

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         /// <summary>
21         /// <para>
22         /// The type.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
27             ConcurrentDictionary<Assembly, Type[]>();
28
29         /// <remarks>
30         /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
31         /// </remarks>
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static Type[] GetLoadableTypes(this Assembly assembly)
34         {
35             Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
36             try
37             {
38                 return assembly.GetTypes();
39             }
40             catch (ReflectionTypeLoadException e)
41             {
42                 return e.Types.ToArray(t => t != null);
43             }
44         }
45
46         /// <summary>
47         /// <para>
48         /// Gets the cached loadable types using the specified assembly.
49         /// </para>
50         /// <para></para>
51         /// </summary>
52         /// <param name="assembly">
53         /// <para>The assembly.</para>
54         /// <para></para>
55         /// </param>
56         /// <returns>
57         /// <para>The type array</para>
58         /// <para></para>
59         /// </returns>
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
62             _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
63     }
64 }
```

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
    ↪ typeMemberMethod)
44      where TDelegate : Delegate
45      {
46          var @delegate = default(TDelegate);
47          try
48          {
49              @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
    ↪ CompileDynamicMethod<TDelegate>(emitCode);
50          }
51          catch (Exception exception)
52          {
53              exception.Ignore();
54          }
55          return @delegate;
56      }
57
58      /// <summary>
59      /// <para>
60      /// Compiles the or default using the specified emit code.
61      /// </para>
62      /// <para></para>
63      /// </summary>
64      /// <typeparam name="TDelegate">
65      /// <para>The delegate.</para>
66      /// <para></para>
67      /// </typeparam>
68      /// <param name="emitCode">
69      /// <para>The emit code.</para>
70      /// <para></para>
71      /// </param>
72      /// <returns>
73      /// <para>The delegate</para>
74      /// <para></para>
75      /// </returns>
76      [MethodImpl(MethodImplOptions.AggressiveInlining)]
77      public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
    ↪ TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);
78
79      /// <summary>
80      /// <para>
81      /// Compiles the emit code.
82      /// </para>
83      /// <para></para>
84      /// </summary>
85      /// <typeparam name="TDelegate">
86      /// <para>The delegate.</para>

```

```

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

```

```

162     /// <summary>
163     /// <para>
164     /// Compiles the type member method using the specified emit code.
165     /// </para>
166     /// <para></para>
167     /// </summary>
168     /// <typeparam name="TDelegate">
169     /// <para>The delegate.</para>
170     /// <para></para>
171     /// </typeparam>
172     /// <param name="emitCode">
173     /// <para>The emit code.</para>
174     /// <para></para>
175     /// </param>
176     /// <returns>
177     /// <para>The delegate</para>
178     /// <para></para>
179     /// </returns>
180     [MethodImpl(MethodImplOptions.AggressiveInlining)]
181     public static TDelegate CompileTypeMemberMethod<TDelegate>(Action<ILGenerator> emitCode)
182     {
183         AssemblyName assemblyName = new AssemblyName(GetNewName());
184         var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
185             ↪ AssemblyBuilderAccess.Run);
186         var module = assembly.DefineDynamicModule(GetNewName());
187         var type = module.DefineType(GetNewName());
188         var methodName = GetNewName();
189         type.EmitStaticMethod<TDelegate>(methodName, emitCode);
190         var typeInfo = type.CreateTypeInfo();
191         return (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(typeof(TDele
192             ↪ gate));
193     }
194     /// <summary>
195     /// <para>
196     /// Gets the new name.
197     /// </para>
198     /// <para></para>
199     /// </summary>
200     /// <returns>
201     /// <para>The string</para>
202     /// <para></para>
203     /// </returns>
204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
205     private static string GetNewName() => Guid.NewGuid().ToString("N");
206 }

```

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>

```

```

59     /// <typeparam name="T">
60     /// <para>The .</para>
61     /// <para></para>
62     /// </typeparam>
63     /// <param name="root">
64     /// <para>The root.</para>
65     /// <para></para>
66     /// </param>
67     /// <param name="message">
68     /// <para>The message.</para>
69     /// <para></para>
70     /// </param>
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
    ↪ message)
73     {
74         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) =>
    ↪ IsUnsignedInteger<T>(root, (string)null);
94
95     /// <summary>
96     /// <para>
97     /// Uses the signed integer using the specified root.
98     /// </para>
99     /// <para></para>
100    /// </summary>
101    /// <typeparam name="T">
102    /// <para>The .</para>
103    /// <para></para>
104    /// </typeparam>
105    /// <param name="root">
106    /// <para>The root.</para>
107    /// <para></para>
108    /// </param>
109    /// <param name="messageBuilder">
110    /// <para>The message builder.</para>
111    /// <para></para>
112    /// </param>
113    /// <exception cref="NotSupportedException">
114    /// <para></para>
115    /// <para></para>
116    /// </exception>
117    [MethodImpl(MethodImplOptions.AggressiveInlining)]
118    public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
    ↪ messageBuilder)
119    {
120        if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
    ↪ NumericType<T>.IsFloatPoint)
121        {
122            throw new NotSupportedException(messageBuilder());
123        }
124    }
125
126    /// <summary>
127    /// <para>
128    /// Uses the signed integer using the specified root.
129    /// </para>
130    /// <para></para>
131    /// </summary>

```

```

132     /// <typeparam name="T">
133     /// <para>The .</para>
134     /// <para></para>
135     /// </typeparam>
136     /// <param name="root">
137     /// <para>The root.</para>
138     /// <para></para>
139     /// </param>
140     /// <param name="message">
141     /// <para>The message.</para>
142     /// <para></para>
143     /// </param>
144     [MethodImpl(MethodImplOptions.AggressiveInlining)]
145     public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
        ↳ message)
146     {
147         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) =>
        ↳ IsSignedInteger<T>(root, (string)null);
167
168     /// <summary>
169     /// <para>
170     /// Uses the signed using the specified root.
171     /// </para>
172     /// <para></para>
173     /// </summary>
174     /// <typeparam name="T">
175     /// <para>The .</para>
176     /// <para></para>
177     /// </typeparam>
178     /// <param name="root">
179     /// <para>The root.</para>
180     /// <para></para>
181     /// </param>
182     /// <param name="messageBuilder">
183     /// <para>The message builder.</para>
184     /// <para></para>
185     /// </param>
186     /// <exception cref="NotSupportedException">
187     /// <para></para>
188     /// <para></para>
189     /// </exception>
190     [MethodImpl(MethodImplOptions.AggressiveInlining)]
191     public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
        ↳ messageBuilder)
192     {
193         if (!NumericType<T>.IsSigned)
194         {
195             throw new NotSupportedException(messageBuilder());
196         }
197     }
198
199     /// <summary>
200     /// <para>
201     /// Uses the signed using the specified root.
202     /// </para>
203     /// <para></para>
204     /// </summary>
205     /// <typeparam name="T">
206     /// <para>The .</para>

```

```

207 /// <para></para>
208 /// </typeparam>
209 /// <param name="root">
210 /// <para>The root.</para>
211 /// <para></para>
212 /// </param>
213 /// <param name="message">
214 /// <para>The message.</para>
215 /// <para></para>
216 /// </param>
217 [MethodImpl(MethodImplOptions.AggressiveInlining)]
218 public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
219 {
220     string messageBuilder() => message;
221     IsSigned<T>(root, messageBuilder);
222 }
223
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 /// Uses 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 /// <para></para>
336 /// </exception>
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
339     ↪ messageBuilder)
340 {
341     if (!NumericType<T>.CanBeNumeric)
342     {
343         throw new NotSupportedException(messageBuilder());
344     }
345 }
346
347 /// <summary>
348 /// <para>
349 /// Cans the be numeric using the specified root.
350 /// </para>
351 /// <para></para>
352 /// </summary>
353 /// <typeparam name="T">
354 /// <para>The .</para>
355 /// <para></para>
356 /// </typeparam>
357 /// <param name="root">
358 /// <para>The root.</para>
359 /// <para></para>
360 /// </param>

```

```

359 /// <param name="message">
360 /// <para>The message.</para>
361 /// <para></para>
362 /// </param>
363 [MethodImpl(MethodImplOptions.AggressiveInlining)]
364 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
365 {
366     string messageBuilder() => message;
367     CanBeNumeric<T>(root, messageBuilder);
368 }
369
370 /// <summary>
371 /// <para>
372 /// Cans the be numeric using the specified root.
373 /// </para>
374 /// <para></para>
375 /// </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 #region OnDebug
390
391 /// <summary>
392 /// <para>
393 /// Ises the unsigned integer using the specified root.
394 /// </para>
395 /// <para></para>
396 /// </summary>
397 /// <typeparam name="T">
398 /// <para>The .</para>
399 /// <para></para>
400 /// </typeparam>
401 /// <param name="root">
402 /// <para>The root.</para>
403 /// <para></para>
404 /// </param>
405 /// <param name="messageBuilder">
406 /// <para>The message builder.</para>
407 /// <para></para>
408 /// </param>
409 [Conditional("DEBUG")]
410 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
411     ↪ Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
412
413 /// <summary>
414 /// <para>
415 /// Ises the unsigned integer using the specified root.
416 /// </para>
417 /// <para></para>
418 /// </summary>
419 /// <typeparam name="T">
420 /// <para>The .</para>
421 /// <para></para>
422 /// </typeparam>
423 /// <param name="root">
424 /// <para>The root.</para>
425 /// <para></para>
426 /// </param>
427 /// <param name="message">
428 /// <para>The message.</para>
429 /// <para></para>
430 /// </param>
431 [Conditional("DEBUG")]
432 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
433     ↪ message) => Ensure.Always.IsUnsignedInteger<T>(message);
434
435 /// <summary>

```

```

434     /// <para>
435     /// Uses the unsigned integer using the specified root.
436     /// </para>
437     /// <para></para>
438     /// </summary>
439     /// <typeparam name="T">
440     /// <para>The .</para>
441     /// <para></para>
442     /// </typeparam>
443     /// <param name="root">
444     /// <para>The root.</para>
445     /// <para></para>
446     /// </param>
447     [Conditional("DEBUG")]
448     public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
449         ↪ Ensure.Always.IsUnsignedInteger<T>();
450
451     /// <summary>
452     /// <para>
453     /// Uses the signed integer using the specified root.
454     /// </para>
455     /// <para></para>
456     /// </summary>
457     /// <typeparam name="T">
458     /// <para>The .</para>
459     /// <para></para>
460     /// </typeparam>
461     /// <param name="root">
462     /// <para>The root.</para>
463     /// <para></para>
464     /// </param>
465     /// <param name="messageBuilder">
466     /// <para>The message builder.</para>
467     /// <para></para>
468     /// </param>
469     [Conditional("DEBUG")]
470     public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
471         ↪ messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
472
473     /// <summary>
474     /// <para>
475     /// Uses the signed integer using the specified root.
476     /// </para>
477     /// <para></para>
478     /// </summary>
479     /// <typeparam name="T">
480     /// <para>The .</para>
481     /// <para></para>
482     /// </typeparam>
483     /// <param name="root">
484     /// <para>The root.</para>
485     /// <para></para>
486     /// </param>
487     /// <param name="message">
488     /// <para>The message.</para>
489     /// <para></para>
490     /// </param>
491     [Conditional("DEBUG")]
492     public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
493         ↪ message) => Ensure.Always.IsSignedInteger<T>(message);
494
495     /// <summary>
496     /// <para>
497     /// Uses the signed integer using the specified root.
498     /// </para>
499     /// <para></para>
500     /// </summary>
501     /// <typeparam name="T">
502     /// <para>The .</para>
503     /// <para></para>
504     /// </typeparam>
505     /// <param name="root">
506     /// <para>The root.</para>
507     /// <para></para>
508     /// </param>
509     [Conditional("DEBUG")]
510     public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
511         ↪ Ensure.Always.IsSignedInteger<T>();

```

```

508     /// <summary>
509     /// <para>
510     /// Uses the signed using the specified root.
511     /// </para>
512     /// <para></para>
513     /// </summary>
514     /// <typeparam name="T">
515     /// <para>The .</para>
516     /// <para></para>
517     /// </typeparam>
518     /// <param name="root">
519     /// <para>The root.</para>
520     /// <para></para>
521     /// </param>
522     /// <param name="messageBuilder">
523     /// <para>The message builder.</para>
524     /// <para></para>
525     /// </param>
526     [Conditional("DEBUG")]
527     public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
528     ↪ messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
529
530     /// <summary>
531     /// <para>
532     /// Uses 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) =>
550     ↪ Ensure.Always.IsSigned<T>(message);
551
552     /// <summary>
553     /// <para>
554     /// Uses the signed using the specified root.
555     /// </para>
556     /// <para></para>
557     /// </summary>
558     /// <typeparam name="T">
559     /// <para>The .</para>
560     /// <para></para>
561     /// </typeparam>
562     /// <param name="root">
563     /// <para>The root.</para>
564     /// <para></para>
565     /// </param>
566     [Conditional("DEBUG")]
567     public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
568     ↪ Ensure.Always.IsSigned<T>();
569
570     /// <summary>
571     /// <para>
572     /// Uses the numeric using the specified root.
573     /// </para>
574     /// <para></para>
575     /// </summary>
576     /// <typeparam name="T">
577     /// <para>The .</para>
578     /// <para></para>
579     /// </typeparam>
580     /// <param name="root">
581     /// <para>The root.</para>
582     /// <para></para>
583     /// </param>
584     /// <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 /// Ises 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>

```

```

79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator, bool
    ↪ extendSign)
81 {
82     var sourceType = typeof(TSource);
83     var targetType = typeof(TTarget);
84     if (sourceType == targetType)
85     {
86         return;
87     }
88     if (extendSign)
89     {
90         if (sourceType == typeof(byte))
91         {
92             generator.Emit(OpCodes.Conv_I1);
93         }
94         if (sourceType == typeof(ushort) || sourceType == typeof(char))
95         {
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
128 /// <summary>
129 /// <para>
130 /// Converts the to boolean using the specified generator.
131 /// </para>
132 /// <para></para>
133 /// </summary>
134 /// <typeparam name="TSource">
135 /// <para>The source.</para>
136 /// <para></para>
137 /// </typeparam>
138 /// <param name="generator">
139 /// <para>The generator.</para>
140 /// <para></para>
141 /// </param>
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 private static void ConvertToBoolean<TSource>(this ILGenerator generator)
144 {
145     generator.LoadConstant<TSource>(default);
146     var sourceType = typeof(TSource);
147     if (sourceType == typeof(float) || sourceType == typeof(double))
148     {
149         generator.Emit(OpCodes.Ceq);
150         // Inversion of the first Ceq instruction
151         generator.LoadConstant<int>(0);
152         generator.Emit(OpCodes.Ceq);
153     }
154     else
155     {

```



```

156         generator.Emit(OpCodes.Cgt_Un);
157     }
158 }
159
160 /// <summary>
161 /// <para>
162 /// Converts the to integer using the specified generator.
163 /// </para>
164 /// <para></para>
165 /// </summary>
166 /// <typeparam name="TSource">
167 /// <para>The source.</para>
168 /// <para></para>
169 /// </typeparam>
170 /// <param name="generator">
171 /// <para>The generator.</para>
172 /// <para></para>
173 /// </param>
174 /// <param name="targetType">
175 /// <para>The target type.</para>
176 /// <para></para>
177 /// </param>
178 /// <param name="extendSign">
179 /// <para>The extend sign.</para>
180 /// <para></para>
181 /// </param>
182 [MethodImpl(MethodImplOptions.AggressiveInlining)]
183 private static void ConvertToInteger<TSource>(this ILGenerator generator, Type
    ↪ targetType, bool extendSign)
184 {
185     if (targetType == typeof(sbyte))
186     {
187         generator.Emit(OpCodes.Conv_I1);
188     }
189     else if (targetType == typeof(byte))
190     {
191         generator.Emit(OpCodes.Conv_U1);
192     }
193     else if (targetType == typeof(short))
194     {
195         generator.Emit(OpCodes.Conv_I2);
196     }
197     else if (targetType == typeof(ushort) || targetType == typeof(char))
198     {
199         var sourceType = typeof(TSource);
200         if (sourceType != typeof(ushort) && sourceType != typeof(char))
201         {
202             generator.Emit(OpCodes.Conv_U2);
203         }
204     }
205     else if (targetType == typeof(int))
206     {
207         generator.Emit(OpCodes.Conv_I4);
208     }
209     else if (targetType == typeof(uint))
210     {
211         generator.Emit(OpCodes.Conv_U4);
212     }
213     else if (targetType == typeof(long) || targetType == typeof(ulong))
214     {
215         if (NumericType<TSource>.IsSigned || extendSign)
216         {
217             generator.Emit(OpCodes.Conv_I8);
218         }
219         else
220         {
221             generator.Emit(OpCodes.Conv_U8);
222         }
223     }
224 }
225
226 /// <summary>
227 /// <para>
228 /// Checkeds the convert using the specified generator.
229 /// </para>
230 /// <para></para>
231 /// </summary>
232 /// <typeparam name="TSource">

```

```

233 /// <para>The source.</para>
234 /// <para></para>
235 /// </typeparam>
236 /// <typeparam name="TTarget">
237 /// <para>The target.</para>
238 /// <para></para>
239 /// </typeparam>
240 /// <param name="generator">
241 /// <para>The generator.</para>
242 /// <para></para>
243 /// </param>
244 /// <exception cref="NotSupportedException">
245 /// <para></para>
246 /// <para></para>
247 /// </exception>
248 [MethodImpl(MethodImplOptions.AggressiveInlining)]
249 public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
250 {
251     var sourceType = typeof(TSource);
252     var targetType = typeof(TTarget);
253     if (sourceType == targetType)
254     {
255         return;
256     }
257     if (targetType == typeof(short))
258     {
259         if (NumericType<TSource>.IsSigned)
260         {
261             generator.Emit(OpCodes.Conv_Ovf_I2);
262         }
263         else
264         {
265             generator.Emit(OpCodes.Conv_Ovf_I2_Un);
266         }
267     }
268     else if (targetType == typeof(ushort) || targetType == typeof(char))
269     {
270         if (sourceType != typeof(ushort) && sourceType != typeof(char))
271         {
272             if (NumericType<TSource>.IsSigned)
273             {
274                 generator.Emit(OpCodes.Conv_Ovf_U2);
275             }
276             else
277             {
278                 generator.Emit(OpCodes.Conv_Ovf_U2_Un);
279             }
280         }
281     }
282     else if (targetType == typeof(sbyte))
283     {
284         if (NumericType<TSource>.IsSigned)
285         {
286             generator.Emit(OpCodes.Conv_Ovf_I1);
287         }
288         else
289         {
290             generator.Emit(OpCodes.Conv_Ovf_I1_Un);
291         }
292     }
293     else if (targetType == typeof(byte))
294     {
295         if (NumericType<TSource>.IsSigned)
296         {
297             generator.Emit(OpCodes.Conv_Ovf_U1);
298         }
299         else
300         {
301             generator.Emit(OpCodes.Conv_Ovf_U1_Un);
302         }
303     }
304     else if (targetType == typeof(int))
305     {
306         if (NumericType<TSource>.IsSigned)
307         {
308             generator.Emit(OpCodes.Conv_Ovf_I4);
309         }
310         else

```

```

311         {
312             generator.Emit(OpCodes.Conv_Ovf_I4_Un);
313         }
314     }
315     else if (targetType == typeof(uint))
316     {
317         if (NumericType<TSource>.IsSigned)
318         {
319             generator.Emit(OpCodes.Conv_Ovf_U4);
320         }
321         else
322         {
323             generator.Emit(OpCodes.Conv_Ovf_U4_Un);
324         }
325     }
326     else if (targetType == typeof(long))
327     {
328         if (NumericType<TSource>.IsSigned)
329         {
330             generator.Emit(OpCodes.Conv_Ovf_I8);
331         }
332         else
333         {
334             generator.Emit(OpCodes.Conv_Ovf_I8_Un);
335         }
336     }
337     else if (targetType == typeof(ulong))
338     {
339         if (NumericType<TSource>.IsSigned)
340         {
341             generator.Emit(OpCodes.Conv_Ovf_U8);
342         }
343         else
344         {
345             generator.Emit(OpCodes.Conv_Ovf_U8_Un);
346         }
347     }
348     else if (targetType == typeof(float))
349     {
350         if (NumericType<TSource>.IsSigned)
351         {
352             generator.Emit(OpCodes.Conv_R4);
353         }
354         else
355         {
356             generator.Emit(OpCodes.Conv_R_Un);
357         }
358     }
359     else if (targetType == typeof(double))
360     {
361         generator.Emit(OpCodes.Conv_R8);
362     }
363     else if (targetType == typeof(bool))
364     {
365         generator.ConvertToBoolean<TSource>();
366     }
367     else
368     {
369         throw new NotSupportedException();
370     }
371 }
372
373 /// <summary>
374 /// <para>
375 /// Loads the constant using the specified generator.
376 /// </para>
377 /// <para></para>
378 /// </summary>
379 /// <param name="generator">
380 /// <para>The generator.</para>
381 /// <para></para>
382 /// </param>
383 /// <param name="value">
384 /// <para>The value.</para>
385 /// <para></para>
386 /// </param>
387 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

388 public static void LoadConstant(this ILGenerator generator, bool value) =>
389     ↪ generator.LoadConstant(value ? 1 : 0);
390
391 /// <summary>
392 /// <para>
393 /// Loads the constant using the specified generator.
394 /// </para>
395 /// </summary>
396 /// <param name="generator">
397 /// <para>The generator.</para>
398 /// </param>
399 /// <param name="value">
400 /// <para>The value.</para>
401 /// </param>
402 [MethodImpl(MethodImplOptions.AggressiveInlining)]
403 public static void LoadConstant(this ILGenerator generator, float value) =>
404     ↪ generator.Emit(OpCodes.Ldc_R4, value);
405
406 /// <summary>
407 /// <para>
408 /// Loads the constant using the specified generator.
409 /// </para>
410 /// </summary>
411 /// <param name="generator">
412 /// <para>The generator.</para>
413 /// </param>
414 /// <param name="value">
415 /// <para>The value.</para>
416 /// </param>
417 [MethodImpl(MethodImplOptions.AggressiveInlining)]
418 public static void LoadConstant(this ILGenerator generator, double value) =>
419     ↪ generator.Emit(OpCodes.Ldc_R8, value);
420
421 /// <summary>
422 /// <para>
423 /// Loads the constant using the specified generator.
424 /// </para>
425 /// </summary>
426 /// <param name="generator">
427 /// <para>The generator.</para>
428 /// </param>
429 /// <param name="value">
430 /// <para>The value.</para>
431 /// </param>
432 [MethodImpl(MethodImplOptions.AggressiveInlining)]
433 public static void LoadConstant(this ILGenerator generator, ulong value) =>
434     ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
435
436 /// <summary>
437 /// <para>
438 /// Loads the constant using the specified generator.
439 /// </para>
440 /// </summary>
441 /// <param name="generator">
442 /// <para>The generator.</para>
443 /// </param>
444 /// <param name="value">
445 /// <para>The value.</para>
446 /// </param>
447 [MethodImpl(MethodImplOptions.AggressiveInlining)]
448 public static void LoadConstant(this ILGenerator generator, long value) =>
449     ↪ generator.Emit(OpCodes.Ldc_I8, value);
450
451 /// <summary>
452 /// <para>

```

```

460    /// Loads the constant using the specified generator.
461    /// </para>
462    /// <para></para>
463    /// </summary>
464    /// <param name="generator">
465    /// <para>The generator.</para>
466    /// <para></para>
467    /// </param>
468    /// <param name="value">
469    /// <para>The value.</para>
470    /// <para></para>
471    /// </param>
472    [MethodImpl(MethodImplOptions.AggressiveInlining)]
473    public static void LoadConstant(this ILGenerator generator, uint value)
474    {
475        switch (value)
476        {
477            case uint.MaxValue:
478                generator.Emit(OpCodes.Ldc_I4_M1);
479                return;
480            case 0:
481                generator.Emit(OpCodes.Ldc_I4_0);
482                return;
483            case 1:
484                generator.Emit(OpCodes.Ldc_I4_1);
485                return;
486            case 2:
487                generator.Emit(OpCodes.Ldc_I4_2);
488                return;
489            case 3:
490                generator.Emit(OpCodes.Ldc_I4_3);
491                return;
492            case 4:
493                generator.Emit(OpCodes.Ldc_I4_4);
494                return;
495            case 5:
496                generator.Emit(OpCodes.Ldc_I4_5);
497                return;
498            case 6:
499                generator.Emit(OpCodes.Ldc_I4_6);
500                return;
501            case 7:
502                generator.Emit(OpCodes.Ldc_I4_7);
503                return;
504            case 8:
505                generator.Emit(OpCodes.Ldc_I4_8);
506                return;
507            default:
508                if (value <= sbyte.MaxValue)
509                {
510                    generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
511                }
512                else
513                {
514                    generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
515                }
516                return;
517        }
518    }
519
520    /// <summary>
521    /// <para>
522    /// Loads the constant using the specified generator.
523    /// </para>
524    /// <para></para>
525    /// </summary>
526    /// <param name="generator">
527    /// <para>The generator.</para>
528    /// <para></para>
529    /// </param>
530    /// <param name="value">
531    /// <para>The value.</para>
532    /// <para></para>
533    /// </param>
534    [MethodImpl(MethodImplOptions.AggressiveInlining)]
535    public static void LoadConstant(this ILGenerator generator, int value)
536    {
537        switch (value)
538        {
539            case -1:

```

```

540         generator.Emit(OpCodes.Ldc_I4_M1);
541         return;
542     case 0:
543         generator.Emit(OpCodes.Ldc_I4_0);
544         return;
545     case 1:
546         generator.Emit(OpCodes.Ldc_I4_1);
547         return;
548     case 2:
549         generator.Emit(OpCodes.Ldc_I4_2);
550         return;
551     case 3:
552         generator.Emit(OpCodes.Ldc_I4_3);
553         return;
554     case 4:
555         generator.Emit(OpCodes.Ldc_I4_4);
556         return;
557     case 5:
558         generator.Emit(OpCodes.Ldc_I4_5);
559         return;
560     case 6:
561         generator.Emit(OpCodes.Ldc_I4_6);
562         return;
563     case 7:
564         generator.Emit(OpCodes.Ldc_I4_7);
565         return;
566     case 8:
567         generator.Emit(OpCodes.Ldc_I4_8);
568         return;
569     default:
570         if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
571         {
572             generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
573         }
574         else
575         {
576             generator.Emit(OpCodes.Ldc_I4, value);
577         }
578         return;
579     }
580 }
581
582 /// <summary>
583 /// <para>
584 /// Loads the constant using the specified generator.
585 /// </para>
586 /// <para></para>
587 /// </summary>
588 /// <param name="generator">
589 /// <para>The generator.</para>
590 /// <para></para>
591 /// </param>
592 /// <param name="value">
593 /// <para>The value.</para>
594 /// <para></para>
595 /// </param>
596 [MethodImpl(MethodImplOptions.AggressiveInlining)]
597 public static void LoadConstant(this ILGenerator generator, short value) =>
598     ↪ generator.LoadConstant((int)value);
599
600 /// <summary>
601 /// <para>
602 /// Loads the constant using the specified generator.
603 /// </para>
604 /// <para></para>
605 /// </summary>
606 /// <param name="generator">
607 /// <para>The generator.</para>
608 /// <para></para>
609 /// </param>
610 /// <param name="value">
611 /// <para>The value.</para>
612 /// <para></para>
613 /// </param>
614 [MethodImpl(MethodImplOptions.AggressiveInlining)]
615 public static void LoadConstant(this ILGenerator generator, ushort value) =>
616     ↪ generator.LoadConstant((int)value);
617
618 /// <summary>
619 /// <para>

```

```

618     /// Loads the constant using the specified generator.
619     /// </para>
620     /// <para></para>
621     /// </summary>
622     /// <param name="generator">
623     /// <para>The generator.</para>
624     /// <para></para>
625     /// </param>
626     /// <param name="value">
627     /// <para>The value.</para>
628     /// <para></para>
629     /// </param>
630     [MethodImpl(MethodImplOptions.AggressiveInlining)]
631     public static void LoadConstant(this ILGenerator generator, sbyte value) =>
        ↪ generator.LoadConstant((int)value);
632
633     /// <summary>
634     /// <para>
635     /// Loads the constant using the specified generator.
636     /// </para>
637     /// <para></para>
638     /// </summary>
639     /// <param name="generator">
640     /// <para>The generator.</para>
641     /// <para></para>
642     /// </param>
643     /// <param name="value">
644     /// <para>The value.</para>
645     /// <para></para>
646     /// </param>
647     [MethodImpl(MethodImplOptions.AggressiveInlining)]
648     public static void LoadConstant(this ILGenerator generator, byte value) =>
        ↪ generator.LoadConstant((int)value);
649
650     /// <summary>
651     /// <para>
652     /// Loads the constant one using the specified generator.
653     /// </para>
654     /// <para></para>
655     /// </summary>
656     /// <typeparam name="TConstant">
657     /// <para>The constant.</para>
658     /// <para></para>
659     /// </typeparam>
660     /// <param name="generator">
661     /// <para>The generator.</para>
662     /// <para></para>
663     /// </param>
664     [MethodImpl(MethodImplOptions.AggressiveInlining)]
665     public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
        ↪ LoadConstantOne(generator, typeof(TConstant));
666
667     /// <summary>
668     /// <para>
669     /// Loads the constant one using the specified generator.
670     /// </para>
671     /// <para></para>
672     /// </summary>
673     /// <param name="generator">
674     /// <para>The generator.</para>
675     /// <para></para>
676     /// </param>
677     /// <param name="constantType">
678     /// <para>The constant type.</para>
679     /// <para></para>
680     /// </param>
681     /// <exception cref="NotSupportedException">
682     /// <para></para>
683     /// <para></para>
684     /// </exception>
685     [MethodImpl(MethodImplOptions.AggressiveInlining)]
686     public static void LoadConstantOne(this ILGenerator generator, Type constantType)
687     {
688         if (constantType == typeof(float))
689         {
690             generator.LoadConstant(1F);
691         }
692         else if (constantType == typeof(double))

```

```

693     {
694         generator.LoadConstant(1D);
695     }
696     else if (constantType == typeof(long))
697     {
698         generator.LoadConstant(1L);
699     }
700     else if (constantType == typeof(ulong))
701     {
702         generator.LoadConstant(1UL);
703     }
704     else if (constantType == typeof(int))
705     {
706         generator.LoadConstant(1);
707     }
708     else if (constantType == typeof(uint))
709     {
710         generator.LoadConstant(1U);
711     }
712     else if (constantType == typeof(short))
713     {
714         generator.LoadConstant((short)1);
715     }
716     else if (constantType == typeof(ushort))
717     {
718         generator.LoadConstant((ushort)1);
719     }
720     else if (constantType == typeof(sbyte))
721     {
722         generator.LoadConstant((sbyte)1);
723     }
724     else if (constantType == typeof(byte))
725     {
726         generator.LoadConstant((byte)1);
727     }
728     else
729     {
730         throw new NotSupportedException();
731     }
732 }
733
734 /// <summary>
735 /// <para>
736 /// Loads the constant using the specified generator.
737 /// </para>
738 /// <para></para>
739 /// </summary>
740 /// <typeparam name="TConstant">
741 /// <para>The constant.</para>
742 /// <para></para>
743 /// </typeparam>
744 /// <param name="generator">
745 /// <para>The generator.</para>
746 /// <para></para>
747 /// </param>
748 /// <param name="constantValue">
749 /// <para>The constant value.</para>
750 /// <para></para>
751 /// </param>
752 [MethodImpl(MethodImplOptions.AggressiveInlining)]
753 public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
    ↪ constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
754
755 /// <summary>
756 /// <para>
757 /// Loads the constant using the specified generator.
758 /// </para>
759 /// <para></para>
760 /// </summary>
761 /// <param name="generator">
762 /// <para>The generator.</para>
763 /// <para></para>
764 /// </param>
765 /// <param name="constantType">
766 /// <para>The constant type.</para>
767 /// <para></para>
768 /// </param>
769 /// <param name="constantValue">

```



```

770 /// <para>The constant value.</para>
771 /// <para></para>
772 /// </param>
773 /// <exception cref="NotSupportedException">
774 /// <para></para>
775 /// <para></para>
776 /// </exception>
777 [MethodImpl(MethodImplOptions.AggressiveInlining)]
778 public static void LoadConstant(this ILGenerator generator, Type constantType, object
779 → constantValue)
780 {
781     constantValue = Convert.ChangeType(constantValue, constantType);
782     if (constantType == typeof(float))
783     {
784         generator.LoadConstant((float)constantValue);
785     }
786     else if (constantType == typeof(double))
787     {
788         generator.LoadConstant((double)constantValue);
789     }
790     else if (constantType == typeof(long))
791     {
792         generator.LoadConstant((long)constantValue);
793     }
794     else if (constantType == typeof(ulong))
795     {
796         generator.LoadConstant((ulong)constantValue);
797     }
798     else if (constantType == typeof(int))
799     {
800         generator.LoadConstant((int)constantValue);
801     }
802     else if (constantType == typeof(uint))
803     {
804         generator.LoadConstant((uint)constantValue);
805     }
806     else if (constantType == typeof(short))
807     {
808         generator.LoadConstant((short)constantValue);
809     }
810     else if (constantType == typeof(ushort))
811     {
812         generator.LoadConstant((ushort)constantValue);
813     }
814     else if (constantType == typeof(sbyte))
815     {
816         generator.LoadConstant((sbyte)constantValue);
817     }
818     else if (constantType == typeof(byte))
819     {
820         generator.LoadConstant((byte)constantValue);
821     }
822     else
823     {
824         throw new NotSupportedException();
825     }
826 }
827 /// <summary>
828 /// <para>
829 /// Increments the generator.
830 /// </para>
831 /// <para></para>
832 /// </summary>
833 /// <typeparam name="TValue">
834 /// <para>The value.</para>
835 /// <para></para>
836 /// </typeparam>
837 /// <param name="generator">
838 /// <para>The generator.</para>
839 /// <para></para>
840 /// </param>
841 [MethodImpl(MethodImplOptions.AggressiveInlining)]
842 public static void Increment<TValue>(this ILGenerator generator) =>
843 → generator.Increment(typeof(TValue));
844 /// <summary>
845 /// <para>

```

```

846     /// Decrements the generator.
847     /// </para>
848     /// <para></para>
849     /// </summary>
850     /// <typeparam name="TValue">
851     /// <para>The value.</para>
852     /// <para></para>
853     /// </typeparam>
854     /// <param name="generator">
855     /// <para>The generator.</para>
856     /// <para></para>
857     /// </param>
858     [MethodImpl(MethodImplOptions.AggressiveInlining)]
859     public static void Decrement<TValue>(this ILGenerator generator) =>
860         ↪ generator.Decrement(typeof(TValue));
861
862     /// <summary>
863     /// <para>
864     /// Increments the generator.
865     /// </para>
866     /// <para></para>
867     /// </summary>
868     /// <param name="generator">
869     /// <para>The generator.</para>
870     /// <para></para>
871     /// </param>
872     /// <param name="valueType">
873     /// <para>The value type.</para>
874     /// <para></para>
875     /// </param>
876     [MethodImpl(MethodImplOptions.AggressiveInlining)]
877     public static void Increment(this ILGenerator generator, Type valueType)
878     {
879         generator.LoadConstantOne(valueType);
880         generator.Add();
881     }
882
883     /// <summary>
884     /// <para>
885     /// Adds the generator.
886     /// </para>
887     /// <para></para>
888     /// </summary>
889     /// <param name="generator">
890     /// <para>The generator.</para>
891     /// <para></para>
892     /// </param>
893     [MethodImpl(MethodImplOptions.AggressiveInlining)]
894     public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
895
896     /// <summary>
897     /// <para>
898     /// Decrements the generator.
899     /// </para>
900     /// <para></para>
901     /// </summary>
902     /// <param name="generator">
903     /// <para>The generator.</para>
904     /// <para></para>
905     /// </param>
906     /// <param name="valueType">
907     /// <para>The value type.</para>
908     /// <para></para>
909     /// </param>
910     [MethodImpl(MethodImplOptions.AggressiveInlining)]
911     public static void Decrement(this ILGenerator generator, Type valueType)
912     {
913         generator.LoadConstantOne(valueType);
914         generator.Subtract();
915     }
916
917     /// <summary>
918     /// <para>
919     /// Subtracts the generator.
920     /// </para>
921     /// <para></para>
922     /// </summary>
923     /// <param name="generator">

```

```

923     /// <para>The generator.</para>
924     /// <para></para>
925     /// </param>
926     [MethodImpl(MethodImplOptions.AggressiveInlining)]
927     public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
928
929     /// <summary>
930     /// <para>
931     /// Negates the generator.
932     /// </para>
933     /// <para></para>
934     /// </summary>
935     /// <param name="generator">
936     /// <para>The generator.</para>
937     /// <para></para>
938     /// </param>
939     [MethodImpl(MethodImplOptions.AggressiveInlining)]
940     public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
941
942     /// <summary>
943     /// <para>
944     /// Ands the generator.
945     /// </para>
946     /// <para></para>
947     /// </summary>
948     /// <param name="generator">
949     /// <para>The generator.</para>
950     /// <para></para>
951     /// </param>
952     [MethodImpl(MethodImplOptions.AggressiveInlining)]
953     public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
954
955     /// <summary>
956     /// <para>
957     /// Ors the generator.
958     /// </para>
959     /// <para></para>
960     /// </summary>
961     /// <param name="generator">
962     /// <para>The generator.</para>
963     /// <para></para>
964     /// </param>
965     [MethodImpl(MethodImplOptions.AggressiveInlining)]
966     public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
967
968     /// <summary>
969     /// <para>
970     /// Nots the generator.
971     /// </para>
972     /// <para></para>
973     /// </summary>
974     /// <param name="generator">
975     /// <para>The generator.</para>
976     /// <para></para>
977     /// </param>
978     [MethodImpl(MethodImplOptions.AggressiveInlining)]
979     public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
980
981     /// <summary>
982     /// <para>
983     /// Shifts the left using the specified generator.
984     /// </para>
985     /// <para></para>
986     /// </summary>
987     /// <param name="generator">
988     /// <para>The generator.</para>
989     /// <para></para>
990     /// </param>
991     [MethodImpl(MethodImplOptions.AggressiveInlining)]
992     public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
993
994     /// <summary>
995     /// <para>
996     /// Shifts the right using the specified generator.
997     /// </para>
998     /// <para></para>
999     /// </summary>
1000    /// <typeparam name="T">

```

```

1001    /// <para>The .</para>
1002    /// <para></para>
1003    /// </typeparam>
1004    /// <param name="generator">
1005    /// <para>The generator.</para>
1006    /// <para></para>
1007    /// </param>
1008    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1009    public static void ShiftRight<T>(this ILGenerator generator)
1010    {
1011        generator.Emit(NumericType<T>.IsSigned ? OpCodes.Shr : OpCodes.Shr_Un);
1012    }
1013
1014    /// <summary>
1015    /// <para>
1016    /// Loads the argument using the specified generator.
1017    /// </para>
1018    /// <para></para>
1019    /// </summary>
1020    /// <param name="generator">
1021    /// <para>The generator.</para>
1022    /// <para></para>
1023    /// </param>
1024    /// <param name="argumentIndex">
1025    /// <para>The argument index.</para>
1026    /// <para></para>
1027    /// </param>
1028    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029    public static void LoadArgument(this ILGenerator generator, int argumentIndex)
1030    {
1031        switch (argumentIndex)
1032        {
1033            case 0:
1034                generator.Emit(OpCodes.Ldarg_0);
1035                break;
1036            case 1:
1037                generator.Emit(OpCodes.Ldarg_1);
1038                break;
1039            case 2:
1040                generator.Emit(OpCodes.Ldarg_2);
1041                break;
1042            case 3:
1043                generator.Emit(OpCodes.Ldarg_3);
1044                break;
1045            default:
1046                generator.Emit(OpCodes.Ldarg, argumentIndex);
1047                break;
1048        }
1049    }
1050
1051    /// <summary>
1052    /// <para>
1053    /// Loads the arguments using the specified generator.
1054    /// </para>
1055    /// <para></para>
1056    /// </summary>
1057    /// <param name="generator">
1058    /// <para>The generator.</para>
1059    /// <para></para>
1060    /// </param>
1061    /// <param name="argumentIndices">
1062    /// <para>The argument indices.</para>
1063    /// <para></para>
1064    /// </param>
1065    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1066    public static void LoadArguments(this ILGenerator generator, params int[]
1067    ↪ argumentIndices)
1068    {
1069        for (var i = 0; i < argumentIndices.Length; i++)
1070        {
1071            generator.LoadArgument(argumentIndices[i]);
1072        }
1073    }
1074
1075    /// <summary>
1076    /// <para>
1077    /// Stores the argument using the specified generator.
1078    /// </para>
1079    /// <para></para>

```

```

1079     /// </summary>
1080     /// <param name="generator">
1081     /// <para>The generator.</para>
1082     /// <para></para>
1083     /// </param>
1084     /// <param name="argumentIndex">
1085     /// <para>The argument index.</para>
1086     /// <para></para>
1087     /// </param>
1088     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1089     public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
1090         ↪ generator.Emit(OpCodes.Starg, argumentIndex);
1091
1092     /// <summary>
1093     /// <para>
1094     /// <para>Compares the greater than using the specified generator.
1095     /// </para>
1096     /// <para></para>
1097     /// </summary>
1098     /// <param name="generator">
1099     /// <para>The generator.</para>
1100     /// <para></para>
1101     /// </param>
1102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1103     public static void CompareGreaterThan(this ILGenerator generator) =>
1104         ↪ generator.Emit(OpCodes.Cgt);
1105
1106     /// <summary>
1107     /// <para>
1108     /// <para>Unsigneds the compare greater 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 UnsignedCompareGreaterThan(this ILGenerator generator) =>
1118         ↪ generator.Emit(OpCodes.Cgt_Un);
1119
1120     /// <summary>
1121     /// <para>
1122     /// <para>Compares the greater 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     /// <param name="isSigned">
1131     /// <para>The is signed.</para>
1132     /// <para></para>
1133     /// </param>
1134     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1135     public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
1136     {
1137         if (isSigned)
1138         {
1139             generator.CompareGreaterThan();
1140         }
1141         else
1142         {
1143             generator.UnsignedCompareGreaterThan();
1144         }
1145     }
1146
1147     /// <summary>
1148     /// <para>
1149     /// <para>Compares the less than using the specified generator.
1150     /// </para>
1151     /// <para></para>
1152     /// </summary>
1153     /// <param name="generator">
1154     /// <para>The generator.</para>
1155     /// <para></para>
1156     /// </param>

```

```

1154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1155 public static void CompareLessThan(this ILGenerator generator) =>
    ↪ generator.Emit(OpCodes.Clt);
1156
1157 /// <summary>
1158 /// <para>
1159 /// Unsigneds the compare less than using the specified generator.
1160 /// </para>
1161 /// <para></para>
1162 /// </summary>
1163 /// <param name="generator">
1164 /// <para>The generator.</para>
1165 /// <para></para>
1166 /// </param>
1167 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1168 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
    ↪ generator.Emit(OpCodes.Clt_Un);
1169
1170 /// <summary>
1171 /// <para>
1172 /// Compares the less than using the specified generator.
1173 /// </para>
1174 /// <para></para>
1175 /// </summary>
1176 /// <param name="generator">
1177 /// <para>The generator.</para>
1178 /// <para></para>
1179 /// </param>
1180 /// <param name="isSigned">
1181 /// <para>The is signed.</para>
1182 /// <para></para>
1183 /// </param>
1184 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1185 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
1186 {
1187     if (isSigned)
1188     {
1189         generator.CompareLessThan();
1190     }
1191     else
1192     {
1193         generator.UnsignedCompareLessThan();
1194     }
1195 }
1196
1197 /// <summary>
1198 /// <para>
1199 /// Branches the if greater or equal using the specified generator.
1200 /// </para>
1201 /// <para></para>
1202 /// </summary>
1203 /// <param name="generator">
1204 /// <para>The generator.</para>
1205 /// <para></para>
1206 /// </param>
1207 /// <param name="label">
1208 /// <para>The label.</para>
1209 /// <para></para>
1210 /// </param>
1211 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1212 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
    ↪ generator.Emit(OpCodes.Bge, label);
1213
1214 /// <summary>
1215 /// <para>
1216 /// Unsigneds the branch if greater or equal using the specified generator.
1217 /// </para>
1218 /// <para></para>
1219 /// </summary>
1220 /// <param name="generator">
1221 /// <para>The generator.</para>
1222 /// <para></para>
1223 /// </param>
1224 /// <param name="label">
1225 /// <para>The label.</para>
1226 /// <para></para>
1227 /// </param>
1228 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

1229 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
    ↳ label) => generator.Emit(OpCodes.Bge_Un, label);
1230
1231 /// <summary>
1232 /// <para>
1233 /// Branches the if greater or equal using the specified generator.
1234 /// </para>
1235 /// <para></para>
1236 /// </summary>
1237 /// <param name="generator">
1238 /// <para>The generator.</para>
1239 /// <para></para>
1240 /// </param>
1241 /// <param name="isSigned">
1242 /// <para>The is signed.</para>
1243 /// <para></para>
1244 /// </param>
1245 /// <param name="label">
1246 /// <para>The label.</para>
1247 /// <para></para>
1248 /// </param>
1249 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1250 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
    ↳ Label label)
1251 {
1252     if (isSigned)
1253     {
1254         generator.BranchIfGreaterOrEqual(label);
1255     }
1256     else
1257     {
1258         generator.UnsignedBranchIfGreaterOrEqual(label);
1259     }
1260 }
1261
1262 /// <summary>
1263 /// <para>
1264 /// Branches the if less or equal using the specified generator.
1265 /// </para>
1266 /// <para></para>
1267 /// </summary>
1268 /// <param name="generator">
1269 /// <para>The generator.</para>
1270 /// <para></para>
1271 /// </param>
1272 /// <param name="label">
1273 /// <para>The label.</para>
1274 /// <para></para>
1275 /// </param>
1276 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1277 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
    ↳ generator.Emit(OpCodes.Ble, label);
1278
1279 /// <summary>
1280 /// <para>
1281 /// Unsigneds the branch if less or equal using the specified generator.
1282 /// </para>
1283 /// <para></para>
1284 /// </summary>
1285 /// <param name="generator">
1286 /// <para>The generator.</para>
1287 /// <para></para>
1288 /// </param>
1289 /// <param name="label">
1290 /// <para>The label.</para>
1291 /// <para></para>
1292 /// </param>
1293 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1294 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
    ↳ => generator.Emit(OpCodes.Ble_Un, label);
1295
1296 /// <summary>
1297 /// <para>
1298 /// Branches the if less or equal using the specified generator.
1299 /// </para>
1300 /// <para></para>
1301 /// </summary>
1302 /// <param name="generator">

```

```

1303     /// <para>The generator.</para>
1304     /// <para></para>
1305     /// </param>
1306     /// <param name="isSigned">
1307     /// <para>The is signed.</para>
1308     /// <para></para>
1309     /// </param>
1310     /// <param name="label">
1311     /// <para>The label.</para>
1312     /// <para></para>
1313     /// </param>
1314     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1315     public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
    ↪ label)
1316     {
1317         if (isSigned)
1318         {
1319             generator.BranchIfLessOrEqual(label);
1320         }
1321         else
1322         {
1323             generator.UnsignedBranchIfLessOrEqual(label);
1324         }
1325     }
1326
1327     /// <summary>
1328     /// <para>
1329     /// Boxes the generator.
1330     /// </para>
1331     /// <para></para>
1332     /// </summary>
1333     /// <typeparam name="TBox">
1334     /// <para>The box.</para>
1335     /// <para></para>
1336     /// </typeparam>
1337     /// <param name="generator">
1338     /// <para>The generator.</para>
1339     /// <para></para>
1340     /// </param>
1341     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1342     public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
1343
1344     /// <summary>
1345     /// <para>
1346     /// Boxes the generator.
1347     /// </para>
1348     /// <para></para>
1349     /// </summary>
1350     /// <param name="generator">
1351     /// <para>The generator.</para>
1352     /// <para></para>
1353     /// </param>
1354     /// <param name="boxedType">
1355     /// <para>The boxed type.</para>
1356     /// <para></para>
1357     /// </param>
1358     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1359     public static void Box(this ILGenerator generator, Type boxedType) =>
    ↪ generator.Emit(OpCodes.Box, boxedType);
1360
1361     /// <summary>
1362     /// <para>
1363     /// Calls the generator.
1364     /// </para>
1365     /// <para></para>
1366     /// </summary>
1367     /// <param name="generator">
1368     /// <para>The generator.</para>
1369     /// <para></para>
1370     /// </param>
1371     /// <param name="method">
1372     /// <para>The method.</para>
1373     /// <para></para>
1374     /// </param>
1375     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1376     public static void Call(this ILGenerator generator, MethodInfo method) =>
    ↪ generator.Emit(OpCodes.Call, method);
1377

```



```

1378     /// <summary>
1379     /// <para>
1380     /// Returns the generator.
1381     /// </para>
1382     /// <para></para>
1383     /// </summary>
1384     /// <param name="generator">
1385     /// <para>The generator.</para>
1386     /// <para></para>
1387     /// </param>
1388     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1389     public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
1390
1391     /// <summary>
1392     /// <para>
1393     /// Unboxes the generator.
1394     /// </para>
1395     /// <para></para>
1396     /// </summary>
1397     /// <typeparam name="TUnbox">
1398     /// <para>The unbox.</para>
1399     /// <para></para>
1400     /// </typeparam>
1401     /// <param name="generator">
1402     /// <para>The generator.</para>
1403     /// <para></para>
1404     /// </param>
1405     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1406     public static void Unbox<TUnbox>(this ILGenerator generator) =>
1407         ↪ generator.Unbox(typeof(TUnbox));
1408
1409     /// <summary>
1410     /// <para>
1411     /// Unboxes the generator.
1412     /// </para>
1413     /// <para></para>
1414     /// </summary>
1415     /// <param name="generator">
1416     /// <para>The generator.</para>
1417     /// <para></para>
1418     /// </param>
1419     /// <param name="typeToUnbox">
1420     /// <para>The type to unbox.</para>
1421     /// <para></para>
1422     /// </param>
1423     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1424     public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
1425         ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
1426
1427     /// <summary>
1428     /// <para>
1429     /// Unboxes the value using the specified generator.
1430     /// </para>
1431     /// <para></para>
1432     /// </summary>
1433     /// <typeparam name="TUnbox">
1434     /// <para>The unbox.</para>
1435     /// <para></para>
1436     /// </typeparam>
1437     /// <param name="generator">
1438     /// <para>The generator.</para>
1439     /// <para></para>
1440     /// </param>
1441     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1442     public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
1443         ↪ generator.UnboxValue(typeof(TUnbox));
1444
1445     /// <summary>
1446     /// <para>
1447     /// Unboxes the value using the specified generator.
1448     /// </para>
1449     /// <para></para>
1450     /// </summary>
1451     /// <param name="generator">
1452     /// <para>The generator.</para>
1453     /// <para></para>
1454     /// </param>
1455     /// <param name="typeToUnbox">

```

```

1453     /// <para>The type to unbox.</para>
1454     /// <para></para>
1455     /// </param>
1456     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1457     public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
        ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);

1458
1459     /// <summary>
1460     /// <para>
1461     /// Declares the local using the specified generator.
1462     /// </para>
1463     /// <para></para>
1464     /// </summary>
1465     /// <typeparam name="T">
1466     /// <para>The .</para>
1467     /// <para></para>
1468     /// </typeparam>
1469     /// <param name="generator">
1470     /// <para>The generator.</para>
1471     /// <para></para>
1472     /// </param>
1473     /// <returns>
1474     /// <para>The local builder</para>
1475     /// <para></para>
1476     /// </returns>
1477     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1478     public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
        ↪ generator.DeclareLocal(typeof(T));

1479
1480     /// <summary>
1481     /// <para>
1482     /// Loads the local using the specified generator.
1483     /// </para>
1484     /// <para></para>
1485     /// </summary>
1486     /// <param name="generator">
1487     /// <para>The generator.</para>
1488     /// <para></para>
1489     /// </param>
1490     /// <param name="local">
1491     /// <para>The local.</para>
1492     /// <para></para>
1493     /// </param>
1494     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1495     public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
        ↪ generator.Emit(OpCodes.Ldloc, local);

1496
1497     /// <summary>
1498     /// <para>
1499     /// Stores the local using the specified generator.
1500     /// </para>
1501     /// <para></para>
1502     /// </summary>
1503     /// <param name="generator">
1504     /// <para>The generator.</para>
1505     /// <para></para>
1506     /// </param>
1507     /// <param name="local">
1508     /// <para>The local.</para>
1509     /// <para></para>
1510     /// </param>
1511     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1512     public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
        ↪ generator.Emit(OpCodes.Stloc, local);

1513
1514     /// <summary>
1515     /// <para>
1516     /// News the object using the specified generator.
1517     /// </para>
1518     /// <para></para>
1519     /// </summary>
1520     /// <param name="generator">
1521     /// <para>The generator.</para>
1522     /// <para></para>
1523     /// </param>
1524     /// <param name="type">
1525     /// <para>The type.</para>
1526     /// <para></para>

```

```

1527     /// </param>
1528     /// <param name="parameterTypes">
1529     /// <para>The parameter types.</para>
1530     /// </para></param>
1531     /// </param>
1532     /// <exception cref="InvalidOperationException">
1533     /// <para></para>
1534     /// <para></para>
1535     /// </exception>
1536     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1537     public static void NewObject(this ILGenerator generator, Type type, params Type[]
        ↪ parameterTypes)
1538     {
1539         var allConstructors = type.GetConstructors(BindingFlags.Public |
        ↪ BindingFlags.NonPublic | BindingFlags.Instance
1540 #if !NETSTANDARD
        ↪ BindingFlags.CreateInstance
1541 #endif
        );
1542         var constructor = allConstructors.Where(c => c.GetParameters().Length ==
        ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
        ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
1543         if (constructor == null)
1544         {
1545             throw new InvalidOperationException("Type " + type + " must have a constructor
        ↪ that matches parameters [" + string.Join(", ",
        ↪ parameterTypes.AsEnumerable()) + "]");
1546         }
1547         generator.NewObject(constructor);
1548     }
1549 }
1550
1551     /// <summary>
1552     /// <para>
1553     /// News the object using the specified generator.
1554     /// </para>
1555     /// </para></summary>
1556     /// <param name="generator">
1557     /// <para>The generator.</para>
1558     /// </para></param>
1559     /// <param name="constructor">
1560     /// <para>The constructor.</para>
1561     /// </para></param>
1562     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1563     public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
        ↪ generator.Emit(OpCodes.Newobj, constructor);
1564
1565     /// <summary>
1566     /// <para>
1567     /// Loads the indirect using the specified generator.
1568     /// </para>
1569     /// </para></summary>
1570     /// <typeparam name="T">
1571     /// <para>The .</para>
1572     /// </para></typeparam>
1573     /// <param name="generator">
1574     /// <para>The generator.</para>
1575     /// </para></param>
1576     /// <param name="isVolatile">
1577     /// <para>The is volatile.</para>
1578     /// </para></param>
1579     /// <param name="unaligned">
1580     /// <para>The unaligned.</para>
1581     /// </para></param>
1582     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1583     public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
        ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
1584
1585     /// <summary>
1586     /// <para>
1587 
```

```

1596     /// Loads the indirect using the specified generator.
1597     /// </para>
1598     /// <para></para>
1599     /// </summary>
1600     /// <param name="generator">
1601     /// <para>The generator.</para>
1602     /// <para></para>
1603     /// </param>
1604     /// <param name="type">
1605     /// <para>The type.</para>
1606     /// <para></para>
1607     /// </param>
1608     /// <param name="isVolatile">
1609     /// <para>The is volatile.</para>
1610     /// <para></para>
1611     /// </param>
1612     /// <param name="unaligned">
1613     /// <para>The unaligned.</para>
1614     /// <para></para>
1615     /// </param>
1616     /// <exception cref="InvalidOperationException">
1617     /// <para></para>
1618     /// <para></para>
1619     /// </exception>
1620     /// <exception cref="ArgumentException">
1621     /// <para>unaligned must be null, 1, 2, or 4</para>
1622     /// <para></para>
1623     /// </exception>
1624     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1625     public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
        ↪ false, byte? unaligned = null)
1626     {
1627         if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
1628         {
1629             throw new ArgumentException("unaligned must be null, 1, 2, or 4");
1630         }
1631         if (isVolatile)
1632         {
1633             generator.Emit(OpCodes.Volatile);
1634         }
1635         if (unaligned.HasValue)
1636         {
1637             generator.Emit(OpCodes.Unaligned, unaligned.Value);
1638         }
1639         if (type.IsPointer)
1640         {
1641             generator.Emit(OpCodes.Ldind_I);
1642         }
1643         else if (!type.IsValueType)
1644         {
1645             generator.Emit(OpCodes.Ldind_Ref);
1646         }
1647         else if (type == typeof(sbyte))
1648         {
1649             generator.Emit(OpCodes.Ldind_I1);
1650         }
1651         else if (type == typeof(bool))
1652         {
1653             generator.Emit(OpCodes.Ldind_I1);
1654         }
1655         else if (type == typeof(byte))
1656         {
1657             generator.Emit(OpCodes.Ldind_U1);
1658         }
1659         else if (type == typeof(short))
1660         {
1661             generator.Emit(OpCodes.Ldind_I2);
1662         }
1663         else if (type == typeof(ushort))
1664         {
1665             generator.Emit(OpCodes.Ldind_U2);
1666         }
1667         else if (type == typeof(char))
1668         {
1669             generator.Emit(OpCodes.Ldind_U2);
1670         }
1671         else if (type == typeof(int))

```

```

1672     {
1673         generator.Emit(OpCodes.Ldind_I4);
1674     }
1675     else if (type == typeof(uint))
1676     {
1677         generator.Emit(OpCodes.Ldind_U4);
1678     }
1679     else if (type == typeof(long) || type == typeof(ulong))
1680     {
1681         generator.Emit(OpCodes.Ldind_I8);
1682     }
1683     else if (type == typeof(float))
1684     {
1685         generator.Emit(OpCodes.Ldind_R4);
1686     }
1687     else if (type == typeof(double))
1688     {
1689         generator.Emit(OpCodes.Ldind_R8);
1690     }
1691     else
1692     {
1693         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
1694             ↪ " ", LoadObject may be more appropriate");
1695     }
1696 }
1697
1698 /// <summary>
1699 /// <para>
1700 /// Stores the indirect using the specified generator.
1701 /// </para>
1702 /// <para></para>
1703 /// </summary>
1704 /// <typeparam name="T">
1705 /// <para>The .</para>
1706 /// <para></para>
1707 /// </typeparam>
1708 /// <param name="generator">
1709 /// <para>The generator.</para>
1710 /// <para></para>
1711 /// </param>
1712 /// <param name="isVolatile">
1713 /// <para>The is volatile.</para>
1714 /// <para></para>
1715 /// </param>
1716 /// <param name="unaligned">
1717 /// <para>The unaligned.</para>
1718 /// <para></para>
1719 /// </param>
1720 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1721 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
1722     ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
1723
1724 /// <summary>
1725 /// <para>
1726 /// Stores the indirect using the specified generator.
1727 /// </para>
1728 /// <para></para>
1729 /// </summary>
1730 /// <param name="generator">
1731 /// <para>The generator.</para>
1732 /// <para></para>
1733 /// </param>
1734 /// <param name="type">
1735 /// <para>The type.</para>
1736 /// <para></para>
1737 /// </param>
1738 /// <param name="isVolatile">
1739 /// <para>The is volatile.</para>
1740 /// <para></para>
1741 /// </param>
1742 /// <param name="unaligned">
1743 /// <para>The unaligned.</para>
1744 /// <para></para>
1745 /// </param>
1746 /// <exception cref="InvalidOperationException">
1747 /// <para></para>
1748 /// <para></para>
1749 /// </exception>

```

```

1748 /// <exception cref="ArgumentException">
1749 /// <para>unaligned must be null, 1, 2, or 4</para>
1750 /// </exception>
1751 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1752 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
1753 ↪ = false, byte? unaligned = null)
1754 {
1755     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
1756     {
1757         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
1758     }
1759     if (isVolatile)
1760     {
1761         generator.Emit(OpCodes.Volatile);
1762     }
1763     if (unaligned.HasValue)
1764     {
1765         generator.Emit(OpCodes.Unaligned, unaligned.Value);
1766     }
1767     if (type.IsPointer)
1768     {
1769         generator.Emit(OpCodes.Stind_I);
1770     }
1771     else if (!type.IsValueType)
1772     {
1773         generator.Emit(OpCodes.Stind_Ref);
1774     }
1775     else if (type == typeof(sbyte) || type == typeof(byte))
1776     {
1777         generator.Emit(OpCodes.Stind_I1);
1778     }
1779     else if (type == typeof(short) || type == typeof(ushort))
1780     {
1781         generator.Emit(OpCodes.Stind_I2);
1782     }
1783     else if (type == typeof(int) || type == typeof(uint))
1784     {
1785         generator.Emit(OpCodes.Stind_I4);
1786     }
1787     else if (type == typeof(long) || type == typeof(ulong))
1788     {
1789         generator.Emit(OpCodes.Stind_I8);
1790     }
1791     else if (type == typeof(float))
1792     {
1793         generator.Emit(OpCodes.Stind_R4);
1794     }
1795     else if (type == typeof(double))
1796     {
1797         generator.Emit(OpCodes.Stind_R8);
1798     }
1799     else
1800     {
1801         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
1802 ↪ + ", StoreObject may be more appropriate");
1803     }
1804 }
1805 /// <summary>
1806 /// <para>
1807 /// Multiplies the generator.
1808 /// </para>
1809 /// </summary>
1810 /// <param name="generator">
1811 /// <para>The generator.</para>
1812 /// </param>
1813 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1814 public static void Multiply(this ILGenerator generator)
1815 {
1816     generator.Emit(OpCodes.Mul);
1817 }
1818 /// <summary>
1819 /// <para>
1820 /// Checks the multiply using the specified generator.

```

```

1824     /// </para>
1825     /// <para></para>
1826     /// </summary>
1827     /// <typeparam name="T">
1828     /// <para>The .</para>
1829     /// <para></para>
1830     /// </typeparam>
1831     /// <param name="generator">
1832     /// <para>The generator.</para>
1833     /// <para></para>
1834     /// </param>
1835     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1836     public static void CheckedMultiply<T>(this ILGenerator generator)
1837     {
1838         if (NumericType<T>.IsSigned)
1839         {
1840             generator.Emit(OpCodes.Mul_Ovf);
1841         }
1842         else
1843         {
1844             generator.Emit(OpCodes.Mul_Ovf_Un);
1845         }
1846     }
1847
1848     /// <summary>
1849     /// <para>
1850     /// Divides the generator.
1851     /// </para>
1852     /// <para></para>
1853     /// </summary>
1854     /// <typeparam name="T">
1855     /// <para>The .</para>
1856     /// <para></para>
1857     /// </typeparam>
1858     /// <param name="generator">
1859     /// <para>The generator.</para>
1860     /// <para></para>
1861     /// </param>
1862     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1863     public static void Divide<T>(this ILGenerator generator)
1864     {
1865         if (NumericType<T>.IsSigned)
1866         {
1867             generator.Emit(OpCodes.Div);
1868         }
1869         else
1870         {
1871             generator.Emit(OpCodes.Div_Un);
1872         }
1873     }
1874 }
1875 }

```

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 }

```

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,
135                ↪ out Type unsignedVersion);
136            Type = type;
137            IsNullable = isNullable;
138            UnderlyingType = underlyingType;
139            CanBeNumeric = canBeNumeric;
140            IsNumeric = isNumeric;
141            IsSigned = isSigned;
142            IsFloatPoint = isFloatPoint;
143            BytesSize = bytesSize;
144            BitsSize = bitsSize;
145            MinValue = minValue;
146            MaxValue = maxValue;
147            SignedVersion = signedVersion;
148            UnsignedVersion = unsignedVersion;
149        }
150        catch (Exception exception)
151        {
152            exception.Ignore();
153        }
154    }

```

```

155     /// <summary>
156     /// <para>
157     /// Gets the min and max values using the specified type.
158     /// </para>
159     /// <para></para>
160     /// </summary>
161     /// <param name="type">
162     /// <para>The type.</para>
163     /// <para></para>
164     /// </param>
165     /// <param name="minValue">
166     /// <para>The min value.</para>
167     /// <para></para>
168     /// </param>
169     /// <param name="maxValue">
170     /// <para>The max value.</para>
171     /// <para></para>
172     /// </param>
173     [MethodImpl(MethodImplOptions.AggressiveInlining)]
174     private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
175     {
176         if (type == typeof(bool))
177         {
178             minValue = (T)(object>false;
179             maxValue = (T)(object>true;
180         }
181         else
182         {
183             minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
184             maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
185         }
186     }
187
188     /// <summary>
189     /// <para>
190     /// Gets the signed and unsigned versions using the specified type.
191     /// </para>
192     /// <para></para>
193     /// </summary>
194     /// <param name="type">
195     /// <para>The type.</para>
196     /// <para></para>
197     /// </param>
198     /// <param name="isSigned">
199     /// <para>The is signed.</para>
200     /// <para></para>
201     /// </param>
202     /// <param name="signedVersion">
203     /// <para>The signed version.</para>
204     /// <para></para>
205     /// </param>
206     /// <param name="unsignedVersion">
207     /// <para>The unsigned version.</para>
208     /// <para></para>
209     /// </param>
210     [MethodImpl(MethodImplOptions.AggressiveInlining)]
211     private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
212     ↪ signedVersion, out Type unsignedVersion)
213     {
214         if (isSigned)
215         {
216             signedVersion = type;
217             unsignedVersion = type.GetUnsignedVersionOrNull();
218         }
219         else
220         {
221             signedVersion = type.GetSignedVersionOrNull();
222             unsignedVersion = type;
223         }
224     }
225 }

```

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) =>
36            ↪ (T)fieldInfo.GetValue(null);
37    }
38 }

```

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 =
25            ↪ MethodAttributes.Public | MethodAttributes.Static;
26        /// <summary>
27        /// <para>
28        /// The hide by sig.
29        /// </para>
30        /// <para></para>
31        /// </summary>
32        public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes =
33            ↪ MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
34            ↪ MethodAttributes.HideBySig;
35        /// <summary>
36        /// <para>
37        /// The aggressive inlining.
38        /// </para>
39        /// <para></para>
40        /// </summary>
41        public static readonly MethodImplAttributes DefaultMethodImplAttributes =
42            ↪ MethodImplAttributes.IL | MethodImplAttributes.Managed |
43            ↪ MethodImplAttributes.AggressiveInlining;
44    }
45 }

```

```

39
40     /// <summary>
41     /// <para>
42     /// Emits the method using the specified type.
43     /// </para>
44     /// <para></para>
45     /// </summary>
46     /// <typeparam name="TDelegate">
47     /// <para>The delegate.</para>
48     /// <para></para>
49     /// </typeparam>
50     /// <param name="type">
51     /// <para>The type.</para>
52     /// <para></para>
53     /// </param>
54     /// <param name="methodName">
55     /// <para>The method name.</para>
56     /// <para></para>
57     /// </param>
58     /// <param name="methodAttributes">
59     /// <para>The method attributes.</para>
60     /// <para></para>
61     /// </param>
62     /// <param name="methodImplAttributes">
63     /// <para>The method impl attributes.</para>
64     /// <para></para>
65     /// </param>
66     /// <param name="emitCode">
67     /// <para>The emit code.</para>
68     /// <para></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     /// <para></para>
86     /// </summary>
87     /// <param name="type">
88     /// <para>The type.</para>
89     /// <para></para>
90     /// </param>
91     /// <param name="methodName">
92     /// <para>The method name.</para>
93     /// <para></para>
94     /// </param>
95     /// <param name="methodAttributes">
96     /// <para>The method attributes.</para>
97     /// <para></para>
98     /// </param>
99     /// <param name="methodImplAttributes">
100    /// <para>The method impl attributes.</para>
101    /// <para></para>
102    /// </param>
103    /// <param name="returnType">
104    /// <para>The return type.</para>
105    /// <para></para>
106    /// </param>
107    /// <param name="parameterTypes">
108    /// <para>The parameter types.</para>
109    /// <para></para>
110    /// </param>
111    /// <param name="emitCode">
112    /// <para>The emit code.</para>
113    /// <para></para>
114    /// </param>
115    [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

112 public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
    ↳ methodAttributes, MethodInfoAttributes methodImplAttributes, Type returnType, Type[]
    ↳ parameterTypes, Action<ILGenerator> emitCode)
113 {
114     MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
    ↳ parameterTypes);
115     method.SetImplementationFlags(methodImplAttributes);
116     var generator = method.GetILGenerator();
117     emitCode(generator);
118 }
119
120 /// <summary>
121 /// <para>
122 /// Emits the static method using the specified type.
123 /// </para>
124 /// <para></para>
125 /// </summary>
126 /// <typeparam name="TDelegate">
127 /// <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) },
107    };
108    signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
109    {
110        { typeof(byte), typeof(sbyte) },
111        { typeof(ushort), typeof(short) },
112        { typeof(uint), typeof(int) },
113        { typeof(ulong), typeof(long) },
114    };
115    }
116
117    /// <summary>
118    /// <para>
119    /// Gets the first field using the specified type.
120    /// </para>
121    /// </summary>
122    /// <param name="type">
123    /// <para>The type.</para>
124    /// </param>
125    /// <returns>
126    /// <para>The field info</para>
127    /// </returns>
128    [MethodImpl(MethodImplOptions.AggressiveInlining)]
129    public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
130
131    /// <summary>
132    /// <para>
133    /// Gets the static field value using the specified type.
134    /// </para>
135    /// </summary>
136    /// <typeparam name="T">
137    /// <para>The .</para>
138    /// </typeparam>
139    /// <param name="type">
140    /// <para>The type.</para>
141    /// </param>
142    /// <param name="name">
143    /// <para>The name.</para>
144    /// </param>
145    /// <returns>
146    /// <para>The</para>
147    /// </returns>
148    [MethodImpl(MethodImplOptions.AggressiveInlining)]
149    public static T GetStaticFieldValue<T>(this Type type, string name) =>
150    ↳ type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();
151
152    /// <summary>
153    /// <para>
154    /// Gets the static property value using the specified type.
155    /// </para>

```



```

159     /// <para></para>
160     /// </summary>
161     /// <typeparam name="T">
162     /// <para>The .</para>
163     /// <para></para>
164     /// </typeparam>
165     /// <param name="type">
166     /// <para>The type.</para>
167     /// <para></para>
168     /// </param>
169     /// <param name="name">
170     /// <para>The name.</para>
171     /// <para></para>
172     /// </param>
173     /// <returns>
174     /// <para>The</para>
175     /// <para></para>
176     /// </returns>
177     [MethodImpl(MethodImplOptions.AggressiveInlining)]
178     public static T GetStaticPropertyValue<T>(this Type type, string name) =>
179         ↪ type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();
180
181     /// <summary>
182     /// <para>
183     /// Gets the generic method using the specified type.
184     /// </para>
185     /// <para></para>
186     /// </summary>
187     /// <param name="type">
188     /// <para>The type.</para>
189     /// <para></para>
190     /// </param>
191     /// <param name="name">
192     /// <para>The name.</para>
193     /// <para></para>
194     /// </param>
195     /// <param name="genericParameterTypes">
196     /// <para>The generic parameter types.</para>
197     /// <para></para>
198     /// </param>
199     /// <param name="argumentTypes">
200     /// <para>The argument types.</para>
201     /// <para></para>
202     /// </param>
203     /// <returns>
204     /// <para>The method.</para>
205     /// <para></para>
206     /// </returns>
207     [MethodImpl(MethodImplOptions.AggressiveInlining)]
208     public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
209         ↪ genericParameterTypes, Type[] argumentTypes)
210     {
211         var methods = from m in type.GetMethods()
212                       where m.Name == name
213                           && m.IsGenericMethodDefinition
214                           let typeParams = m.GetGenericArguments()
215                           let normalParams = m.GetParameters().Select(x => x.ParameterType)
216                           where typeParams.SequenceEqual(genericParameterTypes)
217                           && normalParams.SequenceEqual(argumentTypes)
218                           select m;
219         var method = methods.Single();
220         return method;
221     }
222
223     /// <summary>
224     /// <para>
225     /// Gets the base type using the specified type.
226     /// </para>
227     /// <para></para>
228     /// </summary>
229     /// <param name="type">
230     /// <para>The type.</para>
231     /// <para></para>
232     /// </param>
233     /// <returns>
234     /// <para>The type</para>
235     /// <para></para>
236     /// </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>

```

```

312     /// Determines whether is generic.
313     /// </para>
314     /// <para></para>
315     /// </summary>
316     /// <param name="type">
317     /// <para>The type.</para>
318     /// <para></para>
319     /// </param>
320     /// <param name="genericTypeDefinition">
321     /// <para>The generic type definition.</para>
322     /// <para></para>
323     /// </param>
324     /// <returns>
325     /// <para>The bool</para>
326     /// <para></para>
327     /// </returns>
328     [MethodImpl(MethodImplOptions.AggressiveInlining)]
329     public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
330         ↪ type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
331     /// <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     /// <summary>
348     /// <para>
349     /// Gets the unsigned version or null using the specified signed type.
350     /// </para>
351     /// <para></para>
352     /// </summary>
353     /// <param name="signedType">
354     /// <para>The signed type.</para>
355     /// <para></para>
356     /// </param>
357     /// <returns>
358     /// <para>The type</para>
359     /// <para></para>
360     /// </returns>
361     [MethodImpl(MethodImplOptions.AggressiveInlining)]
362     public static Type GetUnsignedVersionOrNull(this Type signedType) =>
363         ↪ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
364     /// <summary>
365     /// <para>
366     /// Gets the signed version or null using the specified unsigned type.
367     /// </para>
368     /// <para></para>
369     /// </summary>
370     /// <param name="unsignedType">
371     /// <para>The unsigned type.</para>
372     /// <para></para>
373     /// </param>
374     /// <returns>
375     /// <para>The type</para>
376     /// <para></para>
377     /// </returns>
378     [MethodImpl(MethodImplOptions.AggressiveInlining)]
379     public static Type GetSignedVersionOrNull(this Type unsignedType) =>
380         ↪ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
381     /// <summary>
382     /// <para>
383     /// Determines whether can be numeric.
384     /// </para>
385     /// <para></para>
386     /// </summary>

```

```

387     /// </summary>
388     /// <param name="type">
389     /// <para>The type.</para>
390     /// <para></para>
391     /// </param>
392     /// <returns>
393     /// <para>The bool</para>
394     /// <para></para>
395     /// </returns>
396     [MethodImpl(MethodImplOptions.AggressiveInlining)]
397     public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
398
399     /// <summary>
400     /// <para>
401     /// Determines whether is numeric.
402     /// </para>
403     /// <para></para>
404     /// </summary>
405     /// <param name="type">
406     /// <para>The type.</para>
407     /// <para></para>
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) =>
    ↪ delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;
466
467 /// <summary>
468 /// <para>
469 /// Gets the delegate parameter types using the specified delegate type.
470 /// </para>
471 /// <para></para>
472 /// </summary>
473 /// <param name="delegateType">
474 /// <para>The delegate type.</para>
475 /// <para></para>
476 /// </param>
477 /// <returns>
478 /// <para>The type array</para>
479 /// <para></para>
480 /// </returns>
481 [MethodImpl(MethodImplOptions.AggressiveInlining)]
482 public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
    ↪ delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();
483
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
54     /// <summary>
55     /// <para>
56     /// Appends the types using the specified container.
57     /// </para>
58     /// <para></para>
59     /// </summary>
60     /// <param name="container">
61     /// <para>The container.</para>
62     /// <para></para>
63     /// </param>
64     /// <param name="types">
65     /// <para>The types.</para>
66     /// <para></para>
67     /// </param>
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     private static void AppendTypes(List<Type> container, IList<Type> types)
70     {
71         for (var i = 0; i < types.Count; i++)
72         {
73             var element = types[i];
74             if (element != typeof(Types))
75             {
76                 if (element.IsSubclassOf(typeof(Types)))
77                 {
78                     AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<Type>>(nameof(Types<object>.Collection)));
79                 }
80                 else
81                 {
82                     container.Add(element);
83                 }
84             }
85         }
86     }
87 }
88 }

```

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         /// <summary>
35         /// <para>
36         /// Initializes a new <see cref="Types"/> instance.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         private Types() { }
41     }

```

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             /// <summary>
35             /// <para>
36             /// Initializes a new <see cref="Types"/> instance.
37             /// </para>
38             /// <para></para>
39             /// </summary>
40             private Types() { }
41         }

```

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,
        ↪ T4, T5>().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         /// <summary>
34         /// <para>
35         /// Initializes a new <see cref="Types"/> instance.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         private Types() { }
40     }
41 }

```

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,
        ↪ T4>().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         /// <summary>
34         /// <para>
35         /// Initializes a new <see cref="Types"/> instance.
36         /// </para>

```



```

37     /// <para></para>
38     /// </summary>
39     private Types() { }
40 }
41 }

```

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         /// <summary>
35         /// <para>
36         /// Initializes a new <see cref="Types"/> instance.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         private Types() { }
41     }
42 }

```

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,
26             ↪ T2>().ToReadOnlyCollection();
27         /// <summary>

```

```

27     /// <para>
28     /// Gets the array value.
29     /// </para>
30     /// <para></para>
31     /// </summary>
32     public new static Type[] Array => Collection.ToArray();
33     /// <summary>
34     /// <para>
35     /// Initializes a new <see cref="Types"/> instance.
36     /// </para>
37     /// <para></para>
38     /// </summary>
39     private Types() { }
40 }
41 }

```

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
26             ↳ Types<T>().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         /// <summary>
35         /// <para>
36         /// Initializes a new <see cref="Types"/> instance.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         private Types() { }
41     }
42 }

```

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

```

1  using System;
2  using Xunit;
3
4  namespace Platform.Reflection.Tests
5  {
6     /// <summary>
7     /// <para>
8     /// Represents the code generation tests.
9     /// </para>
10    /// <para></para>
11    /// </summary>
12    public class CodeGenerationTests
13    {
14        /// <summary>
15        /// <para>
16        /// Tests that empty action compilation test.
17        /// </para>

```

```

18     /// <para></para>
19     /// </summary>
20     [Fact]
21     public void EmptyActionCompilationTest()
22     {
23         var compiledAction = DelegateHelpers.Compile<Action>(generator =>
24         {
25             generator.Return();
26         });
27         compiledAction();
28     }
29
30     /// <summary>
31     /// <para>
32     /// Tests that failed action compilation test.
33     /// </para>
34     /// <para></para>
35     /// </summary>
36     /// <exception cref="NotImplementedException">
37     /// <para></para>
38     /// <para></para>
39     /// </exception>
40     [Fact]
41     public void FailedActionCompilationTest()
42     {
43         var compiledAction = DelegateHelpers.Compile<Action>(generator =>
44         {
45             throw new NotImplementedException();
46         });
47         Assert.Throws<NotSupportedException>(compiledAction);
48     }
49
50     /// <summary>
51     /// <para>
52     /// Tests that constant loading test.
53     /// </para>
54     /// <para></para>
55     /// </summary>
56     [Fact]
57     public void ConstantLoadingTest()
58     {
59         CheckConstantLoading<byte>(8);
60         CheckConstantLoading<uint>(8);
61         CheckConstantLoading<ushort>(8);
62         CheckConstantLoading<ulong>(8);
63     }
64
65     /// <summary>
66     /// <para>
67     /// Checks the constant loading using the specified value.
68     /// </para>
69     /// <para></para>
70     /// </summary>
71     /// <typeparam name="T">
72     /// <para>The .</para>
73     /// <para></para>
74     /// </typeparam>
75     /// <param name="value">
76     /// <para>The value.</para>
77     /// <para></para>
78     /// </param>
79     private void CheckConstantLoading<T>(T value)
80     {
81         var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
82         {
83             generator.LoadConstant(value);
84             generator.Return();
85         });
86         Assert.Equal(value, compiledFunction());
87     }
88
89     /// <summary>
90     /// <para>
91     /// Tests that unsigned integers conversion with sign extension test.
92     /// </para>
93     /// <para></para>
94     /// </summary>
95     [Fact]

```

```

96 public void UnsignedIntegersConversionWithSignExtensionTest()
97 {
98     object[] withSignExtension = new object[]
99     {
100         CompileUncheckedConverter<byte, sbyte>(extendSign: true)(128),
101         CompileUncheckedConverter<byte, short>(extendSign: true)(128),
102         CompileUncheckedConverter<ushort, short>(extendSign: true)(32768),
103         CompileUncheckedConverter<byte, int>(extendSign: true)(128),
104         CompileUncheckedConverter<ushort, int>(extendSign: true)(32768),
105         CompileUncheckedConverter<uint, int>(extendSign: true)(2147483648),
106         CompileUncheckedConverter<byte, long>(extendSign: true)(128),
107         CompileUncheckedConverter<ushort, long>(extendSign: true)(32768),
108         CompileUncheckedConverter<uint, long>(extendSign: true)(2147483648),
109         CompileUncheckedConverter<ulong, long>(extendSign: true)(9223372036854775808)
110     };
111     object[] withoutSignExtension = new object[]
112     {
113         CompileUncheckedConverter<byte, sbyte>(extendSign: false)(128),
114         CompileUncheckedConverter<byte, short>(extendSign: false)(128),
115         CompileUncheckedConverter<ushort, short>(extendSign: false)(32768),
116         CompileUncheckedConverter<byte, int>(extendSign: false)(128),
117         CompileUncheckedConverter<ushort, int>(extendSign: false)(32768),
118         CompileUncheckedConverter<uint, int>(extendSign: false)(2147483648),
119         CompileUncheckedConverter<byte, long>(extendSign: false)(128),
120         CompileUncheckedConverter<ushort, long>(extendSign: false)(32768),
121         CompileUncheckedConverter<uint, long>(extendSign: false)(2147483648),
122         CompileUncheckedConverter<ulong, long>(extendSign: false)(9223372036854775808)
123     };
124     var i = 0;
125     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
126     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
127     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
128     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
129     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
130     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
131     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
132     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
133     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
134     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
135 }
136
137 /// <summary>
138 /// <para>
139 /// Tests that signed integers conversion of minus one with sign extension test.
140 /// </para>
141 /// <para></para>
142 /// </summary>
143 [Fact]
144 public void SignedIntegersConversionOfMinusOneWithSignExtensionTest()
145 {
146     object[] withSignExtension = new object[]
147     {
148         CompileUncheckedConverter<sbyte, byte>(extendSign: true)(-1),
149         CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(-1),
150         CompileUncheckedConverter<short, ushort>(extendSign: true)(-1),
151         CompileUncheckedConverter<sbyte, uint>(extendSign: true)(-1),
152         CompileUncheckedConverter<short, uint>(extendSign: true)(-1),
153         CompileUncheckedConverter<int, uint>(extendSign: true)(-1),
154         CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(-1),
155         CompileUncheckedConverter<short, ulong>(extendSign: true)(-1),
156         CompileUncheckedConverter<int, ulong>(extendSign: true)(-1),
157         CompileUncheckedConverter<long, ulong>(extendSign: true)(-1)
158     };
159     object[] withoutSignExtension = new object[]
160     {
161         CompileUncheckedConverter<sbyte, byte>(extendSign: false)(-1),
162         CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(-1),
163         CompileUncheckedConverter<short, ushort>(extendSign: false)(-1),
164         CompileUncheckedConverter<sbyte, uint>(extendSign: false)(-1),
165         CompileUncheckedConverter<short, uint>(extendSign: false)(-1),
166         CompileUncheckedConverter<int, uint>(extendSign: false)(-1),
167         CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(-1),
168         CompileUncheckedConverter<short, ulong>(extendSign: false)(-1),
169         CompileUncheckedConverter<int, ulong>(extendSign: false)(-1),
170         CompileUncheckedConverter<long, ulong>(extendSign: false)(-1)
171     };
172     var i = 0;
173     Assert.Equal((byte)255, (byte)withSignExtension[i]);

```

```

174     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
175     Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
176     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
177     Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
178     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
179     Assert.Equal(4294967295, withSignExtension[i]);
180     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
181     Assert.Equal(4294967295, withSignExtension[i]);
182     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
183     Assert.Equal(4294967295, withSignExtension[i]);
184     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
185     Assert.Equal(18446744073709551615, withSignExtension[i]);
186     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
187     Assert.Equal(18446744073709551615, withSignExtension[i]);
188     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
189     Assert.Equal(18446744073709551615, withSignExtension[i]);
190     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
191     Assert.Equal(18446744073709551615, withSignExtension[i]);
192     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
193 }
194
195 /// <summary>
196 /// <para>
197 /// Tests that signed integers conversion of two with sign extension test.
198 /// </para>
199 /// <para></para>
200 /// </summary>
201 [Fact]
202 public void SignedIntegersConversionOfTwoWithSignExtensionTest()
203 {
204     object[] withSignExtension = new object[]
205     {
206         CompileUncheckedConverter<sbyte, byte>(extendSign: true)(2),
207         CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(2),
208         CompileUncheckedConverter<short, ushort>(extendSign: true)(2),
209         CompileUncheckedConverter<sbyte, uint>(extendSign: true)(2),
210         CompileUncheckedConverter<short, uint>(extendSign: true)(2),
211         CompileUncheckedConverter<int, uint>(extendSign: true)(2),
212         CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(2),
213         CompileUncheckedConverter<short, ulong>(extendSign: true)(2),
214         CompileUncheckedConverter<int, ulong>(extendSign: true)(2),
215         CompileUncheckedConverter<long, ulong>(extendSign: true)(2)
216     };
217     object[] withoutSignExtension = new object[]
218     {
219         CompileUncheckedConverter<sbyte, byte>(extendSign: false)(2),
220         CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(2),
221         CompileUncheckedConverter<short, ushort>(extendSign: false)(2),
222         CompileUncheckedConverter<sbyte, uint>(extendSign: false)(2),
223         CompileUncheckedConverter<short, uint>(extendSign: false)(2),
224         CompileUncheckedConverter<int, uint>(extendSign: false)(2),
225         CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(2),
226         CompileUncheckedConverter<short, ulong>(extendSign: false)(2),
227         CompileUncheckedConverter<int, ulong>(extendSign: false)(2),
228         CompileUncheckedConverter<long, ulong>(extendSign: false)(2)
229     };
230     for (var i = 0; i < withSignExtension.Length; i++)
231     {
232         Assert.Equal(2UL, Convert.ToUInt64(withSignExtension[i]));
233         Assert.Equal(withSignExtension[i], withoutSignExtension[i]);
234     }
235 }
236
237 /// <summary>
238 /// <para>
239 /// Compiles the unchecked converter using the specified extend sign.
240 /// </para>
241 /// <para></para>
242 /// </summary>
243 /// <typeparam name="TSource">
244 /// <para>The source.</para>
245 /// <para></para>
246 /// </typeparam>
247 /// <typeparam name="TTarget">
248 /// <para>The target.</para>
249 /// <para></para>
250 /// </typeparam>
251 /// <param name="extendSign">

```

```

252     /// <para>The extend sign.</para>
253     /// <para></para>
254     /// </param>
255     /// <returns>
256     /// <para>A converter of t source and t target</para>
257     /// <para></para>
258     /// </returns>
259     private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
        ↪ TTarget>(bool extendSign)
260     {
261         return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
262         {
263             generator.LoadArgument(0);
264             generator.UncheckedConvert<TSource, TTarget>(extendSign);
265             generator.Return();
266         });
267     }
268 }
269 }

```

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      /// <summary>
10     /// <para>
11     /// Represents the get il bytes method tests.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     public static class GetILBytesMethodTests
16     {
17         /// <summary>
18         /// <para>
19         /// Tests that il bytes for delegate are available test.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         [Fact]
24         public static void ILBytesForDelegateAreAvailableTest()
25         {
26             var function = new Func<object, int>(argument => 0);
27             var bytes = function.GetMethodInfo().GetILBytes();
28             Assert.False(bytes.IsNullOrEmpty());
29         }
30
31         /// <summary>
32         /// <para>
33         /// Tests that il bytes for different delegates are the same test.
34         /// </para>
35         /// <para></para>
36         /// </summary>
37         [Fact]
38         public static void ILBytesForDifferentDelegatesAreTheSameTest()
39         {
40             var firstFunction = new Func<object, int>(argument => 0);
41             var secondFunction = new Func<object, int>(argument => 0);
42             Assert.False(firstFunction == secondFunction);
43             var firstFunctionBytes = firstFunction.GetMethodInfo().GetILBytes();
44             Assert.False(firstFunctionBytes.IsNullOrEmpty());
45             var secondFunctionBytes = secondFunction.GetMethodInfo().GetILBytes();
46             Assert.False(secondFunctionBytes.IsNullOrEmpty());
47             Assert.True(firstFunctionBytes.EqualTo(secondFunctionBytes));
48         }
49     }
50 }

```

1.23 ./csharp/Platform.Reflection.Tests/NumericTypeTests.cs

```

1  using Xunit;
2
3  namespace Platform.Reflection.Tests
4  {
5      /// <summary>

```

```

6      /// <para>
7      /// Represents the numeric type tests.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     public class NumericTypeTests
12     {
13         /// <summary>
14         /// <para>
15         /// Tests that u int 64 is numeric test.
16         /// </para>
17         /// <para></para>
18         /// </summary>
19         [Fact]
20         public void UInt64IsNumericTest()
21         {
22             Assert.True(NumericType<ulong>.IsNumeric);
23         }
24     }
25 }

```

Index

- ./csharp/Platform.Reflection.Tests/CodeGenerationTests.cs, 58
- ./csharp/Platform.Reflection.Tests/GetILBytesMethodTests.cs, 62
- ./csharp/Platform.Reflection.Tests/NumericTypeTests.cs, 62
- ./csharp/Platform.Reflection/AssemblyExtensions.cs, 1
- ./csharp/Platform.Reflection/DelegateHelpers.cs, 1
- ./csharp/Platform.Reflection/DynamicExtensions.cs, 4
- ./csharp/Platform.Reflection/EnsureExtensions.cs, 5
- ./csharp/Platform.Reflection/FieldInfoExtensions.cs, 14
- ./csharp/Platform.Reflection/ILGeneratorExtensions.cs, 14
- ./csharp/Platform.Reflection/MethodInfoExtensions.cs, 39
- ./csharp/Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 40
- ./csharp/Platform.Reflection/NumericType.cs, 41
- ./csharp/Platform.Reflection/PropertyInfoExtensions.cs, 43
- ./csharp/Platform.Reflection/TypeBuilderExtensions.cs, 44
- ./csharp/Platform.Reflection/TypeExtensions.cs, 46
- ./csharp/Platform.Reflection/Types.cs, 53
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 54
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 55
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 55
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4].cs, 56
- ./csharp/Platform.Reflection/Types[T1, T2, T3].cs, 57
- ./csharp/Platform.Reflection/Types[T1, T2].cs, 57
- ./csharp/Platform.Reflection/Types[T].cs, 58