedType) =>
364     ↪ generator.Emit(OpCodes.Box, boxedType);
365
366 public static void Call(this ILGenerator generator, MethodInfo method) =>
367     ↪ generator.Emit(OpCodes.Call, method);
368
369 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);

```

```

361 public static void Unbox<TUnbox>(this ILGenerator generator) =>
362     ↪ generator.Unbox(typeof(TUnbox));
363
364 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
365     ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
366
367 public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
368     ↪ generator.UnboxValue(typeof(TUnbox));
369
370 public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
371     ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
372
373 public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
374     ↪ generator.DeclareLocal(typeof(T));
375
376 public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
377     ↪ generator.Emit(OpCodes.Ldloc, local);
378
379 public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
380     ↪ generator.Emit(OpCodes.Stloc, local);
381
382 public static void NewObject(this ILGenerator generator, Type type, params Type[]
383     ↪ parameterTypes)
384 {
385     var allConstructors = type.GetConstructors(BindingFlags.Public |
386         ↪ BindingFlags.NonPublic | BindingFlags.Instance
387         | BindingFlags.CreateInstance
388     );
389     var constructor = allConstructors.Where(c => c.GetParameters().Length ==
390         ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
391         ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
392     if (constructor == null)
393     {
394         throw new InvalidOperationException("Type " + type + " must have a constructor
395             ↪ that matches parameters [" + string.Join(", ",
396             ↪ parameterTypes.AsEnumerable()) + "]");
397     }
398     generator.NewObject(constructor);
399 }
400
401 public static void NewObject(this ILGenerator generator, ConstructorInfo constructor)
402 {
403     generator.Emit(OpCodes.Newobj, constructor);
404 }
405
406 public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
407     ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
408
409 public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
410     ↪ false, byte? unaligned = null)
411 {
412     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
413     {
414         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
415     }
416     if (isVolatile)
417     {
418         generator.Emit(OpCodes.Volatile);
419     }
420     if (unaligned.HasValue)
421     {
422         generator.Emit(OpCodes.Unaligned, unaligned.Value);
423     }
424     if (type.IsPointer)
425     {
426         generator.Emit(OpCodes.Ldind_I);
427     }
428     else if (!type.IsValueType)
429     {
430         generator.Emit(OpCodes.Ldind_Ref);
431     }
432     else if (type == typeof(sbyte))
433     {
434         generator.Emit(OpCodes.Ldind_I1);
435     }
436 }

```

```

423     else if (type == typeof(bool))
424     {
425         generator.Emit(OpCodes.Ldind_I1);
426     }
427     else if (type == typeof(byte))
428     {
429         generator.Emit(OpCodes.Ldind_U1);
430     }
431     else if (type == typeof(short))
432     {
433         generator.Emit(OpCodes.Ldind_I2);
434     }
435     else if (type == typeof(ushort))
436     {
437         generator.Emit(OpCodes.Ldind_U2);
438     }
439     else if (type == typeof(char))
440     {
441         generator.Emit(OpCodes.Ldind_U2);
442     }
443     else if (type == typeof(int))
444     {
445         generator.Emit(OpCodes.Ldind_I4);
446     }
447     else if (type == typeof(uint))
448     {
449         generator.Emit(OpCodes.Ldind_U4);
450     }
451     else if (type == typeof(long) || type == typeof(ulong))
452     {
453         generator.Emit(OpCodes.Ldind_I8);
454     }
455     else if (type == typeof(float))
456     {
457         generator.Emit(OpCodes.Ldind_R4);
458     }
459     else if (type == typeof(double))
460     {
461         generator.Emit(OpCodes.Ldind_R8);
462     }
463     else
464     {
465         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
466             ↪ " , LoadObject may be more appropriate");
467     }
468 }
469
470 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
471     ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
472
473 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
474     ↪ = false, byte? unaligned = null)
475 {
476     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
477     {
478         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
479     }
480     if (isVolatile)
481     {
482         generator.Emit(OpCodes.Volatile);
483     }
484     if (unaligned.HasValue)
485     {
486         generator.Emit(OpCodes.Unaligned, unaligned.Value);
487     }
488     if (type.IsPointer)
489     {
490         generator.Emit(OpCodes.Stind_I);
491     }
492     else if (!type.IsValueType)
493     {
494         generator.Emit(OpCodes.Stind_Ref);
495     }
496     else if (type == typeof(sbyte) || type == typeof(byte))
497     {
498         generator.Emit(OpCodes.Stind_I1);
499     }
500     else if (type == typeof(short) || type == typeof(ushort))

```

```

498         {
499             generator.Emit(OpCodes.Stind_I2);
500         }
501         else if (type == typeof(int) || type == typeof(uint))
502         {
503             generator.Emit(OpCodes.Stind_I4);
504         }
505         else if (type == typeof(long) || type == typeof(ulong))
506         {
507             generator.Emit(OpCodes.Stind_I8);
508         }
509         else if (type == typeof(float))
510         {
511             generator.Emit(OpCodes.Stind_R4);
512         }
513         else if (type == typeof(double))
514         {
515             generator.Emit(OpCodes.Stind_R8);
516         }
517         else
518         {
519             throw new InvalidOperationException("StoreIndirect cannot be used with " + type
520                 ↪ + ", StoreObject may be more appropriate");
521         }
522     }
523 }

```

./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) =>
10             ↪ methodInfo.GetMethodBody().GetILAsByteArray();
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             {
57                 exception.Ignore();
58             }
59         }
60
61         private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
62         {
63             if (type == typeof(bool))
64             {
65                 minValue = (T)(object>false;
66                 maxValue = (T)(object>true;
67             }
68             else
69             {
70                 minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
71                 maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
72             }
73         }
74
75         private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
76             ↪ signedVersion, out Type unsignedVersion)
77         {
78             if (isSigned)
79             {
80                 signedVersion = type;
81                 unsignedVersion = type.GetUnsignedVersionOrNull();
82             }
83             else
84             {
85                 signedVersion = type.GetSignedVersionOrNull();
86                 unsignedVersion = type;
87             }
88         }
89     }
90 }

```

```
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),
24             ↪ typeof(DateTime), typeof(TimeSpan) };
25             _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
26             ↪ typeof(ulong) };
27             _canBeNumericTypes.UnionWith(_isNumericTypes);
28             _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
29             ↪ typeof(long) };
30             _canBeNumericTypes.UnionWith(_isSignedTypes);
31             _isNumericTypes.UnionWith(_isSignedTypes);
32             _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
33             ↪ typeof(float) };
34             _canBeNumericTypes.UnionWith(_isFloatPointTypes);
35             _isNumericTypes.UnionWith(_isFloatPointTypes);
36             _isSignedTypes.UnionWith(_isFloatPointTypes);
37             _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
38             {
39                 { typeof(sbyte), typeof(byte) },
40                 { typeof(short), typeof(ushort) },
41                 { typeof(int), typeof(uint) },
42                 { typeof(long), typeof(ulong) },
43             };
44             _signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
45             {
46                 { typeof(byte), typeof(sbyte) },
47                 { typeof(ushort), typeof(short) },
48                 { typeof(uint), typeof(int) },
49                 { typeof(ulong), typeof(long) },
50             };
51         }
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public static T GetStaticFieldValue<T>(this Type type, string name) =>
58             ↪ type.GetTypeInfo().GetField(name, BindingFlags.Public | BindingFlags.NonPublic |
59             ↪ BindingFlags.Static).GetStaticValue<T>();
60     }
61 }
```

```

54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 public static T GetStaticPropertyValue<T>(this Type type, string name) =>
56     ↳ 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
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 public static Type GetUnsignedVersionOrNull(this Type signedType) =>
    ↳ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 public static Type GetSignedVersionOrNull(this Type unsignedType) =>
    ↳ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
111 }
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
    ↪     ReadOnlyCollection<Type>(new Type[0]);
13     public static Type[] Array => Collection.ToArray();
14
15     protected ReadOnlyCollection<Type> ToReadOnlyCollection()
16     {
17         var types = GetType().GetGenericArguments();
18         var result = new List<Type>();
19         AppendTypes(result, types);
20         return new ReadOnlyCollection<Type>(result);
21     }
22
23     private static void AppendTypes(List<Type> container, IList<Type> types)
24     {
25         for (var i = 0; i < types.Count; i++)
26         {
27             var element = types[i];
28             if (element != typeof(Types))
29             {
30                 if (element.IsSubclassOf(typeof(Types)))
31                 {
32                     AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<
    ↪     <Type>>(nameof(Types<object>.Collection)));
33                 }
34                 else
35                 {
36                     container.Add(element);
37                 }
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,
            ↪ T4, T5, T6, T7>().ToReadOnlyCollection();
12        public new static Type[] Array => Collection.ToArray();
13        private Types() { }
14    }
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
            ↪ Types<T>().ToReadOnlyCollection();
12        public new static Type[] Array => Collection.ToArray();
13        private Types() { }

```

```
14     }
15 }
```

./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            Action compiledAction = DelegateHelpers.Compile<Action>(generator =>
12            {
13                generator.Return();
14            });
15            compiledAction();
16        }
17
18        [Fact]
19        public static void FailedActionCompilationTest()
20        {
21            Action 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 }
```

12 }
13 }

Index

- ./Platform.Reflection.Tests/CodeGenerationTests.cs, 18
- ./Platform.Reflection.Tests/GetILBytesMethodTests.cs, 18
- ./Platform.Reflection.Tests/NumericTypeTests.cs, 18
- ./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, 12
- ./Platform.Reflection/PropertyInfoExtensions.cs, 14
- ./Platform.Reflection/TypeExtensions.cs, 14
- ./Platform.Reflection/Types.cs, 15
- ./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, 16
- ./Platform.Reflection/Types[T1, T2, T3].cs, 16
- ./Platform.Reflection/Types[T1, T2].cs, 16
- ./Platform.Reflection/Types[T].cs, 17