

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;
5  using System.Reflection.Emit;
6  using Platform.Exceptions;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class DelegateHelpers
13     {
14         public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode, bool
15             ↳ aggressiveInlining)
16             where TDelegate : Delegate
17         {
18             var @delegate = default(TDelegate);
19             try
20             {
21                 @delegate = aggressiveInlining ? CompileUsingMethodBuilder<TDelegate>(emitCode)
22                     ↳ : CompileUsingDynamicMethod<TDelegate>(emitCode);
23             }
24             catch (Exception exception)
25             {
26                 exception.Ignore();
27             }
28             return @delegate;
29         }
30
31         public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
32             ↳ TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);
33
34         public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
35             ↳ aggressiveInlining)
36             where TDelegate : Delegate
37         {
38             var @delegate = CompileOrDefault<TDelegate>(emitCode, aggressiveInlining);
39             if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
40             {

```

```

37         @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
38     }
39     return @delegate;
40 }
41
42 public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
    ↳ : Delegate => Compile<TDelegate>(emitCode, false);
43
44 private static TDelegate CompileUsingDynamicMethod<TDelegate>(Action<ILGenerator>
    ↳ emitCode)
45 {
46     var delegateType = typeof(TDelegate);
47     var invoke = delegateType.GetMethod("Invoke");
48     var returnType = invoke.ReturnType;
49     var parameterTypes = invoke.GetParameters().Select(s => s.ParameterType).ToArray();
50     var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
51     var generator = dynamicMethod.GetILGenerator();
52     emitCode(generator);
53     return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
54 }
55
56 private static TDelegate CompileUsingMethodBuilder<TDelegate>(Action<ILGenerator>
    ↳ emitCode)
57 {
58     AssemblyName assemblyName = new AssemblyName(GetNewName());
59     var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
    ↳ AssemblyBuilderAccess.Run);
60     var module = assembly.DefineDynamicModule(GetNewName());
61     var type = module.DefineType(GetNewName());
62     var delegateType = typeof(TDelegate);
63     var invoke = delegateType.GetMethod("Invoke");
64     var returnType = invoke.ReturnType;
65     var parameterTypes = invoke.GetParameters().Select(s => s.ParameterType).ToArray();
66     var methodName = GetNewName();
67     MethodBuilder method = type.DefineMethod(methodName, MethodAttributes.Public |
    ↳ MethodAttributes.Static, returnType, parameterTypes);
68     method.SetImplementationFlags(MethodImplAttributes.IL | MethodImplAttributes.Managed
    ↳ | MethodImplAttributes.AggressiveInlining);
69     var generator = method.GetILGenerator();
70     emitCode(generator);
71     var typeInfo = type.CreateTypeInfo();
72     return
    ↳ (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(delegateType);
73 }
74
75 private static string GetNewName() => Guid.NewGuid().ToString("N");
76 }
77 }

```

./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 namespace Platform.Reflection
5 {
6     public static class DynamicExtensions
7     {
8         public static bool HasProperty(this object @object, string propertyName)
9         {
10             var type = @object.GetType();
11             if (type is IDictionary<string, object> dictionary)
12             {
13                 return dictionary.ContainsKey(propertyName);
14             }
15             return type.GetProperty(propertyName) != null;
16         }
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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
50             public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
51             ↪ message)
52             {
53                 string messageBuilder() => message;
54                 IsSignedInteger<T>(root, messageBuilder());
55             }
56
57             [MethodImpl(MethodImplOptions.AggressiveInlining)]
58             public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
59             ↪ IsSignedInteger<T>(root, (string)null);
60
61             [MethodImpl(MethodImplOptions.AggressiveInlining)]
62             public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
63             ↪ messageBuilder)
64             {
65                 if (!NumericType<T>.IsSigned)
66                 {
67                     throw new NotSupportedException(messageBuilder());
68                 }
69             }
70
71             [MethodImpl(MethodImplOptions.AggressiveInlining)]
72             public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
73             {
74                 string messageBuilder() => message;
75                 IsSigned<T>(root, messageBuilder());
76             }
77
78             [MethodImpl(MethodImplOptions.AggressiveInlining)]
79             public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
80             ↪ (string)null);
81
82             [MethodImpl(MethodImplOptions.AggressiveInlining)]
83             public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
84             ↪ 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) =>
91     => IsNumeric<T>(root, (string)null);
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
95     => messageBuilder)
96 {
97     if (!NumericType<T>.CanBeNumeric)
98     {
99         throw new NotSupportedException(messageBuilder());
100     }
101 }
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
105 {
106     string messageBuilder() => message;
107     CanBeNumeric<T>(root, messageBuilder());
108 }
109
110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
111 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
112     => CanBeNumeric<T>(root, (string)null);
113
114 #endregion
115
116 #region OnDebug
117
118 [Conditional("DEBUG")]
119 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
120     => Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
121
122 [Conditional("DEBUG")]
123 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
124     => message) => Ensure.Always.IsUnsignedInteger<T>(message);
125
126 [Conditional("DEBUG")]
127 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
128     => Ensure.Always.IsUnsignedInteger<T>();
129
130 [Conditional("DEBUG")]
131 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
132     => messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
133
134 [Conditional("DEBUG")]
135 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
136     => message) => Ensure.Always.IsSignedInteger<T>(message);
137
138 [Conditional("DEBUG")]
139 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
140     => Ensure.Always.IsSignedInteger<T>();
141
142 [Conditional("DEBUG")]
143 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
144     => messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
145
146 [Conditional("DEBUG")]
147 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
148     => Ensure.Always.IsSigned<T>(message);
149
150 [Conditional("DEBUG")]
151 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
152     => Ensure.Always.IsSigned<T>();

```

```

141     [Conditional("DEBUG")]
142     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
143         ↪ messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
144
145     [Conditional("DEBUG")]
146     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
147         ↪ Ensure.Always.IsNumeric<T>(message);
148
149     [Conditional("DEBUG")]
150     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
151         ↪ Ensure.Always.IsNumeric<T>();
152
153     [Conditional("DEBUG")]
154     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
155         ↪ messageBuilder) => Ensure.Always.CanBeNumeric<T>(messageBuilder);
156
157     [Conditional("DEBUG")]
158     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
159         ↪ => Ensure.Always.CanBeNumeric<T>(message);
160
161     [Conditional("DEBUG")]
162     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
163         ↪ Ensure.Always.CanBeNumeric<T>();
164
165     #endregion
166 }

```

./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     }
14 }

```

./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         }
35     }
36 }

```

```

32     }
33     else
34     {
35         throw new NotSupportedException();
36     }
37 }
38
39 public static void LoadConstant(this ILGenerator generator, bool value) =>
40     ↪ generator.LoadConstant(value ? 1 : 0);
41
42 public static void LoadConstant(this ILGenerator generator, float value) =>
43     ↪ generator.Emit(OpCodes.Ldc_R4, value);
44
45 public static void LoadConstant(this ILGenerator generator, double value) =>
46     ↪ generator.Emit(OpCodes.Ldc_R8, value);
47
48 public static void LoadConstant(this ILGenerator generator, ulong value) =>
49     ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
50
51 public static void LoadConstant(this ILGenerator generator, long value) =>
52     ↪ generator.Emit(OpCodes.Ldc_I8, value);
53
54 public static void LoadConstant(this ILGenerator generator, uint value)
55 {
56     switch (value)
57     {
58         case uint.MaxValue:
59             generator.Emit(OpCodes.Ldc_I4_M1);
60             return;
61         case 0:
62             generator.Emit(OpCodes.Ldc_I4_0);
63             return;
64         case 1:
65             generator.Emit(OpCodes.Ldc_I4_1);
66             return;
67         case 2:
68             generator.Emit(OpCodes.Ldc_I4_2);
69             return;
70         case 3:
71             generator.Emit(OpCodes.Ldc_I4_3);
72             return;
73         case 4:
74             generator.Emit(OpCodes.Ldc_I4_4);
75             return;
76         case 5:
77             generator.Emit(OpCodes.Ldc_I4_5);
78             return;
79         case 6:
80             generator.Emit(OpCodes.Ldc_I4_6);
81             return;
82         case 7:
83             generator.Emit(OpCodes.Ldc_I4_7);
84             return;
85         case 8:
86             generator.Emit(OpCodes.Ldc_I4_8);
87             return;
88         default:
89             if (value <= sbyte.MaxValue)
90             {
91                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
92             }
93             else
94             {
95                 generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
96             }
97             return;
98     }
99 }
100
101 public static void LoadConstant(this ILGenerator generator, int value)
102 {
103     switch (value)
104     {
105         case -1:
106             generator.Emit(OpCodes.Ldc_I4_M1);
107             return;
108         case 0:
109             generator.Emit(OpCodes.Ldc_I4_0);
110             return;
111         case 1:

```

```

107         generator.Emit(OpCodes.Ldc_I4_1);
108         return;
109     case 2:
110         generator.Emit(OpCodes.Ldc_I4_2);
111         return;
112     case 3:
113         generator.Emit(OpCodes.Ldc_I4_3);
114         return;
115     case 4:
116         generator.Emit(OpCodes.Ldc_I4_4);
117         return;
118     case 5:
119         generator.Emit(OpCodes.Ldc_I4_5);
120         return;
121     case 6:
122         generator.Emit(OpCodes.Ldc_I4_6);
123         return;
124     case 7:
125         generator.Emit(OpCodes.Ldc_I4_7);
126         return;
127     case 8:
128         generator.Emit(OpCodes.Ldc_I4_8);
129         return;
130     default:
131         if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
132         {
133             generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
134         }
135         else
136         {
137             generator.Emit(OpCodes.Ldc_I4, value);
138         }
139         return;
140     }
141 }
142
143 public static void LoadConstant(this ILGenerator generator, short value)
144 {
145     generator.LoadConstant((int)value);
146 }
147
148 public static void LoadConstant(this ILGenerator generator, ushort value)
149 {
150     generator.LoadConstant((int)value);
151 }
152
153 public static void LoadConstant(this ILGenerator generator, sbyte value)
154 {
155     generator.LoadConstant((int)value);
156 }
157
158 public static void LoadConstant(this ILGenerator generator, byte value)
159 {
160     generator.LoadConstant((int)value);
161 }
162
163 public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
164     ↪ LoadConstantOne(generator, typeof(TConstant));
165
166 public static void LoadConstantOne(this ILGenerator generator, Type constantType)
167 {
168     if (constantType == typeof(float))
169     {
170         generator.LoadConstant(1F);
171     }
172     else if (constantType == typeof(double))
173     {
174         generator.LoadConstant(1D);
175     }
176     else if (constantType == typeof(long))
177     {
178         generator.LoadConstant(1L);
179     }
180     else if (constantType == typeof(ulong))
181     {
182         generator.LoadConstant(1UL);
183     }
184     else if (constantType == typeof(int))
185     {
186         generator.LoadConstant(1);
187     }
188 }

```

```

186     }
187     else if (constantType == typeof(uint))
188     {
189         generator.LoadConstant(1U);
190     }
191     else if (constantType == typeof(short))
192     {
193         generator.LoadConstant((short)1);
194     }
195     else if (constantType == typeof(ushort))
196     {
197         generator.LoadConstant((ushort)1);
198     }
199     else if (constantType == typeof(sbyte))
200     {
201         generator.LoadConstant((sbyte)1);
202     }
203     else if (constantType == typeof(byte))
204     {
205         generator.LoadConstant((byte)1);
206     }
207     else
208     {
209         throw new NotSupportedException();
210     }
211 }
212
213 public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
↪  constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
214
215 public static void LoadConstant(this ILGenerator generator, Type constantType, object
↪  constantValue)
216 {
217     constantValue = Convert.ChangeType(constantValue, constantType);
218     if (constantType == typeof(float))
219     {
220         generator.LoadConstant((float)constantValue);
221     }
222     else if (constantType == typeof(double))
223     {
224         generator.LoadConstant((double)constantValue);
225     }
226     else if (constantType == typeof(long))
227     {
228         generator.LoadConstant((long)constantValue);
229     }
230     else if (constantType == typeof(ulong))
231     {
232         generator.LoadConstant((ulong)constantValue);
233     }
234     else if (constantType == typeof(int))
235     {
236         generator.LoadConstant((int)constantValue);
237     }
238     else if (constantType == typeof(uint))
239     {
240         generator.LoadConstant((uint)constantValue);
241     }
242     else if (constantType == typeof(short))
243     {
244         generator.LoadConstant((short)constantValue);
245     }
246     else if (constantType == typeof(ushort))
247     {
248         generator.LoadConstant((ushort)constantValue);
249     }
250     else if (constantType == typeof(sbyte))
251     {
252         generator.LoadConstant((sbyte)constantValue);
253     }
254     else if (constantType == typeof(byte))
255     {
256         generator.LoadConstant((byte)constantValue);
257     }
258     else
259     {
260         throw new NotSupportedException();
261     }

```



```

262     }
263
264     public static void Increment<TValue>(this ILGenerator generator) =>
265         ↪ generator.Increment(typeof(TValue));
266
267     public static void Decrement<TValue>(this ILGenerator generator) =>
268         ↪ generator.Decrement(typeof(TValue));
269
270     public static void Increment(this ILGenerator generator, Type valueType)
271     {
272         generator.LoadConstantOne(valueType);
273         generator.Add();
274     }
275
276     public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
277
278     public static void Decrement(this ILGenerator generator, Type valueType)
279     {
280         generator.LoadConstantOne(valueType);
281         generator.Subtract();
282     }
283
284     public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
285
286     public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
287
288     public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
289
290     public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
291
292     public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
293
294     public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
295
296     public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
297
298     public static void LoadArgument(this ILGenerator generator, int argumentIndex)
299     {
300         switch (argumentIndex)
301         {
302             case 0:
303                 generator.Emit(OpCodes.Ldarg_0);
304                 break;
305             case 1:
306                 generator.Emit(OpCodes.Ldarg_1);
307                 break;
308             case 2:
309                 generator.Emit(OpCodes.Ldarg_2);
310                 break;
311             case 3:
312                 generator.Emit(OpCodes.Ldarg_3);
313                 break;
314             default:
315                 generator.Emit(OpCodes.Ldarg, argumentIndex);
316                 break;
317         }
318     }
319
320     public static void LoadArguments(this ILGenerator generator, params int[]
321         ↪ argumentIndices)
322     {
323         for (var i = 0; i < argumentIndices.Length; i++)
324         {
325             generator.LoadArgument(argumentIndices[i]);
326         }
327     }
328
329     public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
330         ↪ generator.Emit(OpCodes.Starg, argumentIndex);
331
332     public static void CompareGreaterThan(this ILGenerator generator) =>
333         ↪ generator.Emit(OpCodes.Cgt);
334
335     public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
336         ↪ generator.Emit(OpCodes.Cgt_Un);
337
338     public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
339     {
340         if (isSigned)

```

```

335     {
336         generator.CompareGreaterThan();
337     }
338     else
339     {
340         generator.UnsignedCompareGreaterThan();
341     }
342 }
343
344 public static void CompareLessThan(this ILGenerator generator) =>
345     ↪ generator.Emit(OpCodes.Clt);
346
347 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
348     ↪ generator.Emit(OpCodes.Clt_Un);
349
350 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
351 {
352     if (isSigned)
353     {
354         generator.CompareLessThan();
355     }
356     else
357     {
358         generator.UnsignedCompareLessThan();
359     }
360 }
361
362 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
363     ↪ generator.Emit(OpCodes.Bge, label);
364
365 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
366     ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
367
368 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
369     ↪ 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));

```

```

402     public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
403         ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
404
405     public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
406         ↪ generator.UnboxValue(typeof(TUnbox));
407
408     public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
409         ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
410
411     public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
412         ↪ generator.DeclareLocal(typeof(T));
413
414     public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
415         ↪ generator.Emit(OpCodes.Ldloc, local);
416
417     public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
418         ↪ generator.Emit(OpCodes.Stloc, local);
419
420     public static void NewObject(this ILGenerator generator, Type type, params Type[]
421         ↪ parameterTypes)
422     {
423         var allConstructors = type.GetConstructors(BindingFlags.Public |
424             ↪ BindingFlags.NonPublic | BindingFlags.Instance
425             | BindingFlags.CreateInstance
426         );
427         var constructor = allConstructors.Where(c => c.GetParameters().Length ==
428             ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
429             ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
430         if (constructor == null)
431         {
432             throw new InvalidOperationException("Type " + type + " must have a constructor
433                 ↪ that matches parameters [" + string.Join(", ",
434                 ↪ parameterTypes.AsEnumerable()) + "]");
435         }
436         generator.NewObject(constructor);
437     }
438
439     public static void NewObject(this ILGenerator generator, ConstructorInfo constructor)
440     {
441         generator.Emit(OpCodes.Newobj, constructor);
442     }
443
444     public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
445         ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
446
447     public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
448         ↪ false, byte? unaligned = null)
449     {
450         if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
451         {
452             throw new ArgumentException("unaligned must be null, 1, 2, or 4");
453         }
454         if (isVolatile)
455         {
456             generator.Emit(OpCodes.Volatile);
457         }
458         if (unaligned.HasValue)
459         {
460             generator.Emit(OpCodes.Unaligned, unaligned.Value);
461         }
462         if (type.IsPointer)
463         {
464             generator.Emit(OpCodes.Ldind_I);
465         }
466         else if (!type.IsValueType)
467         {
468             generator.Emit(OpCodes.Ldind_Ref);
469         }
470         else if (type == typeof(sbyte))
471         {
472             generator.Emit(OpCodes.Ldind_I1);
473         }
474         else if (type == typeof(bool))
475         {
476             generator.Emit(OpCodes.Ldind_I1);
477         }
478     }

```

```

465     }
466     else if (type == typeof(byte))
467     {
468         generator.Emit(OpCodes.Ldind_U1);
469     }
470     else if (type == typeof(short))
471     {
472         generator.Emit(OpCodes.Ldind_I2);
473     }
474     else if (type == typeof(ushort))
475     {
476         generator.Emit(OpCodes.Ldind_U2);
477     }
478     else if (type == typeof(char))
479     {
480         generator.Emit(OpCodes.Ldind_U2);
481     }
482     else if (type == typeof(int))
483     {
484         generator.Emit(OpCodes.Ldind_I4);
485     }
486     else if (type == typeof(uint))
487     {
488         generator.Emit(OpCodes.Ldind_U4);
489     }
490     else if (type == typeof(long) || type == typeof(ulong))
491     {
492         generator.Emit(OpCodes.Ldind_I8);
493     }
494     else if (type == typeof(float))
495     {
496         generator.Emit(OpCodes.Ldind_R4);
497     }
498     else if (type == typeof(double))
499     {
500         generator.Emit(OpCodes.Ldind_R8);
501     }
502     else
503     {
504         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
505             ↪ " , LoadObject may be more appropriate");
506     }
507 }
508
509 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
510     ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
511
512 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
513     ↪ = false, byte? unaligned = null)
514 {
515     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
516     {
517         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
518     }
519     if (isVolatile)
520     {
521         generator.Emit(OpCodes.Volatile);
522     }
523     if (unaligned.HasValue)
524     {
525         generator.Emit(OpCodes.Unaligned, unaligned.Value);
526     }
527     if (type.IsPointer)
528     {
529         generator.Emit(OpCodes.Stind_I);
530     }
531     else if (!type.IsValueType)
532     {
533         generator.Emit(OpCodes.Stind_Ref);
534     }
535     else if (type == typeof(sbyte) || type == typeof(byte))
536     {
537         generator.Emit(OpCodes.Stind_I1);
538     }
539     else if (type == typeof(short) || type == typeof(ushort))
540     {
541         generator.Emit(OpCodes.Stind_I2);
542     }
543     else if (type == typeof(int) || type == typeof(uint))
544     {
545         generator.Emit(OpCodes.Stind_I4);
546     }
547     else if (type == typeof(long) || type == typeof(ulong))
548     {
549         generator.Emit(OpCodes.Stind_I8);
550     }
551     else if (type == typeof(float))
552     {
553         generator.Emit(OpCodes.Stind_R4);
554     }
555     else if (type == typeof(double))
556     {
557         generator.Emit(OpCodes.Stind_R8);
558     }
559     else
560     {
561         throw new InvalidOperationException("StoreIndirect cannot be used with " + type +
562             ↪ " , LoadObject may be more appropriate");
563     }
564 }

```

```

540         else if (type == typeof(int) || type == typeof(uint))
541         {
542             generator.Emit(OpCodes.Stind_I4);
543         }
544         else if (type == typeof(long) || type == typeof(ulong))
545         {
546             generator.Emit(OpCodes.Stind_I8);
547         }
548         else if (type == typeof(float))
549         {
550             generator.Emit(OpCodes.Stind_R4);
551         }
552         else if (type == typeof(double))
553         {
554             generator.Emit(OpCodes.Stind_R8);
555         }
556         else
557         {
558             throw new InvalidOperationException("StoreIndirect cannot be used with " + type
559                 ↪ + ", StoreObject may be more appropriate");
560         }
561     }
562 }

```

./Platform.Reflection/MethodInfoExtensions.cs

```

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

```

./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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
53             public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
54
55             [MethodImpl(MethodImplOptions.AggressiveInlining)]
56             public static T GetStaticFieldValue<T>(this Type type, string name) =>
57                 ↪ type.GetTypeInfo().GetField(name, BindingFlags.Public | BindingFlags.NonPublic |
58                 ↪ BindingFlags.Static).GetStaticValue<T>();
59
60             [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         }
62     }
63 }
```

```

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

```

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

```

1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Reflection
8  {
9      public class Types<T1, T2, T3> : Types
10     {
11         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2,
12             ↪ T3>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16

```

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

```

1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Reflection
8  {
9      public class Types<T1, T2, T3, T4> : Types
10     {

```

```

11         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
12             ↪ T4>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }
17
18 ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs
19 using System;
20 using System.Collections.ObjectModel;
21 using Platform.Collections.Lists;
22
23 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
24
25 namespace Platform.Reflection
26 {
27     public class Types<T1, T2, T3, T4, T5> : Types
28     {
29         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
30             ↪ T4, T5>().ToReadOnlyCollection();
31         public new static Type[] Array => Collection.ToArray();
32         private Types() { }
33     }
34 }
35
36 ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs
37 using System;
38 using System.Collections.ObjectModel;
39 using Platform.Collections.Lists;
40
41 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
42
43 namespace Platform.Reflection
44 {
45     public class Types<T1, T2, T3, T4, T5, T6> : Types
46     {
47         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
48             ↪ T4, T5, T6>().ToReadOnlyCollection();
49         public new static Type[] Array => Collection.ToArray();
50         private Types() { }
51     }
52 }
53
54 ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs
55 using System;
56 using System.Collections.ObjectModel;
57 using Platform.Collections.Lists;
58
59 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
60
61 namespace Platform.Reflection
62 {
63     public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
64     {
65         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
66             ↪ T4, T5, T6, T7>().ToReadOnlyCollection();
67         public new static Type[] Array => Collection.ToArray();
68         private Types() { }
69     }
70 }
71
72 ./Platform.Reflection/Types[T].cs
73 using System;
74 using System.Collections.ObjectModel;
75 using Platform.Collections.Lists;
76
77 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
78
79 namespace Platform.Reflection
80 {
81     public class Types<T> : Types
82     {
83         public new static ReadOnlyCollection<Type> Collection { get; } = new
84             ↪ Types<T>().ToReadOnlyCollection();
85         public new static Type[] Array => Collection.ToArray();
86         private Types() { }
87     }
88 }

```

./Platform.Reflection.Tests/CodeGenerationTests.cs

```
1  using Platform.Diagnostics;
2  using System;
3  using System.Runtime.CompilerServices;
4  using Xunit;
5  using Xunit.Abstractions;
6
7  namespace Platform.Reflection.Tests
8  {
9      public class CodeGenerationTests
10     {
11         private readonly ITestOutputHelper _output;
12
13         public CodeGenerationTests(ITestOutputHelper output) => _output = output;
14
15         [Fact]
16         public void EmptyActionCompilationTest()
17         {
18             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
19             {
20                 generator.Return();
21             });
22             compiledAction();
23         }
24
25         [Fact]
26         public void FailedActionCompilationTest()
27         {
28             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
29             {
30                 throw new NotImplementedException();
31             });
32             Assert.Throws<NotSupportedException>(compiledAction);
33         }
34
35         [Fact]
36         public void ConstantLoadingTest()
37         {
38             CheckConstantLoading<byte>(8);
39             CheckConstantLoading<uint>(8);
40             CheckConstantLoading<ushort>(8);
41             CheckConstantLoading<ulong>(8);
42         }
43
44         private void CheckConstantLoading<T>(T value)
45         {
46             var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
47             {
48                 generator.LoadConstant(value);
49                 generator.Return();
50             });
51             Assert.Equal(value, compiledFunction());
52         }
53
54         private class MethodsContainer
55         {
56             public static readonly Func<int> DelegateWithoutAggressiveInlining;
57             public static readonly Func<int> DelegateWithAggressiveInlining;
58
59             static MethodsContainer()
60             {
61                 void emitCode(System.Reflection.Emit.ILGenerator generator)
62                 {
63                     generator.LoadConstant(140314);
64                     generator.Return();
65                 };
66                 DelegateWithoutAggressiveInlining = DelegateHelpers.Compile<Func<int>>(emitCode,
67                     ↪ aggressiveInlining: false);
68                 DelegateWithAggressiveInlining = DelegateHelpers.Compile<Func<int>>(emitCode,
69                     ↪ aggressiveInlining: true);
70             }
71
72             [MethodImpl(MethodImplOptions.AggressiveInlining)]
73             public static int WrapperForDelegateWithoutAggressiveInlining() =>
74                 ↪ DelegateWithoutAggressiveInlining();
75
76             [MethodImpl(MethodImplOptions.AggressiveInlining)]
77             public static int WrapperForDelegateWithAggressiveInlining() =>
78                 ↪ DelegateWithAggressiveInlining();
79         }
80     }
81 }
```

```

75     }
76
77     [Fact]
78     public void AggressiveInliningEffectTest()
79     {
80         const int N = 10000000;
81
82         int result = 0;
83
84         // Warm up
85
86         for (int i = 0; i < N; i++)
87         {
88             result = MethodsContainer.DelegateWithoutAggressiveInlining();
89         }
90         for (int i = 0; i < N; i++)
91         {
92             result = MethodsContainer.DelegateWithAggressiveInlining();
93         }
94         for (int i = 0; i < N; i++)
95         {
96             result = MethodsContainer.WrapperForDelegateWithoutAggressiveInlining();
97         }
98         for (int i = 0; i < N; i++)
99         {
100             result = MethodsContainer.WrapperForDelegateWithAggressiveInlining();
101         }
102         for (int i = 0; i < N; i++)
103         {
104             result = Function();
105         }
106         for (int i = 0; i < N; i++)
107         {
108             result = 140314;
109         }
110
111         // Measure
112         var ts1 = Performance.Measure(() =>
113         {
114             for (int i = 0; i < N; i++)
115             {
116                 result = MethodsContainer.DelegateWithoutAggressiveInlining();
117             }
118         });
119         var ts2 = Performance.Measure(() =>
120         {
121             for (int i = 0; i < N; i++)
122             {
123                 result = MethodsContainer.DelegateWithAggressiveInlining();
124             }
125         });
126         var ts3 = Performance.Measure(() =>
127         {
128             for (int i = 0; i < N; i++)
129             {
130                 result = MethodsContainer.WrapperForDelegateWithoutAggressiveInlining();
131             }
132         });
133         var ts4 = Performance.Measure(() =>
134         {
135             for (int i = 0; i < N; i++)
136             {
137                 result = MethodsContainer.WrapperForDelegateWithAggressiveInlining();
138             }
139         });
140         var ts5 = Performance.Measure(() =>
141         {
142             for (int i = 0; i < N; i++)
143             {
144                 result = Function();
145             }
146         });
147         var ts6 = Performance.Measure(() =>
148         {
149             for (int i = 0; i < N; i++)
150             {
151                 result = 140314;
152             }

```

```

153     });
154
155     var output = $"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {result}";
156     _output.WriteLine(output);
157
158     Assert.True(ts5 < ts1);
159     Assert.True(ts5 < ts2);
160     Assert.True(ts5 < ts3);
161     Assert.True(ts5 < ts4);
162     Assert.True(ts6 < ts1);
163     Assert.True(ts6 < ts2);
164     Assert.True(ts6 < ts3);
165     Assert.True(ts6 < ts4);
166 }
167
168 [MethodImpl(MethodImplOptions.AggressiveInlining)]
169 private static int Function() => 140314;
170 }
171 }

```

./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, 21
- ./Platform.Reflection.Tests/NumericTypeTests.cs, 21
- ./Platform.Reflection/AssemblyExtensions.cs, 1
- ./Platform.Reflection/DelegateHelpers.cs, 1
- ./Platform.Reflection/DynamicExtensions.cs, 2
- ./Platform.Reflection/EnsureExtensions.cs, 2
- ./Platform.Reflection/FieldInfoExtensions.cs, 5
- ./Platform.Reflection/ILGeneratorExtensions.cs, 5
- ./Platform.Reflection/MethodInfoExtensions.cs, 13
- ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 13
- ./Platform.Reflection/NumericType.cs, 13
- ./Platform.Reflection/PropertyInfoExtensions.cs, 14
- ./Platform.Reflection/TypeExtensions.cs, 15
- ./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, 18
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 18
- ./Platform.Reflection/Types[T1, T2, T3, T4].cs, 17
- ./Platform.Reflection/Types[T1, T2, T3].cs, 17
- ./Platform.Reflection/Types[T1, T2].cs, 17
- ./Platform.Reflection/Types[T].cs, 18