

## 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 System.Runtime.CompilerServices;
5 using Platform.Exceptions;
6 using Platform.Collections.Lists;
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 AssemblyExtensions
13     {
14         private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
15             ↳ ConcurrentDictionary<Assembly, Type[]>();
16
17         /// <remarks>
18         /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
19         /// </remarks>
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public static Type[] GetLoadableTypes(this Assembly assembly)
22         {
23             Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
24             try
25             {
26                 return assembly.GetTypes();
27             }
28             catch (ReflectionTypeLoadException e)
29             {
30                 return e.Types.ToArray(t => t != null);
31             }
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
36             ↳ _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
37     }
38 }
```

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

```

35     public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
    ↪     aggressiveInlining)
36     where TDelegate : Delegate
37     {
38         var @delegate = CompileOrDefault<TDelegate>(emitCode, aggressiveInlining);
39         if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
40         {
41             @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
42         }
43         return @delegate;
44     }
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
    ↪     : Delegate => Compile<TDelegate>(emitCode, false);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     private static TDelegate CompileUsingDynamicMethod<TDelegate>(Action<ILGenerator>
    ↪     emitCode)
51     {
52         var delegateType = typeof(TDelegate);
53         var invoke = delegateType.GetMethod("Invoke");
54         var returnType = invoke.ReturnType;
55         var parameterTypes = invoke.GetParameters().Select(s => s.ParameterType).ToArray();
56         var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
57         var generator = dynamicMethod.GetILGenerator();
58         emitCode(generator);
59         return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     private static TDelegate CompileUsingMethodBuilder<TDelegate>(Action<ILGenerator>
    ↪     emitCode)
64     {
65         AssemblyName assemblyName = new AssemblyName(GetNewName());
66         var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
    ↪         AssemblyBuilderAccess.Run);
67         var module = assembly.DefineDynamicModule(GetNewName());
68         var type = module.DefineType(GetNewName());
69         var delegateType = typeof(TDelegate);
70         var invoke = delegateType.GetMethod("Invoke");
71         var returnType = invoke.ReturnType;
72         var parameterTypes = invoke.GetParameters().Select(s => s.ParameterType).ToArray();
73         var methodName = GetNewName();
74         MethodBuilder method = type.DefineMethod(methodName, MethodAttributes.Public |
    ↪         MethodAttributes.Static, returnType, parameterTypes);
75         method.SetImplementationFlags(MethodImplAttributes.IL | MethodImplAttributes.Managed
    ↪         | MethodImplAttributes.AggressiveInlining);
76         var generator = method.GetILGenerator();
77         emitCode(generator);
78         var typeInfo = type.CreateTypeInfo();
79         return
    ↪         (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(delegateType);
80     }
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     private static string GetNewName() => Guid.NewGuid().ToString("N");
84 }
85 }

```

### 1.3 ./Platform.Reflection/DynamicExtensions.cs

```

1  using System.Collections.Generic;
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 DynamicExtensions
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static bool HasProperty(this object @object, string propertyName)
12         {
13             var type = @object.GetType();
14             if (type is IDictionary<string, object> dictionary)
15             {
16                 return dictionary.ContainsKey(propertyName);
17             }

```

```

18         return type.GetProperty(propertyName) != null;
19     }
20 }
21 }

```

#### 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                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34                 public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
35                     ↪ IsUnsignedInteger<T>(root, (string)null);
36
37                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38                 public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
39                     ↪ messageBuilder)
40                 {
41                     if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
42                         ↪ NumericType<T>.IsFloatPoint)
43                     {
44                         throw new NotSupportedException(messageBuilder());
45                     }
46
47                     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48                     public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
49                         ↪ message)
50                     {
51                         string messageBuilder() => message;
52                         IsSignedInteger<T>(root, messageBuilder);
53
54                         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55                         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
56                             ↪ IsSignedInteger<T>(root, (string)null);
57
58                         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59                         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
60                             ↪ messageBuilder)
61                         {
62                             if (!NumericType<T>.IsSigned)
63                             {
64                                 throw new NotSupportedException(messageBuilder());
65                             }
66
67                             [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

64 public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
65 {
66     string messageBuilder() => message;
67     IsSigned<T>(root, messageBuilder);
68 }
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
72     ↪ (string)null);
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
76     ↪ messageBuilder)
77 {
78     if (!NumericType<T>.IsNumeric)
79     {
80         throw new NotSupportedException(messageBuilder());
81     }
82 }
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
86 {
87     string messageBuilder() => message;
88     IsNumeric<T>(root, messageBuilder);
89 }
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
93     ↪ IsNumeric<T>(root, (string)null);
94
95 [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
97     ↪ messageBuilder)
98 {
99     if (!NumericType<T>.CanBeNumeric)
100     {
101         throw new NotSupportedException(messageBuilder());
102     }
103 }
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
107 {
108     string messageBuilder() => message;
109     CanBeNumeric<T>(root, messageBuilder);
110 }
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
114     ↪ CanBeNumeric<T>(root, (string)null);
115
116 #endregion
117
118 #region OnDebug
119
120 [Conditional("DEBUG")]
121 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
122     ↪ Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
123
124 [Conditional("DEBUG")]
125 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
126     ↪ message) => Ensure.Always.IsUnsignedInteger<T>(message);
127
128 [Conditional("DEBUG")]
129 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
130     ↪ Ensure.Always.IsUnsignedInteger<T>();
131
132 [Conditional("DEBUG")]
133 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
134     ↪ messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
135
136 [Conditional("DEBUG")]
137 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
138     ↪ message) => Ensure.Always.IsSignedInteger<T>(message);
139
140 [Conditional("DEBUG")]

```

```

131     public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
132         ↪ Ensure.Always.IsSignedInteger<T>();
133
134     [Conditional("DEBUG")]
135     public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
136         ↪ messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
137
138     [Conditional("DEBUG")]
139     public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
140         ↪ Ensure.Always.IsSigned<T>(message);
141
142     [Conditional("DEBUG")]
143     public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
144         ↪ Ensure.Always.IsSigned<T>();
145
146     [Conditional("DEBUG")]
147     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
148         ↪ messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
149
150     [Conditional("DEBUG")]
151     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
152         ↪ Ensure.Always.IsNumeric<T>(message);
153
154     [Conditional("DEBUG")]
155     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
156         ↪ Ensure.Always.IsNumeric<T>();
157
158     [Conditional("DEBUG")]
159     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
160         ↪ messageBuilder) => Ensure.Always.CanBeNumeric<T>(messageBuilder);
161
162     [Conditional("DEBUG")]
163     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
164         ↪ => Ensure.Always.CanBeNumeric<T>(message);
165
166     [Conditional("DEBUG")]
167     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
168         ↪ Ensure.Always.CanBeNumeric<T>();
169
170     #endregion
171 }
172 }

```

## 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 using System.Runtime.CompilerServices;
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 ILGeneratorExtensions
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void Throw<T>(this ILGenerator generator) =>
15             ↪ generator.ThrowException(typeof(T));
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator)

```

```

18 {
19     var type = typeof(TTarget);
20     if (type == typeof(short))
21     {
22         generator.Emit(OpCodes.Conv_I2);
23     }
24     else if (type == typeof(ushort))
25     {
26         generator.Emit(OpCodes.Conv_U2);
27     }
28     else if (type == typeof(sbyte))
29     {
30         generator.Emit(OpCodes.Conv_I1);
31     }
32     else if (type == typeof(byte))
33     {
34         generator.Emit(OpCodes.Conv_U1);
35     }
36     else if (type == typeof(int))
37     {
38         generator.Emit(OpCodes.Conv_I4);
39     }
40     else if (type == typeof(uint))
41     {
42         generator.Emit(OpCodes.Conv_U4);
43     }
44     else if (type == typeof(long))
45     {
46         generator.Emit(OpCodes.Conv_I8);
47     }
48     else if (type == typeof(ulong))
49     {
50         generator.Emit(OpCodes.Conv_U8);
51     }
52     else if (type == typeof(float))
53     {
54         if (NumericType<TSource>.IsSigned)
55         {
56             generator.Emit(OpCodes.Conv_R4);
57         }
58         else
59         {
60             generator.Emit(OpCodes.Conv_R_Un);
61         }
62     }
63     else if (type == typeof(double))
64     {
65         generator.Emit(OpCodes.Conv_R8);
66     }
67     else
68     {
69         throw new NotSupportedException();
70     }
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
75 {
76     var type = typeof(TTarget);
77     if (type == typeof(short))
78     {
79         if (NumericType<TSource>.IsSigned)
80         {
81             generator.Emit(OpCodes.Conv_Ovf_I2);
82         }
83         else
84         {
85             generator.Emit(OpCodes.Conv_Ovf_I2_Un);
86         }
87     }
88     else if (type == typeof(ushort))
89     {
90         if (NumericType<TSource>.IsSigned)
91         {
92             generator.Emit(OpCodes.Conv_Ovf_U2);
93         }
94         else
95         {

```

```

96         generator.Emit(OpCodes.Conv_Ovf_U2_Un);
97     }
98 }
99 else if (type == typeof(sbyte))
100 {
101     if (NumericType<TSource>.IsSigned)
102     {
103         generator.Emit(OpCodes.Conv_Ovf_I1);
104     }
105     else
106     {
107         generator.Emit(OpCodes.Conv_Ovf_I1_Un);
108     }
109 }
110 else if (type == typeof(byte))
111 {
112     if (NumericType<TSource>.IsSigned)
113     {
114         generator.Emit(OpCodes.Conv_Ovf_U1);
115     }
116     else
117     {
118         generator.Emit(OpCodes.Conv_Ovf_U1_Un);
119     }
120 }
121 else if (type == typeof(int))
122 {
123     if (NumericType<TSource>.IsSigned)
124     {
125         generator.Emit(OpCodes.Conv_Ovf_I4);
126     }
127     else
128     {
129         generator.Emit(OpCodes.Conv_Ovf_I4_Un);
130     }
131 }
132 else if (type == typeof(uint))
133 {
134     if (NumericType<TSource>.IsSigned)
135     {
136         generator.Emit(OpCodes.Conv_Ovf_U4);
137     }
138     else
139     {
140         generator.Emit(OpCodes.Conv_Ovf_U4_Un);
141     }
142 }
143 else if (type == typeof(long))
144 {
145     if (NumericType<TSource>.IsSigned)
146     {
147         generator.Emit(OpCodes.Conv_Ovf_I8);
148     }
149     else
150     {
151         generator.Emit(OpCodes.Conv_Ovf_I8_Un);
152     }
153 }
154 else if (type == typeof(ulong))
155 {
156     if (NumericType<TSource>.IsSigned)
157     {
158         generator.Emit(OpCodes.Conv_Ovf_U8);
159     }
160     else
161     {
162         generator.Emit(OpCodes.Conv_Ovf_U8_Un);
163     }
164 }
165 else if (type == typeof(float))
166 {
167     if (NumericType<TSource>.IsSigned)
168     {
169         generator.Emit(OpCodes.Conv_R4);
170     }
171     else
172     {
173         generator.Emit(OpCodes.Conv_R_Un);

```

```

174     }
175 }
176 else if (type == typeof(double))
177 {
178     generator.Emit(OpCodes.Conv_R8);
179 }
180 else
181 {
182     throw new NotSupportedException();
183 }
184 }
185
186 [MethodImpl(MethodImplOptions.AggressiveInlining)]
187 public static void LoadConstant(this ILGenerator generator, bool value) =>
188     ↪ generator.LoadConstant(value ? 1 : 0);
189
190 [MethodImpl(MethodImplOptions.AggressiveInlining)]
191 public static void LoadConstant(this ILGenerator generator, float value) =>
192     ↪ generator.Emit(OpCodes.Ldc_R4, value);
193
194 [MethodImpl(MethodImplOptions.AggressiveInlining)]
195 public static void LoadConstant(this ILGenerator generator, double value) =>
196     ↪ generator.Emit(OpCodes.Ldc_R8, value);
197
198 [MethodImpl(MethodImplOptions.AggressiveInlining)]
199 public static void LoadConstant(this ILGenerator generator, ulong value) =>
200     ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
201
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 public static void LoadConstant(this ILGenerator generator, long value) =>
204     ↪ generator.Emit(OpCodes.Ldc_I8, value);
205
206 [MethodImpl(MethodImplOptions.AggressiveInlining)]
207 public static void LoadConstant(this ILGenerator generator, uint value)
208 {
209     switch (value)
210     {
211         case uint.MaxValue:
212             generator.Emit(OpCodes.Ldc_I4_M1);
213             return;
214         case 0:
215             generator.Emit(OpCodes.Ldc_I4_0);
216             return;
217         case 1:
218             generator.Emit(OpCodes.Ldc_I4_1);
219             return;
220         case 2:
221             generator.Emit(OpCodes.Ldc_I4_2);
222             return;
223         case 3:
224             generator.Emit(OpCodes.Ldc_I4_3);
225             return;
226         case 4:
227             generator.Emit(OpCodes.Ldc_I4_4);
228             return;
229         case 5:
230             generator.Emit(OpCodes.Ldc_I4_5);
231             return;
232         case 6:
233             generator.Emit(OpCodes.Ldc_I4_6);
234             return;
235         case 7:
236             generator.Emit(OpCodes.Ldc_I4_7);
237             return;
238         case 8:
239             generator.Emit(OpCodes.Ldc_I4_8);
240             return;
241         default:
242             if (value <= sbyte.MaxValue)
243             {
244                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
245             }
246             else
247             {
248                 generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
249             }
250             return;
251     }
252 }
253 }
254

```



```

249 [MethodImpl(MethodImplOptions.AggressiveInlining)]
250 public static void LoadConstant(this ILGenerator generator, int value)
251 {
252     switch (value)
253     {
254         case -1:
255             generator.Emit(OpCodes.Ldc_I4_M1);
256             return;
257         case 0:
258             generator.Emit(OpCodes.Ldc_I4_0);
259             return;
260         case 1:
261             generator.Emit(OpCodes.Ldc_I4_1);
262             return;
263         case 2:
264             generator.Emit(OpCodes.Ldc_I4_2);
265             return;
266         case 3:
267             generator.Emit(OpCodes.Ldc_I4_3);
268             return;
269         case 4:
270             generator.Emit(OpCodes.Ldc_I4_4);
271             return;
272         case 5:
273             generator.Emit(OpCodes.Ldc_I4_5);
274             return;
275         case 6:
276             generator.Emit(OpCodes.Ldc_I4_6);
277             return;
278         case 7:
279             generator.Emit(OpCodes.Ldc_I4_7);
280             return;
281         case 8:
282             generator.Emit(OpCodes.Ldc_I4_8);
283             return;
284         default:
285             if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
286             {
287                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
288             }
289             else
290             {
291                 generator.Emit(OpCodes.Ldc_I4, value);
292             }
293             return;
294     }
295 }
296
297 [MethodImpl(MethodImplOptions.AggressiveInlining)]
298 public static void LoadConstant(this ILGenerator generator, short value) =>
299     ↪ generator.LoadConstant((int)value);
300
301 [MethodImpl(MethodImplOptions.AggressiveInlining)]
302 public static void LoadConstant(this ILGenerator generator, ushort value) =>
303     ↪ generator.LoadConstant((int)value);
304
305 [MethodImpl(MethodImplOptions.AggressiveInlining)]
306 public static void LoadConstant(this ILGenerator generator, sbyte value) =>
307     ↪ generator.LoadConstant((int)value);
308
309 [MethodImpl(MethodImplOptions.AggressiveInlining)]
310 public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
311     ↪ LoadConstantOne(generator, typeof(TConstant));
312
313 [MethodImpl(MethodImplOptions.AggressiveInlining)]
314 public static void LoadConstantOne(this ILGenerator generator, Type constantType)
315 {
316     if (constantType == typeof(float))
317     {
318         generator.LoadConstant(1F);
319     }
320     else if (constantType == typeof(double))
321     {
322         generator.LoadConstant(1D);
323     }
324     else if (constantType == typeof(long))

```

```

324     {
325         generator.LoadConstant(1L);
326     }
327     else if (constantType == typeof(ulong))
328     {
329         generator.LoadConstant(1UL);
330     }
331     else if (constantType == typeof(int))
332     {
333         generator.LoadConstant(1);
334     }
335     else if (constantType == typeof(uint))
336     {
337         generator.LoadConstant(1U);
338     }
339     else if (constantType == typeof(short))
340     {
341         generator.LoadConstant((short)1);
342     }
343     else if (constantType == typeof(ushort))
344     {
345         generator.LoadConstant((ushort)1);
346     }
347     else if (constantType == typeof(sbyte))
348     {
349         generator.LoadConstant((sbyte)1);
350     }
351     else if (constantType == typeof(byte))
352     {
353         generator.LoadConstant((byte)1);
354     }
355     else
356     {
357         throw new NotSupportedException();
358     }
359 }
360
361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
362 public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
    ↪  constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
363
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 public static void LoadConstant(this ILGenerator generator, Type constantType, object
    ↪  constantValue)
366 {
367     constantValue = Convert.ChangeType(constantValue, constantType);
368     if (constantType == typeof(float))
369     {
370         generator.LoadConstant((float)constantValue);
371     }
372     else if (constantType == typeof(double))
373     {
374         generator.LoadConstant((double)constantValue);
375     }
376     else if (constantType == typeof(long))
377     {
378         generator.LoadConstant((long)constantValue);
379     }
380     else if (constantType == typeof(ulong))
381     {
382         generator.LoadConstant((ulong)constantValue);
383     }
384     else if (constantType == typeof(int))
385     {
386         generator.LoadConstant((int)constantValue);
387     }
388     else if (constantType == typeof(uint))
389     {
390         generator.LoadConstant((uint)constantValue);
391     }
392     else if (constantType == typeof(short))
393     {
394         generator.LoadConstant((short)constantValue);
395     }
396     else if (constantType == typeof(ushort))
397     {
398         generator.LoadConstant((ushort)constantValue);
399     }

```

```

400     else if (constantType == typeof(sbyte))
401     {
402         generator.LoadConstant((sbyte)constantValue);
403     }
404     else if (constantType == typeof(byte))
405     {
406         generator.LoadConstant((byte)constantValue);
407     }
408     else
409     {
410         throw new NotSupportedException();
411     }
412 }
413
414 [MethodImpl(MethodImplOptions.AggressiveInlining)]
415 public static void Increment<TValue>(this ILGenerator generator) =>
416     ↪ generator.Increment(typeof(TValue));
417
418 [MethodImpl(MethodImplOptions.AggressiveInlining)]
419 public static void Decrement<TValue>(this ILGenerator generator) =>
420     ↪ generator.Decrement(typeof(TValue));
421
422 [MethodImpl(MethodImplOptions.AggressiveInlining)]
423 public static void Increment(this ILGenerator generator, Type valueType)
424 {
425     generator.LoadConstantOne(valueType);
426     generator.Add();
427 }
428
429 [MethodImpl(MethodImplOptions.AggressiveInlining)]
430 public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
431
432 [MethodImpl(MethodImplOptions.AggressiveInlining)]
433 public static void Decrement(this ILGenerator generator, Type valueType)
434 {
435     generator.LoadConstantOne(valueType);
436     generator.Subtract();
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
441
442 [MethodImpl(MethodImplOptions.AggressiveInlining)]
443 public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
444
445 [MethodImpl(MethodImplOptions.AggressiveInlining)]
446 public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
447
448 [MethodImpl(MethodImplOptions.AggressiveInlining)]
449 public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
450
451 [MethodImpl(MethodImplOptions.AggressiveInlining)]
452 public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
453
454 [MethodImpl(MethodImplOptions.AggressiveInlining)]
455 public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
456
457 [MethodImpl(MethodImplOptions.AggressiveInlining)]
458 public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
459
460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
461 public static void LoadArgument(this ILGenerator generator, int argumentIndex)
462 {
463     switch (argumentIndex)
464     {
465         case 0:
466             generator.Emit(OpCodes.Ldarg_0);
467             break;
468         case 1:
469             generator.Emit(OpCodes.Ldarg_1);
470             break;
471         case 2:
472             generator.Emit(OpCodes.Ldarg_2);
473             break;
474         case 3:
475             generator.Emit(OpCodes.Ldarg_3);
476             break;
477         default:
478             generator.Emit(OpCodes.Ldarg, argumentIndex);
479     }
480 }

```

```

477         break;
478     }
479 }
480
481 [MethodImpl(MethodImplOptions.AggressiveInlining)]
482 public static void LoadArguments(this ILGenerator generator, params int[]
    ↳ argumentIndices)
483 {
484     for (var i = 0; i < argumentIndices.Length; i++)
485     {
486         generator.LoadArgument(argumentIndices[i]);
487     }
488 }
489
490 [MethodImpl(MethodImplOptions.AggressiveInlining)]
491 public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
    ↳ generator.Emit(OpCodes.Starg, argumentIndex);
492
493 [MethodImpl(MethodImplOptions.AggressiveInlining)]
494 public static void CompareGreaterThan(this ILGenerator generator) =>
    ↳ generator.Emit(OpCodes.Cgt);
495
496 [MethodImpl(MethodImplOptions.AggressiveInlining)]
497 public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
    ↳ generator.Emit(OpCodes.Cgt_Un);
498
499 [MethodImpl(MethodImplOptions.AggressiveInlining)]
500 public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
501 {
502     if (isSigned)
503     {
504         generator.CompareGreaterThan();
505     }
506     else
507     {
508         generator.UnsignedCompareGreaterThan();
509     }
510 }
511
512 [MethodImpl(MethodImplOptions.AggressiveInlining)]
513 public static void CompareLessThan(this ILGenerator generator) =>
    ↳ generator.Emit(OpCodes.Clt);
514
515 [MethodImpl(MethodImplOptions.AggressiveInlining)]
516 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
    ↳ generator.Emit(OpCodes.Clt_Un);
517
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
520 {
521     if (isSigned)
522     {
523         generator.CompareLessThan();
524     }
525     else
526     {
527         generator.UnsignedCompareLessThan();
528     }
529 }
530
531 [MethodImpl(MethodImplOptions.AggressiveInlining)]
532 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
    ↳ generator.Emit(OpCodes.Bge, label);
533
534 [MethodImpl(MethodImplOptions.AggressiveInlining)]
535 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
    ↳ label) => generator.Emit(OpCodes.Bge_Un, label);
536
537 [MethodImpl(MethodImplOptions.AggressiveInlining)]
538 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
    ↳ Label label)
539 {
540     if (isSigned)
541     {
542         generator.BranchIfGreaterOrEqual(label);
543     }
544     else
545     {

```

```

546         generator.UnsignedBranchIfGreaterOrEqual(label);
547     }
548 }
549
550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
551 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
552     ↪ generator.Emit(OpCodes.Ble, label);
553
554 [MethodImpl(MethodImplOptions.AggressiveInlining)]
555 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
556     ↪ => generator.Emit(OpCodes.Ble_Un, label);
557
558 [MethodImpl(MethodImplOptions.AggressiveInlining)]
559 public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
560     ↪ label)
561 {
562     if (isSigned)
563     {
564         generator.BranchIfLessOrEqual(label);
565     }
566     else
567     {
568         generator.UnsignedBranchIfLessOrEqual(label);
569     }
570 }
571
572 [MethodImpl(MethodImplOptions.AggressiveInlining)]
573 public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
574
575 [MethodImpl(MethodImplOptions.AggressiveInlining)]
576 public static void Box(this ILGenerator generator, Type boxedType) =>
577     ↪ generator.Emit(OpCodes.Box, boxedType);
578
579 [MethodImpl(MethodImplOptions.AggressiveInlining)]
580 public static void Call(this ILGenerator generator, MethodInfo method) =>
581     ↪ generator.Emit(OpCodes.Call, method);
582
583 [MethodImpl(MethodImplOptions.AggressiveInlining)]
584 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
585
586 [MethodImpl(MethodImplOptions.AggressiveInlining)]
587 public static void Unbox<TUnbox>(this ILGenerator generator) =>
588     ↪ generator.Unbox(typeof(TUnbox));
589
590 [MethodImpl(MethodImplOptions.AggressiveInlining)]
591 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
592     ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
593
594 [MethodImpl(MethodImplOptions.AggressiveInlining)]
595 public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
596     ↪ generator.UnboxValue(typeof(TUnbox));
597
598 [MethodImpl(MethodImplOptions.AggressiveInlining)]
599 public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
600     ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
601
602 [MethodImpl(MethodImplOptions.AggressiveInlining)]
603 public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
604     ↪ generator.DeclareLocal(typeof(T));
605
606 [MethodImpl(MethodImplOptions.AggressiveInlining)]
607 public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
608     ↪ generator.Emit(OpCodes.Ldloc, local);
609
610 [MethodImpl(MethodImplOptions.AggressiveInlining)]
611 public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
612     ↪ generator.Emit(OpCodes.Stloc, local);
613
614 [MethodImpl(MethodImplOptions.AggressiveInlining)]
615 public static void NewObject(this ILGenerator generator, Type type, params Type[]
616     ↪ parameterTypes)
617 {
618     var allConstructors = type.GetConstructors(BindingFlags.Public |
619     ↪ BindingFlags.NonPublic | BindingFlags.Instance
620     | BindingFlags.CreateInstance
621     );
622 }

```

```

610     var constructor = allConstructors.Where(c => c.GetParameters().Length ==
        ↳ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
        ↳ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
611     if (constructor == null)
612     {
613         throw new InvalidOperationException("Type " + type + " must have a constructor
        ↳ that matches parameters [" + string.Join(", ",
        ↳ parameterTypes.AsEnumerable()) + "]");
614     }
615     generator.NewObject(constructor);
616 }
617
618 [MethodImpl(MethodImplOptions.AggressiveInlining)]
619 public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
    ↳ generator.Emit(OpCodes.Newobj, constructor);
620
621 [MethodImpl(MethodImplOptions.AggressiveInlining)]
622 public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
    ↳ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
623
624 [MethodImpl(MethodImplOptions.AggressiveInlining)]
625 public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
    ↳ false, byte? unaligned = null)
626 {
627     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
628     {
629         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
630     }
631     if (isVolatile)
632     {
633         generator.Emit(OpCodes.Volatile);
634     }
635     if (unaligned.HasValue)
636     {
637         generator.Emit(OpCodes.Unaligned, unaligned.Value);
638     }
639     if (type.IsPointer)
640     {
641         generator.Emit(OpCodes.Ldind_I);
642     }
643     else if (!type.IsValueType)
644     {
645         generator.Emit(OpCodes.Ldind_Ref);
646     }
647     else if (type == typeof(sbyte))
648     {
649         generator.Emit(OpCodes.Ldind_I1);
650     }
651     else if (type == typeof(bool))
652     {
653         generator.Emit(OpCodes.Ldind_I1);
654     }
655     else if (type == typeof(byte))
656     {
657         generator.Emit(OpCodes.Ldind_U1);
658     }
659     else if (type == typeof(short))
660     {
661         generator.Emit(OpCodes.Ldind_I2);
662     }
663     else if (type == typeof(ushort))
664     {
665         generator.Emit(OpCodes.Ldind_U2);
666     }
667     else if (type == typeof(char))
668     {
669         generator.Emit(OpCodes.Ldind_U2);
670     }
671     else if (type == typeof(int))
672     {
673         generator.Emit(OpCodes.Ldind_I4);
674     }
675     else if (type == typeof(uint))
676     {
677         generator.Emit(OpCodes.Ldind_U4);
678     }
679     else if (type == typeof(long) || type == typeof(ulong))

```

```

680     {
681         generator.Emit(OpCodes.Ldind_I8);
682     }
683     else if (type == typeof(float))
684     {
685         generator.Emit(OpCodes.Ldind_R4);
686     }
687     else if (type == typeof(double))
688     {
689         generator.Emit(OpCodes.Ldind_R8);
690     }
691     else
692     {
693         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
        ↪      ", LoadObject may be more appropriate");
694     }
695 }
696
697 [MethodImpl(MethodImplOptions.AggressiveInlining)]
698 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
699 ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
700
701 [MethodImpl(MethodImplOptions.AggressiveInlining)]
702 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
703 ↪ = false, byte? unaligned = null)
704 {
705     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
706     {
707         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
708     }
709     if (isVolatile)
710     {
711         generator.Emit(OpCodes.Volatile);
712     }
713     if (unaligned.HasValue)
714     {
715         generator.Emit(OpCodes.Unaligned, unaligned.Value);
716     }
717     if (type.IsPointer)
718     {
719         generator.Emit(OpCodes.Stind_I);
720     }
721     else if (!type.IsValueType)
722     {
723         generator.Emit(OpCodes.Stind_Ref);
724     }
725     else if (type == typeof(sbyte) || type == typeof(byte))
726     {
727         generator.Emit(OpCodes.Stind_I1);
728     }
729     else if (type == typeof(short) || type == typeof(ushort))
730     {
731         generator.Emit(OpCodes.Stind_I2);
732     }
733     else if (type == typeof(int) || type == typeof(uint))
734     {
735         generator.Emit(OpCodes.Stind_I4);
736     }
737     else if (type == typeof(long) || type == typeof(ulong))
738     {
739         generator.Emit(OpCodes.Stind_I8);
740     }
741     else if (type == typeof(float))
742     {
743         generator.Emit(OpCodes.Stind_R4);
744     }
745     else if (type == typeof(double))
746     {
747         generator.Emit(OpCodes.Stind_R8);
748     }
749     else
750     {
751         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
752 ↪      + ", StoreObject may be more appropriate");
753     }
754 }
755 }

```

### 1.7 ./Platform.Reflection/MethodInfoExtensions.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 MethodInfoExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static byte[] GetILBytes(this MethodInfo methodInfo) =>
12             ↪ methodInfo.GetMethodBody().GetILAsByteArray();
13     }
14 }
```

### 1.8 ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Interfaces;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Reflection
9 {
10     public class NotSupportedExceptionDelegateFactory<TDelegate> : IFactory<TDelegate>
11         where TDelegate : Delegate
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public TDelegate Create()
15         {
16             var @delegate = DelegateHelpers.CompileOrDefault<TDelegate>(generator =>
17             {
18                 generator.Throw<NotSupportedException>();
19             });
20             if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
21             {
22                 throw new InvalidOperationException("Unable to compile stub delegate.");
23             }
24             return @delegate;
25         }
26     }
27 }
```

### 1.9 ./Platform.Reflection/NumericType.cs

```
1 using System;
2 using System.Runtime.CompilerServices;
3 using System.Runtime.InteropServices;
4 using Platform.Exceptions;
5
6 // ReSharper disable AssignmentInConditionalExpression
7 // ReSharper disable BuiltInTypeReferenceStyle
8 // ReSharper disable StaticFieldInGenericType
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Reflection
12 {
13     public static class NumericType<T>
14     {
15         public static readonly Type Type;
16         public static readonly Type UnderlyingType;
17         public static readonly Type SignedVersion;
18         public static readonly Type UnsignedVersion;
19         public static readonly bool IsFloatPoint;
20         public static readonly bool IsNumeric;
21         public static readonly bool IsSigned;
22         public static readonly bool CanBeNumeric;
23         public static readonly bool IsNullable;
24         public static readonly int BitsLength;
25         public static readonly T MinValue;
26         public static readonly T MaxValue;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         static NumericType()
30         {
31             try
32             {
33                 var type = typeof(T);
34                 var isNullable = type.IsNullable();
35             }
36         }
37     }
38 }
```



```

35     var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
36     var canBeNumeric = underlyingType.CanBeNumeric();
37     var isNumeric = underlyingType.IsNumeric();
38     var isSigned = underlyingType.IsSigned();
39     var isFloatPoint = underlyingType.IsFloatPoint();
40     var bitsLength = Marshal.SizeOf(underlyingType) * 8;
41     GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
42     GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
    ↪ out Type unsignedVersion);
43     Type = type;
44     IsNullable = isNullable;
45     UnderlyingType = underlyingType;
46     CanBeNumeric = canBeNumeric;
47     IsNumeric = isNumeric;
48     IsSigned = isSigned;
49     IsFloatPoint = isFloatPoint;
50     BitsLength = bitsLength;
51     MinValue = minValue;
52     MaxValue = maxValue;
53     SignedVersion = signedVersion;
54     UnsignedVersion = unsignedVersion;
55 }
56 catch (Exception exception)
57 {
58     exception.Ignore();
59 }
60 }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
64 {
65     if (type == typeof(bool))
66     {
67         minValue = (T)(object>false;
68         maxValue = (T)(object>true;
69     }
70     else
71     {
72         minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
73         maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
74     }
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
    ↪ signedVersion, out Type unsignedVersion)
79 {
80     if (isSigned)
81     {
82         signedVersion = type;
83         unsignedVersion = type.GetUnsignedVersionOrNull();
84     }
85     else
86     {
87         signedVersion = type.GetSignedVersionOrNull();
88         unsignedVersion = type;
89     }
90 }
91 }
92 }

```

## 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) =>
            ↪ (T)fieldInfo.GetValue(null);
12     }
13 }

```

## 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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         static TypeExtensions()
23         {
24             _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
25                 ↳ typeof(DateTime), typeof(TimeSpan) };
26             _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
27                 ↳ typeof(ulong) };
28             _canBeNumericTypes.UnionWith(_isNumericTypes);
29             _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
30                 ↳ typeof(long) };
31             _canBeNumericTypes.UnionWith(_isSignedTypes);
32             _isNumericTypes.UnionWith(_isSignedTypes);
33             _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
34                 ↳ typeof(float) };
35             _canBeNumericTypes.UnionWith(_isFloatPointTypes);
36             _isNumericTypes.UnionWith(_isFloatPointTypes);
37             _isSignedTypes.UnionWith(_isFloatPointTypes);
38             _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
39             {
40                 { typeof(sbyte), typeof(byte) },
41                 { typeof(short), typeof(ushort) },
42                 { typeof(int), typeof(uint) },
43                 { typeof(long), typeof(ulong) },
44             };
45             _signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
46             {
47                 { typeof(byte), typeof(sbyte) },
48                 { typeof(ushort), typeof(short) },
49                 { typeof(uint), typeof(int) },
50                 { typeof(ulong), typeof(long) },
51             };
52         }
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         public static T GetStaticFieldValue<T>(this Type type, string name) =>
59             ↳ type.GetTypeInfo().GetField(name, BindingFlags.Public | BindingFlags.NonPublic |
60             ↳ BindingFlags.Static).GetStaticValue<T>();
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public static T GetStaticPropertyValue<T>(this Type type, string name) =>
64             ↳ type.GetTypeInfo().GetProperty(name, BindingFlags.Public | BindingFlags.NonPublic |
65             ↳ BindingFlags.Static).GetStaticValue<T>();
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
69             ↳ genericParameterTypes, Type[] argumentTypes)
70         {
71             var methods = from m in type.GetMethods()
72                 where m.Name == name
73                     && m.IsGenericMethodDefinition
74                     let typeParams = m.GetGenericArguments()
75                     let normalParams = m.GetParameters().Select(x => x.ParameterType)
76                     where typeParams.SequenceEqual(genericParameterTypes)
77                     && normalParams.SequenceEqual(argumentTypes)
78                     select m;
79             var method = methods.Single();
80             return method;
81         }
82     }
83 }

```

```

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    }
117 }

```

## 1.12 ./Platform.Reflection/Types.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Collections.ObjectModel;
4  using System.Runtime.CompilerServices;
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 abstract class Types
12     {
13         public static ReadOnlyCollection<Type> Collection { get; } = new
14             ↳ ReadOnlyCollection<Type>(System.Array.Empty<Type>());
15         public static Type[] Array => Collection.ToArray();
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected ReadOnlyCollection<Type> ToReadOnlyCollection()
19         {
20             var types = GetType().GetGenericArguments();
21             var result = new List<Type>();
22             AppendTypes(result, types);
23             return new ReadOnlyCollection<Type>(result);
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         private static void AppendTypes(List<Type> container, IList<Type> types)
28         {
29             for (var i = 0; i < types.Count; i++)
30             {
31                 var element = types[i];
32                 if (element != typeof(Types))

```

```

32         {
33             if (element.IsSubclassOf(typeof(Types)))
34             {
35                 AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<
36                     ↳ <Type>>(nameof(Types<object>.Collection)));
37             }
38             else
39             {
40                 container.Add(element);
41             }
42         }
43     }
44 }
45 }

```

### 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     for (int i = 0; i < N; i++)
90     {
91         result = MethodsContainer.DelegateWithoutAggressiveInlining();
92     }
93     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 }

```

## 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, 21
- ./Platform.Reflection.Tests/GetILBytesMethodTests.cs, 24
- ./Platform.Reflection.Tests/NumericTypeTests.cs, 24
- ./Platform.Reflection/AssemblyExtensions.cs, 1
- ./Platform.Reflection/DelegateHelpers.cs, 1
- ./Platform.Reflection/DynamicExtensions.cs, 2
- ./Platform.Reflection/EnsureExtensions.cs, 3
- ./Platform.Reflection/FieldInfoExtensions.cs, 5
- ./Platform.Reflection/ILGeneratorExtensions.cs, 5
- ./Platform.Reflection/MethodInfoExtensions.cs, 16
- ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 16
- ./Platform.Reflection/NumericType.cs, 16
- ./Platform.Reflection/PropertyInfoExtensions.cs, 17
- ./Platform.Reflection/TypeExtensions.cs, 17
- ./Platform.Reflection/Types.cs, 19
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 20
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 20
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 20
- ./Platform.Reflection/Types[T1, T2, T3, T4].cs, 20
- ./Platform.Reflection/Types[T1, T2, T3].cs, 21
- ./Platform.Reflection/Types[T1, T2].cs, 21
- ./Platform.Reflection/Types[T].cs, 21