LinksPlatform's Platform.Reflection Class Library

1.1 ./csharp/Platform.Reflection/AssemblyExtensions.cs

```
using System;
using System.Collections.Concurrent;
using System.Reflection;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the assembly extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class AssemblyExtensions
    {
        private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
        ↪  ConcurrentDictionary<Assembly, Type[]>();

        /// <remarks>
        /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetLoadableTypes(this Assembly assembly)
        {
            Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
            try
            {
                return assembly.GetTypes();
            }
            catch (ReflectionTypeLoadException e)
            {
                return e.Types.ToArray(t => t != null);
            }
        }

        /// <summary>
        /// <para>
        /// Gets the cached loadable types using the specified assembly.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="assembly">
        /// <para>The assembly.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The type array</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
        ↪  _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
    }
}
```

1.2 ./csharp/Platform.Reflection/DelegateHelpers.cs

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Exceptions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the delegate helpers.
    /// </para>
    /// <para></para>
    /// </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>
```

```
        /// <param name="typeMemberMethod">
        /// <para>The type member method.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The delegate.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool typeMemberMethod)
            where TDelegate : Delegate
        {
            var @delegate = CompileOrDefault<TDelegate>(emitCode, typeMemberMethod);
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
                @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
            }
            return @delegate;
        }

        /// <summary>
        /// <para>
        /// Compiles the emit code.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TDelegate">
        /// <para>The delegate.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="emitCode">
        /// <para>The emit code.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The delegate</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate : Delegate => Compile<TDelegate>(emitCode, false);

        /// <summary>
        /// <para>
        /// Compiles the dynamic method using the specified emit code.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TDelegate">
        /// <para>The delegate.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="emitCode">
        /// <para>The emit code.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The delegate</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileDynamicMethod<TDelegate>(Action<ILGenerator> emitCode)
        {
            var delegateType = typeof(TDelegate);
            delegateType.GetDelegateCharacteristics(out Type returnType, out Type[] parameterTypes);
            var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
            emitCode(dynamicMethod.GetILGenerator());
            return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
        }

        /// <summary>
        /// <para>
        /// Compiles the type member method using the specified emit code.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <typeparam name="TDelegate">
        /// <para>The delegate.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="emitCode">
        /// <para>The emit code.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The delegate</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileTypeMemberMethod<TDelegate>(Action<ILGenerator> emitCode)
        {
            AssemblyName assemblyName = new AssemblyName(GetNewName());
            var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
            ↪  AssemblyBuilderAccess.Run);
            var module = assembly.DefineDynamicModule(GetNewName());
            var type = module.DefineType(GetNewName());
            var methodName = GetNewName();
            type.EmitStaticMethod<TDelegate>(methodName, emitCode);
            var typeInfo = type.CreateTypeInfo();
            return (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(typeof(TDele
            ↪  gate));
        }
[MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string GetNewName() => Guid.NewGuid().ToString("N");
    }
}
```

1.3 ./csharp/Platform.Reflection/DynamicExtensions.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the dynamic extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class DynamicExtensions
    {
        /// <summary>
        /// <para>
        /// Determines whether has property.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@object">
        /// <para>The object.</para>
        /// <para></para>
        /// </param>
        /// <param name="propertyName">
        /// <para>The property name.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool HasProperty(this object @object, string propertyName)
        {
            var type = @object.GetType();
            if (type is IDictionary<string, object> dictionary)
            {
                return dictionary.ContainsKey(propertyName);
            }
            return type.GetProperty(propertyName) != null;
        }
    }
}
```

1.4 ./csharp/Platform.Reflection/EnsureExtensions.cs

```
using System;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Exceptions.ExtensionRoots;

#pragma warning disable IDE0060 // Remove unused parameter
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the ensure extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class EnsureExtensions
    {
        #region Always

        /// <summary>
        /// <para>
        /// Ises the unsigned integer using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="messageBuilder">
        /// <para>The message builder.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotSupportedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root,
        ↪  Func<string> messageBuilder)
        {
            if (!NumericType<T>.IsNumeric || NumericType<T>.IsSigned ||
            ↪  NumericType<T>.IsFloatPoint)
            {
                throw new NotSupportedException(messageBuilder());
            }
        }

        /// <summary>
        /// <para>
        /// Ises the unsigned integer using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="message">
        /// <para>The message.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
        ↪  message)
        {
            string messageBuilder() => message;
```

```
            IsUnsignedInteger<T>(root, messageBuilder);
        }

        /// <summary>
        /// <para>
        /// Ises the unsigned integer using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
        ↪  IsUnsignedInteger<T>(root, (string)null);

        /// <summary>
        /// <para>
        /// Ises the signed integer using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="messageBuilder">
        /// <para>The message builder.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotSupportedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
        ↪  messageBuilder)
        {
            if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
            ↪  NumericType<T>.IsFloatPoint)
            {
                throw new NotSupportedException(messageBuilder());
            }
        }

        /// <summary>
        /// <para>
        /// Ises the signed integer using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="message">
        /// <para>The message.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
        ↪  message)
        {
            string messageBuilder() => message;
```

```
148             IsSignedInteger<T>(root, messageBuilder);
149         }
150 
151         /// <summary>
152         /// <para>
153         /// Ises 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         /// Ises 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         /// Ises 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 
```

```
        /// <summary>
        /// <para>
        /// Ises the signed using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
        ↪  (string)null);

        /// <summary>
        /// <para>
        /// Ises the numeric using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="messageBuilder">
        /// <para>The message builder.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotSupportedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
        ↪  messageBuilder)
        {
            if (!NumericType<T>.IsNumeric)
            {
                throw new NotSupportedException(messageBuilder());
            }
        }

        /// <summary>
        /// <para>
        /// Ises the numeric using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="message">
        /// <para>The message.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
        {
            string messageBuilder() => message;
            IsNumeric<T>(root, messageBuilder);
        }

        /// <summary>
        /// <para>
        /// Ises the numeric using the specified root.
```

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

```
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
        ↪  CanBeNumeric<T>(root, (string)null);

        #endregion

        #region OnDebug

        /// <summary>
        /// <para>
        /// Ises the unsigned integer using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="messageBuilder">
        /// <para>The message builder.</para>
        /// <para></para>
        /// </param>
        [Conditional("DEBUG")]
        public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
        ↪  Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);

        /// <summary>
        /// <para>
        /// Ises the unsigned integer using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="message">
        /// <para>The message.</para>
        /// <para></para>
        /// </param>
        [Conditional("DEBUG")]
        public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
        ↪  message) => Ensure.Always.IsUnsignedInteger<T>(message);

        /// <summary>
        /// <para>
        /// Ises the unsigned integer using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        [Conditional("DEBUG")]
        public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
        ↪  Ensure.Always.IsUnsignedInteger<T>();

```

```
450         /// <summary>
451         /// <para>
452         /// Ises the signed integer using the specified root.
453         /// </para>
454         /// <para></para>
455         /// </summary>
456         /// <typeparam name="T">
457         /// <para>The .</para>
458         /// <para></para>
459         /// </typeparam>
460         /// <param name="root">
461         /// <para>The root.</para>
462         /// <para></para>
463         /// </param>
464         /// <param name="messageBuilder">
465         /// <para>The message builder.</para>
466         /// <para></para>
467         /// </param>
468         [Conditional("DEBUG")]
469         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪ messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
470
471         /// <summary>
472         /// <para>
473         /// Ises the signed integer using the specified root.
474         /// </para>
475         /// <para></para>
476         /// </summary>
477         /// <typeparam name="T">
478         /// <para>The .</para>
479         /// <para></para>
480         /// </typeparam>
481         /// <param name="root">
482         /// <para>The root.</para>
483         /// <para></para>
484         /// </param>
485         /// <param name="message">
486         /// <para>The message.</para>
487         /// <para></para>
488         /// </param>
489         [Conditional("DEBUG")]
490         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
            ↪ message) => Ensure.Always.IsSignedInteger<T>(message);
491
492         /// <summary>
493         /// <para>
494         /// Ises the signed integer using the specified root.
495         /// </para>
496         /// <para></para>
497         /// </summary>
498         /// <typeparam name="T">
499         /// <para>The .</para>
500         /// <para></para>
501         /// </typeparam>
502         /// <param name="root">
503         /// <para>The root.</para>
504         /// <para></para>
505         /// </param>
506         [Conditional("DEBUG")]
507         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪ Ensure.Always.IsSignedInteger<T>();
508
509         /// <summary>
510         /// <para>
511         /// Ises the signed using the specified root.
512         /// </para>
513         /// <para></para>
514         /// </summary>
515         /// <typeparam name="T">
516         /// <para>The .</para>
517         /// <para></para>
518         /// </typeparam>
519         /// <param name="root">
520         /// <para>The root.</para>
521         /// <para></para>
522         /// </param>
523         /// <param name="messageBuilder">
524         /// <para>The message builder.</para>
```

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

```
599         /// <param name="root">
600         /// <para>The root.</para>
601         /// <para></para>
602         /// </param>
603         /// <param name="message">
604         /// <para>The message.</para>
605         /// <para></para>
606         /// </param>
607         [Conditional("DEBUG")]
608         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
            ↪ Ensure.Always.IsNumeric<T>(message);
609
610         /// <summary>
611         /// <para>
612         /// 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>
```

```
        /// Throws the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Throw<T>(this ILGenerator generator) =>
        ↪  generator.ThrowException(typeof(T));

        /// <summary>
        /// <para>
        /// Uncheckeds the convert using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TSource">
        /// <para>The source.</para>
        /// <para></para>
        /// </typeparam>
        /// <typeparam name="TTarget">
        /// <para>The target.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator) =>
        ↪  UncheckedConvert<TSource, TTarget>(generator, extendSign: false);

        /// <summary>
        /// <para>
        /// Uncheckeds the convert using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TSource">
        /// <para>The source.</para>
        /// <para></para>
        /// </typeparam>
        /// <typeparam name="TTarget">
        /// <para>The target.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="extendSign">
        /// <para>The extend sign.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator, bool
        ↪  extendSign)
        {
            var sourceType = typeof(TSource);
            var targetType = typeof(TTarget);
            if (sourceType == targetType)
            {
                return;
            }
            if (extendSign)
            {
                if (sourceType == typeof(byte))
                {
                    generator.Emit(OpCodes.Conv_I1);
                }
                if (sourceType == typeof(ushort) || sourceType == typeof(char))
                {
```

```
                    generator.Emit(OpCodes.Conv_I2);
                }
            }
            if (NumericType<TSource>.BitsSize > NumericType<TTarget>.BitsSize)
            {
                generator.ConvertToInteger<TSource>(targetType, extendSign: false);
            }
            else
            {
                generator.ConvertToInteger<TSource>(targetType, extendSign);
            }
            if (targetType == typeof(float))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_R4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_R_Un);
                }
            }
            else if (targetType == typeof(double))
            {
                generator.Emit(OpCodes.Conv_R8);
            }
            else if (targetType == typeof(bool))
            {
                generator.ConvertToBoolean<TSource>();
            }
        }
[MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void ConvertToBoolean<TSource>(this ILGenerator generator)
        {
            generator.LoadConstant<TSource>(default);
            var sourceType = typeof(TSource);
            if (sourceType == typeof(float) || sourceType == typeof(double))
            {
                generator.Emit(OpCodes.Ceq);
                // Inversion of the first Ceq instruction
                generator.LoadConstant<int>(0);
                generator.Emit(OpCodes.Ceq);
            }
            else
            {
                generator.Emit(OpCodes.Cgt_Un);
            }
        }
[MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void ConvertToInteger<TSource>(this ILGenerator generator, Type
        ↪ targetType, bool extendSign)
        {
            if (targetType == typeof(sbyte))
            {
                generator.Emit(OpCodes.Conv_I1);
            }
            else if (targetType == typeof(byte))
            {
                generator.Emit(OpCodes.Conv_U1);
            }
            else if (targetType == typeof(short))
            {
                generator.Emit(OpCodes.Conv_I2);
            }
            else if (targetType == typeof(ushort) || targetType == typeof(char))
            {
                var sourceType = typeof(TSource);
                if (sourceType != typeof(ushort) && sourceType != typeof(char))
                {
                    generator.Emit(OpCodes.Conv_U2);
                }
            }
            else if (targetType == typeof(int))
            {
                generator.Emit(OpCodes.Conv_I4);
            }
            else if (targetType == typeof(uint))
            {
```

```
                generator.Emit(OpCodes.Conv_U4);
            }
            else if (targetType == typeof(long) || targetType == typeof(ulong))
            {
                if (NumericType<TSource>.IsSigned || extendSign)
                {
                    generator.Emit(OpCodes.Conv_I8);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_U8);
                }
            }
        }

        /// <summary>
        /// <para>
        /// Checkeds the convert using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TSource">
        /// <para>The source.</para>
        /// <para></para>
        /// </typeparam>
        /// <typeparam name="TTarget">
        /// <para>The target.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotSupportedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
        {
            var sourceType = typeof(TSource);
            var targetType = typeof(TTarget);
            if (sourceType == targetType)
            {
                return;
            }
            if (targetType == typeof(short))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2_Un);
                }
            }
            else if (targetType == typeof(ushort) || targetType == typeof(char))
            {
                if (sourceType != typeof(ushort) && sourceType != typeof(char))
                {
                    if (NumericType<TSource>.IsSigned)
                    {
                        generator.Emit(OpCodes.Conv_Ovf_U2);
                    }
                    else
                    {
                        generator.Emit(OpCodes.Conv_Ovf_U2_Un);
                    }
                }
            }
            else if (targetType == typeof(sbyte))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I1);
                }
                else
```

```
            {
                generator.Emit(OpCodes.Conv_Ovf_I1_Un);
            }
        }
        else if (targetType == typeof(byte))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_U1);
            }
            else
            {
                generator.Emit(OpCodes.Conv_Ovf_U1_Un);
            }
        }
        else if (targetType == typeof(int))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_I4);
            }
            else
            {
                generator.Emit(OpCodes.Conv_Ovf_I4_Un);
            }
        }
        else if (targetType == typeof(uint))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_U4);
            }
            else
            {
                generator.Emit(OpCodes.Conv_Ovf_U4_Un);
            }
        }
        else if (targetType == typeof(long))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_I8);
            }
            else
            {
                generator.Emit(OpCodes.Conv_Ovf_I8_Un);
            }
        }
        else if (targetType == typeof(ulong))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_Ovf_U8);
            }
            else
            {
                generator.Emit(OpCodes.Conv_Ovf_U8_Un);
            }
        }
        else if (targetType == typeof(float))
        {
            if (NumericType<TSource>.IsSigned)
            {
                generator.Emit(OpCodes.Conv_R4);
            }
            else
            {
                generator.Emit(OpCodes.Conv_R_Un);
            }
        }
        else if (targetType == typeof(double))
        {
            generator.Emit(OpCodes.Conv_R8);
        }
        else if (targetType == typeof(bool))
        {
            generator.ConvertToBoolean<TSource>();
        }
```

```
            else
            {
                throw new NotSupportedException();
            }
        }

        /// <summary>
        /// <para>
        /// Loads the constant using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, bool value) =>
        ↪  generator.LoadConstant(value ? 1 : 0);

        /// <summary>
        /// <para>
        /// Loads the constant using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, float value) =>
        ↪  generator.Emit(OpCodes.Ldc_R4, value);

        /// <summary>
        /// <para>
        /// Loads the constant using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, double value) =>
        ↪  generator.Emit(OpCodes.Ldc_R8, value);

        /// <summary>
        /// <para>
        /// Loads the constant using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, ulong value) =>
        ↪  generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));

```

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

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

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

```
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="constantType">
        /// <para>The constant type.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotSupportedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstantOne(this ILGenerator generator, Type constantType)
        {
            if (constantType == typeof(float))
            {
                generator.LoadConstant(1F);
            }
            else if (constantType == typeof(double))
            {
                generator.LoadConstant(1D);
            }
            else if (constantType == typeof(long))
            {
                generator.LoadConstant(1L);
            }
            else if (constantType == typeof(ulong))
            {
                generator.LoadConstant(1UL);
            }
            else if (constantType == typeof(int))
            {
                generator.LoadConstant(1);
            }
            else if (constantType == typeof(uint))
            {
                generator.LoadConstant(1U);
            }
            else if (constantType == typeof(short))
            {
                generator.LoadConstant((short)1);
            }
            else if (constantType == typeof(ushort))
            {
                generator.LoadConstant((ushort)1);
            }
            else if (constantType == typeof(sbyte))
            {
                generator.LoadConstant((sbyte)1);
            }
            else if (constantType == typeof(byte))
            {
                generator.LoadConstant((byte)1);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        /// <summary>
        /// <para>
        /// Loads the constant using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TConstant">
        /// <para>The constant.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="constantValue">
        /// <para>The constant value.</para>
        /// <para></para>
```

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

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

```
        /// <para></para>
        /// </param>
        /// <param name="valueType">
        /// <para>The value type.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Decrement(this ILGenerator generator, Type valueType)
        {
            generator.LoadConstantOne(valueType);
            generator.Subtract();
        }

        /// <summary>
        /// <para>
        /// Subtracts the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);

        /// <summary>
        /// <para>
        /// Negates the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);

        /// <summary>
        /// <para>
        /// Ands the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);

        /// <summary>
        /// <para>
        /// Ors the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);

        /// <summary>
        /// <para>
        /// Nots the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);

```

```
        /// <summary>
        /// <para>
        /// Shifts the left using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);

        /// <summary>
        /// <para>
        /// Shifts the right using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ShiftRight<T>(this ILGenerator generator)
        {
            generator.Emit(NumericType<T>.IsSigned ? OpCodes.Shr : OpCodes.Shr_Un);
        }

        /// <summary>
        /// <para>
        /// Loads the argument using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="argumentIndex">
        /// <para>The argument index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadArgument(this ILGenerator generator, int argumentIndex)
        {
            switch (argumentIndex)
            {
                case 0:
                    generator.Emit(OpCodes.Ldarg_0);
                    break;
                case 1:
                    generator.Emit(OpCodes.Ldarg_1);
                    break;
                case 2:
                    generator.Emit(OpCodes.Ldarg_2);
                    break;
                case 3:
                    generator.Emit(OpCodes.Ldarg_3);
                    break;
                default:
                    generator.Emit(OpCodes.Ldarg, argumentIndex);
                    break;
            }
        }

        /// <summary>
        /// <para>
        /// Loads the arguments using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
```

```
        /// </param>
        /// <param name="argumentIndices">
        /// <para>The argument indices.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadArguments(this ILGenerator generator, params int[]
        ↪  argumentIndices)
        {
            for (var i = 0; i < argumentIndices.Length; i++)
            {
                generator.LoadArgument(argumentIndices[i]);
            }
        }

        /// <summary>
        /// <para>
        /// Stores the argument using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="argumentIndex">
        /// <para>The argument index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
        ↪  generator.Emit(OpCodes.Starg, argumentIndex);

        /// <summary>
        /// <para>
        /// Compares the greater than using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareGreaterThan(this ILGenerator generator) =>
        ↪  generator.Emit(OpCodes.Cgt);

        /// <summary>
        /// <para>
        /// Unsigneds the compare greater than using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
        ↪  generator.Emit(OpCodes.Cgt_Un);

        /// <summary>
        /// <para>
        /// Compares the greater than using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="isSigned">
        /// <para>The is signed.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
        {
```

```
            if (isSigned)
            {
                generator.CompareGreaterThan();
            }
            else
            {
                generator.UnsignedCompareGreaterThan();
            }
        }

        /// <summary>
        /// <para>
        /// Compares the less than using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareLessThan(this ILGenerator generator) =>
        ↪  generator.Emit(OpCodes.Clt);

        /// <summary>
        /// <para>
        /// Unsigneds the compare less than using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnsignedCompareLessThan(this ILGenerator generator) =>
        ↪  generator.Emit(OpCodes.Clt_Un);

        /// <summary>
        /// <para>
        /// Compares the less than using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="isSigned">
        /// <para>The is signed.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareLessThan(this ILGenerator generator, bool isSigned)
        {
            if (isSigned)
            {
                generator.CompareLessThan();
            }
            else
            {
                generator.UnsignedCompareLessThan();
            }
        }

        /// <summary>
        /// <para>
        /// Branches the if greater or equal using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="label">
        /// <para>The label.</para>
        /// <para></para>
```

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

```
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="label">
        /// <para>The label.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
        ↪  => generator.Emit(OpCodes.Ble_Un, label);

        /// <summary>
        /// <para>
        /// Branches the if less or equal using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="isSigned">
        /// <para>The is signed.</para>
        /// <para></para>
        /// </param>
        /// <param name="label">
        /// <para>The label.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
        ↪  label)
        {
            if (isSigned)
            {
                generator.BranchIfLessOrEqual(label);
            }
            else
            {
                generator.UnsignedBranchIfLessOrEqual(label);
            }
        }

        /// <summary>
        /// <para>
        /// Boxes the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TBox">
        /// <para>The box.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));

        /// <summary>
        /// <para>
        /// Boxes the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="boxedType">
        /// <para>The boxed type.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
1321            public static void Box(this ILGenerator generator, Type boxedType) =>
                ↪  generator.Emit(OpCodes.Box, boxedType);
1322
1323            /// <summary>
1324            /// <para>
1325            /// Calls the generator.
1326            /// </para>
1327            /// <para></para>
1328            /// </summary>
1329            /// <param name="generator">
1330            /// <para>The generator.</para>
1331            /// <para></para>
1332            /// </param>
1333            /// <param name="method">
1334            /// <para>The method.</para>
1335            /// <para></para>
1336            /// </param>
1337            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1338            public static void Call(this ILGenerator generator, MethodInfo method) =>
                ↪  generator.Emit(OpCodes.Call, method);
1339
1340            /// <summary>
1341            /// <para>
1342            /// Returns the generator.
1343            /// </para>
1344            /// <para></para>
1345            /// </summary>
1346            /// <param name="generator">
1347            /// <para>The generator.</para>
1348            /// <para></para>
1349            /// </param>
1350            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1351            public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
1352
1353            /// <summary>
1354            /// <para>
1355            /// Unboxes the generator.
1356            /// </para>
1357            /// <para></para>
1358            /// </summary>
1359            /// <typeparam name="TUnbox">
1360            /// <para>The unbox.</para>
1361            /// <para></para>
1362            /// </typeparam>
1363            /// <param name="generator">
1364            /// <para>The generator.</para>
1365            /// <para></para>
1366            /// </param>
1367            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1368            public static void Unbox<TUnbox>(this ILGenerator generator) =>
                ↪  generator.Unbox(typeof(TUnbox));
1369
1370            /// <summary>
1371            /// <para>
1372            /// Unboxes the generator.
1373            /// </para>
1374            /// <para></para>
1375            /// </summary>
1376            /// <param name="generator">
1377            /// <para>The generator.</para>
1378            /// <para></para>
1379            /// </param>
1380            /// <param name="typeToUnbox">
1381            /// <para>The type to unbox.</para>
1382            /// <para></para>
1383            /// </param>
1384            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1385            public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
                ↪  generator.Emit(OpCodes.Unbox, typeToUnbox);
1386
1387            /// <summary>
1388            /// <para>
1389            /// Unboxes the value using the specified generator.
1390            /// </para>
1391            /// <para></para>
1392            /// </summary>
1393            /// <typeparam name="TUnbox">
1394            /// <para>The unbox.</para>
```

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

```
1469          /// <param name="local">
1470          /// <para>The local.</para>
1471          /// <para></para>
1472          /// </param>
1473          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1474          public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
              ↪ generator.Emit(OpCodes.Stloc, local);
1475
1476          /// <summary>
1477          /// <para>
1478          /// News the object using the specified generator.
1479          /// </para>
1480          /// <para></para>
1481          /// </summary>
1482          /// <param name="generator">
1483          /// <para>The generator.</para>
1484          /// <para></para>
1485          /// </param>
1486          /// <param name="type">
1487          /// <para>The type.</para>
1488          /// <para></para>
1489          /// </param>
1490          /// <param name="parameterTypes">
1491          /// <para>The parameter types.</para>
1492          /// <para></para>
1493          /// </param>
1494          /// <exception cref="InvalidOperationException">
1495          /// <para></para>
1496          /// <para></para>
1497          /// </exception>
1498          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1499          public static void NewObject(this ILGenerator generator, Type type, params Type[]
              ↪ parameterTypes)
1500          {
1501              var allConstructors = type.GetConstructors(BindingFlags.Public |
                  ↪ BindingFlags.NonPublic | BindingFlags.Instance
1502  #if !NETSTANDARD
1503                  | BindingFlags.CreateInstance
1504  #endif
1505                  );
1506              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();
1507              if (constructor == null)
1508              {
1509                  throw new InvalidOperationException("Type " + type + " must have a constructor
                      ↪ that matches parameters [" + string.Join(", ",
                      ↪ parameterTypes.AsEnumerable()) + "]");
1510              }
1511              generator.NewObject(constructor);
1512          }
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="constructor">
1525          /// <para>The constructor.</para>
1526          /// <para></para>
1527          /// </param>
1528          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1529          public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
              ↪ generator.Emit(OpCodes.Newobj, constructor);
1530
1531          /// <summary>
1532          /// <para>
1533          /// Loads the indirect using the specified generator.
1534          /// </para>
1535          /// <para></para>
1536          /// </summary>
1537          /// <typeparam name="T">
```

```
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="isVolatile">
        /// <para>The is volatile.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaligned">
        /// <para>The unaligned.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
        ↪  byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);

        /// <summary>
        /// <para>
        /// Loads the indirect using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="isVolatile">
        /// <para>The is volatile.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaligned">
        /// <para>The unaligned.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        /// <exception cref="ArgumentException">
        /// <para>unaligned must be null, 1, 2, or 4</para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
        ↪  false, byte? unaligned = null)
        {
            if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
            {
                throw new ArgumentException("unaligned must be null, 1, 2, or 4");
            }
            if (isVolatile)
            {
                generator.Emit(OpCodes.Volatile);
            }
            if (unaligned.HasValue)
            {
                generator.Emit(OpCodes.Unaligned, unaligned.Value);
            }
            if (type.IsPointer)
            {
                generator.Emit(OpCodes.Ldind_I);
            }
            else if (!type.IsValueType)
            {
                generator.Emit(OpCodes.Ldind_Ref);
            }
            else if (type == typeof(sbyte))
            {
                generator.Emit(OpCodes.Ldind_I1);
            }
```

```
            else if (type == typeof(bool))
            {
                generator.Emit(OpCodes.Ldind_I1);
            }
            else if (type == typeof(byte))
            {
                generator.Emit(OpCodes.Ldind_U1);
            }
            else if (type == typeof(short))
            {
                generator.Emit(OpCodes.Ldind_I2);
            }
            else if (type == typeof(ushort))
            {
                generator.Emit(OpCodes.Ldind_U2);
            }
            else if (type == typeof(char))
            {
                generator.Emit(OpCodes.Ldind_U2);
            }
            else if (type == typeof(int))
            {
                generator.Emit(OpCodes.Ldind_I4);
            }
            else if (type == typeof(uint))
            {
                generator.Emit(OpCodes.Ldind_U4);
            }
            else if (type == typeof(long) || type == typeof(ulong))
            {
                generator.Emit(OpCodes.Ldind_I8);
            }
            else if (type == typeof(float))
            {
                generator.Emit(OpCodes.Ldind_R4);
            }
            else if (type == typeof(double))
            {
                generator.Emit(OpCodes.Ldind_R8);
            }
            else
            {
                throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
                ↪  ", LoadObject may be more appropriate");
            }
        }

        /// <summary>
        /// <para>
        /// Stores the indirect using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="isVolatile">
        /// <para>The is volatile.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaligned">
        /// <para>The unaligned.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
        ↪  byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);

        /// <summary>
        /// <para>
        /// Stores the indirect using the specified generator.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="isVolatile">
        /// <para>The is volatile.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaligned">
        /// <para>The unaligned.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        /// <exception cref="ArgumentException">
        /// <para>unaligned must be null, 1, 2, or 4</para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile = false, byte? unaligned = null)
        {
            if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
            {
                throw new ArgumentException("unaligned must be null, 1, 2, or 4");
            }
            if (isVolatile)
            {
                generator.Emit(OpCodes.Volatile);
            }
            if (unaligned.HasValue)
            {
                generator.Emit(OpCodes.Unaligned, unaligned.Value);
            }
            if (type.IsPointer)
            {
                generator.Emit(OpCodes.Stind_I);
            }
            else if (!type.IsValueType)
            {
                generator.Emit(OpCodes.Stind_Ref);
            }
            else if (type == typeof(sbyte) || type == typeof(byte))
            {
                generator.Emit(OpCodes.Stind_I1);
            }
            else if (type == typeof(short) || type == typeof(ushort))
            {
                generator.Emit(OpCodes.Stind_I2);
            }
            else if (type == typeof(int) || type == typeof(uint))
            {
                generator.Emit(OpCodes.Stind_I4);
            }
            else if (type == typeof(long) || type == typeof(ulong))
            {
                generator.Emit(OpCodes.Stind_I8);
            }
            else if (type == typeof(float))
            {
                generator.Emit(OpCodes.Stind_R4);
            }
            else if (type == typeof(double))
            {
                generator.Emit(OpCodes.Stind_R8);
            }
            else
            {
                throw new InvalidOperationException("StoreIndirect cannot be used with " + type + ", StoreObject may be more appropriate");
```

```
            }
        }

        /// <summary>
        /// <para>
        /// Multiplies the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Multiply(this ILGenerator generator)
        {
            generator.Emit(OpCodes.Mul);
        }

        /// <summary>
        /// <para>
        /// Checkeds the multiply using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CheckedMultiply<T>(this ILGenerator generator)
        {
            if (NumericType<T>.IsSigned)
            {
                generator.Emit(OpCodes.Mul_Ovf);
            }
            else
            {
                generator.Emit(OpCodes.Mul_Ovf_Un);
            }
        }

        /// <summary>
        /// <para>
        /// Divides the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Divide<T>(this ILGenerator generator)
        {
            if (NumericType<T>.IsSigned)
            {
                generator.Emit(OpCodes.Div);
            }
            else
            {
                generator.Emit(OpCodes.Div_Un);
            }
        }
    }
}
```

1.7 ./csharp/Platform.Reflection/MethodInfoExtensions.cs

```
using System;
using System.Linq;
```

```
using System.Reflection;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the method info extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class MethodInfoExtensions
    {
        /// <summary>
        /// <para>
        /// Gets the il bytes using the specified method info.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="methodInfo">
        /// <para>The method info.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The byte array</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static byte[] GetILBytes(this MethodInfo methodInfo) =>
        ↪  methodInfo.GetMethodBody().GetILAsByteArray();

        /// <summary>
        /// <para>
        /// Gets the parameter types using the specified method info.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="methodInfo">
        /// <para>The method info.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The type array</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
        ↪  methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
    }
}
```

1.8 ./csharp/Platform.Reflection/NotSupportedExceptionDelegateFactory.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the not supported exception delegate factory.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IFactory{TDelegate}"/>
    public class NotSupportedExceptionDelegateFactory<TDelegate> : IFactory<TDelegate>
        where TDelegate : Delegate
    {
        /// <summary>
        /// <para>
        /// Creates this instance.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <exception cref="InvalidOperationException">
        /// <para>Unable to compile stub delegate.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The delegate.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TDelegate Create()
        {
            var @delegate = DelegateHelpers.CompileOrDefault<TDelegate>(generator =>
            {
                generator.Throw<NotSupportedException>();
            });
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
                throw new InvalidOperationException("Unable to compile stub delegate.");
            }
            return @delegate;
        }
    }
}
```

1.9 ./csharp/Platform.Reflection/NumericType.cs

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using Platform.Exceptions;

// ReSharper disable AssignmentInConditionalExpression
// ReSharper disable BuiltInTypeReferenceStyle
// ReSharper disable StaticFieldInGenericType
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the numeric type.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class NumericType<T>
    {
        /// <summary>
        /// <para>
        /// The type.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly Type Type;
        /// <summary>
        /// <para>
        /// The underlying type.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly Type UnderlyingType;
        /// <summary>
        /// <para>
        /// The signed version.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly Type SignedVersion;
        /// <summary>
        /// <para>
        /// The unsigned version.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly Type UnsignedVersion;
        /// <summary>
        /// <para>
        /// The is float point.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        public static readonly bool IsFloatPoint;
        /// <summary>
        /// <para>
        /// The is numeric.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly bool IsNumeric;
        /// <summary>
        /// <para>
        /// The is signed.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly bool IsSigned;
        /// <summary>
        /// <para>
        /// The can be numeric.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly bool CanBeNumeric;
        /// <summary>
        /// <para>
        /// The is nullable.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly bool IsNullable;
        /// <summary>
        /// <para>
        /// The bytes size.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly int BytesSize;
        /// <summary>
        /// <para>
        /// The bits size.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly int BitsSize;
        /// <summary>
        /// <para>
        /// The min value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly T MinValue;
        /// <summary>
        /// <para>
        /// The max value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly T MaxValue;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="NumericType"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static NumericType()
        {
            try
            {
                var type = typeof(T);
                var isNullable = type.IsNullable();
                var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
                var canBeNumeric = underlyingType.CanBeNumeric();
                var isNumeric = underlyingType.IsNumeric();
                var isSigned = underlyingType.IsSigned();
                var isFloatPoint = underlyingType.IsFloatPoint();
                var bytesSize = Marshal.SizeOf(underlyingType);
                var bitsSize = bytesSize * 8;
```

```
                GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
                GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
                ↪  out Type unsignedVersion);
                Type = type;
                IsNullable = isNullable;
                UnderlyingType = underlyingType;
                CanBeNumeric = canBeNumeric;
                IsNumeric = isNumeric;
                IsSigned = isSigned;
                IsFloatPoint = isFloatPoint;
                BytesSize = bytesSize;
                BitsSize = bitsSize;
                MinValue = minValue;
                MaxValue = maxValue;
                SignedVersion = signedVersion;
                UnsignedVersion = unsignedVersion;
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
        }
[MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
        {
            if (type == typeof(bool))
            {
                minValue = (T)(object)false;
                maxValue = (T)(object)true;
            }
            else
            {
                minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
                maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
            }
        }
[MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
        ↪  signedVersion, out Type unsignedVersion)
        {
            if (isSigned)
            {
                signedVersion = type;
                unsignedVersion = type.GetUnsignedVersionOrNull();
            }
            else
            {
                signedVersion = type.GetSignedVersionOrNull();
                unsignedVersion = type;
            }
        }
    }
}
```

1.10 ./csharp/Platform.Reflection/PropertyInfoExtensions.cs

```
using System.Reflection;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the property info extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class PropertyInfoExtensions
    {
        /// <summary>
        /// <para>
        /// Gets the static value using the specified field info.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
```

```
        /// </typeparam>
        /// <param name="fieldInfo">
        /// <para>The field info.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticValue<T>(this PropertyInfo fieldInfo) =>
        ↪  (T)fieldInfo.GetValue(null);
    }
}
```

1.11 ./csharp/Platform.Reflection/TypeBuilderExtensions.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the type builder extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class TypeBuilderExtensions
    {
        /// <summary>
        /// <para>
        /// The static.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly MethodAttributes DefaultStaticMethodAttributes =
        ↪  MethodAttributes.Public | MethodAttributes.Static;
        /// <summary>
        /// <para>
        /// The hide by sig.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes =
        ↪  MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
        ↪  MethodAttributes.HideBySig;
        /// <summary>
        /// <para>
        /// The aggressive inlining.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly MethodImplAttributes DefaultMethodImplAttributes =
        ↪  MethodImplAttributes.IL | MethodImplAttributes.Managed |
        ↪  MethodImplAttributes.AggressiveInlining;

        /// <summary>
        /// <para>
        /// Emits the method using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TDelegate">
        /// <para>The delegate.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="methodName">
        /// <para>The method name.</para>
        /// <para></para>
        /// </param>
```

```
        /// <param name="methodAttributes">
        /// <para>The method attributes.</para>
        /// <para></para>
        /// </param>
        /// <param name="methodImplAttributes">
        /// <para>The method impl attributes.</para>
        /// <para></para>
        /// </param>
        /// <param name="emitCode">
        /// <para>The emit code.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName,
        ↪  MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes,
        ↪  Action<ILGenerator> emitCode)
        {
            typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[]
            ↪  parameterTypes);
            EmitMethod(type, methodName, methodAttributes, methodImplAttributes, returnType,
            ↪  parameterTypes, emitCode);
        }

        /// <summary>
        /// <para>
        /// Emits the method using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="methodName">
        /// <para>The method name.</para>
        /// <para></para>
        /// </param>
        /// <param name="methodAttributes">
        /// <para>The method attributes.</para>
        /// <para></para>
        /// </param>
        /// <param name="methodImplAttributes">
        /// <para>The method impl attributes.</para>
        /// <para></para>
        /// </param>
        /// <param name="returnType">
        /// <para>The return type.</para>
        /// <para></para>
        /// </param>
        /// <param name="parameterTypes">
        /// <para>The parameter types.</para>
        /// <para></para>
        /// </param>
        /// <param name="emitCode">
        /// <para>The emit code.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
        ↪  methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[]
        ↪  parameterTypes, Action<ILGenerator> emitCode)
        {
            MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
            ↪  parameterTypes);
            method.SetImplementationFlags(methodImplAttributes);
            var generator = method.GetILGenerator();
            emitCode(generator);
        }

        /// <summary>
        /// <para>
        /// Emits the static method using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TDelegate">
        /// <para>The delegate.</para>
```

```
        /// <para></para>
        /// </typeparam>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="methodName">
        /// <para>The method name.</para>
        /// <para></para>
        /// </param>
        /// <param name="emitCode">
        /// <para>The emit code.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName,
        ↪  Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
        ↪  DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);

        /// <summary>
        /// <para>
        /// Emits the final virtual method using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TDelegate">
        /// <para>The delegate.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="methodName">
        /// <para>The method name.</para>
        /// <para></para>
        /// </param>
        /// <param name="emitCode">
        /// <para>The emit code.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitFinalVirtualMethod<TDelegate>(this TypeBuilder type, string
        ↪  methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
        ↪  DefaultFinalVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
    }
}
```

1.12 ./csharp/Platform.Reflection/TypeExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Runtime.CompilerServices;
using Platform.Collections;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the type extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class TypeExtensions
    {
        /// <summary>
        /// <para>
        /// The static.
        /// </para>
        /// <para></para>
        /// </summary>
        static public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
        ↪  BindingFlags.NonPublic | BindingFlags.Static;
        /// <summary>
        /// <para>
```

```
        /// The default delegate method name.
        /// </para>
        /// <para></para>
        /// </summary>
        static public readonly string DefaultDelegateMethodName = "Invoke";

        /// <summary>
        /// <para>
        /// The can be numeric types.
        /// </para>
        /// <para></para>
        /// </summary>
        static private readonly HashSet<Type> _canBeNumericTypes;
        /// <summary>
        /// <para>
        /// The is numeric types.
        /// </para>
        /// <para></para>
        /// </summary>
        static private readonly HashSet<Type> _isNumericTypes;
        /// <summary>
        /// <para>
        /// The is signed types.
        /// </para>
        /// <para></para>
        /// </summary>
        static private readonly HashSet<Type> _isSignedTypes;
        /// <summary>
        /// <para>
        /// The is float point types.
        /// </para>
        /// <para></para>
        /// </summary>
        static private readonly HashSet<Type> _isFloatPointTypes;
        /// <summary>
        /// <para>
        /// The unsigned versions of signed types.
        /// </para>
        /// <para></para>
        /// </summary>
        static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
        /// <summary>
        /// <para>
        /// The signed versions of unsigned types.
        /// </para>
        /// <para></para>
        /// </summary>
        static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TypeExtensions"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static TypeExtensions()
        {
            _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
            ↪  typeof(DateTime), typeof(TimeSpan) };
            _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
            ↪  typeof(ulong) };
            _canBeNumericTypes.UnionWith(_isNumericTypes);
            _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
            ↪  typeof(long) };
            _canBeNumericTypes.UnionWith(_isSignedTypes);
            _isNumericTypes.UnionWith(_isSignedTypes);
            _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
            ↪  typeof(float) };
            _canBeNumericTypes.UnionWith(_isFloatPointTypes);
            _isNumericTypes.UnionWith(_isFloatPointTypes);
            _isSignedTypes.UnionWith(_isFloatPointTypes);
            _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
            {
                { typeof(sbyte), typeof(byte) },
                { typeof(short), typeof(ushort) },
                { typeof(int), typeof(uint) },
                { typeof(long), typeof(ulong) },
```

```
            };
            _signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
            {
                { typeof(byte), typeof(sbyte)},
                { typeof(ushort), typeof(short) },
                { typeof(uint), typeof(int) },
                { typeof(ulong), typeof(long) },
            };
        }

        /// <summary>
        /// <para>
        /// Gets the first field using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The field info</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];

        /// <summary>
        /// <para>
        /// Gets the static field value using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="name">
        /// <para>The name.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticFieldValue<T>(this Type type, string name) =>
        ↪  type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();

        /// <summary>
        /// <para>
        /// Gets the static property value using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="name">
        /// <para>The name.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticPropertyValue<T>(this Type type, string name) =>
        ↪  type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();
```

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

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

```
        /// <para>
        /// Determines whether is nullable.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));

        /// <summary>
        /// <para>
        /// Gets the unsigned version or null using the specified signed type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="signedType">
        /// <para>The signed type.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The type</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetUnsignedVersionOrNull(this Type signedType) =>
        ↪  _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);

        /// <summary>
        /// <para>
        /// Gets the signed version or null using the specified unsigned type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="unsignedType">
        /// <para>The unsigned type.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The type</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetSignedVersionOrNull(this Type unsignedType) =>
        ↪  _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);

        /// <summary>
        /// <para>
        /// Determines whether can be numeric.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);

        /// <summary>
        /// <para>
        /// Determines whether is numeric.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);

        /// <summary>
        /// <para>
        /// Determines whether is signed.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);

        /// <summary>
        /// <para>
        /// Determines whether is float point.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);

        /// <summary>
        /// <para>
        /// Gets the delegate return type using the specified delegate type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="delegateType">
        /// <para>The delegate type.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The type</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetDelegateReturnType(this Type delegateType) =>
        ↪  delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;

        /// <summary>
        /// <para>
        /// Gets the delegate parameter types using the specified delegate type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="delegateType">
        /// <para>The delegate type.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The type array</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
        ↪  delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();

```

```
        /// <summary>
        /// <para>
        /// Gets the delegate characteristics using the specified delegate type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="delegateType">
        /// <para>The delegate type.</para>
        /// <para></para>
        /// </param>
        /// <param name="returnType">
        /// <para>The return type.</para>
        /// <para></para>
        /// </param>
        /// <param name="parameterTypes">
        /// <para>The parameter types.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetDelegateCharacteristics(this Type delegateType, out Type
        ↪  returnType, out Type[] parameterTypes)
        {
            var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
            returnType = invoke.ReturnType;
            parameterTypes = invoke.GetParameterTypes();
        }
    }
}
```

1.13 ./csharp/Platform.Reflection/Types.cs

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Runtime.CompilerServices;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the types.
    /// </para>
    /// <para></para>
    /// </summary>
    public abstract class Types
    {
        /// <summary>
        /// <para>
        /// Gets the collection value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static ReadOnlyCollection<Type> Collection { get; } = new
        ↪  ReadOnlyCollection<Type>(System.Array.Empty<Type>());
        /// <summary>
        /// <para>
        /// Gets the array value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Type[] Array => Collection.ToArray();

        /// <summary>
        /// <para>
        /// Returns the read only collection.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A read only collection of type</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected ReadOnlyCollection<Type> ToReadOnlyCollection()
        {
```

```
            var types = GetType().GetGenericArguments();
            var result = new List<Type>();
            AppendTypes(result, types);
            return new ReadOnlyCollection<Type>(result);
        }
[MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void AppendTypes(List<Type> container, IList<Type> types)
        {
            for (var i = 0; i < types.Count; i++)
            {
                var element = types[i];
                if (element != typeof(Types))
                {
                    if (element.IsSubclassOf(typeof(Types)))
                    {
                        AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection
                        ↪  <Type>>(nameof(Types<object>.Collection)));
                    }
                    else
                    {
                        container.Add(element);
                    }
                }
            }
        }
    }
}
```

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

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the types.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="Types"/>
    public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
    {
        /// <summary>
        /// <para>
        /// Gets the collection value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
        ↪  T4, T5, T6, T7>().ToReadOnlyCollection();
        /// <summary>
        /// <para>
        /// Gets the array value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
```

```
    /// Represents the types.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="Types"/>
    public class Types<T1, T2, T3, T4, T5, T6> : Types
    {
        /// <summary>
        /// <para>
        /// Gets the collection value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
        ↪  T4, T5, T6>().ToReadOnlyCollection();
        /// <summary>
        /// <para>
        /// Gets the array value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the types.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="Types"/>
    public class Types<T1, T2, T3, T4, T5> : Types
    {
        /// <summary>
        /// <para>
        /// Gets the collection value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
        ↪  T4, T5>().ToReadOnlyCollection();
        /// <summary>
        /// <para>
        /// Gets the array value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.17 ./csharp/Platform.Reflection/Types[T1, T2, T3, T4].cs

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the types.
    /// </para>
```

```
    /// <para></para>
    /// </summary>
    /// <seealso cref="Types"/>
    public class Types<T1, T2, T3, T4> : Types
    {
        /// <summary>
        /// <para>
        /// Gets the collection value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
        ↪  T4>().ToReadOnlyCollection();
        /// <summary>
        /// <para>
        /// Gets the array value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.18 ./csharp/Platform.Reflection/Types[T1, T2, T3].cs

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the types.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="Types"/>
    public class Types<T1, T2, T3> : Types
    {
        /// <summary>
        /// <para>
        /// Gets the collection value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2,
        ↪  T3>().ToReadOnlyCollection();
        /// <summary>
        /// <para>
        /// Gets the array value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.19 ./csharp/Platform.Reflection/Types[T1, T2].cs

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the types.
    /// </para>
    /// <para></para>
    /// </summary>
```

```
    /// <seealso cref="Types"/>
    public class Types<T1, T2> : Types
    {
        /// <summary>
        /// <para>
        /// Gets the collection value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1,
        ↪  T2>().ToReadOnlyCollection();
        /// <summary>
        /// <para>
        /// Gets the array value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.20 ./csharp/Platform.Reflection/Types[T].cs

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the types.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="Types"/>
    public class Types<T> : Types
    {
        /// <summary>
        /// <para>
        /// Gets the collection value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static ReadOnlyCollection<Type> Collection { get; } = new
        ↪  Types<T>().ToReadOnlyCollection();
        /// <summary>
        /// <para>
        /// Gets the array value.
        /// </para>
        /// <para></para>
        /// </summary>
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
using Xunit;

namespace Platform.Reflection.Tests
{
    /// <summary>
    /// <para>
    /// Represents the code generation tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public class CodeGenerationTests
    {
        /// <summary>
        /// <para>
        /// Tests that empty action compilation test.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        [Fact]
        public void EmptyActionCompilationTest()
        {
            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                generator.Return();
            });
            compiledAction();
        }

        /// <summary>
        /// <para>
        /// Tests that failed action compilation test.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <exception cref="NotImplementedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [Fact]
        public void FailedActionCompilationTest()
        {
            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                throw new NotImplementedException();
            });
            Assert.Throws<NotSupportedException>(compiledAction);
        }

        /// <summary>
        /// <para>
        /// Tests that constant loading test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void ConstantLoadingTest()
        {
            CheckConstantLoading<byte>(8);
            CheckConstantLoading<uint>(8);
            CheckConstantLoading<ushort>(8);
            CheckConstantLoading<ulong>(8);
        }
        private void CheckConstantLoading<T>(T value)
        {
            var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
            {
                generator.LoadConstant(value);
                generator.Return();
            });
            Assert.Equal(value, compiledFunction());
        }

        /// <summary>
        /// <para>
        /// Tests that unsigned integers conversion with sign extension test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void UnsignedIntegersConversionWithSignExtensionTest()
        {
            object[] withSignExtension = new object[]
            {
                CompileUncheckedConverter<byte, sbyte>(extendSign: true)(128),
                CompileUncheckedConverter<byte, short>(extendSign: true)(128),
                CompileUncheckedConverter<ushort, short>(extendSign: true)(32768),
                CompileUncheckedConverter<byte, int>(extendSign: true)(128),
                CompileUncheckedConverter<ushort, int>(extendSign: true)(32768),
                CompileUncheckedConverter<uint, int>(extendSign: true)(2147483648),
                CompileUncheckedConverter<byte, long>(extendSign: true)(128),
                CompileUncheckedConverter<ushort, long>(extendSign: true)(32768),
                CompileUncheckedConverter<uint, long>(extendSign: true)(2147483648),
                CompileUncheckedConverter<ulong, long>(extendSign: true)(9223372036854775808)
            };
```

```
            object[] withoutSignExtension = new object[]
            {
                CompileUncheckedConverter<byte, sbyte>(extendSign: false)(128),
                CompileUncheckedConverter<byte, short>(extendSign: false)(128),
                CompileUncheckedConverter<ushort, short>(extendSign: false)(32768),
                CompileUncheckedConverter<byte, int>(extendSign: false)(128),
                CompileUncheckedConverter<ushort, int>(extendSign: false)(32768),
                CompileUncheckedConverter<uint, int>(extendSign: false)(2147483648),
                CompileUncheckedConverter<byte, long>(extendSign: false)(128),
                CompileUncheckedConverter<ushort, long>(extendSign: false)(32768),
                CompileUncheckedConverter<uint, long>(extendSign: false)(2147483648),
                CompileUncheckedConverter<ulong, long>(extendSign: false)(9223372036854775808)
            };
            var i = 0;
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
        }

        /// <summary>
        /// <para>
        /// Tests that signed integers conversion of minus one with sign extension test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void SignedIntegersConversionOfMinusOneWithSignExtensionTest()
        {
            object[] withSignExtension = new object[]
            {
                CompileUncheckedConverter<sbyte, byte>(extendSign: true)(-1),
                CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(-1),
                CompileUncheckedConverter<short, ushort>(extendSign: true)(-1),
                CompileUncheckedConverter<sbyte, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<short, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<int, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<short, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<int, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<long, ulong>(extendSign: true)(-1)
            };
            object[] withoutSignExtension = new object[]
            {
                CompileUncheckedConverter<sbyte, byte>(extendSign: false)(-1),
                CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(-1),
                CompileUncheckedConverter<short, ushort>(extendSign: false)(-1),
                CompileUncheckedConverter<sbyte, uint>(extendSign: false)(-1),
                CompileUncheckedConverter<short, uint>(extendSign: false)(-1),
                CompileUncheckedConverter<int, uint>(extendSign: false)(-1),
                CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(-1),
                CompileUncheckedConverter<short, ulong>(extendSign: false)(-1),
                CompileUncheckedConverter<int, ulong>(extendSign: false)(-1),
                CompileUncheckedConverter<long, ulong>(extendSign: false)(-1)
            };
            var i = 0;
            Assert.Equal((byte)255, (byte)withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(4294967295, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(4294967295, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(4294967295, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(18446744073709551615, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(18446744073709551615, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
```

```
            Assert.Equal(18446744073709551615, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(18446744073709551615, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
        }

        /// <summary>
        /// <para>
        /// Tests that signed integers conversion of two with sign extension test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void SignedIntegersConversionOfTwoWithSignExtensionTest()
        {
            object[] withSignExtension = new object[]
            {
                CompileUncheckedConverter<sbyte, byte>(extendSign: true)(2),
                CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(2),
                CompileUncheckedConverter<short, ushort>(extendSign: true)(2),
                CompileUncheckedConverter<sbyte, uint>(extendSign: true)(2),
                CompileUncheckedConverter<short, uint>(extendSign: true)(2),
                CompileUncheckedConverter<int, uint>(extendSign: true)(2),
                CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(2),
                CompileUncheckedConverter<short, ulong>(extendSign: true)(2),
                CompileUncheckedConverter<int, ulong>(extendSign: true)(2),
                CompileUncheckedConverter<long, ulong>(extendSign: true)(2)
            };
            object[] withoutSignExtension = new object[]
            {
                CompileUncheckedConverter<sbyte, byte>(extendSign: false)(2),
                CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(2),
                CompileUncheckedConverter<short, ushort>(extendSign: false)(2),
                CompileUncheckedConverter<sbyte, uint>(extendSign: false)(2),
                CompileUncheckedConverter<short, uint>(extendSign: false)(2),
                CompileUncheckedConverter<int, uint>(extendSign: false)(2),
                CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(2),
                CompileUncheckedConverter<short, ulong>(extendSign: false)(2),
                CompileUncheckedConverter<int, ulong>(extendSign: false)(2),
                CompileUncheckedConverter<long, ulong>(extendSign: false)(2)
            };
            for (var i = 0; i < withSignExtension.Length; i++)
            {
                Assert.Equal(2UL, Convert.ToUInt64(withSignExtension[i]));
                Assert.Equal(withSignExtension[i], withoutSignExtension[i]);
            }
        }
        private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
        ↪  TTarget>(bool extendSign)
        {
            return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
            {
                generator.LoadArgument(0);
                generator.UncheckedConvert<TSource, TTarget>(extendSign);
                generator.Return();
            });
        }
    }
}
```

1.22 ./csharp/Platform.Reflection.Tests/GetILBytesMethodTests.cs

```
using System;
using System.Reflection;
using Xunit;
using Platform.Collections;
using Platform.Collections.Lists;

namespace Platform.Reflection.Tests
{
    /// <summary>
    /// <para>
    /// Represents the get il bytes method tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class GetILBytesMethodTests
    {
        /// <summary>
```

```
        /// <para>
        /// Tests that il bytes for delegate are available test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void ILBytesForDelegateAreAvailableTest()
        {
            var function = new Func<object, int>(argument => 0);
            var bytes = function.GetMethodInfo().GetILBytes();
            Assert.False(bytes.IsNullOrEmpty());
        }

        /// <summary>
        /// <para>
        /// Tests that il bytes for different delegates are the same test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void ILBytesForDifferentDelegatesAreTheSameTest()
        {
            var firstFunction = new Func<object, int>(argument => 0);
            var secondFunction = new Func<object, int>(argument => 0);
            Assert.False(firstFunction == secondFunction);
            var firstFunctionBytes = firstFunction.GetMethodInfo().GetILBytes();
            Assert.False(firstFunctionBytes.IsNullOrEmpty());
            var secondFunctionBytes = secondFunction.GetMethodInfo().GetILBytes();
            Assert.False(secondFunctionBytes.IsNullOrEmpty());
            Assert.True(firstFunctionBytes.EqualTo(secondFunctionBytes));
        }
    }
}
```

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

```
using Xunit;

namespace Platform.Reflection.Tests
{
    /// <summary>
    /// <para>
    /// Represents the numeric type tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public class NumericTypeTests
    {
        /// <summary>
        /// <para>
        /// Tests that u int 64 is numeric test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void UInt64IsNumericTest()
        {
            Assert.True(NumericType<ulong>.IsNumeric);
        }
    }
}
```

Index