

LinksPlatform's Platform.Reflection Class Library

1.1 ./Platform.Reflection/AssemblyExtensions.cs

```
1 using System;
2 using System.Collections.Concurrent;
3 using System.Reflection;
4 using 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 }
```

1.2 ./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             {
41             }
42         }
43     }
44 }
```

```

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 }

```

1.3 ./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 }

```

1.4 ./Platform.Reflection/EnsureExtensions.cs

```

1 using System;
2 using System.Diagnostics;
3 using System.Runtime.CompilerServices;
4 using Platform.Exceptions;
5 using Platform.Exceptions.ExtensionRoots;
6
7 #pragma warning disable IDE0060 // Remove unused parameter

```

```

8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class EnsureExtensions
13     {
14         #region Always
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root,
18         ↪ Func<string> messageBuilder)
19         {
20             if (!NumericType<T>.IsNumeric || NumericType<T>.IsSigned ||
21             ↪ NumericType<T>.IsFloatPoint)
22             {
23                 throw new NotSupportedException(messageBuilder());
24             }
25
26             [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 }

```

1.5 ./Platform.Reflection/FieldInfoExtensions.cs

```

1 using System.Reflection;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Reflection
7 {
8     public static class FieldInfoExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T GetStaticValue<T>(this FieldInfo fieldInfo) =>
12             ↪ (T)fieldInfo.GetValue(null);
13     }
14 }

```

1.6 ./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 UncheckedConvert<TSource, TTarget>(this ILGenerator generator)
16         {
17             var type = typeof(TTarget);
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 if (type == typeof(int))
34     {
35         generator.Emit(OpCodes.Conv_I4);
36     }
37     else if (type == typeof(uint))
38     {
39         generator.Emit(OpCodes.Conv_U4);
40     }
41     else if (type == typeof(long))
42     {
43         generator.Emit(OpCodes.Conv_I8);
44     }
45     else if (type == typeof(ulong))
46     {
47         generator.Emit(OpCodes.Conv_U8);
48     }
49     else if (type == typeof(float))
50     {
51         if (NumericType<TSource>.IsSigned)
52         {
53             generator.Emit(OpCodes.Conv_R4);
54         }
55         else
56         {
57             generator.Emit(OpCodes.Conv_R_Un);
58         }
59     }
60     else if (type == typeof(double))
61     {
62         generator.Emit(OpCodes.Conv_R8);
63     }
64     else
65     {
66         throw new NotSupportedException();
67     }
68 }
69
70 public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
71 {
72     var type = typeof(TTarget);
73     if (type == typeof(short))
74     {
75         if (NumericType<TSource>.IsSigned)
76         {
77             generator.Emit(OpCodes.Conv_Ovf_I2);
78         }
79         else
80         {
81             generator.Emit(OpCodes.Conv_Ovf_I2_Un);
82         }
83     }
84     else if (type == typeof(ushort))
85     {
86         if (NumericType<TSource>.IsSigned)
87         {
88             generator.Emit(OpCodes.Conv_Ovf_U2);
89         }
90         else
91         {
92             generator.Emit(OpCodes.Conv_Ovf_U2_Un);
93         }
94     }
95     else if (type == typeof(sbyte))
96     {
97         if (NumericType<TSource>.IsSigned)
98         {
99             generator.Emit(OpCodes.Conv_Ovf_I1);
100         }
101         else
102         {
103             generator.Emit(OpCodes.Conv_Ovf_I1_Un);
104         }
105     }
106     else if (type == typeof(byte))
107     {
108         if (NumericType<TSource>.IsSigned)
109         {

```

```

110         generator.Emit(OpCodes.Conv_Ovf_U1);
111     }
112     else
113     {
114         generator.Emit(OpCodes.Conv_Ovf_U1_Un);
115     }
116 }
117 else if (type == typeof(int))
118 {
119     if (NumericType<TSource>.IsSigned)
120     {
121         generator.Emit(OpCodes.Conv_Ovf_I4);
122     }
123     else
124     {
125         generator.Emit(OpCodes.Conv_Ovf_I4_Un);
126     }
127 }
128 else if (type == typeof(uint))
129 {
130     if (NumericType<TSource>.IsSigned)
131     {
132         generator.Emit(OpCodes.Conv_Ovf_U4);
133     }
134     else
135     {
136         generator.Emit(OpCodes.Conv_Ovf_U4_Un);
137     }
138 }
139 else if (type == typeof(long))
140 {
141     if (NumericType<TSource>.IsSigned)
142     {
143         generator.Emit(OpCodes.Conv_Ovf_I8);
144     }
145     else
146     {
147         generator.Emit(OpCodes.Conv_Ovf_I8_Un);
148     }
149 }
150 else if (type == typeof(ulong))
151 {
152     if (NumericType<TSource>.IsSigned)
153     {
154         generator.Emit(OpCodes.Conv_Ovf_U8);
155     }
156     else
157     {
158         generator.Emit(OpCodes.Conv_Ovf_U8_Un);
159     }
160 }
161 else if (type == typeof(float))
162 {
163     if (NumericType<TSource>.IsSigned)
164     {
165         generator.Emit(OpCodes.Conv_R4);
166     }
167     else
168     {
169         generator.Emit(OpCodes.Conv_R_Un);
170     }
171 }
172 else if (type == typeof(double))
173 {
174     generator.Emit(OpCodes.Conv_R8);
175 }
176 else
177 {
178     throw new NotSupportedException();
179 }
180 }
181
182 public static void LoadConstant(this ILGenerator generator, bool value) =>
183     ↪ generator.LoadConstant(value ? 1 : 0);
184
185 public static void LoadConstant(this ILGenerator generator, float value) =>
186     ↪ generator.Emit(OpCodes.Ldc_R4, value);

```

```

186 public static void LoadConstant(this ILGenerator generator, double value) =>
187     ↪ generator.Emit(OpCodes.Ldc_R8, value);
188
189 public static void LoadConstant(this ILGenerator generator, ulong value) =>
190     ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
191
192 public static void LoadConstant(this ILGenerator generator, long value) =>
193     ↪ generator.Emit(OpCodes.Ldc_I8, value);
194
195 public static void LoadConstant(this ILGenerator generator, uint value)
196 {
197     switch (value)
198     {
199         case uint.MaxValue:
200             generator.Emit(OpCodes.Ldc_I4_M1);
201             return;
202         case 0:
203             generator.Emit(OpCodes.Ldc_I4_0);
204             return;
205         case 1:
206             generator.Emit(OpCodes.Ldc_I4_1);
207             return;
208         case 2:
209             generator.Emit(OpCodes.Ldc_I4_2);
210             return;
211         case 3:
212             generator.Emit(OpCodes.Ldc_I4_3);
213             return;
214         case 4:
215             generator.Emit(OpCodes.Ldc_I4_4);
216             return;
217         case 5:
218             generator.Emit(OpCodes.Ldc_I4_5);
219             return;
220         case 6:
221             generator.Emit(OpCodes.Ldc_I4_6);
222             return;
223         case 7:
224             generator.Emit(OpCodes.Ldc_I4_7);
225             return;
226         case 8:
227             generator.Emit(OpCodes.Ldc_I4_8);
228             return;
229         default:
230             if (value <= sbyte.MaxValue)
231             {
232                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
233             }
234             else
235             {
236                 generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
237             }
238             return;
239     }
240 }
241
242 public static void LoadConstant(this ILGenerator generator, int value)
243 {
244     switch (value)
245     {
246         case -1:
247             generator.Emit(OpCodes.Ldc_I4_M1);
248             return;
249         case 0:
250             generator.Emit(OpCodes.Ldc_I4_0);
251             return;
252         case 1:
253             generator.Emit(OpCodes.Ldc_I4_1);
254             return;
255         case 2:
256             generator.Emit(OpCodes.Ldc_I4_2);
257             return;
258         case 3:
259             generator.Emit(OpCodes.Ldc_I4_3);
260             return;
261         case 4:
262             generator.Emit(OpCodes.Ldc_I4_4);
263             return;
264         case 5:
265             generator.Emit(OpCodes.Ldc_I4_5);
266             return;

```



```

264         case 6:
265             generator.Emit(OpCodes.Ldc_I4_6);
266             return;
267         case 7:
268             generator.Emit(OpCodes.Ldc_I4_7);
269             return;
270         case 8:
271             generator.Emit(OpCodes.Ldc_I4_8);
272             return;
273         default:
274             if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
275             {
276                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
277             }
278             else
279             {
280                 generator.Emit(OpCodes.Ldc_I4, value);
281             }
282             return;
283     }
284 }
285
286 public static void LoadConstant(this ILGenerator generator, short value)
287 {
288     generator.LoadConstant((int)value);
289 }
290
291 public static void LoadConstant(this ILGenerator generator, ushort value)
292 {
293     generator.LoadConstant((int)value);
294 }
295
296 public static void LoadConstant(this ILGenerator generator, sbyte value)
297 {
298     generator.LoadConstant((int)value);
299 }
300
301 public static void LoadConstant(this ILGenerator generator, byte value)
302 {
303     generator.LoadConstant((int)value);
304 }
305
306 public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
307     ↪ LoadConstantOne(generator, typeof(TConstant));
308
309 public static void LoadConstantOne(this ILGenerator generator, Type constantType)
310 {
311     if (constantType == typeof(float))
312     {
313         generator.LoadConstant(1F);
314     }
315     else if (constantType == typeof(double))
316     {
317         generator.LoadConstant(1D);
318     }
319     else if (constantType == typeof(long))
320     {
321         generator.LoadConstant(1L);
322     }
323     else if (constantType == typeof(ulong))
324     {
325         generator.LoadConstant(1UL);
326     }
327     else if (constantType == typeof(int))
328     {
329         generator.LoadConstant(1);
330     }
331     else if (constantType == typeof(uint))
332     {
333         generator.LoadConstant(1U);
334     }
335     else if (constantType == typeof(short))
336     {
337         generator.LoadConstant((short)1);
338     }
339     else if (constantType == typeof(ushort))
340     {
341         generator.LoadConstant((ushort)1);
342     }
343 }

```

```

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

```

```

416 public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
417
418 public static void Decrement(this ILGenerator generator, Type valueType)
419 {
420     generator.LoadConstantOne(valueType);
421     generator.Subtract();
422 }
423
424 public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
425
426 public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
427
428 public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
429
430 public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
431
432 public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
433
434 public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
435
436 public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
437
438 public static void LoadArgument(this ILGenerator generator, int argumentIndex)
439 {
440     switch (argumentIndex)
441     {
442         case 0:
443             generator.Emit(OpCodes.Ldarg_0);
444             break;
445         case 1:
446             generator.Emit(OpCodes.Ldarg_1);
447             break;
448         case 2:
449             generator.Emit(OpCodes.Ldarg_2);
450             break;
451         case 3:
452             generator.Emit(OpCodes.Ldarg_3);
453             break;
454         default:
455             generator.Emit(OpCodes.Ldarg, argumentIndex);
456             break;
457     }
458 }
459
460 public static void LoadArguments(this ILGenerator generator, params int[]
461     ↪ argumentIndices)
462 {
463     for (var i = 0; i < argumentIndices.Length; i++)
464     {
465         generator.LoadArgument(argumentIndices[i]);
466     }
467 }
468
469 public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
470     ↪ generator.Emit(OpCodes.Starg, argumentIndex);
471
472 public static void CompareGreaterThan(this ILGenerator generator) =>
473     ↪ generator.Emit(OpCodes.Cgt);
474
475 public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
476     ↪ generator.Emit(OpCodes.Cgt_Un);
477
478 public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
479 {
480     if (isSigned)
481     {
482         generator.CompareGreaterThan();
483     }
484     else
485     {
486         generator.UnsignedCompareGreaterThan();
487     }
488 }
489
490 public static void CompareLessThan(this ILGenerator generator) =>
491     ↪ generator.Emit(OpCodes.Clt);
492
493 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
494     ↪ generator.Emit(OpCodes.Clt_Un);

```

```

490 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
491 {
492     if (isSigned)
493     {
494         generator.CompareLessThan();
495     }
496     else
497     {
498         generator.UnsignedCompareLessThan();
499     }
500 }
501
502 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
503     ↪ generator.Emit(OpCodes.Bge, label);
504
505 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
506     ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
507
508 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
509     ↪ Label label)
510 {
511     if (isSigned)
512     {
513         generator.BranchIfGreaterOrEqual(label);
514     }
515     else
516     {
517         generator.UnsignedBranchIfGreaterOrEqual(label);
518     }
519 }
520
521 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
522     ↪ generator.Emit(OpCodes.Ble, label);
523
524 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
525     ↪ => generator.Emit(OpCodes.Ble_Un, label);
526
527 public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
528     ↪ label)
529 {
530     if (isSigned)
531     {
532         generator.BranchIfLessOrEqual(label);
533     }
534     else
535     {
536         generator.UnsignedBranchIfLessOrEqual(label);
537     }
538 }
539
540 public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
541
542 public static void Box(this ILGenerator generator, Type boxedType) =>
543     ↪ generator.Emit(OpCodes.Box, boxedType);
544
545 public static void Call(this ILGenerator generator, MethodInfo method) =>
546     ↪ generator.Emit(OpCodes.Call, method);
547
548 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
549
550 public static void Unbox<TUnbox>(this ILGenerator generator) =>
551     ↪ generator.Unbox(typeof(TUnbox));
552
553 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
554     ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
555
556 public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
557     ↪ generator.UnboxValue(typeof(TUnbox));
558
559 public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
560     ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
561
562 public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
563     ↪ generator.DeclareLocal(typeof(T));
564
565 public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
566     ↪ generator.Emit(OpCodes.Ldloc, local);

```

```

554 public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
555     ↪ generator.Emit(OpCodes.Stloc, local);
556
557 public static void NewObject(this ILGenerator generator, Type type, params Type[]
558     ↪ parameterTypes)
559 {
560     var allConstructors = type.GetConstructors(BindingFlags.Public |
561     ↪ BindingFlags.NonPublic | BindingFlags.Instance
562     ↪ | BindingFlags.CreateInstance
563     ↪ );
564     var constructor = allConstructors.Where(c => c.GetParameters().Length ==
565     ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
566     ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
567     if (constructor == null)
568     {
569         throw new InvalidOperationException("Type " + type + " must have a constructor
570         ↪ that matches parameters [" + string.Join(", ",
571         ↪ parameterTypes.AsEnumerable()) + "]");
572     }
573     generator.NewObject(constructor);
574 }
575
576 public static void NewObject(this ILGenerator generator, ConstructorInfo constructor)
577 {
578     generator.Emit(OpCodes.Newobj, constructor);
579 }
580
581 public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
582     ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
583
584 public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
585     ↪ false, byte? unaligned = null)
586 {
587     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
588     {
589         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
590     }
591     if (isVolatile)
592     {
593         generator.Emit(OpCodes.Volatile);
594     }
595     if (unaligned.HasValue)
596     {
597         generator.Emit(OpCodes.Unaligned, unaligned.Value);
598     }
599     if (type.IsPointer)
600     {
601         generator.Emit(OpCodes.Ldind_I);
602     }
603     else if (!type.IsValueType)
604     {
605         generator.Emit(OpCodes.Ldind_Ref);
606     }
607     else if (type == typeof(sbyte))
608     {
609         generator.Emit(OpCodes.Ldind_I1);
610     }
611     else if (type == typeof(bool))
612     {
613         generator.Emit(OpCodes.Ldind_I1);
614     }
615     else if (type == typeof(byte))
616     {
617         generator.Emit(OpCodes.Ldind_U1);
618     }
619     else if (type == typeof(short))
620     {
621         generator.Emit(OpCodes.Ldind_I2);
622     }
623     else if (type == typeof(ushort))
624     {
625         generator.Emit(OpCodes.Ldind_U2);
626     }
627     else if (type == typeof(char))
628     {
629         generator.Emit(OpCodes.Ldind_U2);
630     }
631     else if (type == typeof(int))
632     {
633         generator.Emit(OpCodes.Ldind_I4);
634     }
635     else if (type == typeof(uint))
636     {
637         generator.Emit(OpCodes.Ldind_U4);
638     }
639     else if (type == typeof(long))
640     {
641         generator.Emit(OpCodes.Ldind_I8);
642     }
643     else if (type == typeof(ulong))
644     {
645         generator.Emit(OpCodes.Ldind_U8);
646     }
647     else
648     {
649         throw new InvalidOperationException("Unsupported type for LoadIndirect: " + type);
650     }
651 }

```

```

622     {
623         generator.Emit(OpCodes.Ldind_U2);
624     }
625     else if (type == typeof(int))
626     {
627         generator.Emit(OpCodes.Ldind_I4);
628     }
629     else if (type == typeof(uint))
630     {
631         generator.Emit(OpCodes.Ldind_U4);
632     }
633     else if (type == typeof(long) || type == typeof(ulong))
634     {
635         generator.Emit(OpCodes.Ldind_I8);
636     }
637     else if (type == typeof(float))
638     {
639         generator.Emit(OpCodes.Ldind_R4);
640     }
641     else if (type == typeof(double))
642     {
643         generator.Emit(OpCodes.Ldind_R8);
644     }
645     else
646     {
647         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
        ↪      ", LoadObject may be more appropriate");
648     }
649 }
650
651 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
652 ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
653
654 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
655 ↪ = false, byte? unaligned = null)
656 {
657     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
658     {
659         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
660     }
661     if (isVolatile)
662     {
663         generator.Emit(OpCodes.Volatile);
664     }
665     if (unaligned.HasValue)
666     {
667         generator.Emit(OpCodes.Unaligned, unaligned.Value);
668     }
669     if (type.IsPointer)
670     {
671         generator.Emit(OpCodes.Stind_I);
672     }
673     else if (!type.IsValueType)
674     {
675         generator.Emit(OpCodes.Stind_Ref);
676     }
677     else if (type == typeof(sbyte) || type == typeof(byte))
678     {
679         generator.Emit(OpCodes.Stind_I1);
680     }
681     else if (type == typeof(short) || type == typeof(ushort))
682     {
683         generator.Emit(OpCodes.Stind_I2);
684     }
685     else if (type == typeof(int) || type == typeof(uint))
686     {
687         generator.Emit(OpCodes.Stind_I4);
688     }
689     else if (type == typeof(long) || type == typeof(ulong))
690     {
691         generator.Emit(OpCodes.Stind_I8);
692     }
693     else if (type == typeof(float))
694     {
695         generator.Emit(OpCodes.Stind_R4);
696     }
697     else if (type == typeof(double))
698     {
699         generator.Emit(OpCodes.Stind_R8);
700     }
701 }

```

```

697         generator.Emit(OpCodes.Stind_R8);
698     }
699     else
700     {
701         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
702             ↳ + ", StoreObject may be more appropriate");
703     }
704 }
705 }

```

1.7 ./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      }

```

1.8 ./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 }

```

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

```

1.10 ./Platform.Reflection/PropertyInfoExtensions.cs

```

1 using System.Reflection;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Reflection
7 {
8     public static class PropertyInfoExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T GetStaticValue<T>(this PropertyInfo fieldInfo) =>
12             ↪ (T)fieldInfo.GetValue(null);
13     }
14 }

```


1.11 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         public static T GetStaticPropertyValue<T>(this Type type, string name) =>
63             ↳ type.GetTypeInfo().GetProperty(name, BindingFlags.Public | BindingFlags.NonPublic |
64             ↳ BindingFlags.Static).GetStaticValue<T>();
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
68             ↳ genericParameterTypes, Type[] argumentTypes)
69         {
70             var methods = from m in type.GetMethods()
71                 where m.Name == name
72                     && m.IsGenericMethodDefinition
73                     let typeParams = m.GetGenericArguments()
74                     let normalParams = m.GetParameters().Select(x => x.ParameterType)
75                     where typeParams.SequenceEqual(genericParameterTypes)
76                     && normalParams.SequenceEqual(argumentTypes)
77                     select m;
78             var method = methods.Single();
79         }
80     }
81 }

```

```

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) =>
81         ↳ type.GetTypeInfo().IsSubclassOf(superClass);
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     public static bool IsValueType(this Type type) => type.GetTypeInfo().IsValueType;
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     public static bool IsGeneric(this Type type) => type.GetTypeInfo().IsGenericType;
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
91         ↳ type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public static Type GetUnsignedVersionOrNull(this Type signedType) =>
98         ↳ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    public static Type GetSignedVersionOrNull(this Type unsignedType) =>
102        ↳ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
103
104    [MethodImpl(MethodImplOptions.AggressiveInlining)]
105    public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
106
107    [MethodImpl(MethodImplOptions.AggressiveInlining)]
108    public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
109
110    [MethodImpl(MethodImplOptions.AggressiveInlining)]
111    public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
112
113    [MethodImpl(MethodImplOptions.AggressiveInlining)]
114    public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
115
116 }

```

1.12 ./Platform.Reflection/Types.cs

```

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

```

```

30         if (element.IsSubclassOf(typeof(Types)))
31         {
32             AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<
33                 ↳ <Type>>(nameof(Types<object>.Collection)));
34         }
35         else
36         {
37             container.Add(element);
38         }
39     }
40 }
41 }
42 }

```

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

```

1 using System;
2 using System.Collections.ObjectModel;
3 using Platform.Collections.Lists;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Reflection
8 {
9     public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
10     {
11         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
12             ↳ T4, T5, T6, T7>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

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

```

1 using System;
2 using System.Collections.ObjectModel;
3 using Platform.Collections.Lists;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Reflection
8 {
9     public class Types<T1, T2, T3, T4, T5, T6> : Types
10     {
11         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
12             ↳ T4, T5, T6>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

1.15 ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs

```

1 using System;
2 using System.Collections.ObjectModel;
3 using Platform.Collections.Lists;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Reflection
8 {
9     public class Types<T1, T2, T3, T4, T5> : Types
10     {
11         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
12             ↳ T4, T5>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

1.16 ./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,
            ↪ T4>().ToReadOnlyCollection();
12         public new static Type[] Array => Collection.ToArray();
13         private Types() { }
14     }
15 }

```

1.17 ./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 }

```

1.18 ./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 }

```

1.19 ./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 }

```

1.20 ./Platform.Reflection.Tests/CodeGenerationTests.cs

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Xunit;
4 using Xunit.Abstractions;
5 using Platform.Diagnostics;
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 [Fact]
82 public void AggressiveInliningEffectTest()
83 {
84     const int N = 10000000;
85
86     int result = 0;
87
88     // Warm up
89
90     for (int i = 0; i < N; i++)
91     {
92         result = MethodsContainer.DelegateWithoutAggressiveInlining();
93     }
94     for (int i = 0; i < N; i++)
95     {

```

```

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 }
```

1.21 ./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 }
```

1.22 ./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, 20
- ./Platform.Reflection.Tests/GetILBytesMethodTests.cs, 23
- ./Platform.Reflection.Tests/NumericTypeTests.cs, 23
- ./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, 15
- ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 15
- ./Platform.Reflection/NumericType.cs, 15
- ./Platform.Reflection/PropertyInfoExtensions.cs, 16
- ./Platform.Reflection/TypeExtensions.cs, 16
- ./Platform.Reflection/Types.cs, 18
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 19
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 19
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 19
- ./Platform.Reflection/Types[T1, T2, T3, T4].cs, 19
- ./Platform.Reflection/Types[T1, T2, T3].cs, 20
- ./Platform.Reflection/Types[T1, T2].cs, 20
- ./Platform.Reflection/Types[T].cs, 20