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));
        }

        /// <summary>
        /// <para>
        /// Gets the new name.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [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>
```

```
138         /// <para></para>
139         /// </param>
140         /// <param name="message">
141         /// <para>The message.</para>
142         /// <para></para>
143         /// </param>
144         [MethodImpl(MethodImplOptions.AggressiveInlining)]
145         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
            ↪ message)
146         {
147             string messageBuilder() => message;
148             IsSignedInteger<T>(root, messageBuilder);
149         }
150 
151         /// <summary>
152         /// <para>
153         /// 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>
```

```
        /// <param name="message">
        /// <para>The message.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
        {
            string messageBuilder() => message;
            IsSigned<T>(root, messageBuilder);
        }

        /// <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.
        /// </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 IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
        ↪  IsNumeric<T>(root, (string)null);

        /// <summary>
        /// <para>
        /// Cans the be 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 CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
        ↪  messageBuilder)
        {
            if (!NumericType<T>.CanBeNumeric)
            {
                throw new NotSupportedException(messageBuilder());
            }
        }

        /// <summary>
        /// <para>
        /// Cans the be 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 CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
```

```
365         {
366             string messageBuilder() => message;
367             CanBeNumeric<T>(root, messageBuilder);
368         }
369 
370         /// <summary>
371         /// <para>
372         /// Cans the be numeric using the specified root.
373         /// </para>
374         /// <para></para>
375         /// </summary>
376         /// <typeparam name="T">
377         /// <para>The .</para>
378         /// <para></para>
379         /// </typeparam>
380         /// <param name="root">
381         /// <para>The root.</para>
382         /// <para></para>
383         /// </param>
384         [MethodImpl(MethodImplOptions.AggressiveInlining)]
385         public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪ CanBeNumeric<T>(root, (string)null);
386 
387         #endregion
388 
389         #region OnDebug
390 
391         /// <summary>
392         /// <para>
393         /// Ises the unsigned integer using the specified root.
394         /// </para>
395         /// <para></para>
396         /// </summary>
397         /// <typeparam name="T">
398         /// <para>The .</para>
399         /// <para></para>
400         /// </typeparam>
401         /// <param name="root">
402         /// <para>The root.</para>
403         /// <para></para>
404         /// </param>
405         /// <param name="messageBuilder">
406         /// <para>The message builder.</para>
407         /// <para></para>
408         /// </param>
409         [Conditional("DEBUG")]
410         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
            ↪ Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
411 
412         /// <summary>
413         /// <para>
414         /// Ises the unsigned integer using the specified root.
415         /// </para>
416         /// <para></para>
417         /// </summary>
418         /// <typeparam name="T">
419         /// <para>The .</para>
420         /// <para></para>
421         /// </typeparam>
422         /// <param name="root">
423         /// <para>The root.</para>
424         /// <para></para>
425         /// </param>
426         /// <param name="message">
427         /// <para>The message.</para>
428         /// <para></para>
429         /// </param>
430         [Conditional("DEBUG")]
431         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
            ↪ message) => Ensure.Always.IsUnsignedInteger<T>(message);
432 
433         /// <summary>
434         /// <para>
435         /// Ises the unsigned integer using the specified root.
436         /// </para>
437         /// <para></para>
438         /// </summary>
439         /// <typeparam name="T">
```

```
440         /// <para>The .</para>
441         /// <para></para>
442         /// </typeparam>
443         /// <param name="root">
444         /// <para>The root.</para>
445         /// <para></para>
446         /// </param>
447         [Conditional("DEBUG")]
448         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪ Ensure.Always.IsUnsignedInteger<T>();
449
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
```

```
namespace Platform.Reflection
{
    /// <summary>
    /// <para>
    /// Represents the il generator extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class ILGeneratorExtensions
    {
        /// <summary>
        /// <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>();
            }
        }

        /// <summary>
        /// <para>
        /// Converts the to boolean using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TSource">
        /// <para>The source.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [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);
            }
        }

        /// <summary>
        /// <para>
```

```
        /// Converts the to integer using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TSource">
        /// <para>The source.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetType">
        /// <para>The target type.</para>
        /// <para></para>
        /// </param>
        /// <param name="extendSign">
        /// <para>The extend sign.</para>
        /// <para></para>
        /// </param>
        [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))
            {
```

```
317                 if (NumericType<TSource>.IsSigned)
318                 {
319                     generator.Emit(OpCodes.Conv_Ovf_U4);
320                 }
321                 else
322                 {
323                     generator.Emit(OpCodes.Conv_Ovf_U4_Un);
324                 }
325             }
326             else if (targetType == typeof(long))
327             {
328                 if (NumericType<TSource>.IsSigned)
329                 {
330                     generator.Emit(OpCodes.Conv_Ovf_I8);
331                 }
332                 else
333                 {
334                     generator.Emit(OpCodes.Conv_Ovf_I8_Un);
335                 }
336             }
337             else if (targetType == typeof(ulong))
338             {
339                 if (NumericType<TSource>.IsSigned)
340                 {
341                     generator.Emit(OpCodes.Conv_Ovf_U8);
342                 }
343                 else
344                 {
345                     generator.Emit(OpCodes.Conv_Ovf_U8_Un);
346                 }
347             }
348             else if (targetType == typeof(float))
349             {
350                 if (NumericType<TSource>.IsSigned)
351                 {
352                     generator.Emit(OpCodes.Conv_R4);
353                 }
354                 else
355                 {
356                     generator.Emit(OpCodes.Conv_R_Un);
357                 }
358             }
359             else if (targetType == typeof(double))
360             {
361                 generator.Emit(OpCodes.Conv_R8);
362             }
363             else if (targetType == typeof(bool))
364             {
365                 generator.ConvertToBoolean<TSource>();
366             }
367             else
368             {
369                 throw new NotSupportedException();
370             }
371         }
372 
373         /// <summary>
374         /// <para>
375         /// Loads the constant using the specified generator.
376         /// </para>
377         /// <para></para>
378         /// </summary>
379         /// <param name="generator">
380         /// <para>The generator.</para>
381         /// <para></para>
382         /// </param>
383         /// <param name="value">
384         /// <para>The value.</para>
385         /// <para></para>
386         /// </param>
387         [MethodImpl(MethodImplOptions.AggressiveInlining)]
388         public static void LoadConstant(this ILGenerator generator, bool value) =>
            ↪  generator.LoadConstant(value ? 1 : 0);
389 
390         /// <summary>
391         /// <para>
392         /// Loads the constant using the specified generator.
393         /// </para>
```

```
394             /// <para></para>
395             /// </summary>
396             /// <param name="generator">
397             /// <para>The generator.</para>
398             /// <para></para>
399             /// </param>
400             /// <param name="value">
401             /// <para>The value.</para>
402             /// <para></para>
403             /// </param>
404             [MethodImpl(MethodImplOptions.AggressiveInlining)]
405             public static void LoadConstant(this ILGenerator generator, float value) =>
                ↪  generator.Emit(OpCodes.Ldc_R4, value);
406
407             /// <summary>
408             /// <para>
409             /// Loads the constant using the specified generator.
410             /// </para>
411             /// <para></para>
412             /// </summary>
413             /// <param name="generator">
414             /// <para>The generator.</para>
415             /// <para></para>
416             /// </param>
417             /// <param name="value">
418             /// <para>The value.</para>
419             /// <para></para>
420             /// </param>
421             [MethodImpl(MethodImplOptions.AggressiveInlining)]
422             public static void LoadConstant(this ILGenerator generator, double value) =>
                ↪  generator.Emit(OpCodes.Ldc_R8, value);
423
424             /// <summary>
425             /// <para>
426             /// Loads the constant using the specified generator.
427             /// </para>
428             /// <para></para>
429             /// </summary>
430             /// <param name="generator">
431             /// <para>The generator.</para>
432             /// <para></para>
433             /// </param>
434             /// <param name="value">
435             /// <para>The value.</para>
436             /// <para></para>
437             /// </param>
438             [MethodImpl(MethodImplOptions.AggressiveInlining)]
439             public static void LoadConstant(this ILGenerator generator, ulong value) =>
                ↪  generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
440
441             /// <summary>
442             /// <para>
443             /// Loads the constant using the specified generator.
444             /// </para>
445             /// <para></para>
446             /// </summary>
447             /// <param name="generator">
448             /// <para>The generator.</para>
449             /// <para></para>
450             /// </param>
451             /// <param name="value">
452             /// <para>The value.</para>
453             /// <para></para>
454             /// </param>
455             [MethodImpl(MethodImplOptions.AggressiveInlining)]
456             public static void LoadConstant(this ILGenerator generator, long value) =>
                ↪  generator.Emit(OpCodes.Ldc_I8, value);
457
458             /// <summary>
459             /// <para>
460             /// Loads the constant using the specified generator.
461             /// </para>
462             /// <para></para>
463             /// </summary>
464             /// <param name="generator">
465             /// <para>The generator.</para>
466             /// <para></para>
467             /// </param>
```

```
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, uint value)
        {
            switch (value)
            {
                case uint.MaxValue:
                    generator.Emit(OpCodes.Ldc_I4_M1);
                    return;
                case 0:
                    generator.Emit(OpCodes.Ldc_I4_0);
                    return;
                case 1:
                    generator.Emit(OpCodes.Ldc_I4_1);
                    return;
                case 2:
                    generator.Emit(OpCodes.Ldc_I4_2);
                    return;
                case 3:
                    generator.Emit(OpCodes.Ldc_I4_3);
                    return;
                case 4:
                    generator.Emit(OpCodes.Ldc_I4_4);
                    return;
                case 5:
                    generator.Emit(OpCodes.Ldc_I4_5);
                    return;
                case 6:
                    generator.Emit(OpCodes.Ldc_I4_6);
                    return;
                case 7:
                    generator.Emit(OpCodes.Ldc_I4_7);
                    return;
                case 8:
                    generator.Emit(OpCodes.Ldc_I4_8);
                    return;
                default:
                    if (value <= sbyte.MaxValue)
                    {
                        generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
                    }
                    else
                    {
                        generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
                    }
                    return;
            }
        }

        /// <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, int value)
        {
            switch (value)
            {
                case -1:
                    generator.Emit(OpCodes.Ldc_I4_M1);
                    return;
                case 0:
                    generator.Emit(OpCodes.Ldc_I4_0);
                    return;
                case 1:
                    generator.Emit(OpCodes.Ldc_I4_1);
                    return;
```

```
                case 2:
                    generator.Emit(OpCodes.Ldc_I4_2);
                    return;
                case 3:
                    generator.Emit(OpCodes.Ldc_I4_3);
                    return;
                case 4:
                    generator.Emit(OpCodes.Ldc_I4_4);
                    return;
                case 5:
                    generator.Emit(OpCodes.Ldc_I4_5);
                    return;
                case 6:
                    generator.Emit(OpCodes.Ldc_I4_6);
                    return;
                case 7:
                    generator.Emit(OpCodes.Ldc_I4_7);
                    return;
                case 8:
                    generator.Emit(OpCodes.Ldc_I4_8);
                    return;
                default:
                    if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
                    {
                        generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
                    }
                    else
                    {
                        generator.Emit(OpCodes.Ldc_I4, value);
                    }
                    return;
            }
        }

        /// <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, short value) =>
        ↪  generator.LoadConstant((int)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, ushort value) =>
        ↪  generator.LoadConstant((int)value);

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

```
625             /// </param>
626             /// <param name="value">
627             /// <para>The value.</para>
628             /// <para></para>
629             /// </param>
630             [MethodImpl(MethodImplOptions.AggressiveInlining)]
631             public static void LoadConstant(this ILGenerator generator, sbyte value) =>
                ↪  generator.LoadConstant((int)value);
632
633             /// <summary>
634             /// <para>
635             /// Loads the constant using the specified generator.
636             /// </para>
637             /// <para></para>
638             /// </summary>
639             /// <param name="generator">
640             /// <para>The generator.</para>
641             /// <para></para>
642             /// </param>
643             /// <param name="value">
644             /// <para>The value.</para>
645             /// <para></para>
646             /// </param>
647             [MethodImpl(MethodImplOptions.AggressiveInlining)]
648             public static void LoadConstant(this ILGenerator generator, byte value) =>
                ↪  generator.LoadConstant((int)value);
649
650             /// <summary>
651             /// <para>
652             /// Loads the constant one using the specified generator.
653             /// </para>
654             /// <para></para>
655             /// </summary>
656             /// <typeparam name="TConstant">
657             /// <para>The constant.</para>
658             /// <para></para>
659             /// </typeparam>
660             /// <param name="generator">
661             /// <para>The generator.</para>
662             /// <para></para>
663             /// </param>
664             [MethodImpl(MethodImplOptions.AggressiveInlining)]
665             public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
                ↪  LoadConstantOne(generator, typeof(TConstant));
666
667             /// <summary>
668             /// <para>
669             /// Loads the constant one using the specified generator.
670             /// </para>
671             /// <para></para>
672             /// </summary>
673             /// <param name="generator">
674             /// <para>The generator.</para>
675             /// <para></para>
676             /// </param>
677             /// <param name="constantType">
678             /// <para>The constant type.</para>
679             /// <para></para>
680             /// </param>
681             /// <exception cref="NotSupportedException">
682             /// <para></para>
683             /// <para></para>
684             /// </exception>
685             [MethodImpl(MethodImplOptions.AggressiveInlining)]
686             public static void LoadConstantOne(this ILGenerator generator, Type constantType)
687             {
688                 if (constantType == typeof(float))
689                 {
690                     generator.LoadConstant(1F);
691                 }
692                 else if (constantType == typeof(double))
693                 {
694                     generator.LoadConstant(1D);
695                 }
696                 else if (constantType == typeof(long))
697                 {
698                     generator.LoadConstant(1L);
699                 }
```

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

        /// <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="constantType">
        /// <para>The constant type.</para>
        /// <para></para>
        /// </param>
        /// <param name="constantValue">
        /// <para>The constant value.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotSupportedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, Type constantType, object
        ↪  constantValue)
        {
            constantValue = Convert.ChangeType(constantValue, constantType);
            if (constantType == typeof(float))
            {
                generator.LoadConstant((float)constantValue);
            }
            else if (constantType == typeof(double))
            {
                generator.LoadConstant((double)constantValue);
            }
            else if (constantType == typeof(long))
            {
                generator.LoadConstant((long)constantValue);
            }
            else if (constantType == typeof(ulong))
            {
                generator.LoadConstant((ulong)constantValue);
            }
            else if (constantType == typeof(int))
            {
                generator.LoadConstant((int)constantValue);
            }
            else if (constantType == typeof(uint))
            {
                generator.LoadConstant((uint)constantValue);
            }
            else if (constantType == typeof(short))
            {
                generator.LoadConstant((short)constantValue);
            }
            else if (constantType == typeof(ushort))
            {
                generator.LoadConstant((ushort)constantValue);
            }
            else if (constantType == typeof(sbyte))
            {
                generator.LoadConstant((sbyte)constantValue);
            }
            else if (constantType == typeof(byte))
            {
                generator.LoadConstant((byte)constantValue);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        /// <summary>
        /// <para>
        /// Increments the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TValue">
        /// <para>The value.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Increment<TValue>(this ILGenerator generator) =>
        ↪  generator.Increment(typeof(TValue));

        /// <summary>
        /// <para>
        /// Decrements the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TValue">
        /// <para>The value.</para>
        /// <para></para>
```

```
853         /// </typeparam>
854         /// <param name="generator">
855         /// <para>The generator.</para>
856         /// <para></para>
857         /// </param>
858         [MethodImpl(MethodImplOptions.AggressiveInlining)]
859         public static void Decrement<TValue>(this ILGenerator generator) =>
            ↪ generator.Decrement(typeof(TValue));
860
861         /// <summary>
862         /// <para>
863         /// Increments the generator.
864         /// </para>
865         /// <para></para>
866         /// </summary>
867         /// <param name="generator">
868         /// <para>The generator.</para>
869         /// <para></para>
870         /// </param>
871         /// <param name="valueType">
872         /// <para>The value type.</para>
873         /// <para></para>
874         /// </param>
875         [MethodImpl(MethodImplOptions.AggressiveInlining)]
876         public static void Increment(this ILGenerator generator, Type valueType)
877         {
878             generator.LoadConstantOne(valueType);
879             generator.Add();
880         }
881
882         /// <summary>
883         /// <para>
884         /// Adds the generator.
885         /// </para>
886         /// <para></para>
887         /// </summary>
888         /// <param name="generator">
889         /// <para>The generator.</para>
890         /// <para></para>
891         /// </param>
892         [MethodImpl(MethodImplOptions.AggressiveInlining)]
893         public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
894
895         /// <summary>
896         /// <para>
897         /// Decrements the generator.
898         /// </para>
899         /// <para></para>
900         /// </summary>
901         /// <param name="generator">
902         /// <para>The generator.</para>
903         /// <para></para>
904         /// </param>
905         /// <param name="valueType">
906         /// <para>The value type.</para>
907         /// <para></para>
908         /// </param>
909         [MethodImpl(MethodImplOptions.AggressiveInlining)]
910         public static void Decrement(this ILGenerator generator, Type valueType)
911         {
912             generator.LoadConstantOne(valueType);
913             generator.Subtract();
914         }
915
916         /// <summary>
917         /// <para>
918         /// Subtracts the generator.
919         /// </para>
920         /// <para></para>
921         /// </summary>
922         /// <param name="generator">
923         /// <para>The generator.</para>
924         /// <para></para>
925         /// </param>
926         [MethodImpl(MethodImplOptions.AggressiveInlining)]
927         public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
928
929         /// <summary>
```

```
        /// <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>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
        ↪  generator.Emit(OpCodes.Bge, label);

        /// <summary>
        /// <para>
        /// Unsigneds the branch 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>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
        ↪  label) => generator.Emit(OpCodes.Bge_Un, label);

        /// <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="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 BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
        ↪  Label label)
        {
            if (isSigned)
            {
                generator.BranchIfGreaterOrEqual(label);
            }
            else
            {
                generator.UnsignedBranchIfGreaterOrEqual(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="label">
        /// <para>The label.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
        ↪  generator.Emit(OpCodes.Ble, label);

        /// <summary>
        /// <para>
        /// Unsigneds the branch if less 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>
        /// </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>
```

```
1310            /// <param name="label">
1311            /// <para>The label.</para>
1312            /// <para></para>
1313            /// </param>
1314            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1315            public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
                ↪  label)
1316            {
1317                if (isSigned)
1318                {
1319                    generator.BranchIfLessOrEqual(label);
1320                }
1321                else
1322                {
1323                    generator.UnsignedBranchIfLessOrEqual(label);
1324                }
1325            }
1326
1327            /// <summary>
1328            /// <para>
1329            /// Boxes the generator.
1330            /// </para>
1331            /// <para></para>
1332            /// </summary>
1333            /// <typeparam name="TBox">
1334            /// <para>The box.</para>
1335            /// <para></para>
1336            /// </typeparam>
1337            /// <param name="generator">
1338            /// <para>The generator.</para>
1339            /// <para></para>
1340            /// </param>
1341            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1342            public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
1343
1344            /// <summary>
1345            /// <para>
1346            /// Boxes the generator.
1347            /// </para>
1348            /// <para></para>
1349            /// </summary>
1350            /// <param name="generator">
1351            /// <para>The generator.</para>
1352            /// <para></para>
1353            /// </param>
1354            /// <param name="boxedType">
1355            /// <para>The boxed type.</para>
1356            /// <para></para>
1357            /// </param>
1358            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1359            public static void Box(this ILGenerator generator, Type boxedType) =>
                ↪  generator.Emit(OpCodes.Box, boxedType);
1360
1361            /// <summary>
1362            /// <para>
1363            /// Calls the generator.
1364            /// </para>
1365            /// <para></para>
1366            /// </summary>
1367            /// <param name="generator">
1368            /// <para>The generator.</para>
1369            /// <para></para>
1370            /// </param>
1371            /// <param name="method">
1372            /// <para>The method.</para>
1373            /// <para></para>
1374            /// </param>
1375            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1376            public static void Call(this ILGenerator generator, MethodInfo method) =>
                ↪  generator.Emit(OpCodes.Call, method);
1377
1378            /// <summary>
1379            /// <para>
1380            /// Returns the generator.
1381            /// </para>
1382            /// <para></para>
1383            /// </summary>
1384            /// <param name="generator">
```

```
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);

        /// <summary>
        /// <para>
        /// Unboxes the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TUnbox">
        /// <para>The unbox.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Unbox<TUnbox>(this ILGenerator generator) =>
        ↪  generator.Unbox(typeof(TUnbox));

        /// <summary>
        /// <para>
        /// Unboxes the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="typeToUnbox">
        /// <para>The type to unbox.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
        ↪  generator.Emit(OpCodes.Unbox, typeToUnbox);

        /// <summary>
        /// <para>
        /// Unboxes the value using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TUnbox">
        /// <para>The unbox.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
        ↪  generator.UnboxValue(typeof(TUnbox));

        /// <summary>
        /// <para>
        /// Unboxes the value using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="typeToUnbox">
        /// <para>The type to unbox.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
        ↪  generator.Emit(OpCodes.Unbox_Any, typeToUnbox);

```

```
1459            /// <summary>
1460            /// <para>
1461            /// Declares the local using the specified generator.
1462            /// </para>
1463            /// <para></para>
1464            /// </summary>
1465            /// <typeparam name="T">
1466            /// <para>The .</para>
1467            /// <para></para>
1468            /// </typeparam>
1469            /// <param name="generator">
1470            /// <para>The generator.</para>
1471            /// <para></para>
1472            /// </param>
1473            /// <returns>
1474            /// <para>The local builder</para>
1475            /// <para></para>
1476            /// </returns>
1477            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1478            public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
                ↪  generator.DeclareLocal(typeof(T));
1479
1480            /// <summary>
1481            /// <para>
1482            /// Loads the local using the specified generator.
1483            /// </para>
1484            /// <para></para>
1485            /// </summary>
1486            /// <param name="generator">
1487            /// <para>The generator.</para>
1488            /// <para></para>
1489            /// </param>
1490            /// <param name="local">
1491            /// <para>The local.</para>
1492            /// <para></para>
1493            /// </param>
1494            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1495            public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
                ↪  generator.Emit(OpCodes.Ldloc, local);
1496
1497            /// <summary>
1498            /// <para>
1499            /// Stores the local using the specified generator.
1500            /// </para>
1501            /// <para></para>
1502            /// </summary>
1503            /// <param name="generator">
1504            /// <para>The generator.</para>
1505            /// <para></para>
1506            /// </param>
1507            /// <param name="local">
1508            /// <para>The local.</para>
1509            /// <para></para>
1510            /// </param>
1511            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1512            public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
                ↪  generator.Emit(OpCodes.Stloc, local);
1513
1514            /// <summary>
1515            /// <para>
1516            /// News the object using the specified generator.
1517            /// </para>
1518            /// <para></para>
1519            /// </summary>
1520            /// <param name="generator">
1521            /// <para>The generator.</para>
1522            /// <para></para>
1523            /// </param>
1524            /// <param name="type">
1525            /// <para>The type.</para>
1526            /// <para></para>
1527            /// </param>
1528            /// <param name="parameterTypes">
1529            /// <para>The parameter types.</para>
1530            /// <para></para>
1531            /// </param>
1532            /// <exception cref="InvalidOperationException">
1533            /// <para></para>
```

```
1534          /// <para></para>
1535          /// </exception>
1536          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1537          public static void NewObject(this ILGenerator generator, Type type, params Type[]
              ↪ parameterTypes)
1538          {
1539              var allConstructors = type.GetConstructors(BindingFlags.Public |
                  ↪ BindingFlags.NonPublic | BindingFlags.Instance
1540  #if !NETSTANDARD
1541                  | BindingFlags.CreateInstance
1542  #endif
1543                  );
1544              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();
1545              if (constructor == null)
1546              {
1547                  throw new InvalidOperationException("Type " + type + " must have a constructor
                      ↪ that matches parameters [" + string.Join(", ",
                      ↪ parameterTypes.AsEnumerable()) + "]");
1548              }
1549              generator.NewObject(constructor);
1550          }
1551
1552          /// <summary>
1553          /// <para>
1554          /// News the object using the specified generator.
1555          /// </para>
1556          /// <para></para>
1557          /// </summary>
1558          /// <param name="generator">
1559          /// <para>The generator.</para>
1560          /// <para></para>
1561          /// </param>
1562          /// <param name="constructor">
1563          /// <para>The constructor.</para>
1564          /// <para></para>
1565          /// </param>
1566          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1567          public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
              ↪ generator.Emit(OpCodes.Newobj, constructor);
1568
1569          /// <summary>
1570          /// <para>
1571          /// Loads the indirect using the specified generator.
1572          /// </para>
1573          /// <para></para>
1574          /// </summary>
1575          /// <typeparam name="T">
1576          /// <para>The .</para>
1577          /// <para></para>
1578          /// </typeparam>
1579          /// <param name="generator">
1580          /// <para>The generator.</para>
1581          /// <para></para>
1582          /// </param>
1583          /// <param name="isVolatile">
1584          /// <para>The is volatile.</para>
1585          /// <para></para>
1586          /// </param>
1587          /// <param name="unaligned">
1588          /// <para>The unaligned.</para>
1589          /// <para></para>
1590          /// </param>
1591          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1592          public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
              ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
1593
1594          /// <summary>
1595          /// <para>
1596          /// Loads the indirect using the specified generator.
1597          /// </para>
1598          /// <para></para>
1599          /// </summary>
1600          /// <param name="generator">
1601          /// <para>The generator.</para>
1602          /// <para></para>
```

```
        /// </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;
```

```
5
6  // ReSharper disable AssignmentInConditionalExpression
7  // ReSharper disable BuiltInTypeReferenceStyle
8  // ReSharper disable StaticFieldInGenericType
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Reflection
12 {
13     /// <summary>
14     /// <para>
15     /// Represents the numeric type.
16     /// </para>
17     /// <para></para>
18     /// </summary>
19     public static class NumericType<T>
20     {
21         /// <summary>
22         /// <para>
23         /// The type.
24         /// </para>
25         /// <para></para>
26         /// </summary>
27         public static readonly Type Type;
28         /// <summary>
29         /// <para>
30         /// The underlying type.
31         /// </para>
32         /// <para></para>
33         /// </summary>
34         public static readonly Type UnderlyingType;
35         /// <summary>
36         /// <para>
37         /// The signed version.
38         /// </para>
39         /// <para></para>
40         /// </summary>
41         public static readonly Type SignedVersion;
42         /// <summary>
43         /// <para>
44         /// The unsigned version.
45         /// </para>
46         /// <para></para>
47         /// </summary>
48         public static readonly Type UnsignedVersion;
49         /// <summary>
50         /// <para>
51         /// The is float point.
52         /// </para>
53         /// <para></para>
54         /// </summary>
55         public static readonly bool IsFloatPoint;
56         /// <summary>
57         /// <para>
58         /// The is numeric.
59         /// </para>
60         /// <para></para>
61         /// </summary>
62         public static readonly bool IsNumeric;
63         /// <summary>
64         /// <para>
65         /// The is signed.
66         /// </para>
67         /// <para></para>
68         /// </summary>
69         public static readonly bool IsSigned;
70         /// <summary>
71         /// <para>
72         /// The can be numeric.
73         /// </para>
74         /// <para></para>
75         /// </summary>
76         public static readonly bool CanBeNumeric;
77         /// <summary>
78         /// <para>
79         /// The is nullable.
80         /// </para>
81         /// <para></para>
82         /// </summary>
83         public static readonly bool IsNullable;
```

```
        /// <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();
            }
        }

        /// <summary>
        /// <para>
        /// Gets the min and max values using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
```

```
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="minValue">
        /// <para>The min value.</para>
        /// <para></para>
        /// </param>
        /// <param name="maxValue">
        /// <para>The max value.</para>
        /// <para></para>
        /// </param>
        [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));
            }
        }

        /// <summary>
        /// <para>
        /// Gets the signed and unsigned versions using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="isSigned">
        /// <para>The is signed.</para>
        /// <para></para>
        /// </param>
        /// <param name="signedVersion">
        /// <para>The signed version.</para>
        /// <para></para>
        /// </param>
        /// <param name="unsignedVersion">
        /// <para>The unsigned version.</para>
        /// <para></para>
        /// </param>
        [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>
```

```
18      public static class TypeExtensions
19      {
20          /// <summary>
21          /// <para>
22          /// The static.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          static public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
            ↪  BindingFlags.NonPublic | BindingFlags.Static;
27          /// <summary>
28          /// <para>
29          /// The default delegate method name.
30          /// </para>
31          /// <para></para>
32          /// </summary>
33          static public readonly string DefaultDelegateMethodName = "Invoke";
34
35          /// <summary>
36          /// <para>
37          /// The can be numeric types.
38          /// </para>
39          /// <para></para>
40          /// </summary>
41          static private readonly HashSet<Type> _canBeNumericTypes;
42          /// <summary>
43          /// <para>
44          /// The is numeric types.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          static private readonly HashSet<Type> _isNumericTypes;
49          /// <summary>
50          /// <para>
51          /// The is signed types.
52          /// </para>
53          /// <para></para>
54          /// </summary>
55          static private readonly HashSet<Type> _isSignedTypes;
56          /// <summary>
57          /// <para>
58          /// The is float point types.
59          /// </para>
60          /// <para></para>
61          /// </summary>
62          static private readonly HashSet<Type> _isFloatPointTypes;
63          /// <summary>
64          /// <para>
65          /// The unsigned versions of signed types.
66          /// </para>
67          /// <para></para>
68          /// </summary>
69          static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
70          /// <summary>
71          /// <para>
72          /// The signed versions of unsigned types.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;
77
78          /// <summary>
79          /// <para>
80          /// Initializes a new <see cref="TypeExtensions"/> instance.
81          /// </para>
82          /// <para></para>
83          /// </summary>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          static TypeExtensions()
86          {
87              _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
                ↪  typeof(DateTime), typeof(TimeSpan) };
88              _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
                ↪  typeof(ulong) };
89              _canBeNumericTypes.UnionWith(_isNumericTypes);
90              _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
                ↪  typeof(long) };
91              _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>();

        /// <summary>
        /// <para>
        /// Gets the generic method using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="name">
        /// <para>The name.</para>
        /// <para></para>
        /// </param>
        /// <param name="genericParameterTypes">
        /// <para>The generic parameter types.</para>
        /// <para></para>
        /// </param>
        /// <param name="argumentTypes">
        /// <para>The argument types.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The method.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
        ↪  genericParameterTypes, Type[] argumentTypes)
        {
            var methods = from m in type.GetMethods()
                          where m.Name == name
                             && m.IsGenericMethodDefinition
                          let typeParams = m.GetGenericArguments()
                          let normalParams = m.GetParameters().Select(x => x.ParameterType)
                          where typeParams.SequenceEqual(genericParameterTypes)
                             && normalParams.SequenceEqual(argumentTypes)
                          select m;
            var method = methods.Single();
            return method;
        }

        /// <summary>
        /// <para>
        /// Gets the base type using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The type</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetBaseType(this Type type) => type.BaseType;

        /// <summary>
        /// <para>
        /// Gets the assembly using the specified type.
        /// </para>
        /// <para></para>
        /// </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">
```

```
        /// <para>The generic type definition.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
        ↪  type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;

        /// <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);
        }

        /// <summary>
        /// <para>
        /// Appends the types using the specified container.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="container">
        /// <para>The container.</para>
        /// <para></para>
        /// </param>
        /// <param name="types">
        /// <para>The types.</para>
        /// <para></para>
        /// </param>
        [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>
```

```
15          /// Tests that u int 64 is numeric test.
16          /// </para>
17          /// <para></para>
18          /// </summary>
19          [Fact]
20          public void UInt64IsNumericTest()
21          {
22              Assert.True(NumericType<ulong>.IsNumeric);
23          }
24      }
25  }
```

Index

./csharp/Platform.Reflection.Tests/CodeGenerationTests.cs, 58
./csharp/Platform.Reflection.Tests/GetILBytesMethodTests.cs, 61
./csharp/Platform.Reflection.Tests/NumericTypeTests.cs, 61
./csharp/Platform.Reflection/AssemblyExtensions.cs, 1
./csharp/Platform.Reflection/DelegateHelpers.cs, 1
./csharp/Platform.Reflection/DynamicExtensions.cs, 4
./csharp/Platform.Reflection/EnsureExtensions.cs, 5
./csharp/Platform.Reflection/FieldInfoExtensions.cs, 14
./csharp/Platform.Reflection/ILGeneratorExtensions.cs, 14
./csharp/Platform.Reflection/MethodInfoExtensions.cs, 39
./csharp/Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 40
./csharp/Platform.Reflection/NumericType.cs, 40
./csharp/Platform.Reflection/PropertyInfoExtensions.cs, 43
./csharp/Platform.Reflection/TypeBuilderExtensions.cs, 44
./csharp/Platform.Reflection/TypeExtensions.cs, 46
./csharp/Platform.Reflection/Types.cs, 53
./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 54
./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 55
./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 55
./csharp/Platform.Reflection/Types[T1, T2, T3, T4].cs, 56
./csharp/Platform.Reflection/Types[T1, T2, T3].cs, 56
./csharp/Platform.Reflection/Types[T1, T2].cs, 57
./csharp/Platform.Reflection/Types[T].cs, 57