

# LinksPlatform's Platform.Reflection Class Library

## 1.1 ./csharp/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     /// <summary>
13     /// <para>
14     /// Represents the assembly extensions.
15     /// </para>
16     /// <para></para>
17     /// </summary>
18     public static class AssemblyExtensions
19     {
20         private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
21             ↳ ConcurrentDictionary<Assembly, Type[]>();
22
23         /// <remarks>
24         /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
25         /// </remarks>
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public static Type[] GetLoadableTypes(this Assembly assembly)
28         {
29             Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
30             try
31             {
32                 return assembly.GetTypes();
33             }
34             catch (ReflectionTypeLoadException e)
35             {
36                 return e.Types.ToArray(t => t != null);
37             }
38         }
39
40         /// <summary>
41         /// <para>
42         /// Gets the cached loadable types using the specified assembly.
43         /// </para>
44         /// <para></para>
45         /// </summary>
46         /// <param name="assembly">
47         /// <para>The assembly.</para>
48         /// </param>
49         /// <returns>
50         /// <para>The type array</para>
51         /// <para></para>
52         /// </returns>
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
55             ↳ _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
56     }
57 }
```

## 1.2 ./csharp/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     /// <summary>
13     /// <para>
14     /// Represents the delegate helpers.
15     /// </para>
16     /// <para></para>
17     /// </summary>
```

```

18 public static class DelegateHelpers
19 {
20     /// <summary>
21     /// <para>
22     /// Compiles the or default using the specified emit code.
23     /// </para>
24     /// <para></para>
25     /// </summary>
26     /// <typeparam name="TDelegate">
27     /// <para>The delegate.</para>
28     /// <para></para>
29     /// </typeparam>
30     /// <param name="emitCode">
31     /// <para>The emit code.</para>
32     /// <para></para>
33     /// </param>
34     /// <param name="typeMemberMethod">
35     /// <para>The type member method.</para>
36     /// <para></para>
37     /// </param>
38     /// <returns>
39     /// <para>The delegate.</para>
40     /// <para></para>
41     /// </returns>
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode, bool
44     → typeMemberMethod)
45     where TDelegate : Delegate
46     {
47         var @delegate = default(TDelegate);
48         try
49         {
50             @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
51             → CompileDynamicMethod<TDelegate>(emitCode);
52         }
53         catch (Exception exception)
54         {
55             exception.Ignore();
56         }
57         return @delegate;
58     }
59     /// <summary>
60     /// <para>
61     /// Compiles the or default using the specified emit code.
62     /// </para>
63     /// <para></para>
64     /// </summary>
65     /// <typeparam name="TDelegate">
66     /// <para>The delegate.</para>
67     /// <para></para>
68     /// </typeparam>
69     /// <param name="emitCode">
70     /// <para>The emit code.</para>
71     /// <para></para>
72     /// </param>
73     /// <returns>
74     /// <para>The delegate.</para>
75     /// <para></para>
76     /// </returns>
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
79     → TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);
80     /// <summary>
81     /// <para>
82     /// Compiles the emit code.
83     /// </para>
84     /// <para></para>
85     /// </summary>
86     /// <typeparam name="TDelegate">
87     /// <para>The delegate.</para>
88     /// <para></para>
89     /// </typeparam>
90     /// <param name="emitCode">
91     /// <para>The emit code.</para>
92     /// <para></para>
93     /// </param>

```

```

93     /// <param name="typeMemberMethod">
94     /// <para>The type member method.</para>
95     /// </para>
96     /// </param>
97     /// <returns>
98     /// <para>The delegate.</para>
99     /// </para>
100    /// </returns>
101    [MethodImpl(MethodImplOptions.AggressiveInlining)]
102    public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
    ↪ typeMemberMethod)
103    where TDelegate : Delegate
104    {
105        var @delegate = CompileOrDefault<TDelegate>(emitCode, typeMemberMethod);
106        if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
107        {
108            @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
109        }
110        return @delegate;
111    }
112
113    /// <summary>
114    /// <para>
115    /// Compiles the emit code.
116    /// </para>
117    /// </para>
118    /// </summary>
119    /// <typeparam name="TDelegate">
120    /// <para>The delegate.</para>
121    /// </para>
122    /// </typeparam>
123    /// <param name="emitCode">
124    /// <para>The emit code.</para>
125    /// </para>
126    /// </param>
127    /// <returns>
128    /// <para>The delegate</para>
129    /// </para>
130    /// </returns>
131    [MethodImpl(MethodImplOptions.AggressiveInlining)]
132    public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
    ↪ : Delegate => Compile<TDelegate>(emitCode, false);
133
134    /// <summary>
135    /// <para>
136    /// Compiles the dynamic method using the specified emit code.
137    /// </para>
138    /// </para>
139    /// </summary>
140    /// <typeparam name="TDelegate">
141    /// <para>The delegate.</para>
142    /// </para>
143    /// </typeparam>
144    /// <param name="emitCode">
145    /// <para>The emit code.</para>
146    /// </para>
147    /// </param>
148    /// <returns>
149    /// <para>The delegate</para>
150    /// </para>
151    /// </returns>
152    [MethodImpl(MethodImplOptions.AggressiveInlining)]
153    public static TDelegate CompileDynamicMethod<TDelegate>(Action<ILGenerator> emitCode)
154    {
155        var delegateType = typeof(TDelegate);
156        delegateType.GetDelegateCharacteristics(out Type returnType, out Type[]
    ↪ parameterTypes);
157        var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
158        emitCode(dynamicMethod.GetILGenerator());
159        return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
160    }
161
162    /// <summary>
163    /// <para>
164    /// Compiles the type member method using the specified emit code.
165    /// </para>
166    /// </para>
167    /// </summary>

```

```

168     /// <typeparam name="TDelegate">
169     /// <para>The delegate.</para>
170     /// </typeparam>
171     /// <param name="emitCode">
172     /// <para>The emit code.</para>
173     /// </param>
174     /// <returns>
175     /// <para>The delegate</para>
176     /// </returns>
177     [MethodImpl(MethodImplOptions.AggressiveInlining)]
178     public static TDelegate CompileTypeMemberMethod<TDelegate>(Action<ILGenerator> emitCode)
179     {
180         AssemblyName assemblyName = new AssemblyName(GetNewName());
181         var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
182             ↪ AssemblyBuilderAccess.Run);
183         var module = assembly.DefineDynamicModule(GetNewName());
184         var type = module.DefineType(GetNewName());
185         var methodName = GetNewName();
186         type.EmitStaticMethod<TDelegate>(methodName, emitCode);
187         var typeInfo = type.CreateTypeInfo();
188         return (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(typeof(TDele
189             ↪ gate));
190     }
191     [MethodImpl(MethodImplOptions.AggressiveInlining)]
192     private static string GetNewName() => Guid.NewGuid().ToString("N");
193 }
194 }
195 }

```

### 1.3 ./csharp/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     /// <summary>
9     /// <para>
10     /// Represents the dynamic extensions.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public static class DynamicExtensions
15     {
16         /// <summary>
17         /// <para>
18         /// Determines whether has property.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         /// <param name="@object">
23         /// <para>The object.</para>
24         /// <para></para>
25         /// </param>
26         /// <param name="propertyName">
27         /// <para>The property name.</para>
28         /// <para></para>
29         /// </param>
30         /// <returns>
31         /// <para>The bool</para>
32         /// <para></para>
33         /// </returns>
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public static bool HasProperty(this object @object, string propertyName)
36         {
37             var type = @object.GetType();
38             if (type is IDictionary<string, object> dictionary)
39             {
40                 return dictionary.ContainsKey(propertyName);
41             }
42             return type.GetProperty(propertyName) != null;
43         }
44     }
45 }

```

#### 1.4 ./csharp/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     /// <summary>
13     /// <para>
14     /// Represents the ensure extensions.
15     /// </para>
16     /// <para></para>
17     /// </summary>
18     public static class EnsureExtensions
19     {
20         #region Always
21
22         /// <summary>
23         /// <para>
24         /// Ises the unsigned integer using the specified root.
25         /// </para>
26         /// <para></para>
27         /// </summary>
28         /// <typeparam name="T">
29         /// <para>The .</para>
30         /// <para></para>
31         /// </typeparam>
32         /// <param name="root">
33         /// <para>The root.</para>
34         /// <para></para>
35         /// </param>
36         /// <param name="messageBuilder">
37         /// <para>The message builder.</para>
38         /// <para></para>
39         /// </param>
40         /// <exception cref="NotSupportedException">
41         /// <para></para>
42         /// <para></para>
43         /// </exception>
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root,
46         ↪ Func<string> messageBuilder)
47         {
48             if (!NumericType<T>.IsNumeric || NumericType<T>.IsSigned ||
49             ↪ NumericType<T>.IsFloatPoint)
50             {
51                 throw new NotSupportedException(messageBuilder());
52             }
53         }
54
55         /// <summary>
56         /// <para>
57         /// Ises the unsigned integer using the specified root.
58         /// </para>
59         /// <para></para>
60         /// </summary>
61         /// <typeparam name="T">
62         /// <para>The .</para>
63         /// <para></para>
64         /// </typeparam>
65         /// <param name="root">
66         /// <para>The root.</para>
67         /// <para></para>
68         /// </param>
69         /// <param name="message">
70         /// <para>The message.</para>
71         /// <para></para>
72         /// </param>
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
75         ↪ message)
76         {
77             string messageBuilder() => message;
```

```

75         IsUnsignedInteger<T>(root, messageBuilder);
76     }
77
78     /// <summary>
79     /// <para>
80     /// Uses the unsigned integer using the specified root.
81     /// </para>
82     /// <para></para>
83     /// </summary>
84     /// <typeparam name="T">
85     /// <para>The .</para>
86     /// <para></para>
87     /// </typeparam>
88     /// <param name="root">
89     /// <para>The root.</para>
90     /// <para></para>
91     /// </param>
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
94         ↪ IsUnsignedInteger<T>(root, (string)null);
95
96     /// <summary>
97     /// <para>
98     /// Uses the signed integer using the specified root.
99     /// </para>
100    /// <para></para>
101    /// </summary>
102    /// <typeparam name="T">
103    /// <para>The .</para>
104    /// <para></para>
105    /// </typeparam>
106    /// <param name="root">
107    /// <para>The root.</para>
108    /// <para></para>
109    /// </param>
110    /// <param name="messageBuilder">
111    /// <para>The message builder.</para>
112    /// <para></para>
113    /// </param>
114    /// <exception cref="NotSupportedException">
115    /// <para></para>
116    /// <para></para>
117    /// </exception>
118    [MethodImpl(MethodImplOptions.AggressiveInlining)]
119    public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
120        ↪ messageBuilder)
121    {
122        if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
123            ↪ NumericType<T>.IsFloatPoint)
124        {
125            throw new NotSupportedException(messageBuilder());
126        }
127    }
128
129    /// <summary>
130    /// <para>
131    /// Uses the signed integer using the specified root.
132    /// </para>
133    /// <para></para>
134    /// </summary>
135    /// <typeparam name="T">
136    /// <para>The .</para>
137    /// <para></para>
138    /// </typeparam>
139    /// <param name="root">
140    /// <para>The root.</para>
141    /// <para></para>
142    /// </param>
143    /// <param name="message">
144    /// <para>The message.</para>
145    /// <para></para>
146    /// </param>
147    [MethodImpl(MethodImplOptions.AggressiveInlining)]
148    public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
149        ↪ message)
150    {
151        string messageBuilder() => message;

```

```

148         IsSignedInteger<T>(root, messageBuilder);
149     }
150
151     /// <summary>
152     /// <para>
153     /// Uses the signed integer using the specified root.
154     /// </para>
155     /// <para></para>
156     /// </summary>
157     /// <typeparam name="T">
158     /// <para>The .</para>
159     /// <para></para>
160     /// </typeparam>
161     /// <param name="root">
162     /// <para>The root.</para>
163     /// <para></para>
164     /// </param>
165     [MethodImpl(MethodImplOptions.AggressiveInlining)]
166     public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
167         ↪ IsSignedInteger<T>(root, (string)null);
168
169     /// <summary>
170     /// <para>
171     /// Uses the signed using the specified root.
172     /// </para>
173     /// <para></para>
174     /// </summary>
175     /// <typeparam name="T">
176     /// <para>The .</para>
177     /// <para></para>
178     /// </typeparam>
179     /// <param name="root">
180     /// <para>The root.</para>
181     /// <para></para>
182     /// </param>
183     /// <param name="messageBuilder">
184     /// <para>The message builder.</para>
185     /// <para></para>
186     /// </param>
187     /// <exception cref="NotSupportedException">
188     /// <para></para>
189     /// <para></para>
190     /// </exception>
191     [MethodImpl(MethodImplOptions.AggressiveInlining)]
192     public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
193         ↪ messageBuilder)
194     {
195         if (!NumericType<T>.IsSigned)
196         {
197             throw new NotSupportedException(messageBuilder());
198         }
199     }
200
201     /// <summary>
202     /// <para>
203     /// Uses the signed using the specified root.
204     /// </para>
205     /// <para></para>
206     /// </summary>
207     /// <typeparam name="T">
208     /// <para>The .</para>
209     /// <para></para>
210     /// </typeparam>
211     /// <param name="root">
212     /// <para>The root.</para>
213     /// <para></para>
214     /// </param>
215     /// <param name="message">
216     /// <para>The message.</para>
217     /// <para></para>
218     /// </param>
219     [MethodImpl(MethodImplOptions.AggressiveInlining)]
220     public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
221     {
222         string messageBuilder() => message;
223         IsSigned<T>(root, messageBuilder);
224     }

```

```

224    /// <summary>
225    /// <para>
226    /// Ises the signed using the specified root.
227    /// </para>
228    /// <para></para>
229    /// </summary>
230    /// <typeparam name="T">
231    /// <para>The .</para>
232    /// <para></para>
233    /// </typeparam>
234    /// <param name="root">
235    /// <para>The root.</para>
236    /// <para></para>
237    /// </param>
238    [MethodImpl(MethodImplOptions.AggressiveInlining)]
239    public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
    ↪ (string)null);

240
241    /// <summary>
242    /// <para>
243    /// Ises the numeric using the specified root.
244    /// </para>
245    /// <para></para>
246    /// </summary>
247    /// <typeparam name="T">
248    /// <para>The .</para>
249    /// <para></para>
250    /// </typeparam>
251    /// <param name="root">
252    /// <para>The root.</para>
253    /// <para></para>
254    /// </param>
255    /// <param name="messageBuilder">
256    /// <para>The message builder.</para>
257    /// <para></para>
258    /// </param>
259    /// <exception cref="NotSupportedException">
260    /// <para></para>
261    /// <para></para>
262    /// </exception>
263    [MethodImpl(MethodImplOptions.AggressiveInlining)]
264    public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
    ↪ messageBuilder)
265    {
266        if (!NumericType<T>.IsNumeric)
267        {
268            throw new NotSupportedException(messageBuilder());
269        }
270    }

271
272    /// <summary>
273    /// <para>
274    /// Ises the numeric using the specified root.
275    /// </para>
276    /// <para></para>
277    /// </summary>
278    /// <typeparam name="T">
279    /// <para>The .</para>
280    /// <para></para>
281    /// </typeparam>
282    /// <param name="root">
283    /// <para>The root.</para>
284    /// <para></para>
285    /// </param>
286    /// <param name="message">
287    /// <para>The message.</para>
288    /// <para></para>
289    /// </param>
290    [MethodImpl(MethodImplOptions.AggressiveInlining)]
291    public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
292    {
293        string messageBuilder() => message;
294        IsNumeric<T>(root, messageBuilder);
295    }

296
297    /// <summary>
298    /// <para>
299    /// Ises the numeric using the specified root.

```



```

300    /// </para>
301    /// <para></para>
302    /// </summary>
303    /// <typeparam name="T">
304    /// <para>The .</para>
305    /// <para></para>
306    /// </typeparam>
307    /// <param name="root">
308    /// <para>The root.</para>
309    /// <para></para>
310    /// </param>
311    [MethodImpl(MethodImplOptions.AggressiveInlining)]
312    public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
313        ↪ IsNumeric<T>(root, (string)null);
314
315    /// <summary>
316    /// <para>
317    /// Cans the be numeric using the specified root.
318    /// </para>
319    /// <para></para>
320    /// </summary>
321    /// <typeparam name="T">
322    /// <para>The .</para>
323    /// <para></para>
324    /// </typeparam>
325    /// <param name="root">
326    /// <para>The root.</para>
327    /// <para></para>
328    /// </param>
329    /// <param name="messageBuilder">
330    /// <para>The message builder.</para>
331    /// <para></para>
332    /// </param>
333    /// <exception cref="NotSupportedException">
334    /// <para></para>
335    /// </exception>
336    [MethodImpl(MethodImplOptions.AggressiveInlining)]
337    public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
338        ↪ messageBuilder)
339    {
340        if (!NumericType<T>.CanBeNumeric)
341        {
342            throw new NotSupportedException(messageBuilder());
343        }
344    }
345
346    /// <summary>
347    /// <para>
348    /// Cans the be numeric using the specified root.
349    /// </para>
350    /// <para></para>
351    /// </summary>
352    /// <typeparam name="T">
353    /// <para>The .</para>
354    /// <para></para>
355    /// </typeparam>
356    /// <param name="root">
357    /// <para>The root.</para>
358    /// <para></para>
359    /// </param>
360    /// <param name="message">
361    /// <para>The message.</para>
362    /// <para></para>
363    /// </param>
364    [MethodImpl(MethodImplOptions.AggressiveInlining)]
365    public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
366    {
367        string messageBuilder() => message;
368        CanBeNumeric<T>(root, messageBuilder);
369    }
370
371    /// <summary>
372    /// <para>
373    /// Cans the be numeric using the specified root.
374    /// </para>
375    /// <para></para>
376    /// </summary>

```

```

376     /// <typeparam name="T">
377     /// <para>The .</para>
378     /// <para></para>
379     /// </typeparam>
380     /// <param name="root">
381     /// <para>The root.</para>
382     /// <para></para>
383     /// </param>
384     [MethodImpl(MethodImplOptions.AggressiveInlining)]
385     public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
386         ↪ CanBeNumeric<T>(root, (string)null);
387
388 #endregion
389
390 #region OnDebug
391
392     /// <summary>
393     /// <para>
394     /// Ises the unsigned integer using the specified root.
395     /// </para>
396     /// <para></para>
397     /// </summary>
398     /// <typeparam name="T">
399     /// <para>The .</para>
400     /// <para></para>
401     /// </typeparam>
402     /// <param name="root">
403     /// <para>The root.</para>
404     /// <para></para>
405     /// </param>
406     /// <param name="messageBuilder">
407     /// <para>The message builder.</para>
408     /// <para></para>
409     /// </param>
410     [Conditional("DEBUG")]
411     public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
412         ↪ Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
413
414     /// <summary>
415     /// <para>
416     /// Ises the unsigned integer using the specified root.
417     /// </para>
418     /// <para></para>
419     /// </summary>
420     /// <typeparam name="T">
421     /// <para>The .</para>
422     /// <para></para>
423     /// </typeparam>
424     /// <param name="root">
425     /// <para>The root.</para>
426     /// <para></para>
427     /// </param>
428     /// <param name="message">
429     /// <para>The message.</para>
430     /// <para></para>
431     /// </param>
432     [Conditional("DEBUG")]
433     public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
434         ↪ message) => Ensure.Always.IsUnsignedInteger<T>(message);
435
436     /// <summary>
437     /// <para>
438     /// Ises the unsigned integer using the specified root.
439     /// </para>
440     /// <para></para>
441     /// </summary>
442     /// <typeparam name="T">
443     /// <para>The .</para>
444     /// <para></para>
445     /// </typeparam>
446     /// <param name="root">
447     /// <para>The root.</para>
448     /// <para></para>
449     /// </param>
450     [Conditional("DEBUG")]
451     public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
452         ↪ Ensure.Always.IsUnsignedInteger<T>();

```

```

450    /// <summary>
451    /// <para>
452    /// Ises the signed integer using the specified root.
453    /// </para>
454    /// <para></para>
455    /// </summary>
456    /// <typeparam name="T">
457    /// <para>The .</para>
458    /// <para></para>
459    /// </typeparam>
460    /// <param name="root">
461    /// <para>The root.</para>
462    /// <para></para>
463    /// </param>
464    /// <param name="messageBuilder">
465    /// <para>The message builder.</para>
466    /// <para></para>
467    /// </param>
468    [Conditional("DEBUG")]
469    public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
    ↳ messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);

470
471    /// <summary>
472    /// <para>
473    /// Ises the signed integer using the specified root.
474    /// </para>
475    /// <para></para>
476    /// </summary>
477    /// <typeparam name="T">
478    /// <para>The .</para>
479    /// <para></para>
480    /// </typeparam>
481    /// <param name="root">
482    /// <para>The root.</para>
483    /// <para></para>
484    /// </param>
485    /// <param name="message">
486    /// <para>The message.</para>
487    /// <para></para>
488    /// </param>
489    [Conditional("DEBUG")]
490    public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
    ↳ message) => Ensure.Always.IsSignedInteger<T>(message);

491
492    /// <summary>
493    /// <para>
494    /// Ises the signed integer using the specified root.
495    /// </para>
496    /// <para></para>
497    /// </summary>
498    /// <typeparam name="T">
499    /// <para>The .</para>
500    /// <para></para>
501    /// </typeparam>
502    /// <param name="root">
503    /// <para>The root.</para>
504    /// <para></para>
505    /// </param>
506    [Conditional("DEBUG")]
507    public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
    ↳ Ensure.Always.IsSignedInteger<T>();

508
509    /// <summary>
510    /// <para>
511    /// Ises the signed using the specified root.
512    /// </para>
513    /// <para></para>
514    /// </summary>
515    /// <typeparam name="T">
516    /// <para>The .</para>
517    /// <para></para>
518    /// </typeparam>
519    /// <param name="root">
520    /// <para>The root.</para>
521    /// <para></para>
522    /// </param>
523    /// <param name="messageBuilder">
524    /// <para>The message builder.</para>

```

```

525 /// <para></para>
526 /// </param>
527 [Conditional("DEBUG")]
528 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
    ↳ messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
529
530 /// <summary>
531 /// <para>
532 /// Ises the signed using the specified root.
533 /// </para>
534 /// <para></para>
535 /// </summary>
536 /// <typeparam name="T">
537 /// <para>The .</para>
538 /// <para></para>
539 /// </typeparam>
540 /// <param name="root">
541 /// <para>The root.</para>
542 /// <para></para>
543 /// </param>
544 /// <param name="message">
545 /// <para>The message.</para>
546 /// <para></para>
547 /// </param>
548 [Conditional("DEBUG")]
549 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
    ↳ Ensure.Always.IsSigned<T>(message);
550
551 /// <summary>
552 /// <para>
553 /// Ises the signed using the specified root.
554 /// </para>
555 /// <para></para>
556 /// </summary>
557 /// <typeparam name="T">
558 /// <para>The .</para>
559 /// <para></para>
560 /// </typeparam>
561 /// <param name="root">
562 /// <para>The root.</para>
563 /// <para></para>
564 /// </param>
565 [Conditional("DEBUG")]
566 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
    ↳ Ensure.Always.IsSigned<T>();
567
568 /// <summary>
569 /// <para>
570 /// Ises the numeric using the specified root.
571 /// </para>
572 /// <para></para>
573 /// </summary>
574 /// <typeparam name="T">
575 /// <para>The .</para>
576 /// <para></para>
577 /// </typeparam>
578 /// <param name="root">
579 /// <para>The root.</para>
580 /// <para></para>
581 /// </param>
582 /// <param name="messageBuilder">
583 /// <para>The message builder.</para>
584 /// <para></para>
585 /// </param>
586 [Conditional("DEBUG")]
587 public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
    ↳ messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
588
589 /// <summary>
590 /// <para>
591 /// Ises the numeric using the specified root.
592 /// </para>
593 /// <para></para>
594 /// </summary>
595 /// <typeparam name="T">
596 /// <para>The .</para>
597 /// <para></para>
598 /// </typeparam>

```

```

599     /// <param name="root">
600     /// <para>The root.</para>
601     /// <para></para>
602     /// </param>
603     /// <param name="message">
604     /// <para>The message.</para>
605     /// <para></para>
606     /// </param>
607     [Conditional("DEBUG")]
608     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
        ↳ Ensure.Always.IsNumeric<T>(message);
609
610     /// <summary>
611     /// <para>
612     /// Uses the numeric using the specified root.
613     /// </para>
614     /// <para></para>
615     /// </summary>
616     /// <typeparam name="T">
617     /// <para>The .</para>
618     /// <para></para>
619     /// </typeparam>
620     /// <param name="root">
621     /// <para>The root.</para>
622     /// <para></para>
623     /// </param>
624     [Conditional("DEBUG")]
625     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
        ↳ Ensure.Always.IsNumeric<T>();
626
627     /// <summary>
628     /// <para>
629     /// Cans the be numeric using the specified root.
630     /// </para>
631     /// <para></para>
632     /// </summary>
633     /// <typeparam name="T">
634     /// <para>The .</para>
635     /// <para></para>
636     /// </typeparam>
637     /// <param name="root">
638     /// <para>The root.</para>
639     /// <para></para>
640     /// </param>
641     /// <param name="messageBuilder">
642     /// <para>The message builder.</para>
643     /// <para></para>
644     /// </param>
645     [Conditional("DEBUG")]
646     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
        ↳ messageBuilder) => Ensure.Always.CanBeNumeric<T>(messageBuilder);
647
648     /// <summary>
649     /// <para>
650     /// Cans the be numeric using the specified root.
651     /// </para>
652     /// <para></para>
653     /// </summary>
654     /// <typeparam name="T">
655     /// <para>The .</para>
656     /// <para></para>
657     /// </typeparam>
658     /// <param name="root">
659     /// <para>The root.</para>
660     /// <para></para>
661     /// </param>
662     /// <param name="message">
663     /// <para>The message.</para>
664     /// <para></para>
665     /// </param>
666     [Conditional("DEBUG")]
667     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
        ↳ => Ensure.Always.CanBeNumeric<T>(message);
668
669     /// <summary>
670     /// <para>
671     /// Cans the be numeric using the specified root.
672     /// </para>

```

```

673     /// <para></para>
674     /// </summary>
675     /// <typeparam name="T">
676     /// <para>The .</para>
677     /// <para></para>
678     /// </typeparam>
679     /// <param name="root">
680     /// <para>The root.</para>
681     /// <para></para>
682     /// </param>
683     [Conditional("DEBUG")]
684     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
        ↪ Ensure.Always.CanBeNumeric<T>();
685
686     #endregion
687 }
688 }

```

## 1.5 ./csharp/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      /// <summary>
9      /// <para>
10     /// Represents the field info extensions.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public static class FieldInfoExtensions
15     {
16         /// <summary>
17         /// <para>
18         /// Gets the static value using the specified field info.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         /// <typeparam name="T">
23         /// <para>The .</para>
24         /// <para></para>
25         /// </typeparam>
26         /// <param name="fieldInfo">
27         /// <para>The field info.</para>
28         /// <para></para>
29         /// </param>
30         /// <returns>
31         /// <para>The</para>
32         /// <para></para>
33         /// </returns>
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public static T GetStaticValue<T>(this FieldInfo fieldInfo) =>
            ↪ (T)fieldInfo.GetValue(null);
36     }
37 }

```

## 1.6 ./csharp/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     /// <summary>
12     /// <para>
13     /// Represents the il generator extensions.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     public static class ILGeneratorExtensions
18     {
19         /// <summary>
20         /// <para>

```

```

21     /// Throws the generator.
22     /// </para>
23     /// <para></para>
24     /// </summary>
25     /// <typeparam name="T">
26     /// <para>The .</para>
27     /// <para></para>
28     /// </typeparam>
29     /// <param name="generator">
30     /// <para>The generator.</para>
31     /// <para></para>
32     /// </param>
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     public static void Throw<T>(this ILGenerator generator) =>
35         ↪ generator.ThrowException(typeof(T));
36
37     /// <summary>
38     /// <para>
39     /// Unchecked the convert using the specified generator.
40     /// </para>
41     /// <para></para>
42     /// </summary>
43     /// <typeparam name="TSource">
44     /// <para>The source.</para>
45     /// <para></para>
46     /// </typeparam>
47     /// <typeparam name="TTarget">
48     /// <para>The target.</para>
49     /// <para></para>
50     /// </typeparam>
51     /// <param name="generator">
52     /// <para>The generator.</para>
53     /// <para></para>
54     /// </param>
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator) =>
57         ↪ UncheckedConvert<TSource, TTarget>(generator, extendSign: false);
58
59     /// <summary>
60     /// <para>
61     /// Unchecked the convert using the specified generator.
62     /// </para>
63     /// <para></para>
64     /// </summary>
65     /// <typeparam name="TSource">
66     /// <para>The source.</para>
67     /// <para></para>
68     /// </typeparam>
69     /// <typeparam name="TTarget">
70     /// <para>The target.</para>
71     /// <para></para>
72     /// </typeparam>
73     /// <param name="generator">
74     /// <para>The generator.</para>
75     /// <para></para>
76     /// </param>
77     /// <param name="extendSign">
78     /// <para>The extend sign.</para>
79     /// <para></para>
80     /// </param>
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator, bool
83         ↪ extendSign)
84     {
85         var sourceType = typeof(TSource);
86         var targetType = typeof(TTarget);
87         if (sourceType == targetType)
88         {
89             return;
90         }
91         if (extendSign)
92         {
93             if (sourceType == typeof(byte))
94             {
95                 generator.Emit(OpCodes.Conv_I1);
96             }
97             if (sourceType == typeof(ushort) || sourceType == typeof(char))
98             {
99

```

```

96         generator.Emit(OpCodes.Conv_I2);
97     }
98 }
99 if (NumericType<TSource>.BitsSize > NumericType<TTarget>.BitsSize)
100 {
101     generator.ConvertToInteger<TSource>(targetType, extendSign: false);
102 }
103 else
104 {
105     generator.ConvertToInteger<TSource>(targetType, extendSign);
106 }
107 if (targetType == typeof(float))
108 {
109     if (NumericType<TSource>.IsSigned)
110     {
111         generator.Emit(OpCodes.Conv_R4);
112     }
113     else
114     {
115         generator.Emit(OpCodes.Conv_R_Un);
116     }
117 }
118 else if (targetType == typeof(double))
119 {
120     generator.Emit(OpCodes.Conv_R8);
121 }
122 else if (targetType == typeof(bool))
123 {
124     generator.ConvertToBoolean<TSource>();
125 }
126 }
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 private static void ConvertToBoolean<TSource>(this ILGenerator generator)
129 {
130     generator.LoadConstant<TSource>(default);
131     var sourceType = typeof(TSource);
132     if (sourceType == typeof(float) || sourceType == typeof(double))
133     {
134         generator.Emit(OpCodes.Ceq);
135         // Inversion of the first Ceq instruction
136         generator.LoadConstant<int>(0);
137         generator.Emit(OpCodes.Ceq);
138     }
139     else
140     {
141         generator.Emit(OpCodes.Cgt_Un);
142     }
143 }
144 [MethodImpl(MethodImplOptions.AggressiveInlining)]
145 private static void ConvertToInteger<TSource>(this ILGenerator generator, Type
→ targetType, bool extendSign)
146 {
147     if (targetType == typeof(sbyte))
148     {
149         generator.Emit(OpCodes.Conv_I1);
150     }
151     else if (targetType == typeof(byte))
152     {
153         generator.Emit(OpCodes.Conv_U1);
154     }
155     else if (targetType == typeof(short))
156     {
157         generator.Emit(OpCodes.Conv_I2);
158     }
159     else if (targetType == typeof(ushort) || targetType == typeof(char))
160     {
161         var sourceType = typeof(TSource);
162         if (sourceType != typeof(ushort) && sourceType != typeof(char))
163         {
164             generator.Emit(OpCodes.Conv_U2);
165         }
166     }
167     else if (targetType == typeof(int))
168     {
169         generator.Emit(OpCodes.Conv_I4);
170     }
171     else if (targetType == typeof(uint))
172     {

```



```

173         generator.Emit(OpCodes.Conv_U4);
174     }
175     else if (targetType == typeof(long) || targetType == typeof(ulong))
176     {
177         if (NumericType<TSource>.IsSigned || extendSign)
178         {
179             generator.Emit(OpCodes.Conv_I8);
180         }
181         else
182         {
183             generator.Emit(OpCodes.Conv_U8);
184         }
185     }
186 }
187
188 /// <summary>
189 /// <para>
190 /// Checkeds the convert using the specified generator.
191 /// </para>
192 /// </summary>
193 /// <typeparam name="TSource">
194 /// <para>The source.</para>
195 /// </typeparam>
196 /// <typeparam name="TTarget">
197 /// <para>The target.</para>
198 /// </typeparam>
199 /// <param name="generator">
200 /// <para>The generator.</para>
201 /// </param>
202 /// <exception cref="NotSupportedException">
203 /// <para></para>
204 /// </exception>
205 [MethodImpl(MethodImplOptions.AggressiveInlining)]
206 public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
207 {
208     var sourceType = typeof(TSource);
209     var targetType = typeof(TTarget);
210     if (sourceType == targetType)
211     {
212         return;
213     }
214     if (targetType == typeof(short))
215     {
216         if (NumericType<TSource>.IsSigned)
217         {
218             generator.Emit(OpCodes.Conv_Ovf_I2);
219         }
220         else
221         {
222             generator.Emit(OpCodes.Conv_Ovf_I2_Un);
223         }
224     }
225     else if (targetType == typeof(ushort) || targetType == typeof(char))
226     {
227         if (sourceType != typeof(ushort) && sourceType != typeof(char))
228         {
229             if (NumericType<TSource>.IsSigned)
230             {
231                 generator.Emit(OpCodes.Conv_Ovf_U2);
232             }
233             else
234             {
235                 generator.Emit(OpCodes.Conv_Ovf_U2_Un);
236             }
237         }
238     }
239     else if (targetType == typeof(sbyte))
240     {
241         if (NumericType<TSource>.IsSigned)
242         {
243             generator.Emit(OpCodes.Conv_Ovf_I1);
244         }
245         else
246     }

```

```

251     {
252         generator.Emit(OpCodes.Conv_Ovf_I1_Un);
253     }
254 }
255 else if (targetType == typeof(byte))
256 {
257     if (NumericType<TSource>.IsSigned)
258     {
259         generator.Emit(OpCodes.Conv_Ovf_U1);
260     }
261     else
262     {
263         generator.Emit(OpCodes.Conv_Ovf_U1_Un);
264     }
265 }
266 else if (targetType == typeof(int))
267 {
268     if (NumericType<TSource>.IsSigned)
269     {
270         generator.Emit(OpCodes.Conv_Ovf_I4);
271     }
272     else
273     {
274         generator.Emit(OpCodes.Conv_Ovf_I4_Un);
275     }
276 }
277 else if (targetType == typeof(uint))
278 {
279     if (NumericType<TSource>.IsSigned)
280     {
281         generator.Emit(OpCodes.Conv_Ovf_U4);
282     }
283     else
284     {
285         generator.Emit(OpCodes.Conv_Ovf_U4_Un);
286     }
287 }
288 else if (targetType == typeof(long))
289 {
290     if (NumericType<TSource>.IsSigned)
291     {
292         generator.Emit(OpCodes.Conv_Ovf_I8);
293     }
294     else
295     {
296         generator.Emit(OpCodes.Conv_Ovf_I8_Un);
297     }
298 }
299 else if (targetType == typeof(ulong))
300 {
301     if (NumericType<TSource>.IsSigned)
302     {
303         generator.Emit(OpCodes.Conv_Ovf_U8);
304     }
305     else
306     {
307         generator.Emit(OpCodes.Conv_Ovf_U8_Un);
308     }
309 }
310 else if (targetType == typeof(float))
311 {
312     if (NumericType<TSource>.IsSigned)
313     {
314         generator.Emit(OpCodes.Conv_R4);
315     }
316     else
317     {
318         generator.Emit(OpCodes.Conv_R_Un);
319     }
320 }
321 else if (targetType == typeof(double))
322 {
323     generator.Emit(OpCodes.Conv_R8);
324 }
325 else if (targetType == typeof(bool))
326 {
327     generator.ConvertToBoolean<TSource>();
328 }

```

```

329         else
330         {
331             throw new NotSupportedException();
332         }
333     }
334
335     /// <summary>
336     /// <para>
337     /// Loads the constant using the specified generator.
338     /// </para>
339     /// <para></para>
340     /// </summary>
341     /// <param name="generator">
342     /// <para>The generator.</para>
343     /// <para></para>
344     /// </param>
345     /// <param name="value">
346     /// <para>The value.</para>
347     /// <para></para>
348     /// </param>
349     [MethodImpl(MethodImplOptions.AggressiveInlining)]
350     public static void LoadConstant(this ILGenerator generator, bool value) =>
351         ↪ generator.LoadConstant(value ? 1 : 0);
352
353     /// <summary>
354     /// <para>
355     /// Loads the constant using the specified generator.
356     /// </para>
357     /// <para></para>
358     /// </summary>
359     /// <param name="generator">
360     /// <para>The generator.</para>
361     /// <para></para>
362     /// </param>
363     /// <param name="value">
364     /// <para>The value.</para>
365     /// <para></para>
366     /// </param>
367     [MethodImpl(MethodImplOptions.AggressiveInlining)]
368     public static void LoadConstant(this ILGenerator generator, float value) =>
369         ↪ generator.Emit(OpCodes.Ldc_R4, value);
370
371     /// <summary>
372     /// <para>
373     /// Loads the constant using the specified generator.
374     /// </para>
375     /// <para></para>
376     /// </summary>
377     /// <param name="generator">
378     /// <para>The generator.</para>
379     /// <para></para>
380     /// </param>
381     /// <param name="value">
382     /// <para>The value.</para>
383     /// <para></para>
384     /// </param>
385     [MethodImpl(MethodImplOptions.AggressiveInlining)]
386     public static void LoadConstant(this ILGenerator generator, double value) =>
387         ↪ generator.Emit(OpCodes.Ldc_R8, value);
388
389     /// <summary>
390     /// <para>
391     /// Loads the constant using the specified generator.
392     /// </para>
393     /// <para></para>
394     /// </summary>
395     /// <param name="generator">
396     /// <para>The generator.</para>
397     /// <para></para>
398     /// </param>
399     /// <param name="value">
400     /// <para>The value.</para>
401     /// <para></para>
402     /// </param>
403     [MethodImpl(MethodImplOptions.AggressiveInlining)]
404     public static void LoadConstant(this ILGenerator generator, ulong value) =>
405         ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));

```

```

403     /// <summary>
404     /// <para>
405     /// Loads the constant using the specified generator.
406     /// </para>
407     /// <para></para>
408     /// </summary>
409     /// <param name="generator">
410     /// <para>The generator.</para>
411     /// <para></para>
412     /// </param>
413     /// <param name="value">
414     /// <para>The value.</para>
415     /// <para></para>
416     /// </param>
417     [MethodImpl(MethodImplOptions.AggressiveInlining)]
418     public static void LoadConstant(this ILGenerator generator, long value) =>
419         ↪ generator.Emit(OpCodes.Ldc_I8, value);
420
421     /// <summary>
422     /// <para>
423     /// Loads the constant using the specified generator.
424     /// </para>
425     /// <para></para>
426     /// </summary>
427     /// <param name="generator">
428     /// <para>The generator.</para>
429     /// <para></para>
430     /// </param>
431     /// <param name="value">
432     /// <para>The value.</para>
433     /// <para></para>
434     /// </param>
435     [MethodImpl(MethodImplOptions.AggressiveInlining)]
436     public static void LoadConstant(this ILGenerator generator, uint value)
437     {
438         switch (value)
439         {
440             case uint.MaxValue:
441                 generator.Emit(OpCodes.Ldc_I4_M1);
442                 return;
443             case 0:
444                 generator.Emit(OpCodes.Ldc_I4_0);
445                 return;
446             case 1:
447                 generator.Emit(OpCodes.Ldc_I4_1);
448                 return;
449             case 2:
450                 generator.Emit(OpCodes.Ldc_I4_2);
451                 return;
452             case 3:
453                 generator.Emit(OpCodes.Ldc_I4_3);
454                 return;
455             case 4:
456                 generator.Emit(OpCodes.Ldc_I4_4);
457                 return;
458             case 5:
459                 generator.Emit(OpCodes.Ldc_I4_5);
460                 return;
461             case 6:
462                 generator.Emit(OpCodes.Ldc_I4_6);
463                 return;
464             case 7:
465                 generator.Emit(OpCodes.Ldc_I4_7);
466                 return;
467             case 8:
468                 generator.Emit(OpCodes.Ldc_I4_8);
469                 return;
470             default:
471                 if (value <= sbyte.MaxValue)
472                 {
473                     generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
474                 }
475                 else
476                 {
477                     generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
478                 }
479                 return;
480         }
481     }

```

```

482    /// <summary>
483    /// <para>
484    /// Loads the constant using the specified generator.
485    /// </para>
486    /// <para></para>
487    /// </summary>
488    /// <param name="generator">
489    /// <para>The generator.</para>
490    /// <para></para>
491    /// </param>
492    /// <param name="value">
493    /// <para>The value.</para>
494    /// <para></para>
495    /// </param>
496    [MethodImpl(MethodImplOptions.AggressiveInlining)]
497    public static void LoadConstant(this ILGenerator generator, int value)
498    {
499        switch (value)
500        {
501            case -1:
502                generator.Emit(OpCodes.Ldc_I4_M1);
503                return;
504            case 0:
505                generator.Emit(OpCodes.Ldc_I4_0);
506                return;
507            case 1:
508                generator.Emit(OpCodes.Ldc_I4_1);
509                return;
510            case 2:
511                generator.Emit(OpCodes.Ldc_I4_2);
512                return;
513            case 3:
514                generator.Emit(OpCodes.Ldc_I4_3);
515                return;
516            case 4:
517                generator.Emit(OpCodes.Ldc_I4_4);
518                return;
519            case 5:
520                generator.Emit(OpCodes.Ldc_I4_5);
521                return;
522            case 6:
523                generator.Emit(OpCodes.Ldc_I4_6);
524                return;
525            case 7:
526                generator.Emit(OpCodes.Ldc_I4_7);
527                return;
528            case 8:
529                generator.Emit(OpCodes.Ldc_I4_8);
530                return;
531            default:
532                if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
533                {
534                    generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
535                }
536                else
537                {
538                    generator.Emit(OpCodes.Ldc_I4, value);
539                }
540                return;
541        }
542    }
543
544    /// <summary>
545    /// <para>
546    /// Loads the constant using the specified generator.
547    /// </para>
548    /// <para></para>
549    /// </summary>
550    /// <param name="generator">
551    /// <para>The generator.</para>
552    /// <para></para>
553    /// </param>
554    /// <param name="value">
555    /// <para>The value.</para>
556    /// <para></para>
557    /// </param>
558    [MethodImpl(MethodImplOptions.AggressiveInlining)]
559    public static void LoadConstant(this ILGenerator generator, short value) =>
560        ↪ generator.LoadConstant((int)value);

```

```

561    /// <summary>
562    /// <para>
563    /// Loads the constant using the specified generator.
564    /// </para>
565    /// <para></para>
566    /// </summary>
567    /// <param name="generator">
568    /// <para>The generator.</para>
569    /// <para></para>
570    /// </param>
571    /// <param name="value">
572    /// <para>The value.</para>
573    /// <para></para>
574    /// </param>
575    [MethodImpl(MethodImplOptions.AggressiveInlining)]
576    public static void LoadConstant(this ILGenerator generator, ushort value) =>
577        ↪ generator.LoadConstant((int)value);
578
579    /// <summary>
580    /// <para>
581    /// Loads the constant using the specified generator.
582    /// </para>
583    /// <para></para>
584    /// </summary>
585    /// <param name="generator">
586    /// <para>The generator.</para>
587    /// <para></para>
588    /// </param>
589    /// <param name="value">
590    /// <para>The value.</para>
591    /// <para></para>
592    /// </param>
593    [MethodImpl(MethodImplOptions.AggressiveInlining)]
594    public static void LoadConstant(this ILGenerator generator, sbyte value) =>
595        ↪ generator.LoadConstant((int)value);
596
597    /// <summary>
598    /// <para>
599    /// Loads the constant using the specified generator.
600    /// </para>
601    /// <para></para>
602    /// </summary>
603    /// <param name="generator">
604    /// <para>The generator.</para>
605    /// <para></para>
606    /// </param>
607    /// <param name="value">
608    /// <para>The value.</para>
609    /// <para></para>
610    /// </param>
611    [MethodImpl(MethodImplOptions.AggressiveInlining)]
612    public static void LoadConstant(this ILGenerator generator, byte value) =>
613        ↪ generator.LoadConstant((int)value);
614
615    /// <summary>
616    /// <para>
617    /// Loads the constant one using the specified generator.
618    /// </para>
619    /// <para></para>
620    /// </summary>
621    /// <typeparam name="TConstant">
622    /// <para>The constant.</para>
623    /// <para></para>
624    /// </typeparam>
625    /// <param name="generator">
626    /// <para>The generator.</para>
627    /// <para></para>
628    /// </param>
629    [MethodImpl(MethodImplOptions.AggressiveInlining)]
630    public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
631        ↪ LoadConstantOne(generator, typeof(TConstant));
632
633    /// <summary>
634    /// <para>
635    /// Loads the constant one using the specified generator.
636    /// </para>
637    /// <para></para>
638    /// </summary>

```

```

635 /// <param name="generator">
636 /// <para>The generator.</para>
637 /// </para>
638 /// </param>
639 /// <param name="constantType">
640 /// <para>The constant type.</para>
641 /// </para>
642 /// </param>
643 /// <exception cref="NotSupportedException">
644 /// <para></para>
645 /// </exception>
646 [MethodImpl(MethodImplOptions.AggressiveInlining)]
647 public static void LoadConstantOne(this ILGenerator generator, Type constantType)
648 {
649     if (constantType == typeof(float))
650     {
651         generator.LoadConstant(1F);
652     }
653     else if (constantType == typeof(double))
654     {
655         generator.LoadConstant(1D);
656     }
657     else if (constantType == typeof(long))
658     {
659         generator.LoadConstant(1L);
660     }
661     else if (constantType == typeof(ulong))
662     {
663         generator.LoadConstant(1UL);
664     }
665     else if (constantType == typeof(int))
666     {
667         generator.LoadConstant(1);
668     }
669     else if (constantType == typeof(uint))
670     {
671         generator.LoadConstant(1U);
672     }
673     else if (constantType == typeof(short))
674     {
675         generator.LoadConstant((short)1);
676     }
677     else if (constantType == typeof(ushort))
678     {
679         generator.LoadConstant((ushort)1);
680     }
681     else if (constantType == typeof(sbyte))
682     {
683         generator.LoadConstant((sbyte)1);
684     }
685     else if (constantType == typeof(byte))
686     {
687         generator.LoadConstant((byte)1);
688     }
689     else
690     {
691         throw new NotSupportedException();
692     }
693 }
694
695
696 /// <summary>
697 /// <para>
698 /// Loads the constant using the specified generator.
699 /// </para>
700 /// </summary>
701 /// <typeparam name="TConstant">
702 /// <para>The constant.</para>
703 /// </typeparam>
704 /// <param name="generator">
705 /// <para>The generator.</para>
706 /// </param>
707 /// <param name="constantValue">
708 /// <para>The constant value.</para>
709 /// </param>
710
711
712

```

```

713 /// </param>
714 [MethodImpl(MethodImplOptions.AggressiveInlining)]
715 public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
    ↳ constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);

716
717 /// <summary>
718 /// <para>
719 /// Loads the constant using the specified generator.
720 /// </para>
721 /// <para></para>
722 /// </summary>
723 /// <param name="generator">
724 /// <para>The generator.</para>
725 /// <para></para>
726 /// </param>
727 /// <param name="constantType">
728 /// <para>The constant type.</para>
729 /// <para></para>
730 /// </param>
731 /// <param name="constantValue">
732 /// <para>The constant value.</para>
733 /// <para></para>
734 /// </param>
735 /// <exception cref="NotSupportedException">
736 /// <para></para>
737 /// <para></para>
738 /// </exception>
739 [MethodImpl(MethodImplOptions.AggressiveInlining)]
740 public static void LoadConstant(this ILGenerator generator, Type constantType, object
    ↳ constantValue)
741 {
742     constantValue = Convert.ChangeType(constantValue, constantType);
743     if (constantType == typeof(float))
744     {
745         generator.LoadConstant((float)constantValue);
746     }
747     else if (constantType == typeof(double))
748     {
749         generator.LoadConstant((double)constantValue);
750     }
751     else if (constantType == typeof(long))
752     {
753         generator.LoadConstant((long)constantValue);
754     }
755     else if (constantType == typeof(ulong))
756     {
757         generator.LoadConstant((ulong)constantValue);
758     }
759     else if (constantType == typeof(int))
760     {
761         generator.LoadConstant((int)constantValue);
762     }
763     else if (constantType == typeof(uint))
764     {
765         generator.LoadConstant((uint)constantValue);
766     }
767     else if (constantType == typeof(short))
768     {
769         generator.LoadConstant((short)constantValue);
770     }
771     else if (constantType == typeof(ushort))
772     {
773         generator.LoadConstant((ushort)constantValue);
774     }
775     else if (constantType == typeof(sbyte))
776     {
777         generator.LoadConstant((sbyte)constantValue);
778     }
779     else if (constantType == typeof(byte))
780     {
781         generator.LoadConstant((byte)constantValue);
782     }
783     else
784     {
785         throw new NotSupportedException();
786     }
787 }
788

```



```

789     /// <summary>
790     /// <para>
791     /// Increments the generator.
792     /// </para>
793     /// <para></para>
794     /// </summary>
795     /// <typeparam name="TValue">
796     /// <para>The value.</para>
797     /// <para></para>
798     /// </typeparam>
799     /// <param name="generator">
800     /// <para>The generator.</para>
801     /// <para></para>
802     /// </param>
803     [MethodImpl(MethodImplOptions.AggressiveInlining)]
804     public static void Increment<TValue>(this ILGenerator generator) =>
805         ↪ generator.Increment(typeof(TValue));
806
807     /// <summary>
808     /// <para>
809     /// Decrements the generator.
810     /// </para>
811     /// <para></para>
812     /// </summary>
813     /// <typeparam name="TValue">
814     /// <para>The value.</para>
815     /// <para></para>
816     /// </typeparam>
817     /// <param name="generator">
818     /// <para>The generator.</para>
819     /// <para></para>
820     /// </param>
821     [MethodImpl(MethodImplOptions.AggressiveInlining)]
822     public static void Decrement<TValue>(this ILGenerator generator) =>
823         ↪ generator.Decrement(typeof(TValue));
824
825     /// <summary>
826     /// <para>
827     /// Increments the generator.
828     /// </para>
829     /// <para></para>
830     /// </summary>
831     /// <param name="generator">
832     /// <para>The generator.</para>
833     /// <para></para>
834     /// </param>
835     /// <param name="valueType">
836     /// <para>The value type.</para>
837     /// <para></para>
838     /// </param>
839     [MethodImpl(MethodImplOptions.AggressiveInlining)]
840     public static void Increment(this ILGenerator generator, Type valueType)
841     {
842         generator.LoadConstantOne(valueType);
843         generator.Add();
844     }
845
846     /// <summary>
847     /// <para>
848     /// Adds the generator.
849     /// </para>
850     /// <para></para>
851     /// </summary>
852     /// <param name="generator">
853     /// <para>The generator.</para>
854     /// <para></para>
855     /// </param>
856     [MethodImpl(MethodImplOptions.AggressiveInlining)]
857     public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
858
859     /// <summary>
860     /// <para>
861     /// Decrements the generator.
862     /// </para>
863     /// <para></para>
864     /// </summary>
865     /// <param name="generator">
866     /// <para>The generator.</para>

```

```

865     /// <para></para>
866     /// </param>
867     /// <param name="valueType">
868     /// <para>The value type.</para>
869     /// <para></para>
870     /// </param>
871     [MethodImpl(MethodImplOptions.AggressiveInlining)]
872     public static void Decrement(this ILGenerator generator, Type valueType)
873     {
874         generator.LoadConstantOne(valueType);
875         generator.Subtract();
876     }
877
878     /// <summary>
879     /// <para>
880     /// Subtracts the generator.
881     /// </para>
882     /// <para></para>
883     /// </summary>
884     /// <param name="generator">
885     /// <para>The generator.</para>
886     /// <para></para>
887     /// </param>
888     [MethodImpl(MethodImplOptions.AggressiveInlining)]
889     public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
890
891     /// <summary>
892     /// <para>
893     /// Negates the generator.
894     /// </para>
895     /// <para></para>
896     /// </summary>
897     /// <param name="generator">
898     /// <para>The generator.</para>
899     /// <para></para>
900     /// </param>
901     [MethodImpl(MethodImplOptions.AggressiveInlining)]
902     public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
903
904     /// <summary>
905     /// <para>
906     /// Ands the generator.
907     /// </para>
908     /// <para></para>
909     /// </summary>
910     /// <param name="generator">
911     /// <para>The generator.</para>
912     /// <para></para>
913     /// </param>
914     [MethodImpl(MethodImplOptions.AggressiveInlining)]
915     public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
916
917     /// <summary>
918     /// <para>
919     /// Ors the generator.
920     /// </para>
921     /// <para></para>
922     /// </summary>
923     /// <param name="generator">
924     /// <para>The generator.</para>
925     /// <para></para>
926     /// </param>
927     [MethodImpl(MethodImplOptions.AggressiveInlining)]
928     public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
929
930     /// <summary>
931     /// <para>
932     /// Nots the generator.
933     /// </para>
934     /// <para></para>
935     /// </summary>
936     /// <param name="generator">
937     /// <para>The generator.</para>
938     /// <para></para>
939     /// </param>
940     [MethodImpl(MethodImplOptions.AggressiveInlining)]
941     public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
942

```

```

943     /// <summary>
944     /// <para>
945     /// Shifts the left using the specified generator.
946     /// </para>
947     /// <para></para>
948     /// </summary>
949     /// <param name="generator">
950     /// <para>The generator.</para>
951     /// <para></para>
952     /// </param>
953     [MethodImpl(MethodImplOptions.AggressiveInlining)]
954     public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
955
956     /// <summary>
957     /// <para>
958     /// Shifts the right using the specified generator.
959     /// </para>
960     /// <para></para>
961     /// </summary>
962     /// <typeparam name="T">
963     /// <para>The .</para>
964     /// <para></para>
965     /// </typeparam>
966     /// <param name="generator">
967     /// <para>The generator.</para>
968     /// <para></para>
969     /// </param>
970     [MethodImpl(MethodImplOptions.AggressiveInlining)]
971     public static void ShiftRight<T>(this ILGenerator generator)
972     {
973         generator.Emit(NumericType<T>.IsSigned ? OpCodes.Shr : OpCodes.Shr_Un);
974     }
975
976     /// <summary>
977     /// <para>
978     /// Loads the argument using the specified generator.
979     /// </para>
980     /// <para></para>
981     /// </summary>
982     /// <param name="generator">
983     /// <para>The generator.</para>
984     /// <para></para>
985     /// </param>
986     /// <param name="argumentIndex">
987     /// <para>The argument index.</para>
988     /// <para></para>
989     /// </param>
990     [MethodImpl(MethodImplOptions.AggressiveInlining)]
991     public static void LoadArgument(this ILGenerator generator, int argumentIndex)
992     {
993         switch (argumentIndex)
994         {
995             case 0:
996                 generator.Emit(OpCodes.Ldarg_0);
997                 break;
998             case 1:
999                 generator.Emit(OpCodes.Ldarg_1);
1000                 break;
1001             case 2:
1002                 generator.Emit(OpCodes.Ldarg_2);
1003                 break;
1004             case 3:
1005                 generator.Emit(OpCodes.Ldarg_3);
1006                 break;
1007             default:
1008                 generator.Emit(OpCodes.Ldarg, argumentIndex);
1009                 break;
1010         }
1011     }
1012
1013     /// <summary>
1014     /// <para>
1015     /// Loads the arguments using the specified generator.
1016     /// </para>
1017     /// <para></para>
1018     /// </summary>
1019     /// <param name="generator">
1020     /// <para>The generator.</para>
1021     /// <para></para>

```

```

1022     /// </param>
1023     /// <param name="argumentIndices">
1024     /// <para>The argument indices.</para>
1025     /// </para></param>
1026     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1027     public static void LoadArguments(this ILGenerator generator, params int[]
1028     ↪ argumentIndices)
1029     {
1030         for (var i = 0; i < argumentIndices.Length; i++)
1031         {
1032             generator.LoadArgument(argumentIndices[i]);
1033         }
1034     }
1035
1036     /// <summary>
1037     /// <para>
1038     /// Stores the argument using the specified generator.
1039     /// </para>
1040     /// </para></summary>
1041     /// <param name="generator">
1042     /// <para>The generator.</para>
1043     /// </para></param>
1044     /// <param name="argumentIndex">
1045     /// <para>The argument index.</para>
1046     /// </para></param>
1047     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1048     public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
1049     ↪ generator.Emit(OpCodes.Starg, argumentIndex);
1050
1051     /// <summary>
1052     /// <para>
1053     /// Compares the greater than using the specified generator.
1054     /// </para>
1055     /// </para></summary>
1056     /// <param name="generator">
1057     /// <para>The generator.</para>
1058     /// </para></param>
1059     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1060     public static void CompareGreaterThan(this ILGenerator generator) =>
1061     ↪ generator.Emit(OpCodes.Cgt);
1062
1063     /// <summary>
1064     /// <para>
1065     /// Unsigneds the compare greater than using the specified generator.
1066     /// </para>
1067     /// </para></summary>
1068     /// <param name="generator">
1069     /// <para>The generator.</para>
1070     /// </para></param>
1071     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1072     public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
1073     ↪ generator.Emit(OpCodes.Cgt_Un);
1074
1075     /// <summary>
1076     /// <para>
1077     /// Compares the greater than using the specified generator.
1078     /// </para>
1079     /// </para></summary>
1080     /// <param name="generator">
1081     /// <para>The generator.</para>
1082     /// </para></param>
1083     /// <param name="isSigned">
1084     /// <para>The is signed.</para>
1085     /// </para></param>
1086     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1087     public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
1088     {
1089

```

```

1096         if (isSigned)
1097         {
1098             generator.CompareGreaterThan();
1099         }
1100         else
1101         {
1102             generator.UnsignedCompareGreaterThan();
1103         }
1104     }
1105
1106     /// <summary>
1107     /// <para>
1108     /// Compares the less than using the specified generator.
1109     /// </para>
1110     /// <para></para>
1111     /// </summary>
1112     /// <param name="generator">
1113     /// <para>The generator.</para>
1114     /// <para></para>
1115     /// </param>
1116     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1117     public static void CompareLessThan(this ILGenerator generator) =>
1118         ↪ generator.Emit(OpCodes.Clt);
1119
1120     /// <summary>
1121     /// <para>
1122     /// Unsigneds the compare less than using the specified generator.
1123     /// </para>
1124     /// <para></para>
1125     /// </summary>
1126     /// <param name="generator">
1127     /// <para>The generator.</para>
1128     /// <para></para>
1129     /// </param>
1130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1131     public static void UnsignedCompareLessThan(this ILGenerator generator) =>
1132         ↪ generator.Emit(OpCodes.Clt_Un);
1133
1134     /// <summary>
1135     /// <para>
1136     /// Compares the less than using the specified generator.
1137     /// </para>
1138     /// <para></para>
1139     /// </summary>
1140     /// <param name="generator">
1141     /// <para>The generator.</para>
1142     /// <para></para>
1143     /// </param>
1144     /// <param name="isSigned">
1145     /// <para>The is signed.</para>
1146     /// <para></para>
1147     /// </param>
1148     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1149     public static void CompareLessThan(this ILGenerator generator, bool isSigned)
1150     {
1151         if (isSigned)
1152         {
1153             generator.CompareLessThan();
1154         }
1155         else
1156         {
1157             generator.UnsignedCompareLessThan();
1158         }
1159     }
1160
1161     /// <summary>
1162     /// <para>
1163     /// Branches the if greater or equal using the specified generator.
1164     /// </para>
1165     /// <para></para>
1166     /// </summary>
1167     /// <param name="generator">
1168     /// <para>The generator.</para>
1169     /// <para></para>
1170     /// </param>
1171     /// <param name="label">
1172     /// <para>The label.</para>
1173     /// <para></para>

```

```

1172     /// </param>
1173     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1174     public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
1175         ↪ generator.Emit(OpCodes.Bge, label);
1176
1177     /// <summary>
1178     /// <para>
1179     /// Unsigneds the branch if greater or equal using the specified generator.
1180     /// </para>
1181     /// <para></para>
1182     /// </summary>
1183     /// <param name="generator">
1184     /// <para>The generator.</para>
1185     /// <para></para>
1186     /// </param>
1187     /// <param name="label">
1188     /// <para>The label.</para>
1189     /// <para></para>
1190     /// </param>
1191     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1192     public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
1193         ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
1194
1195     /// <summary>
1196     /// <para>
1197     /// Branches the if greater or equal using the specified generator.
1198     /// </para>
1199     /// <para></para>
1200     /// </summary>
1201     /// <param name="generator">
1202     /// <para>The generator.</para>
1203     /// <para></para>
1204     /// </param>
1205     /// <param name="isSigned">
1206     /// <para>The is signed.</para>
1207     /// <para></para>
1208     /// </param>
1209     /// <param name="label">
1210     /// <para>The label.</para>
1211     /// <para></para>
1212     /// </param>
1213     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1214     public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
1215         ↪ Label label)
1216     {
1217         if (isSigned)
1218         {
1219             generator.BranchIfGreaterOrEqual(label);
1220         }
1221         else
1222         {
1223             generator.UnsignedBranchIfGreaterOrEqual(label);
1224         }
1225     }
1226
1227     /// <summary>
1228     /// <para>
1229     /// Branches the if less or equal using the specified generator.
1230     /// </para>
1231     /// <para></para>
1232     /// </summary>
1233     /// <param name="generator">
1234     /// <para>The generator.</para>
1235     /// <para></para>
1236     /// </param>
1237     /// <param name="label">
1238     /// <para>The label.</para>
1239     /// <para></para>
1240     /// </param>
1241     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1242     public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
1243         ↪ generator.Emit(OpCodes.Ble, label);
1244
1245     /// <summary>
1246     /// <para>
1247     /// Unsigneds the branch if less or equal using the specified generator.
1248     /// </para>
1249     /// <para></para>

```

```

1246    /// </summary>
1247    /// <param name="generator">
1248    /// <para>The generator.</para>
1249    /// <para></para>
1250    /// </param>
1251    /// <param name="label">
1252    /// <para>The label.</para>
1253    /// <para></para>
1254    /// </param>
1255    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1256    public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
1257        => generator.Emit(OpCodes.Ble_Un, label);
1258
1259    /// <summary>
1260    /// <para>
1261    /// Branches the if less or equal using the specified generator.
1262    /// </para>
1263    /// <para></para>
1264    /// </summary>
1265    /// <param name="generator">
1266    /// <para>The generator.</para>
1267    /// <para></para>
1268    /// </param>
1269    /// <param name="isSigned">
1270    /// <para>The is signed.</para>
1271    /// <para></para>
1272    /// </param>
1273    /// <param name="label">
1274    /// <para>The label.</para>
1275    /// <para></para>
1276    /// </param>
1277    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1278    public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
1279        label)
1280    {
1281        if (isSigned)
1282        {
1283            generator.BranchIfLessOrEqual(label);
1284        }
1285        else
1286        {
1287            generator.UnsignedBranchIfLessOrEqual(label);
1288        }
1289    }
1290
1291    /// <summary>
1292    /// <para>
1293    /// Boxes the generator.
1294    /// </para>
1295    /// <para></para>
1296    /// </summary>
1297    /// <typeparam name="TBox">
1298    /// <para>The box.</para>
1299    /// <para></para>
1300    /// </typeparam>
1301    /// <param name="generator">
1302    /// <para>The generator.</para>
1303    /// <para></para>
1304    /// </param>
1305    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1306    public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
1307
1308    /// <summary>
1309    /// <para>
1310    /// Boxes the generator.
1311    /// </para>
1312    /// <para></para>
1313    /// </summary>
1314    /// <param name="generator">
1315    /// <para>The generator.</para>
1316    /// <para></para>
1317    /// </param>
1318    /// <param name="boxedType">
1319    /// <para>The boxed type.</para>
1320    /// <para></para>
1321    /// </param>
1322    [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

1321 public static void Box(this ILGenerator generator, Type boxedType) =>
1322     ↪ generator.Emit(OpCodes.Box, boxedType);
1323
1324 /// <summary>
1325 /// <para>
1326 /// Calls the generator.
1327 /// </para>
1328 /// </summary>
1329 /// <param name="generator">
1330 /// <para>The generator.</para>
1331 /// </param>
1332 /// <param name="method">
1333 /// <para>The method.</para>
1334 /// </param>
1335 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1336 public static void Call(this ILGenerator generator, MethodInfo method) =>
1337     ↪ generator.Emit(OpCodes.Call, method);
1338
1339 /// <summary>
1340 /// <para>
1341 /// Returns the generator.
1342 /// </para>
1343 /// </summary>
1344 /// <param name="generator">
1345 /// <para>The generator.</para>
1346 /// </param>
1347 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1348 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
1349
1350 /// <summary>
1351 /// <para>
1352 /// Unboxes the generator.
1353 /// </para>
1354 /// </summary>
1355 /// <typeparam name="TUnbox">
1356 /// <para>The unbox.</para>
1357 /// </typeparam>
1358 /// <param name="generator">
1359 /// <para>The generator.</para>
1360 /// </param>
1361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1362 public static void Unbox<TUnbox>(this ILGenerator generator) =>
1363     ↪ generator.Unbox(typeof(TUnbox));
1364
1365 /// <summary>
1366 /// <para>
1367 /// Unboxes the generator.
1368 /// </para>
1369 /// </summary>
1370 /// <param name="generator">
1371 /// <para>The generator.</para>
1372 /// </param>
1373 /// <param name="typeToUnbox">
1374 /// <para>The type to unbox.</para>
1375 /// </param>
1376 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1377 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
1378     ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
1379
1380 /// <summary>
1381 /// <para>
1382 /// Unboxes the value using the specified generator.
1383 /// </para>
1384 /// </summary>
1385 /// <typeparam name="TUnbox">
1386 /// <para>The unbox.</para>
1387

```



```

1395     /// <para></para>
1396     /// </typeparam>
1397     /// <param name="generator">
1398     /// <para>The generator.</para>
1399     /// <para></para>
1400     /// </param>
1401     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1402     public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
1403         ↪ generator.UnboxValue(typeof(TUnbox));
1404
1405     /// <summary>
1406     /// <para>
1407     /// Unboxes the value using the specified generator.
1408     /// </para>
1409     /// </summary>
1410     /// <param name="generator">
1411     /// <para>The generator.</para>
1412     /// <para></para>
1413     /// </param>
1414     /// <param name="typeToUnbox">
1415     /// <para>The type to unbox.</para>
1416     /// <para></para>
1417     /// </param>
1418     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1419     public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
1420         ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
1421
1422     /// <summary>
1423     /// <para>
1424     /// Declares the local using the specified generator.
1425     /// </para>
1426     /// <para></para>
1427     /// </summary>
1428     /// <typeparam name="T">
1429     /// <para>The .</para>
1430     /// <para></para>
1431     /// </typeparam>
1432     /// <param name="generator">
1433     /// <para>The generator.</para>
1434     /// <para></para>
1435     /// </param>
1436     /// <returns>
1437     /// <para>The local builder</para>
1438     /// <para></para>
1439     /// </returns>
1440     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1441     public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
1442         ↪ generator.DeclareLocal(typeof(T));
1443
1444     /// <summary>
1445     /// <para>
1446     /// Loads the local using the specified generator.
1447     /// </para>
1448     /// <para></para>
1449     /// </summary>
1450     /// <param name="generator">
1451     /// <para>The generator.</para>
1452     /// <para></para>
1453     /// </param>
1454     /// <param name="local">
1455     /// <para>The local.</para>
1456     /// <para></para>
1457     /// </param>
1458     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1459     public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
1460         ↪ generator.Emit(OpCodes.Ldloc, local);
1461
1462     /// <summary>
1463     /// <para>
1464     /// Stores the local using the specified generator.
1465     /// </para>
1466     /// <para></para>
1467     /// </summary>
1468     /// <param name="generator">
1469     /// <para>The generator.</para>
1470     /// <para></para>
1471     /// </param>

```

```

1469     /// <param name="local">
1470     /// <para>The local.</para>
1471     /// <para></para>
1472     /// </param>
1473     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1474     public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
1475         ↪ generator.Emit(OpCodes.Stloc, local);
1476
1477     /// <summary>
1478     /// <para>
1479     /// News the object using the specified generator.
1480     /// </para>
1481     /// <para></para>
1482     /// </summary>
1483     /// <param name="generator">
1484     /// <para>The generator.</para>
1485     /// <para></para>
1486     /// </param>
1487     /// <param name="type">
1488     /// <para>The type.</para>
1489     /// <para></para>
1490     /// </param>
1491     /// <param name="parameterTypes">
1492     /// <para>The parameter types.</para>
1493     /// <para></para>
1494     /// </param>
1495     /// <exception cref="InvalidOperationException">
1496     /// <para></para>
1497     /// <para></para>
1498     /// </exception>
1499     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1500     public static void NewObject(this ILGenerator generator, Type type, params Type[]
1501         ↪ parameterTypes)
1502     {
1503         var allConstructors = type.GetConstructors(BindingFlags.Public |
1504             ↪ BindingFlags.NonPublic | BindingFlags.Instance
1505             | BindingFlags.CreateInstance
1506             );
1507         var constructor = allConstructors.Where(c => c.GetParameters().Length ==
1508             ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
1509             ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
1510         if (constructor == null)
1511         {
1512             throw new InvalidOperationException("Type " + type + " must have a constructor
1513                 ↪ that matches parameters [" + string.Join(", ",
1514                 ↪ parameterTypes.AsEnumerable()) + "]");
1515         }
1516         generator.NewObject(constructor);
1517     }
1518
1519     /// <summary>
1520     /// <para>
1521     /// News the object using the specified generator.
1522     /// </para>
1523     /// <para></para>
1524     /// </summary>
1525     /// <param name="generator">
1526     /// <para>The generator.</para>
1527     /// <para></para>
1528     /// </param>
1529     /// <param name="constructor">
1530     /// <para>The constructor.</para>
1531     /// <para></para>
1532     /// </param>
1533     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1534     public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
1535         ↪ generator.Emit(OpCodes.Newobj, constructor);
1536
1537     /// <summary>
1538     /// <para>
1539     /// Loads the indirect using the specified generator.
1540     /// </para>
1541     /// <para></para>
1542     /// </summary>
1543     /// <typeparam name="T">

```

```

1538    /// <para>The .</para>
1539    /// <para></para>
1540    /// </typeparam>
1541    /// <param name="generator">
1542    /// <para>The generator.</para>
1543    /// <para></para>
1544    /// </param>
1545    /// <param name="isVolatile">
1546    /// <para>The is volatile.</para>
1547    /// <para></para>
1548    /// </param>
1549    /// <param name="unaligned">
1550    /// <para>The unaligned.</para>
1551    /// <para></para>
1552    /// </param>
1553    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1554    public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
    ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
1555
1556    /// <summary>
1557    /// <para>
1558    /// Loads the indirect using the specified generator.
1559    /// </para>
1560    /// <para></para>
1561    /// </summary>
1562    /// <param name="generator">
1563    /// <para>The generator.</para>
1564    /// <para></para>
1565    /// </param>
1566    /// <param name="type">
1567    /// <para>The type.</para>
1568    /// <para></para>
1569    /// </param>
1570    /// <param name="isVolatile">
1571    /// <para>The is volatile.</para>
1572    /// <para></para>
1573    /// </param>
1574    /// <param name="unaligned">
1575    /// <para>The unaligned.</para>
1576    /// <para></para>
1577    /// </param>
1578    /// <exception cref="InvalidOperationException">
1579    /// <para></para>
1580    /// <para></para>
1581    /// </exception>
1582    /// <exception cref="ArgumentException">
1583    /// <para>unaligned must be null, 1, 2, or 4</para>
1584    /// <para></para>
1585    /// </exception>
1586    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1587    public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
    ↪ false, byte? unaligned = null)
1588    {
1589        if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
1590        {
1591            throw new ArgumentException("unaligned must be null, 1, 2, or 4");
1592        }
1593        if (isVolatile)
1594        {
1595            generator.Emit(OpCodes.Volatile);
1596        }
1597        if (unaligned.HasValue)
1598        {
1599            generator.Emit(OpCodes.Unaligned, unaligned.Value);
1600        }
1601        if (type.IsPointer)
1602        {
1603            generator.Emit(OpCodes.Ldind_I);
1604        }
1605        else if (!type.IsValueType)
1606        {
1607            generator.Emit(OpCodes.Ldind_Ref);
1608        }
1609        else if (type == typeof(sbyte))
1610        {
1611            generator.Emit(OpCodes.Ldind_I1);
1612        }

```

```

1613     else if (type == typeof(bool))
1614     {
1615         generator.Emit(OpCodes.Ldind_I1);
1616     }
1617     else if (type == typeof(byte))
1618     {
1619         generator.Emit(OpCodes.Ldind_U1);
1620     }
1621     else if (type == typeof(short))
1622     {
1623         generator.Emit(OpCodes.Ldind_I2);
1624     }
1625     else if (type == typeof(ushort))
1626     {
1627         generator.Emit(OpCodes.Ldind_U2);
1628     }
1629     else if (type == typeof(char))
1630     {
1631         generator.Emit(OpCodes.Ldind_U2);
1632     }
1633     else if (type == typeof(int))
1634     {
1635         generator.Emit(OpCodes.Ldind_I4);
1636     }
1637     else if (type == typeof(uint))
1638     {
1639         generator.Emit(OpCodes.Ldind_U4);
1640     }
1641     else if (type == typeof(long) || type == typeof(ulong))
1642     {
1643         generator.Emit(OpCodes.Ldind_I8);
1644     }
1645     else if (type == typeof(float))
1646     {
1647         generator.Emit(OpCodes.Ldind_R4);
1648     }
1649     else if (type == typeof(double))
1650     {
1651         generator.Emit(OpCodes.Ldind_R8);
1652     }
1653     else
1654     {
1655         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
            ↪ " , LoadObject may be more appropriate");
1656     }
1657 }
1658
1659 /// <summary>
1660 /// <para>
1661 /// Stores the indirect using the specified generator.
1662 /// </para>
1663 /// <para></para>
1664 /// </summary>
1665 /// <typeparam name="T">
1666 /// <para>The .</para>
1667 /// <para></para>
1668 /// </typeparam>
1669 /// <param name="generator">
1670 /// <para>The generator.</para>
1671 /// <para></para>
1672 /// </param>
1673 /// <param name="isVolatile">
1674 /// <para>The is volatile.</para>
1675 /// <para></para>
1676 /// </param>
1677 /// <param name="unaligned">
1678 /// <para>The unaligned.</para>
1679 /// <para></para>
1680 /// </param>
1681 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1682 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
    ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
1683
1684 /// <summary>
1685 /// <para>
1686 /// Stores the indirect using the specified generator.
1687 /// </para>
1688 /// <para></para>

```

```

1689 /// </summary>
1690 /// <param name="generator">
1691 /// <para>The generator.</para>
1692 /// <para></para>
1693 /// </param>
1694 /// <param name="type">
1695 /// <para>The type.</para>
1696 /// <para></para>
1697 /// </param>
1698 /// <param name="isVolatile">
1699 /// <para>The is volatile.</para>
1700 /// <para></para>
1701 /// </param>
1702 /// <param name="unaligned">
1703 /// <para>The unaligned.</para>
1704 /// <para></para>
1705 /// </param>
1706 /// <exception cref="InvalidOperationException">
1707 /// <para></para>
1708 /// <para></para>
1709 /// </exception>
1710 /// <exception cref="ArgumentException">
1711 /// <para>unaligned must be null, 1, 2, or 4</para>
1712 /// <para></para>
1713 /// </exception>
1714 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1715 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
↵ = false, byte? unaligned = null)
1716 {
1717     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
1718     {
1719         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
1720     }
1721     if (isVolatile)
1722     {
1723         generator.Emit(OpCodes.Volatile);
1724     }
1725     if (unaligned.HasValue)
1726     {
1727         generator.Emit(OpCodes.Unaligned, unaligned.Value);
1728     }
1729     if (type.IsPointer)
1730     {
1731         generator.Emit(OpCodes.Stind_I);
1732     }
1733     else if (!type.IsValueType)
1734     {
1735         generator.Emit(OpCodes.Stind_Ref);
1736     }
1737     else if (type == typeof(sbyte) || type == typeof(byte))
1738     {
1739         generator.Emit(OpCodes.Stind_I1);
1740     }
1741     else if (type == typeof(short) || type == typeof(ushort))
1742     {
1743         generator.Emit(OpCodes.Stind_I2);
1744     }
1745     else if (type == typeof(int) || type == typeof(uint))
1746     {
1747         generator.Emit(OpCodes.Stind_I4);
1748     }
1749     else if (type == typeof(long) || type == typeof(ulong))
1750     {
1751         generator.Emit(OpCodes.Stind_I8);
1752     }
1753     else if (type == typeof(float))
1754     {
1755         generator.Emit(OpCodes.Stind_R4);
1756     }
1757     else if (type == typeof(double))
1758     {
1759         generator.Emit(OpCodes.Stind_R8);
1760     }
1761     else
1762     {
1763         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
↵ + ", StoreObject may be more appropriate");

```

```

1764     }
1765 }
1766
1767     /// <summary>
1768     /// <para>
1769     /// Multiplies the generator.
1770     /// </para>
1771     /// <para></para>
1772     /// </summary>
1773     /// <param name="generator">
1774     /// <para>The generator.</para>
1775     /// <para></para>
1776     /// </param>
1777     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1778     public static void Multiply(this ILGenerator generator)
1779     {
1780         generator.Emit(OpCodes.Mul);
1781     }
1782
1783     /// <summary>
1784     /// <para>
1785     /// Checkeds the multiply using the specified generator.
1786     /// </para>
1787     /// <para></para>
1788     /// </summary>
1789     /// <typeparam name="T">
1790     /// <para>The .</para>
1791     /// <para></para>
1792     /// </typeparam>
1793     /// <param name="generator">
1794     /// <para>The generator.</para>
1795     /// <para></para>
1796     /// </param>
1797     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1798     public static void CheckedMultiply<T>(this ILGenerator generator)
1799     {
1800         if (NumericType<T>.IsSigned)
1801         {
1802             generator.Emit(OpCodes.Mul_Ovf);
1803         }
1804         else
1805         {
1806             generator.Emit(OpCodes.Mul_Ovf_Un);
1807         }
1808     }
1809
1810     /// <summary>
1811     /// <para>
1812     /// Divides the generator.
1813     /// </para>
1814     /// <para></para>
1815     /// </summary>
1816     /// <typeparam name="T">
1817     /// <para>The .</para>
1818     /// <para></para>
1819     /// </typeparam>
1820     /// <param name="generator">
1821     /// <para>The generator.</para>
1822     /// <para></para>
1823     /// </param>
1824     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1825     public static void Divide<T>(this ILGenerator generator)
1826     {
1827         if (NumericType<T>.IsSigned)
1828         {
1829             generator.Emit(OpCodes.Div);
1830         }
1831         else
1832         {
1833             generator.Emit(OpCodes.Div_Un);
1834         }
1835     }
1836 }
1837 }

```

## 1.7 ./csharp/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     /// <summary>
11     /// <para>
12     /// Represents the method info extensions.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     public static class MethodInfoExtensions
17     {
18         /// <summary>
19         /// <para>
20         /// Gets the il bytes using the specified method info.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         /// <param name="methodInfo">
25         /// <para>The method info.</para>
26         /// <para></para>
27         /// </param>
28         /// <returns>
29         /// <para>The byte array</para>
30         /// <para></para>
31         /// </returns>
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static byte[] GetILBytes(this MethodInfo methodInfo) =>
34             ↪ methodInfo.GetMethodBody().GetILAsByteArray();
35
36         /// <summary>
37         /// <para>
38         /// Gets the parameter types using the specified method info.
39         /// </para>
40         /// <para></para>
41         /// </summary>
42         /// <param name="methodInfo">
43         /// <para>The method info.</para>
44         /// <para></para>
45         /// </param>
46         /// <returns>
47         /// <para>The type array</para>
48         /// <para></para>
49         /// </returns>
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
52             ↪ methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
53     }
54 }

```

## 1.8 ./csharp/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     /// <summary>
11     /// <para>
12     /// Represents the not supported exception delegate factory.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="IFactory{TDelegate}">
17     public class NotSupportedExceptionDelegateFactory<TDelegate> : IFactory<TDelegate>
18         where TDelegate : Delegate
19     {
20         /// <summary>
21         /// <para>
22         /// Creates this instance.
23         /// </para>
24         /// <para></para>
25         /// </summary>

```

```

26     /// <exception cref="InvalidOperationException">
27     /// <para>Unable to compile stub delegate.</para>
28     /// <para></para>
29     /// </exception>
30     /// <returns>
31     /// <para>The delegate.</para>
32     /// <para></para>
33     /// </returns>
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     public TDelegate Create()
36     {
37         var @delegate = DelegateHelpers.CompileOrDefault<TDelegate>(generator =>
38         {
39             generator.Throw<NotSupportedException>();
40         });
41         if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
42         {
43             throw new InvalidOperationException("Unable to compile stub delegate.");
44         }
45         return @delegate;
46     }
47 }
48 }

```

## 1.9 ./csharp/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     /// <summary>
14     /// <para>
15     /// Represents the numeric type.
16     /// </para>
17     /// <para></para>
18     /// </summary>
19     public static class NumericType<T>
20     {
21         /// <summary>
22         /// <para>
23         /// The type.
24         /// </para>
25         /// <para></para>
26         /// </summary>
27         public static readonly Type Type;
28         /// <summary>
29         /// <para>
30         /// The underlying type.
31         /// </para>
32         /// <para></para>
33         /// </summary>
34         public static readonly Type UnderlyingType;
35         /// <summary>
36         /// <para>
37         /// The signed version.
38         /// </para>
39         /// <para></para>
40         /// </summary>
41         public static readonly Type SignedVersion;
42         /// <summary>
43         /// <para>
44         /// The unsigned version.
45         /// </para>
46         /// <para></para>
47         /// </summary>
48         public static readonly Type UnsignedVersion;
49         /// <summary>
50         /// <para>
51         /// The is float point.
52         /// </para>
53         /// <para></para>
54         /// </summary>

```



```

55     public static readonly bool IsFloatPoint;
56     /// <summary>
57     /// <para>
58     /// The is numeric.
59     /// </para>
60     /// <para></para>
61     /// </summary>
62     public static readonly bool IsNumeric;
63     /// <summary>
64     /// <para>
65     /// The is signed.
66     /// </para>
67     /// <para></para>
68     /// </summary>
69     public static readonly bool IsSigned;
70     /// <summary>
71     /// <para>
72     /// The can be numeric.
73     /// </para>
74     /// <para></para>
75     /// </summary>
76     public static readonly bool CanBeNumeric;
77     /// <summary>
78     /// <para>
79     /// The is nullable.
80     /// </para>
81     /// <para></para>
82     /// </summary>
83     public static readonly bool IsNullable;
84     /// <summary>
85     /// <para>
86     /// The bytes size.
87     /// </para>
88     /// <para></para>
89     /// </summary>
90     public static readonly int BytesSize;
91     /// <summary>
92     /// <para>
93     /// The bits size.
94     /// </para>
95     /// <para></para>
96     /// </summary>
97     public static readonly int BitsSize;
98     /// <summary>
99     /// <para>
100    /// The min value.
101    /// </para>
102    /// <para></para>
103    /// </summary>
104    public static readonly T MinValue;
105    /// <summary>
106    /// <para>
107    /// The max value.
108    /// </para>
109    /// <para></para>
110    /// </summary>
111    public static readonly T MaxValue;
112
113    /// <summary>
114    /// <para>
115    /// Initializes a new <see cref="NumericType"/> instance.
116    /// </para>
117    /// <para></para>
118    /// </summary>
119    [MethodImpl(MethodImplOptions.AggressiveInlining)]
120    static NumericType()
121    {
122        try
123        {
124            var type = typeof(T);
125            var isNullable = type.IsNullable();
126            var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
127            var canBeNumeric = underlyingType.CanBeNumeric();
128            var isNumeric = underlyingType.IsNumeric();
129            var isSigned = underlyingType.IsSigned();
130            var isFloatPoint = underlyingType.IsFloatPoint();
131            var bytesSize = Marshal.SizeOf(underlyingType);
132            var bitsSize = bytesSize * 8;

```

```

133         GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
134         GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
        ↳ out Type unsignedVersion);
135         Type = type;
136         IsNullable = isNullable;
137         UnderlyingType = underlyingType;
138         CanBeNumeric = canBeNumeric;
139         IsNumeric = isNumeric;
140         IsSigned = isSigned;
141         IsFloatPoint = isFloatPoint;
142         BytesSize = bytesSize;
143         BitsSize = bitsSize;
144         MinValue = minValue;
145         MaxValue = maxValue;
146         SignedVersion = signedVersion;
147         UnsignedVersion = unsignedVersion;
148     }
149     catch (Exception exception)
150     {
151         exception.Ignore();
152     }
153 }
154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
156 {
157     if (type == typeof(bool))
158     {
159         minValue = (T)(object)false;
160         maxValue = (T)(object>true;
161     }
162     else
163     {
164         minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
165         maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
166     }
167 }
168 [MethodImpl(MethodImplOptions.AggressiveInlining)]
169 private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
        ↳ signedVersion, out Type unsignedVersion)
170 {
171     if (isSigned)
172     {
173         signedVersion = type;
174         unsignedVersion = type.GetUnsignedVersionOrNull();
175     }
176     else
177     {
178         signedVersion = type.GetSignedVersionOrNull();
179         unsignedVersion = type;
180     }
181 }
182 }
183 }

```

## 1.10 ./csharp/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     /// <summary>
9     /// <para>
10    /// Represents the property info extensions.
11    /// </para>
12    /// <para></para>
13    /// </summary>
14    public static class PropertyInfoExtensions
15    {
16        /// <summary>
17        /// <para>
18        /// Gets the static value using the specified field info.
19        /// </para>
20        /// <para></para>
21        /// </summary>
22        /// <typeparam name="T">
23        /// <para>The .</para>
24        /// <para></para>

```

```

25     /// </typeparam>
26     /// <param name="fieldInfo">
27     /// <para>The field info.</para>
28     /// <para></para>
29     /// </param>
30     /// <returns>
31     /// <para>The</para>
32     /// <para></para>
33     /// </returns>
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     public static T GetStaticValue<T>(<this> PropertyInfo fieldInfo) =>
        ↪ (T)fieldInfo.GetValue(null);
36 }
37 }

```

### 1.11 ./csharp/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     /// <summary>
11     /// <para>
12     /// Represents the type builder extensions.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     public static class TypeBuilderExtensions
17     {
18         /// <summary>
19         /// <para>
20         /// The static.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         public static readonly MethodAttributes DefaultStaticMethodAttributes =
            ↪ MethodAttributes.Public | MethodAttributes.Static;
25
26         /// <summary>
27         /// <para>
28         /// The hide by sig.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes =
            ↪ MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
            ↪ MethodAttributes.HideBySig;
33
34         /// <summary>
35         /// <para>
36         /// The aggressive inlining.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         public static readonly MethodImplAttributes DefaultMethodImplAttributes =
            ↪ MethodImplAttributes.IL | MethodImplAttributes.Managed |
            ↪ MethodImplAttributes.AggressiveInlining;
41
42         /// <summary>
43         /// <para>
44         /// Emits the method using the specified type.
45         /// </para>
46         /// <para></para>
47         /// </summary>
48         /// <typeparam name="TDelegate">
49         /// <para>The delegate.</para>
50         /// <para></para>
51         /// </typeparam>
52         /// <param name="type">
53         /// <para>The type.</para>
54         /// <para></para>
55         /// </param>
56         /// <param name="methodName">
57         /// <para>The method name.</para>
58         /// <para></para>
59         /// </param>

```

```

58     /// <param name="methodAttributes">
59     /// <para>The method attributes.</para>
60     /// </para>
61     /// </param>
62     /// <param name="methodImplAttributes">
63     /// <para>The method impl attributes.</para>
64     /// </para>
65     /// </param>
66     /// <param name="emitCode">
67     /// <para>The emit code.</para>
68     /// </para>
69     /// </param>
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName,
72     ↪ MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes,
73     ↪ Action<ILGenerator> emitCode)
74     {
75         typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[]
76         ↪ parameterTypes);
77         EmitMethod(type, methodName, methodAttributes, methodImplAttributes, returnType,
78         ↪ parameterTypes, emitCode);
79     }
80
81     /// <summary>
82     /// <para>
83     /// Emits the method using the specified type.
84     /// </para>
85     /// </summary>
86     /// <param name="type">
87     /// <para>The type.</para>
88     /// </param>
89     /// <param name="methodName">
90     /// <para>The method name.</para>
91     /// </param>
92     /// <param name="methodAttributes">
93     /// <para>The method attributes.</para>
94     /// </param>
95     /// <param name="methodImplAttributes">
96     /// <para>The method impl attributes.</para>
97     /// </param>
98     /// <param name="returnType">
99     /// <para>The return type.</para>
100    /// </param>
101    /// <param name="parameterTypes">
102    /// <para>The parameter types.</para>
103    /// </param>
104    /// <param name="emitCode">
105    /// <para>The emit code.</para>
106    /// </param>
107    [MethodImpl(MethodImplOptions.AggressiveInlining)]
108    public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
109    ↪ methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[]
110    ↪ parameterTypes, Action<ILGenerator> emitCode)
111    {
112        MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
113        ↪ parameterTypes);
114        method.SetImplementationFlags(methodImplAttributes);
115        var generator = method.GetILGenerator();
116        emitCode(generator);
117    }
118
119    /// <summary>
120    /// <para>
121    /// Emits the static method using the specified type.
122    /// </para>
123    /// </summary>
124    /// <typeparam name="TDelegate">
125    /// <para>The delegate.</para>

```

```

128     /// <para></para>
129     /// </typeparam>
130     /// <param name="type">
131     /// <para>The type.</para>
132     /// <para></para>
133     /// </param>
134     /// <param name="methodName">
135     /// <para>The method name.</para>
136     /// <para></para>
137     /// </param>
138     /// <param name="emitCode">
139     /// <para>The emit code.</para>
140     /// <para></para>
141     /// </param>
142     [MethodImpl(MethodImplOptions.AggressiveInlining)]
143     public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName,
        ↪ Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
        ↪ DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);
144
145     /// <summary>
146     /// <para>
147     /// Emits the final virtual method using the specified type.
148     /// </para>
149     /// <para></para>
150     /// </summary>
151     /// <typeparam name="TDelegate">
152     /// <para>The delegate.</para>
153     /// <para></para>
154     /// </typeparam>
155     /// <param name="type">
156     /// <para>The type.</para>
157     /// <para></para>
158     /// </param>
159     /// <param name="methodName">
160     /// <para>The method name.</para>
161     /// <para></para>
162     /// </param>
163     /// <param name="emitCode">
164     /// <para>The emit code.</para>
165     /// <para></para>
166     /// </param>
167     [MethodImpl(MethodImplOptions.AggressiveInlining)]
168     public static void EmitFinalVirtualMethod<TDelegate>(this TypeBuilder type, string
        ↪ methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
        ↪ DefaultFinalVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
169 }
170 }

```

## 1.12 ./csharp/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     /// <summary>
13     /// <para>
14     /// Represents the type extensions.
15     /// </para>
16     /// <para></para>
17     /// </summary>
18     public static class TypeExtensions
19     {
20         /// <summary>
21         /// <para>
22         /// The static.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         static public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
            ↪ BindingFlags.NonPublic | BindingFlags.Static;
27         /// <summary>
28         /// <para>

```

```

29     /// The default delegate method name.
30     /// </para>
31     /// <para></para>
32     /// </summary>
33     static public readonly string DefaultDelegateMethodName = "Invoke";
34
35     /// <summary>
36     /// <para>
37     /// The can be numeric types.
38     /// </para>
39     /// <para></para>
40     /// </summary>
41     static private readonly HashSet<Type> _canBeNumericTypes;
42     /// <summary>
43     /// <para>
44     /// The is numeric types.
45     /// </para>
46     /// <para></para>
47     /// </summary>
48     static private readonly HashSet<Type> _isNumericTypes;
49     /// <summary>
50     /// <para>
51     /// The is signed types.
52     /// </para>
53     /// <para></para>
54     /// </summary>
55     static private readonly HashSet<Type> _isSignedTypes;
56     /// <summary>
57     /// <para>
58     /// The is float point types.
59     /// </para>
60     /// <para></para>
61     /// </summary>
62     static private readonly HashSet<Type> _isFloatPointTypes;
63     /// <summary>
64     /// <para>
65     /// The unsigned versions of signed types.
66     /// </para>
67     /// <para></para>
68     /// </summary>
69     static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
70     /// <summary>
71     /// <para>
72     /// The signed versions of unsigned types.
73     /// </para>
74     /// <para></para>
75     /// </summary>
76     static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;
77
78     /// <summary>
79     /// <para>
80     /// Initializes a new <see cref="TypeExtensions"/> instance.
81     /// </para>
82     /// <para></para>
83     /// </summary>
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     static TypeExtensions()
86     {
87         _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
88             ↳ typeof(DateTime), typeof(TimeSpan) };
89         _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
90             ↳ typeof(ulong) };
91         _canBeNumericTypes.UnionWith(_isNumericTypes);
92         _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
93             ↳ typeof(long) };
94         _canBeNumericTypes.UnionWith(_isSignedTypes);
95         _isNumericTypes.UnionWith(_isSignedTypes);
96         _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
97             ↳ typeof(float) };
98         _canBeNumericTypes.UnionWith(_isFloatPointTypes);
99         _isNumericTypes.UnionWith(_isFloatPointTypes);
100        _isSignedTypes.UnionWith(_isFloatPointTypes);
101        _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
102        {
103            { typeof(sbyte), typeof(byte) },
104            { typeof(short), typeof(ushort) },
105            { typeof(int), typeof(uint) },
106            { typeof(long), typeof(ulong) },

```

```

103     };
104     signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
105     {
106         { typeof(byte), typeof(sbyte) },
107         { typeof(ushort), typeof(short) },
108         { typeof(uint), typeof(int) },
109         { typeof(ulong), typeof(long) },
110     };
111 }
112
113 /// <summary>
114 /// <para>
115 /// Gets the first field using the specified type.
116 /// </para>
117 /// <para></para>
118 /// </summary>
119 /// <param name="type">
120 /// <para>The type.</para>
121 /// <para></para>
122 /// </param>
123 /// <returns>
124 /// <para>The field info</para>
125 /// <para></para>
126 /// </returns>
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
129
130 /// <summary>
131 /// <para>
132 /// Gets the static field value using the specified type.
133 /// </para>
134 /// <para></para>
135 /// </summary>
136 /// <typeparam name="T">
137 /// <para>The .</para>
138 /// <para></para>
139 /// </typeparam>
140 /// <param name="type">
141 /// <para>The type.</para>
142 /// <para></para>
143 /// </param>
144 /// <param name="name">
145 /// <para>The name.</para>
146 /// <para></para>
147 /// </param>
148 /// <returns>
149 /// <para>The</para>
150 /// <para></para>
151 /// </returns>
152 [MethodImpl(MethodImplOptions.AggressiveInlining)]
153 public static T GetStaticFieldValue<T>(this Type type, string name) =>
154     ↪ type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();
155
156 /// <summary>
157 /// <para>
158 /// Gets the static property value using the specified type.
159 /// </para>
160 /// <para></para>
161 /// </summary>
162 /// <typeparam name="T">
163 /// <para>The .</para>
164 /// <para></para>
165 /// </typeparam>
166 /// <param name="type">
167 /// <para>The type.</para>
168 /// <para></para>
169 /// </param>
170 /// <param name="name">
171 /// <para>The name.</para>
172 /// <para></para>
173 /// </param>
174 /// <returns>
175 /// <para>The</para>
176 /// <para></para>
177 /// </returns>
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public static T GetStaticPropertyValue<T>(this Type type, string name) =>
180     ↪ type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();

```

```

179
180     /// <summary>
181     /// <para>
182     /// Gets the generic method using the specified type.
183     /// </para>
184     /// <para></para>
185     /// </summary>
186     /// <param name="type">
187     /// <para>The type.</para>
188     /// <para></para>
189     /// </param>
190     /// <param name="name">
191     /// <para>The name.</para>
192     /// <para></para>
193     /// </param>
194     /// <param name="genericParameterTypes">
195     /// <para>The generic parameter types.</para>
196     /// <para></para>
197     /// </param>
198     /// <param name="argumentTypes">
199     /// <para>The argument types.</para>
200     /// <para></para>
201     /// </param>
202     /// <returns>
203     /// <para>The method.</para>
204     /// <para></para>
205     /// </returns>
206     [MethodImpl(MethodImplOptions.AggressiveInlining)]
207     public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
        ↳ genericParameterTypes, Type[] argumentTypes)
208     {
209         var methods = from m in type.GetMethods()
210                       where m.Name == name
211                           && m.IsGenericMethodDefinition
212                       let typeParams = m.GetGenericArguments()
213                       let normalParams = m.GetParameters().Select(x => x.ParameterType)
214                       where typeParams.SequenceEqual(genericParameterTypes)
215                           && normalParams.SequenceEqual(argumentTypes)
216                       select m;
217         var method = methods.Single();
218         return method;
219     }
220
221     /// <summary>
222     /// <para>
223     /// Gets the base type using the specified type.
224     /// </para>
225     /// <para></para>
226     /// </summary>
227     /// <param name="type">
228     /// <para>The type.</para>
229     /// <para></para>
230     /// </param>
231     /// <returns>
232     /// <para>The type</para>
233     /// <para></para>
234     /// </returns>
235     [MethodImpl(MethodImplOptions.AggressiveInlining)]
236     public static Type GetBaseType(this Type type) => type.BaseType;
237
238     /// <summary>
239     /// <para>
240     /// Gets the assembly using the specified type.
241     /// </para>
242     /// <para></para>
243     /// </summary>
244     /// <param name="type">
245     /// <para>The type.</para>
246     /// <para></para>
247     /// </param>
248     /// <returns>
249     /// <para>The assembly</para>
250     /// <para></para>
251     /// </returns>
252     [MethodImpl(MethodImplOptions.AggressiveInlining)]
253     public static Assembly GetAssembly(this Type type) => type.Assembly;
254
255     /// <summary>

```



```

256    /// <para>
257    /// Determines whether is subclass of.
258    /// </para>
259    /// <para></para>
260    /// </summary>
261    /// <param name="type">
262    /// <para>The type.</para>
263    /// <para></para>
264    /// </param>
265    /// <param name="superClass">
266    /// <para>The super.</para>
267    /// <para></para>
268    /// </param>
269    /// <returns>
270    /// <para>The bool</para>
271    /// <para></para>
272    /// </returns>
273    [MethodImpl(MethodImplOptions.AggressiveInlining)]
274    public static bool IsSubclassOf(this Type type, Type superClass) =>
275        ↪ type.IsSubclassOf(superClass);
276
277    /// <summary>
278    /// <para>
279    /// Determines whether is value type.
280    /// </para>
281    /// <para></para>
282    /// </summary>
283    /// <param name="type">
284    /// <para>The type.</para>
285    /// <para></para>
286    /// </param>
287    /// <returns>
288    /// <para>The bool</para>
289    /// <para></para>
290    /// </returns>
291    [MethodImpl(MethodImplOptions.AggressiveInlining)]
292    public static bool IsValueType(this Type type) => type.IsValueType;
293
294    /// <summary>
295    /// <para>
296    /// Determines whether is generic.
297    /// </para>
298    /// <para></para>
299    /// </summary>
300    /// <param name="type">
301    /// <para>The type.</para>
302    /// <para></para>
303    /// </param>
304    /// <returns>
305    /// <para>The bool</para>
306    /// <para></para>
307    /// </returns>
308    [MethodImpl(MethodImplOptions.AggressiveInlining)]
309    public static bool IsGeneric(this Type type) => type.IsGenericType;
310
311    /// <summary>
312    /// <para>
313    /// Determines whether is generic.
314    /// </para>
315    /// <para></para>
316    /// </summary>
317    /// <param name="type">
318    /// <para>The type.</para>
319    /// <para></para>
320    /// </param>
321    /// <param name="genericTypeDefinition">
322    /// <para>The generic type definition.</para>
323    /// <para></para>
324    /// </param>
325    /// <returns>
326    /// <para>The bool</para>
327    /// <para></para>
328    /// </returns>
329    [MethodImpl(MethodImplOptions.AggressiveInlining)]
330    public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
331        ↪ type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
332
333    /// <summary>

```

```

332     /// <para>
333     /// Determines whether is nullable.
334     /// </para>
335     /// <para></para>
336     /// </summary>
337     /// <param name="type">
338     /// <para>The type.</para>
339     /// <para></para>
340     /// </param>
341     /// <returns>
342     /// <para>The bool</para>
343     /// <para></para>
344     /// </returns>
345     [MethodImpl(MethodImplOptions.AggressiveInlining)]
346     public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
347
348     /// <summary>
349     /// <para>
350     /// Gets the unsigned version or null using the specified signed type.
351     /// </para>
352     /// <para></para>
353     /// </summary>
354     /// <param name="signedType">
355     /// <para>The signed type.</para>
356     /// <para></para>
357     /// </param>
358     /// <returns>
359     /// <para>The type</para>
360     /// <para></para>
361     /// </returns>
362     [MethodImpl(MethodImplOptions.AggressiveInlining)]
363     public static Type GetUnsignedVersionOrNull(this Type signedType) =>
364         ↪ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
365
366     /// <summary>
367     /// <para>
368     /// Gets the signed version or null using the specified unsigned type.
369     /// </para>
370     /// <para></para>
371     /// </summary>
372     /// <param name="unsignedType">
373     /// <para>The unsigned type.</para>
374     /// <para></para>
375     /// </param>
376     /// <returns>
377     /// <para>The type</para>
378     /// <para></para>
379     /// </returns>
380     [MethodImpl(MethodImplOptions.AggressiveInlining)]
381     public static Type GetSignedVersionOrNull(this Type unsignedType) =>
382         ↪ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
383
384     /// <summary>
385     /// <para>
386     /// Determines whether can be numeric.
387     /// </para>
388     /// <para></para>
389     /// </summary>
390     /// <param name="type">
391     /// <para>The type.</para>
392     /// <para></para>
393     /// </param>
394     /// <returns>
395     /// <para>The bool</para>
396     /// <para></para>
397     /// </returns>
398     [MethodImpl(MethodImplOptions.AggressiveInlining)]
399     public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
400
401     /// <summary>
402     /// <para>
403     /// Determines whether is numeric.
404     /// </para>
405     /// <para></para>
406     /// </summary>
407     /// <param name="type">
408     /// <para>The type.</para>
409     /// <para></para>
410     /// </param>
411     /// <returns>
412     /// <para>The bool</para>
413     /// <para></para>
414     /// </returns>

```

```

408     /// </param>
409     /// <returns>
410     /// <para>The bool</para>
411     /// <para></para>
412     /// </returns>
413     [MethodImpl(MethodImplOptions.AggressiveInlining)]
414     public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
415
416     /// <summary>
417     /// <para>
418     /// Determines whether is signed.
419     /// </para>
420     /// <para></para>
421     /// </summary>
422     /// <param name="type">
423     /// <para>The type.</para>
424     /// <para></para>
425     /// </param>
426     /// <returns>
427     /// <para>The bool</para>
428     /// <para></para>
429     /// </returns>
430     [MethodImpl(MethodImplOptions.AggressiveInlining)]
431     public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
432
433     /// <summary>
434     /// <para>
435     /// Determines whether is float point.
436     /// </para>
437     /// <para></para>
438     /// </summary>
439     /// <param name="type">
440     /// <para>The type.</para>
441     /// <para></para>
442     /// </param>
443     /// <returns>
444     /// <para>The bool</para>
445     /// <para></para>
446     /// </returns>
447     [MethodImpl(MethodImplOptions.AggressiveInlining)]
448     public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
449
450     /// <summary>
451     /// <para>
452     /// Gets the delegate return type using the specified delegate type.
453     /// </para>
454     /// <para></para>
455     /// </summary>
456     /// <param name="delegateType">
457     /// <para>The delegate type.</para>
458     /// <para></para>
459     /// </param>
460     /// <returns>
461     /// <para>The type</para>
462     /// <para></para>
463     /// </returns>
464     [MethodImpl(MethodImplOptions.AggressiveInlining)]
465     public static Type GetDelegateReturnType(this Type delegateType) =>
466         ↪ delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;
467
468     /// <summary>
469     /// <para>
470     /// Gets the delegate parameter types using the specified delegate type.
471     /// </para>
472     /// <para></para>
473     /// </summary>
474     /// <param name="delegateType">
475     /// <para>The delegate type.</para>
476     /// <para></para>
477     /// </param>
478     /// <returns>
479     /// <para>The type array</para>
480     /// <para></para>
481     /// </returns>
482     [MethodImpl(MethodImplOptions.AggressiveInlining)]
483     public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
484         ↪ delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();

```

```

484     /// <summary>
485     /// <para>
486     /// Gets the delegate characteristics using the specified delegate type.
487     /// </para>
488     /// <para></para>
489     /// </summary>
490     /// <param name="delegateType">
491     /// <para>The delegate type.</para>
492     /// <para></para>
493     /// </param>
494     /// <param name="returnType">
495     /// <para>The return type.</para>
496     /// <para></para>
497     /// </param>
498     /// <param name="parameterTypes">
499     /// <para>The parameter types.</para>
500     /// <para></para>
501     /// </param>
502     [MethodImpl(MethodImplOptions.AggressiveInlining)]
503     public static void GetDelegateCharacteristics(this Type delegateType, out Type
    ↪ returnType, out Type[] parameterTypes)
504     {
505         var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
506         returnType = invoke.ReturnType;
507         parameterTypes = invoke.GetParameterTypes();
508     }
509 }
510 }

```

### 1.13 ./csharp/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     /// <summary>
13     /// <para>
14     /// Represents the types.
15     /// </para>
16     /// <para></para>
17     /// </summary>
18     public abstract class Types
19     {
20         /// <summary>
21         /// <para>
22         /// Gets the collection value.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         public static ReadOnlyCollection<Type> Collection { get; } = new
    ↪ ReadOnlyCollection<Type>(System.Array.Empty<Type>());
27         /// <summary>
28         /// <para>
29         /// Gets the array value.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         public static Type[] Array => Collection.ToArray();
34
35         /// <summary>
36         /// <para>
37         /// Returns the read only collection.
38         /// </para>
39         /// <para></para>
40         /// </summary>
41         /// <returns>
42         /// <para>A read only collection of type</para>
43         /// <para></para>
44         /// </returns>
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected ReadOnlyCollection<Type> ToReadOnlyCollection()
47         {

```

```

48     var types = GetType().GetGenericArguments();
49     var result = new List<Type>();
50     AppendTypes(result, types);
51     return new ReadOnlyCollection<Type>(result);
52 }
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 private static void AppendTypes(List<Type> container, IList<Type> types)
55 {
56     for (var i = 0; i < types.Count; i++)
57     {
58         var element = types[i];
59         if (element != typeof(Types))
60         {
61             if (element.IsSubclassOf(typeof(Types)))
62             {
63                 AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<Type>>(nameof(Types.Collection)));
64             }
65             else
66             {
67                 container.Add(element);
68             }
69         }
70     }
71 }
72 }
73 }

```

#### 1.14 ./csharp/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     /// <summary>
11     /// <para>
12     /// Represents the types.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="Types"/>
17     public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
18     {
19         /// <summary>
20         /// <para>
21         /// Gets the collection value.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
26             ↪ T4, T5, T6, T7>().ToReadOnlyCollection();
27         /// <summary>
28         /// <para>
29         /// Gets the array value.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         public new static Type[] Array => Collection.ToArray();
34         private Types() { }
35     }
36 }

```

#### 1.15 ./csharp/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     /// <summary>
11     /// <para>

```

```

12     /// Represents the types.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="Types"/>
17     public class Types<T1, T2, T3, T4, T5, T6> : Types
18     {
19         /// <summary>
20         /// <para>
21         /// Gets the collection value.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
26             ↪ T4, T5, T6>().ToReadOnlyCollection();
27         /// <summary>
28         /// <para>
29         /// Gets the array value.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         public new static Type[] Array => Collection.ToArray();
34         private Types() { }
35     }

```

#### 1.16 ./csharp/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     /// <summary>
11     /// <para>
12     /// Represents the types.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="Types"/>
17     public class Types<T1, T2, T3, T4, T5> : Types
18     {
19         /// <summary>
20         /// <para>
21         /// Gets the collection value.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
26             ↪ T4, T5>().ToReadOnlyCollection();
27         /// <summary>
28         /// <para>
29         /// Gets the array value.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         public new static Type[] Array => Collection.ToArray();
34         private Types() { }
35     }

```

#### 1.17 ./csharp/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     /// <summary>
11     /// <para>
12     /// Represents the types.
13     /// </para>

```

```

14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="Types"/>
17     public class Types<T1, T2, T3, T4> : Types
18     {
19         /// <summary>
20         /// <para>
21         /// Gets the collection value.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
26             ↪ T4>().ToReadOnlyCollection();
27         /// <summary>
28         /// <para>
29         /// Gets the array value.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         public new static Type[] Array => Collection.ToArray();
34         private Types() { }
35     }

```

### 1.18 ./csharp/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     /// <summary>
11     /// <para>
12     /// Represents the types.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="Types"/>
17     public class Types<T1, T2, T3> : Types
18     {
19         /// <summary>
20         /// <para>
21         /// Gets the collection value.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2,
26             ↪ T3>().ToReadOnlyCollection();
27         /// <summary>
28         /// <para>
29         /// Gets the array value.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         public new static Type[] Array => Collection.ToArray();
34         private Types() { }
35     }

```

### 1.19 ./csharp/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     /// <summary>
11     /// <para>
12     /// Represents the types.
13     /// </para>
14     /// <para></para>
15     /// </summary>

```

```

16     /// <seealso cref="Types"/>
17     public class Types<T1, T2> : Types
18     {
19         /// <summary>
20         /// <para>
21         /// Gets the collection value.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1,
            ↪ T2>().ToReadOnlyCollection();
26         /// <summary>
27         /// <para>
28         /// Gets the array value.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         public new static Type[] Array => Collection.ToArray();
33         private Types() { }
34     }
35 }

```

## 1.20 ./csharp/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     /// <summary>
11     /// <para>
12     /// Represents the types.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="Types"/>
17     public class Types<T> : Types
18     {
19         /// <summary>
20         /// <para>
21         /// Gets the collection value.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         public new static ReadOnlyCollection<Type> Collection { get; } = new
            ↪ Types<T>().ToReadOnlyCollection();
26         /// <summary>
27         /// <para>
28         /// Gets the array value.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         public new static Type[] Array => Collection.ToArray();
33         private Types() { }
34     }
35 }

```

## 1.21 ./csharp/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 private void CheckConstantLoading<T>(T value)
37 {
38     var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
39     {
40         generator.LoadConstant(value);
41         generator.Return();
42     });
43     Assert.Equal(value, compiledFunction());
44 }
45
46 [Fact]
47 public void UnsignedIntegersConversionWithSignExtensionTest()
48 {
49     object[] withSignExtension = new object[]
50     {
51         CompileUncheckedConverter<byte, sbyte>(extendSign: true)(128),
52         CompileUncheckedConverter<byte, short>(extendSign: true)(128),
53         CompileUncheckedConverter<ushort, short>(extendSign: true)(32768),
54         CompileUncheckedConverter<byte, int>(extendSign: true)(128),
55         CompileUncheckedConverter<ushort, int>(extendSign: true)(32768),
56         CompileUncheckedConverter<uint, int>(extendSign: true)(2147483648),
57         CompileUncheckedConverter<byte, long>(extendSign: true)(128),
58         CompileUncheckedConverter<ushort, long>(extendSign: true)(32768),
59         CompileUncheckedConverter<uint, long>(extendSign: true)(2147483648),
60         CompileUncheckedConverter<ulong, long>(extendSign: true)(9223372036854775808)
61     };
62     object[] withoutSignExtension = new object[]
63     {
64         CompileUncheckedConverter<byte, sbyte>(extendSign: false)(128),
65         CompileUncheckedConverter<byte, short>(extendSign: false)(128),
66         CompileUncheckedConverter<ushort, short>(extendSign: false)(32768),
67         CompileUncheckedConverter<byte, int>(extendSign: false)(128),
68         CompileUncheckedConverter<ushort, int>(extendSign: false)(32768),
69         CompileUncheckedConverter<uint, int>(extendSign: false)(2147483648),
70         CompileUncheckedConverter<byte, long>(extendSign: false)(128),
71         CompileUncheckedConverter<ushort, long>(extendSign: false)(32768),
72         CompileUncheckedConverter<uint, long>(extendSign: false)(2147483648),
73         CompileUncheckedConverter<ulong, long>(extendSign: false)(9223372036854775808)
74     };
75     var i = 0;
76     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
77     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
78     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
79     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
80     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
81     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
82     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
83     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
84     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
85     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
86 }
87
88 [Fact]
89 public void SignedIntegersConversionOfMinusOneWithSignExtensionTest()
90 {
91     object[] withSignExtension = new object[]
92     {
93         CompileUncheckedConverter<sbyte, byte>(extendSign: true)(-1),
94         CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(-1),
95         CompileUncheckedConverter<short, ushort>(extendSign: true)(-1),

```

```

96     CompileUncheckedConverter<sbyte, uint>(extendSign: true)(-1),
97     CompileUncheckedConverter<short, uint>(extendSign: true)(-1),
98     CompileUncheckedConverter<int, uint>(extendSign: true)(-1),
99     CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(-1),
100    CompileUncheckedConverter<short, ulong>(extendSign: true)(-1),
101    CompileUncheckedConverter<int, ulong>(extendSign: true)(-1),
102    CompileUncheckedConverter<long, ulong>(extendSign: true)(-1)
103 };
104 object[] withoutSignExtension = new object[]
105 {
106     CompileUncheckedConverter<sbyte, byte>(extendSign: false)(-1),
107     CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(-1),
108     CompileUncheckedConverter<short, ushort>(extendSign: false)(-1),
109     CompileUncheckedConverter<sbyte, uint>(extendSign: false)(-1),
110     CompileUncheckedConverter<short, uint>(extendSign: false)(-1),
111     CompileUncheckedConverter<int, uint>(extendSign: false)(-1),
112     CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(-1),
113     CompileUncheckedConverter<short, ulong>(extendSign: false)(-1),
114     CompileUncheckedConverter<int, ulong>(extendSign: false)(-1),
115     CompileUncheckedConverter<long, ulong>(extendSign: false)(-1)
116 };
117 var i = 0;
118 Assert.Equal((byte)255, (byte)withSignExtension[i]);
119 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
120 Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
121 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
122 Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
123 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
124 Assert.Equal(4294967295, withSignExtension[i]);
125 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
126 Assert.Equal(4294967295, withSignExtension[i]);
127 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
128 Assert.Equal(4294967295, withSignExtension[i]);
129 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
130 Assert.Equal(18446744073709551615, withSignExtension[i]);
131 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
132 Assert.Equal(18446744073709551615, withSignExtension[i]);
133 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
134 Assert.Equal(18446744073709551615, withSignExtension[i]);
135 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
136 Assert.Equal(18446744073709551615, withSignExtension[i]);
137 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
138 }

```

[Fact]

```

141 public void SignedIntegersConversionOfTwoWithSignExtensionTest()

```

```

142 {
143     object[] withSignExtension = new object[]
144     {
145         CompileUncheckedConverter<sbyte, byte>(extendSign: true)(2),
146         CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(2),
147         CompileUncheckedConverter<short, ushort>(extendSign: true)(2),
148         CompileUncheckedConverter<sbyte, uint>(extendSign: true)(2),
149         CompileUncheckedConverter<short, uint>(extendSign: true)(2),
150         CompileUncheckedConverter<int, uint>(extendSign: true)(2),
151         CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(2),
152         CompileUncheckedConverter<short, ulong>(extendSign: true)(2),
153         CompileUncheckedConverter<int, ulong>(extendSign: true)(2),
154         CompileUncheckedConverter<long, ulong>(extendSign: true)(2)
155     };
156     object[] withoutSignExtension = new object[]
157     {
158         CompileUncheckedConverter<sbyte, byte>(extendSign: false)(2),
159         CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(2),
160         CompileUncheckedConverter<short, ushort>(extendSign: false)(2),
161         CompileUncheckedConverter<sbyte, uint>(extendSign: false)(2),
162         CompileUncheckedConverter<short, uint>(extendSign: false)(2),
163         CompileUncheckedConverter<int, uint>(extendSign: false)(2),
164         CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(2),
165         CompileUncheckedConverter<short, ulong>(extendSign: false)(2),
166         CompileUncheckedConverter<int, ulong>(extendSign: false)(2),
167         CompileUncheckedConverter<long, ulong>(extendSign: false)(2)
168     };
169     for (var i = 0; i < withSignExtension.Length; i++)
170     {
171         Assert.Equal(2UL, Convert.ToUInt64(withSignExtension[i]));
172         Assert.Equal(withSignExtension[i], withoutSignExtension[i]);
173     }

```

```

174     }
175     private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
    ↪ TTarget>(bool extendSign)
176     {
177         return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
178         {
179             generator.LoadArgument(0);
180             generator.UncheckedConvert<TSource, TTarget>(extendSign);
181             generator.Return();
182         });
183     }
184 }
185 }

```

## 1.22 ./csharp/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 ./csharp/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

- ./csharp/Platform.Reflection.Tests/CodeGenerationTests.cs, 56
- ./csharp/Platform.Reflection.Tests/GetILBytesMethodTests.cs, 59
- ./csharp/Platform.Reflection.Tests/NumericTypeTests.cs, 59
- ./csharp/Platform.Reflection/AssemblyExtensions.cs, 1
- ./csharp/Platform.Reflection/DelegateHelpers.cs, 1
- ./csharp/Platform.Reflection/DynamicExtensions.cs, 4
- ./csharp/Platform.Reflection/EnsureExtensions.cs, 4
- ./csharp/Platform.Reflection/FieldInfoExtensions.cs, 14
- ./csharp/Platform.Reflection/ILGeneratorExtensions.cs, 14
- ./csharp/Platform.Reflection/MethodInfoExtensions.cs, 38
- ./csharp/Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 39
- ./csharp/Platform.Reflection/NumericType.cs, 40
- ./csharp/Platform.Reflection/PropertyInfoExtensions.cs, 42
- ./csharp/Platform.Reflection/TypeBuilderExtensions.cs, 43
- ./csharp/Platform.Reflection/TypeExtensions.cs, 45
- ./csharp/Platform.Reflection/Types.cs, 52
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 53
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 53
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 54
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4].cs, 54
- ./csharp/Platform.Reflection/Types[T1, T2, T3].cs, 55
- ./csharp/Platform.Reflection/Types[T1, T2].cs, 55
- ./csharp/Platform.Reflection/Types[T].cs, 56