

# 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.Reflection;
4 using System.Reflection.Emit;
5 using System.Runtime.CompilerServices;
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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode, bool
16             ↳ typeMemberMethod)
17             where TDelegate : Delegate
18         {
19             var @delegate = default(TDelegate);
20             try
21             {
22                 @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
23                     ↳ CompileDynamicMethod<TDelegate>(emitCode);
24             }
25             catch (Exception exception)
26             {
27                 exception.Ignore();
28             }
29             return @delegate;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
34             ↳ TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
38             ↳ typeMemberMethod)
39         {
40             // ...
41         }
42     }
43 }
```

```

35     where TDelegate : Delegate
36 {
37     var @delegate = CompileOrDefault<TDelegate>(emitCode, typeMemberMethod);
38     if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
39     {
40         @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
41     }
42     return @delegate;
43 }
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
    ↳ : Delegate => Compile<TDelegate>(emitCode, false);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 public static TDelegate CompileDynamicMethod<TDelegate>(Action<ILGenerator> emitCode)
50 {
51     var delegateType = typeof(TDelegate);
52     delegateType.GetDelegateCharacteristics(out Type returnType, out Type[]
        ↳ parameterTypes);
53     var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
54     emitCode(dynamicMethod.GetILGenerator());
55     return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public static TDelegate CompileTypeMemberMethod<TDelegate>(Action<ILGenerator> emitCode)
60 {
61     AssemblyName assemblyName = new AssemblyName(GetNewName());
62     var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
        ↳ AssemblyBuilderAccess.Run);
63     var module = assembly.DefineDynamicModule(GetNewName());
64     var type = module.DefineType(GetNewName());
65     var methodName = GetNewName();
66     type.EmitStaticMethod<TDelegate>(methodName, emitCode);
67     var typeInfo = type.CreateTypeInfo();
68     return (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(typeof(TDele
        ↳ gate));
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 private static string GetNewName() => Guid.NewGuid().ToString("N");
73 }
74 }

```

### 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
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
29         ↪ message)
30         {
31             string messageBuilder() => message;
32             IsUnsignedInteger<T>(root, messageBuilder());
33         }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
37         ↪ IsUnsignedInteger<T>(root, (string)null);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
41         ↪ messageBuilder)
42         {
43             if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
44             ↪ NumericType<T>.IsFloatPoint)
45             {
46                 throw new NotSupportedException(messageBuilder());
47             }
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
52         ↪ message)
53         {
54             string messageBuilder() => message;
55             IsSignedInteger<T>(root, messageBuilder());
56         }
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
60         ↪ IsSignedInteger<T>(root, (string)null);
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
64         ↪ messageBuilder)
65         {
66             if (!NumericType<T>.IsSigned)
67             {
68                 throw new NotSupportedException(messageBuilder());
69             }
70         }
71
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
74         {
75             string messageBuilder() => message;
76             IsSigned<T>(root, messageBuilder());
77         }
78
79         [MethodImpl(MethodImplOptions.AggressiveInlining)]
80         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
81         ↪ (string)null);
82
83         [MethodImpl(MethodImplOptions.AggressiveInlining)]
84         public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
85         ↪ messageBuilder)
86         {
87             if (!NumericType<T>.IsNumeric)
88             {
89                 throw new NotSupportedException(messageBuilder());
90             }
91         }
92     }
93 }

```

```

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>();
153
154     [Conditional("DEBUG")]
155 
```

```

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

```

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

```

### 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) =>
            ↳ generator.ThrowException(typeof(T));
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         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         }
37     }
38 }

```

```

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)]

```

```

190 public static void LoadConstant(this ILGenerator generator, float value) =>
191     ↪ generator.Emit(OpCodes.Ldc_R4, value);
192
193 [MethodImpl(MethodImplOptions.AggressiveInlining)]
194 public static void LoadConstant(this ILGenerator generator, double value) =>
195     ↪ generator.Emit(OpCodes.Ldc_R8, value);
196
197 [MethodImpl(MethodImplOptions.AggressiveInlining)]
198 public static void LoadConstant(this ILGenerator generator, ulong value) =>
199     ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
200
201 [MethodImpl(MethodImplOptions.AggressiveInlining)]
202 public static void LoadConstant(this ILGenerator generator, long value) =>
203     ↪ generator.Emit(OpCodes.Ldc_I8, value);
204
205 [MethodImpl(MethodImplOptions.AggressiveInlining)]
206 public static void LoadConstant(this ILGenerator generator, uint value)
207 {
208     switch (value)
209     {
210         case uint.MaxValue:
211             generator.Emit(OpCodes.Ldc_I4_M1);
212             return;
213         case 0:
214             generator.Emit(OpCodes.Ldc_I4_0);
215             return;
216         case 1:
217             generator.Emit(OpCodes.Ldc_I4_1);
218             return;
219         case 2:
220             generator.Emit(OpCodes.Ldc_I4_2);
221             return;
222         case 3:
223             generator.Emit(OpCodes.Ldc_I4_3);
224             return;
225         case 4:
226             generator.Emit(OpCodes.Ldc_I4_4);
227             return;
228         case 5:
229             generator.Emit(OpCodes.Ldc_I4_5);
230             return;
231         case 6:
232             generator.Emit(OpCodes.Ldc_I4_6);
233             return;
234         case 7:
235             generator.Emit(OpCodes.Ldc_I4_7);
236             return;
237         case 8:
238             generator.Emit(OpCodes.Ldc_I4_8);
239             return;
240         default:
241             if (value <= sbyte.MaxValue)
242             {
243                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
244             }
245             else
246             {
247                 generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
248             }
249             return;
250     }
251 }
252
253 [MethodImpl(MethodImplOptions.AggressiveInlining)]
254 public static void LoadConstant(this ILGenerator generator, int value)
255 {
256     switch (value)
257     {
258         case -1:
259             generator.Emit(OpCodes.Ldc_I4_M1);
260             return;
261         case 0:
262             generator.Emit(OpCodes.Ldc_I4_0);
263             return;
264         case 1:
265             generator.Emit(OpCodes.Ldc_I4_1);
266             return;
267         case 2:
268             generator.Emit(OpCodes.Ldc_I4_2);
269             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))
325     {
326         generator.LoadConstant(1L);
327     }
328     else if (constantType == typeof(ulong))
329     {
330         generator.LoadConstant(1UL);
331     }
332     else if (constantType == typeof(int))
333     {
334         generator.LoadConstant(1);
335     }
336     else if (constantType == typeof(uint))
337     {
338         generator.LoadConstant(1U);
339     }
340     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             break;
480     }
481 }
482
483 [MethodImpl(MethodImplOptions.AggressiveInlining)]
484 public static void LoadArguments(this ILGenerator generator, params int[]
485     ↪ argumentIndices)
486 {
487     for (var i = 0; i < argumentIndices.Length; i++)
488     {
489         generator.LoadArgument(argumentIndices[i]);
490     }
491 }
492
493 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

491 public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
492     ↪ generator.Emit(OpCodes.Starg, argumentIndex);
493
494 [MethodImpl(MethodImplOptions.AggressiveInlining)]
495 public static void CompareGreaterThan(this ILGenerator generator) =>
496     ↪ generator.Emit(OpCodes.Cgt);
497
498 [MethodImpl(MethodImplOptions.AggressiveInlining)]
499 public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
500     ↪ generator.Emit(OpCodes.Cgt_Un);
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
504 {
505     if (isSigned)
506     {
507         generator.CompareGreaterThan();
508     }
509     else
510     {
511         generator.UnsignedCompareGreaterThan();
512     }
513 }
514
515 [MethodImpl(MethodImplOptions.AggressiveInlining)]
516 public static void CompareLessThan(this ILGenerator generator) =>
517     ↪ generator.Emit(OpCodes.Clt);
518
519 [MethodImpl(MethodImplOptions.AggressiveInlining)]
520 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
521     ↪ generator.Emit(OpCodes.Clt_Un);
522
523 [MethodImpl(MethodImplOptions.AggressiveInlining)]
524 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
525 {
526     if (isSigned)
527     {
528         generator.CompareLessThan();
529     }
530     else
531     {
532         generator.UnsignedCompareLessThan();
533     }
534 }
535
536 [MethodImpl(MethodImplOptions.AggressiveInlining)]
537 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
538     ↪ generator.Emit(OpCodes.Bge, label);
539
540 [MethodImpl(MethodImplOptions.AggressiveInlining)]
541 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
542     ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
543
544 [MethodImpl(MethodImplOptions.AggressiveInlining)]
545 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
546     ↪ Label label)
547 {
548     if (isSigned)
549     {
550         generator.BranchIfGreaterOrEqual(label);
551     }
552     else
553     {
554         generator.UnsignedBranchIfGreaterOrEqual(label);
555     }
556 }
557
558 [MethodImpl(MethodImplOptions.AggressiveInlining)]
559 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
560     ↪ generator.Emit(OpCodes.Ble, label);
561
562 [MethodImpl(MethodImplOptions.AggressiveInlining)]
563 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
564     ↪ => generator.Emit(OpCodes.Ble_Un, label);
565
566 [MethodImpl(MethodImplOptions.AggressiveInlining)]
567 public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
568     ↪ label)

```

```

558 {
559     if (isSigned)
560     {
561         generator.BranchIfLessOrEqual(label);
562     }
563     else
564     {
565         generator.UnsignedBranchIfLessOrEqual(label);
566     }
567 }
568
569 [MethodImpl(MethodImplOptions.AggressiveInlining)]
570 public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
571
572 [MethodImpl(MethodImplOptions.AggressiveInlining)]
573 public static void Box(this ILGenerator generator, Type boxedType) =>
574     ↪ generator.Emit(OpCodes.Box, boxedType);
575
576 [MethodImpl(MethodImplOptions.AggressiveInlining)]
577 public static void Call(this ILGenerator generator, MethodInfo method) =>
578     ↪ generator.Emit(OpCodes.Call, method);
579
580 [MethodImpl(MethodImplOptions.AggressiveInlining)]
581 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
582
583 [MethodImpl(MethodImplOptions.AggressiveInlining)]
584 public static void Unbox<TUnbox>(this ILGenerator generator) =>
585     ↪ generator.Unbox(typeof(TUnbox));
586
587 [MethodImpl(MethodImplOptions.AggressiveInlining)]
588 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
589     ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
590
591 [MethodImpl(MethodImplOptions.AggressiveInlining)]
592 public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
593     ↪ generator.UnboxValue(typeof(TUnbox));
594
595 [MethodImpl(MethodImplOptions.AggressiveInlining)]
596 public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
597     ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
598
599 [MethodImpl(MethodImplOptions.AggressiveInlining)]
600 public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
601     ↪ generator.DeclareLocal(typeof(T));
602
603 [MethodImpl(MethodImplOptions.AggressiveInlining)]
604 public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
605     ↪ generator.Emit(OpCodes.Ldloc, local);
606
607 [MethodImpl(MethodImplOptions.AggressiveInlining)]
608 public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
609     ↪ generator.Emit(OpCodes.Stloc, local);
610
611 [MethodImpl(MethodImplOptions.AggressiveInlining)]
612 public static void NewObject(this ILGenerator generator, Type type, params Type[]
613     ↪ parameterTypes)
614 {
615     var allConstructors = type.GetConstructors(BindingFlags.Public |
616     ↪ BindingFlags.NonPublic | BindingFlags.Instance
617     #if !NETSTANDARD
618     | BindingFlags.CreateInstance
619     #endif
620     );
621     var constructor = allConstructors.Where(c => c.GetParameters().Length ==
622     ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
623     ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
624     if (constructor == null)
625     {
626         throw new InvalidOperationException("Type " + type + " must have a constructor
627         ↪ that matches parameters [" + string.Join(", ",
628         ↪ parameterTypes.AsEnumerable()) + "]");
629     }
630     generator.NewObject(constructor);
631 }
632
633 [MethodImpl(MethodImplOptions.AggressiveInlining)]
634 public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
635     ↪ generator.Emit(OpCodes.Newobj, constructor);

```

```

620 [MethodImpl(MethodImplOptions.AggressiveInlining)]
621 public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
622 ↪ 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 =
626 ↪ false, byte? unaligned = null)
627 {
628     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
629     {
630         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
631     }
632     if (isVolatile)
633     {
634         generator.Emit(OpCodes.Volatile);
635     }
636     if (unaligned.HasValue)
637     {
638         generator.Emit(OpCodes.Unaligned, unaligned.Value);
639     }
640     if (type.IsPointer)
641     {
642         generator.Emit(OpCodes.Ldind_I);
643     }
644     else if (!type.IsValueType)
645     {
646         generator.Emit(OpCodes.Ldind_Ref);
647     }
648     else if (type == typeof(sbyte))
649     {
650         generator.Emit(OpCodes.Ldind_I1);
651     }
652     else if (type == typeof(bool))
653     {
654         generator.Emit(OpCodes.Ldind_I1);
655     }
656     else if (type == typeof(byte))
657     {
658         generator.Emit(OpCodes.Ldind_U1);
659     }
660     else if (type == typeof(short))
661     {
662         generator.Emit(OpCodes.Ldind_I2);
663     }
664     else if (type == typeof(ushort))
665     {
666         generator.Emit(OpCodes.Ldind_U2);
667     }
668     else if (type == typeof(char))
669     {
670         generator.Emit(OpCodes.Ldind_U2);
671     }
672     else if (type == typeof(int))
673     {
674         generator.Emit(OpCodes.Ldind_I4);
675     }
676     else if (type == typeof(uint))
677     {
678         generator.Emit(OpCodes.Ldind_U4);
679     }
680     else if (type == typeof(long) || type == typeof(ulong))
681     {
682         generator.Emit(OpCodes.Ldind_I8);
683     }
684     else if (type == typeof(float))
685     {
686         generator.Emit(OpCodes.Ldind_R4);
687     }
688     else if (type == typeof(double))
689     {
690         generator.Emit(OpCodes.Ldind_R8);
691     }
692     else
693     {
694         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
695 ↪ ", LoadObject may be more appropriate");

```

```

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

```

## 1.7 ./Platform.Reflection/MethodInfoExtensions.cs

```

1  using System;
2  using System.Linq;
3  using System.Reflection;
4  using System.Runtime.CompilerServices;
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 MethodInfoExtensions
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static byte[] GetILBytes(this MethodInfo methodInfo) =>
   ↪ methodInfo.GetMethodBody().GetILAsByteArray();

```

```

14     [MethodImpl(MethodImplOptions.AggressiveInlining)]
15     public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
16         ↪ methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
17 }
18 }

```

## 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                 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,
43                     ↪ 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/TypeBuilderExtensions.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  using System;
4  using System.Reflection;
5  using System.Reflection.Emit;
6  using System.Runtime.CompilerServices;
7
8  namespace Platform.Reflection
9  {
10     public static class TypeBuilderExtensions
11     {

```

```

12     public static readonly MethodAttributes DefaultStaticMethodAttributes =
13         ↪ MethodAttributes.Public | MethodAttributes.Static;
14     public static readonly MethodAttributes DefaultVirtualMethodAttributes =
15         ↪ MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
16         ↪ MethodAttributes.HideBySig;
17     public static readonly MethodImplAttributes DefaultMethodImplAttributes =
18         ↪ MethodImplAttributes.IL | MethodImplAttributes.Managed |
19         ↪ MethodImplAttributes.AggressiveInlining;
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName,
23         ↪ MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes,
24         ↪ Action<ILGenerator> emitCode)
25     {
26         typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[]
27             ↪ parameterTypes);
28         EmitMethod(type, methodName, methodAttributes, methodImplAttributes, returnType,
29             ↪ parameterTypes, emitCode);
30     }
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
34         ↪ methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[]
35         ↪ parameterTypes, Action<ILGenerator> emitCode)
36     {
37         MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
38             ↪ parameterTypes);
39         method.SetImplementationFlags(methodImplAttributes);
40         var generator = method.GetILGenerator();
41         emitCode(generator);
42     }
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName,
46         ↪ Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
47         ↪ DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     public static void EmitVirtualMethod<TDelegate>(this TypeBuilder type, string
51         ↪ methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
52         ↪ DefaultVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
53 }
54 }

```

## 1.12 ./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 public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
15             ↪ BindingFlags.NonPublic | BindingFlags.Static;
16         static public readonly string DefaultDelegateMethodName = "Invoke";
17
18         static private readonly HashSet<Type> _canBeNumericTypes;
19         static private readonly HashSet<Type> _isNumericTypes;
20         static private readonly HashSet<Type> _isSignedTypes;
21         static private readonly HashSet<Type> _isFloatPointTypes;
22         static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
23         static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         static TypeExtensions()
27         {
28             _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
29                 ↪ typeof(DateTime), typeof(TimeSpan) };
30             _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
31                 ↪ typeof(ulong) };
32             _canBeNumericTypes.UnionWith(_isNumericTypes);
33             _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
34                 ↪ typeof(long) };
35         }
36     }
37 }

```

```

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 [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.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public static T GetStaticPropertyValue<T>(this Type type, string name) =>
62     ↳ type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
66     ↳ genericParameterTypes, Type[] argumentTypes)
67 {
68     var methods = from m in type.GetMethods()
69         where m.Name == name
70             && m.IsGenericMethodDefinition
71             let typeParams = m.GetGenericArguments()
72             let normalParams = m.GetParameters().Select(x => x.ParameterType)
73             where typeParams.SequenceEqual(genericParameterTypes)
74             && normalParams.SequenceEqual(argumentTypes)
75             select m;
76     var method = methods.Single();
77     return method;
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public static Type GetBaseType(this Type type) => type.BaseType;
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 public static Assembly GetAssembly(this Type type) => type.Assembly;
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 public static bool IsSubclassOf(this Type type, Type superClass) =>
88     ↳ type.IsSubclassOf(superClass);
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public static bool IsValueType(this Type type) => type.IsValueType;
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 public static bool IsGeneric(this Type type) => type.IsGenericType;
95
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
98     ↳ type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public static Type GetUnsignedVersionOrNull(this Type signedType) =>
105     ↳ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

102     public static Type GetSignedVersionOrNull(this Type unsignedType) =>
103         ↪ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
104
105     [MethodImpl(MethodImplOptions.AggressiveInlining)]
106     public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
110
111     [MethodImpl(MethodImplOptions.AggressiveInlining)]
112     public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
113
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
116
117     [MethodImpl(MethodImplOptions.AggressiveInlining)]
118     public static Type GetDelegateReturnType(this Type delegateType) =>
119         ↪ delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;
120
121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
123         ↪ delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();
124
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]
126     public static void GetDelegateCharacteristics(this Type delegateType, out Type
127         ↪ returnType, out Type[] parameterTypes)
128     {
129         var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
130         returnType = invoke.ReturnType;
131         parameterTypes = invoke.GetParameterTypes();
132     }
133 }

```

### 1.13 ./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  #pragma warning disable CA1819 // Properties should not return arrays
9
10 namespace Platform.Reflection
11 {
12     public abstract class Types
13     {
14         public static ReadOnlyCollection<Type> Collection { get; } = new
15             ↪ ReadOnlyCollection<Type>(System.Array.Empty<Type>());
16         public static Type[] Array => Collection.ToArray();
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected ReadOnlyCollection<Type> ToReadOnlyCollection()
20         {
21             var types = GetType().GetGenericArguments();
22             var result = new List<Type>();
23             AppendTypes(result, types);
24             return new ReadOnlyCollection<Type>(result);
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         private static void AppendTypes(List<Type> container, IList<Type> types)
29         {
30             for (var i = 0; i < types.Count; i++)
31             {
32                 var element = types[i];
33                 if (element != typeof(Types))
34                 {
35                     if (element.IsSubclassOf(typeof(Types)))
36                     {
37                         AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<
38                             ↪ <Type>>(nameof(Types<object>.Collection)));
39                     }
40                     else
41                     {
42                         container.Add(element);
43                     }
44                 }
45             }
46         }
47     }
48 }

```

```

43     }
44 }
45 }
46 }

```

#### 1.14 ./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 #pragma warning disable CA1819 // Properties should not return arrays
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
13             ↪ T4, T5, T6, T7>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

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

```

1 using System;
2 using System.Collections.ObjectModel;
3 using Platform.Collections.Lists;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6 #pragma warning disable CA1819 // Properties should not return arrays
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2, T3, T4, T5, T6> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
13             ↪ T4, T5, T6>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

#### 1.16 ./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 #pragma warning disable CA1819 // Properties should not return arrays
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2, T3, T4, T5> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
13             ↪ T4, T5>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

#### 1.17 ./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 #pragma warning disable CA1819 // Properties should not return arrays
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2, T3, T4> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
13             ↪ T4>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

### 1.18 ./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 #pragma warning disable CA1819 // Properties should not return arrays
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2, T3> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2,
13             ↪ T3>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }
```

### 1.19 ./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 #pragma warning disable CA1819 // Properties should not return arrays
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1,
13             ↪ T2>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }
```

### 1.20 ./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 #pragma warning disable CA1819 // Properties should not return arrays
7
8 namespace Platform.Reflection
9 {
10     public class Types<T> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new
13             ↪ Types<T>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }
```

### 1.21 ./Platform.Reflection.Tests/CodeGenerationTests.cs

```
1 using System;
2 using Xunit;
3
4 namespace Platform.Reflection.Tests
5 {
6     public class CodeGenerationTests
7     {
8         [Fact]
9         public void EmptyActionCompilationTest()
10         {
11             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
12             {
13                 generator.Return();
14             });
15             compiledAction();
16         }
17
18         [Fact]
19         public void FailedActionCompilationTest()
20         {
21             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
```

```

22     {
23         throw new NotImplementedException();
24     });
25     Assert.Throws<NotSupportedException>(compiledAction);
26 }
27
28 [Fact]
29 public void ConstantLoadingTest()
30 {
31     CheckConstantLoading<byte>(8);
32     CheckConstantLoading<uint>(8);
33     CheckConstantLoading<ushort>(8);
34     CheckConstantLoading<ulong>(8);
35 }
36
37 private void CheckConstantLoading<T>(T value)
38 {
39     var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
40     {
41         generator.LoadConstant(value);
42         generator.Return();
43     });
44     Assert.Equal(value, compiledFunction());
45 }
46 }
47 }

```

## 1.22 ./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 }

```

## 1.23 ./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, 22
- ./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, 16
- ./Platform.Reflection/NumericType.cs, 16
- ./Platform.Reflection/PropertyInfoExtensions.cs, 17
- ./Platform.Reflection/TypeBuilderExtensions.cs, 17
- ./Platform.Reflection/TypeExtensions.cs, 18
- ./Platform.Reflection/Types.cs, 20
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 21
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 21
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 21
- ./Platform.Reflection/Types[T1, T2, T3, T4].cs, 21
- ./Platform.Reflection/Types[T1, T2, T3].cs, 22
- ./Platform.Reflection/Types[T1, T2].cs, 22
- ./Platform.Reflection/Types[T].cs, 22