LinksPlatform's Platform.Reflection Class Library

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

```csharp
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

```csharp
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>
```

```csharp
public static class DelegateHelpers
{
    /// <summary>
    /// <para>
    /// Compiles the or default 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>
    /// <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 CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode, bool
        typeMemberMethod)
        where TDelegate : Delegate
    {
        var @delegate = default(TDelegate);
        try
        {
            @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
                CompileDynamicMethod<TDelegate>(emitCode);
        }
        catch (Exception exception)
        {
            exception.Ignore();
        }
        return @delegate;
    }

    /// <summary>
    /// <para>
    /// Compiles the or default 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 CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
        TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);

    /// <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>
```

```csharp
        /// <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>
```

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

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

```csharp
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;
        }
    }
}
```

```csharp
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;
```

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

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

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

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

```csharp
                IsSignedInteger<T>(root, messageBuilder);
        }

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

        /// <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>
        /// <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 IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
        ↪  messageBuilder)
        {
            if (!NumericType<T>.IsSigned)
            {
                throw new NotSupportedException(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>
        /// <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);
        }
```

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

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

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

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

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

```
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);

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);

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>();

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>
```

```csharp
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="message">
        /// <para>The message.</para>
        /// <para></para>
        /// </param>
        [Conditional("DEBUG")]
        public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
        ↪   Ensure.Always.IsNumeric<T>(message);

        /// <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>
        [Conditional("DEBUG")]
        public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
        ↪   Ensure.Always.IsNumeric<T>();

        /// <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>
        [Conditional("DEBUG")]
        public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
        ↪   messageBuilder) => Ensure.Always.CanBeNumeric<T>(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>
        [Conditional("DEBUG")]
        public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
        ↪   => Ensure.Always.CanBeNumeric<T>(message);

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

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

1.5 ./csharp/Platform.Reflection/FieldInfoExtensions.cs

```
1  using System.Reflection;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Reflection
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the field info extensions.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public static class FieldInfoExtensions
15     {
16         /// <summary>
17         /// <para>
18         /// Gets the static value using the specified field info.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         /// <typeparam name="T">
23         /// <para>The .</para>
24         /// <para></para>
25         /// </typeparam>
26         /// <param name="fieldInfo">
27         /// <para>The field info.</para>
28         /// <para></para>
29         /// </param>
30         /// <returns>
31         /// <para>The</para>
32         /// <para></para>
33         /// </returns>
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public static T GetStaticValue<T>(this FieldInfo fieldInfo) =>
           ↪  (T)fieldInfo.GetValue(null);
36     }
37 }
```

1.6 ./csharp/Platform.Reflection/ILGeneratorExtensions.cs

```
1  using System;
2  using System.Linq;
3  using System.Reflection;
4  using System.Reflection.Emit;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Reflection
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the il generator extensions.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     public static class ILGeneratorExtensions
18     {
19         /// <summary>
20         /// <para>
```

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

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

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

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

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

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

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

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

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

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

```csharp
        /// <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, long value) =>
            generator.Emit(OpCodes.Ldc_I8, 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, 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;
            }
        }
```

```csharp
        /// <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);
```

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

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

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

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

```csharp
        /// <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>
        /// </typeparam>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Decrement<TValue>(this ILGenerator generator) =>
            generator.Decrement(typeof(TValue));

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

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

        /// <summary>
        /// <para>
        /// Decrements the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
```

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

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

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

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

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

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

```csharp
        /// <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>
```

```csharp
        /// </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)
        {
```

```csharp
            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>
```

```csharp
        /// </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>
        /// <param name="label">
        /// <para>The label.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
        ↪  label)
        {
            if (isSigned)
            {
                generator.BranchIfLessOrEqual(label);
            }
            else
            {
                generator.UnsignedBranchIfLessOrEqual(label);
            }
        }

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

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

```csharp
        public static void Box(this ILGenerator generator, Type boxedType) =>
        ↪    generator.Emit(OpCodes.Box, boxedType);

        /// <summary>
        /// <para>
        /// Calls the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="method">
        /// <para>The method.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Call(this ILGenerator generator, MethodInfo method) =>
        ↪    generator.Emit(OpCodes.Call, method);

        /// <summary>
        /// <para>
        /// Returns the generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <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>
```

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

```csharp
        /// <param name="local">
        /// <para>The local.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
        ↪   generator.Emit(OpCodes.Stloc, local);

        /// <summary>
        /// <para>
        /// News the object 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="parameterTypes">
        /// <para>The parameter types.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void NewObject(this ILGenerator generator, Type type, params Type[]
        ↪   parameterTypes)
        {
            var allConstructors = type.GetConstructors(BindingFlags.Public |
            ↪   BindingFlags.NonPublic | BindingFlags.Instance
#if !NETSTANDARD
                | BindingFlags.CreateInstance
#endif
            );
            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();
            if (constructor == null)
            {
                throw new InvalidOperationException("Type " + type + " must have a constructor
                ↪    that matches parameters [" + string.Join(", ",
                ↪    parameterTypes.AsEnumerable()) + "]");
            }
            generator.NewObject(constructor);
        }

        /// <summary>
        /// <para>
        /// News the object using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="generator">
        /// <para>The generator.</para>
        /// <para></para>
        /// </param>
        /// <param name="constructor">
        /// <para>The constructor.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
        ↪   generator.Emit(OpCodes.Newobj, constructor);

        /// <summary>
        /// <para>
        /// Loads the indirect using the specified generator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
```

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

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

```csharp
            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>
```

```csharp
        /// </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");
```

```csharp
                }
            }

            /// <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

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

```csharp
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

```csharp
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>
```

```csharp
        /// <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

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

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

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

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

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

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

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

```csharp
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>
```

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

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

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  using System;
4  using System.Reflection;
5  using System.Reflection.Emit;
6  using System.Runtime.CompilerServices;
7
8  namespace Platform.Reflection
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the type builder extensions.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     public static class TypeBuilderExtensions
17     {
18         /// <summary>
19         /// <para>
20         /// The static.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         public static readonly MethodAttributes DefaultStaticMethodAttributes =
           ↪  MethodAttributes.Public | MethodAttributes.Static;
25         /// <summary>
26         /// <para>
27         /// The hide by sig.
28         /// </para>
29         /// <para></para>
30         /// </summary>
31         public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes =
           ↪  MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
           ↪  MethodAttributes.HideBySig;
32         /// <summary>
33         /// <para>
34         /// The aggressive inlining.
35         /// </para>
36         /// <para></para>
37         /// </summary>
38         public static readonly MethodImplAttributes DefaultMethodImplAttributes =
           ↪  MethodImplAttributes.IL | MethodImplAttributes.Managed |
           ↪  MethodImplAttributes.AggressiveInlining;
39
40         /// <summary>
41         /// <para>
42         /// Emits the method using the specified type.
43         /// </para>
44         /// <para></para>
45         /// </summary>
46         /// <typeparam name="TDelegate">
47         /// <para>The delegate.</para>
48         /// <para></para>
49         /// </typeparam>
50         /// <param name="type">
51         /// <para>The type.</para>
52         /// <para></para>
53         /// </param>
54         /// <param name="methodName">
55         /// <para>The method name.</para>
56         /// <para></para>
57         /// </param>
```

```csharp
        /// <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>
```

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

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Reflection;
5  using System.Runtime.CompilerServices;
6  using Platform.Collections;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     /// <summary>
13     /// <para>
14     /// Represents the type extensions.
15     /// </para>
16     /// <para></para>
17     /// </summary>
18     public static class TypeExtensions
19     {
20         /// <summary>
21         /// <para>
22         /// The static.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         static public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
        ↪   BindingFlags.NonPublic | BindingFlags.Static;
27         /// <summary>
28         /// <para>
```

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

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

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

```csharp
            };
            _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>();
```

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

```csharp
        /// <para>
        /// Determines whether is subclass of.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        /// <param name="superClass">
        /// <para>The super.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsSubclassOf(this Type type, Type superClass) =>
            type.IsSubclassOf(superClass);

        /// <summary>
        /// <para>
        /// Determines whether is value type.
        /// </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 IsValueType(this Type type) => type.IsValueType;

        /// <summary>
        /// <para>
        /// Determines whether is generic.
        /// </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 IsGeneric(this Type type) => type.IsGenericType;

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

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

```csharp
        /// </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();
```

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

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

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Collections.ObjectModel;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8  #pragma warning disable CA1819 // Properties should not return arrays
9
10  namespace Platform.Reflection
11  {
12      /// <summary>
13      /// <para>
14      /// Represents the types.
15      /// </para>
16      /// <para></para>
17      /// </summary>
18      public abstract class Types
19      {
20          /// <summary>
21          /// <para>
22          /// Gets the collection value.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          public static ReadOnlyCollection<Type> Collection { get; } = new
     ↪    ReadOnlyCollection<Type>(System.Array.Empty<Type>());
27          /// <summary>
28          /// <para>
29          /// Gets the array value.
30          /// </para>
31          /// <para></para>
32          /// </summary>
33          public static Type[] Array => Collection.ToArray();
34
35          /// <summary>
36          /// <para>
37          /// Returns the read only collection.
38          /// </para>
39          /// <para></para>
40          /// </summary>
41          /// <returns>
42          /// <para>A read only collection of type</para>
43          /// <para></para>
44          /// </returns>
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected ReadOnlyCollection<Type> ToReadOnlyCollection()
47          {
```

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

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

```
1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  #pragma warning disable CA1819 // Properties should not return arrays
7
8  namespace Platform.Reflection
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the types.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="Types"/>
17     public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
18     {
19         /// <summary>
20         /// <para>
21         /// Gets the collection value.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
                  ↪  T4, T5, T6, T7>().ToReadOnlyCollection();
26         /// <summary>
27         /// <para>
28         /// Gets the array value.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         public new static Type[] Array => Collection.ToArray();
33         private Types() { }
34     }
35 }
```

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

```
1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  #pragma warning disable CA1819 // Properties should not return arrays
7
8  namespace Platform.Reflection
9  {
10     /// <summary>
11     /// <para>
```

```csharp
     /// 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

```csharp
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

```csharp
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>
```

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

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

```
1   using System;
2   using System.Collections.ObjectModel;
3   using Platform.Collections.Lists;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6   #pragma warning disable CA1819 // Properties should not return arrays
7
8   namespace Platform.Reflection
9   {
10      /// <summary>
11      /// <para>
12      /// Represents the types.
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      /// <seealso cref="Types"/>
17      public class Types<T1, T2, T3> : Types
18      {
19          /// <summary>
20          /// <para>
21          /// Gets the collection value.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2,
            ↪  T3>().ToReadOnlyCollection();
26          /// <summary>
27          /// <para>
28          /// Gets the array value.
29          /// </para>
30          /// <para></para>
31          /// </summary>
32          public new static Type[] Array => Collection.ToArray();
33          private Types() { }
34      }
35  }
```

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

```
1   using System;
2   using System.Collections.ObjectModel;
3   using Platform.Collections.Lists;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6   #pragma warning disable CA1819 // Properties should not return arrays
7
8   namespace Platform.Reflection
9   {
10      /// <summary>
11      /// <para>
12      /// Represents the types.
13      /// </para>
14      /// <para></para>
15      /// </summary>
```

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

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

```csharp
1   using System;
2   using System.Collections.ObjectModel;
3   using Platform.Collections.Lists;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6   #pragma warning disable CA1819 // Properties should not return arrays
7
8   namespace Platform.Reflection
9   {
10      /// <summary>
11      /// <para>
12      /// Represents the types.
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      /// <seealso cref="Types"/>
17      public class Types<T> : Types
18      {
19          /// <summary>
20          /// <para>
21          /// Gets the collection value.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          public new static ReadOnlyCollection<Type> Collection { get; } = new
        ↪   Types<T>().ToReadOnlyCollection();
26          /// <summary>
27          /// <para>
28          /// Gets the array value.
29          /// </para>
30          /// <para></para>
31          /// </summary>
32          public new static Type[] Array => Collection.ToArray();
33          private Types() { }
34      }
35  }
```

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

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

```csharp
        [Fact]
        public void FailedActionCompilationTest()
        {
            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                throw new NotImplementedException();
            });
            Assert.Throws<NotSupportedException>(compiledAction);
        }

        [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());
        }

        [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++]);
        }

        [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),
```

```
 96          CompileUncheckedConverter<sbyte, uint>(extendSign: true)(-1),
 97          CompileUncheckedConverter<short, uint>(extendSign: true)(-1),
 98          CompileUncheckedConverter<int, uint>(extendSign: true)(-1),
 99          CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(-1),
100          CompileUncheckedConverter<short, ulong>(extendSign: true)(-1),
101          CompileUncheckedConverter<int, ulong>(extendSign: true)(-1),
102          CompileUncheckedConverter<long, ulong>(extendSign: true)(-1)
103      };
104      object[] withoutSignExtension = new object[]
105      {
106          CompileUncheckedConverter<sbyte, byte>(extendSign: false)(-1),
107          CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(-1),
108          CompileUncheckedConverter<short, ushort>(extendSign: false)(-1),
109          CompileUncheckedConverter<sbyte, uint>(extendSign: false)(-1),
110          CompileUncheckedConverter<short, uint>(extendSign: false)(-1),
111          CompileUncheckedConverter<int, uint>(extendSign: false)(-1),
112          CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(-1),
113          CompileUncheckedConverter<short, ulong>(extendSign: false)(-1),
114          CompileUncheckedConverter<int, ulong>(extendSign: false)(-1),
115          CompileUncheckedConverter<long, ulong>(extendSign: false)(-1)
116      };
117      var i = 0;
118      Assert.Equal((byte)255, (byte)withSignExtension[i]);
119      Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
120      Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
121      Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
122      Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
123      Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
124      Assert.Equal(4294967295, withSignExtension[i]);
125      Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
126      Assert.Equal(4294967295, withSignExtension[i]);
127      Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
128      Assert.Equal(4294967295, withSignExtension[i]);
129      Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
130      Assert.Equal(18446744073709551615, withSignExtension[i]);
131      Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
132      Assert.Equal(18446744073709551615, withSignExtension[i]);
133      Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
134      Assert.Equal(18446744073709551615, withSignExtension[i]);
135      Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
136      Assert.Equal(18446744073709551615, withSignExtension[i]);
137      Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
138  }
139
140  [Fact]
141  public void SignedIntegersConversionOfTwoWithSignExtensionTest()
142  {
143      object[] withSignExtension = new object[]
144      {
145          CompileUncheckedConverter<sbyte, byte>(extendSign: true)(2),
146          CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(2),
147          CompileUncheckedConverter<short, ushort>(extendSign: true)(2),
148          CompileUncheckedConverter<sbyte, uint>(extendSign: true)(2),
149          CompileUncheckedConverter<short, uint>(extendSign: true)(2),
150          CompileUncheckedConverter<int, uint>(extendSign: true)(2),
151          CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(2),
152          CompileUncheckedConverter<short, ulong>(extendSign: true)(2),
153          CompileUncheckedConverter<int, ulong>(extendSign: true)(2),
154          CompileUncheckedConverter<long, ulong>(extendSign: true)(2)
155      };
156      object[] withoutSignExtension = new object[]
157      {
158          CompileUncheckedConverter<sbyte, byte>(extendSign: false)(2),
159          CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(2),
160          CompileUncheckedConverter<short, ushort>(extendSign: false)(2),
161          CompileUncheckedConverter<sbyte, uint>(extendSign: false)(2),
162          CompileUncheckedConverter<short, uint>(extendSign: false)(2),
163          CompileUncheckedConverter<int, uint>(extendSign: false)(2),
164          CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(2),
165          CompileUncheckedConverter<short, ulong>(extendSign: false)(2),
166          CompileUncheckedConverter<int, ulong>(extendSign: false)(2),
167          CompileUncheckedConverter<long, ulong>(extendSign: false)(2)
168      };
169      for (var i = 0; i < withSignExtension.Length; i++)
170      {
171          Assert.Equal(2UL, Convert.ToUInt64(withSignExtension[i]));
172          Assert.Equal(withSignExtension[i], withoutSignExtension[i]);
173      }
```

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

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

```
1  using System;
2  using System.Reflection;
3  using Xunit;
4  using Platform.Collections;
5  using Platform.Collections.Lists;
6
7  namespace Platform.Reflection.Tests
8  {
9      public static class GetILBytesMethodTests
10     {
11         [Fact]
12         public static void ILBytesForDelegateAreAvailableTest()
13         {
14             var function = new Func<object, int>(argument => 0);
15             var bytes = function.GetMethodInfo().GetILBytes();
16             Assert.False(bytes.IsNullOrEmpty());
17         }
18
19         [Fact]
20         public static void ILBytesForDifferentDelegatesAreTheSameTest()
21         {
22             var firstFunction = new Func<object, int>(argument => 0);
23             var secondFunction = new Func<object, int>(argument => 0);
24             Assert.False(firstFunction == secondFunction);
25             var firstFunctionBytes = firstFunction.GetMethodInfo().GetILBytes();
26             Assert.False(firstFunctionBytes.IsNullOrEmpty());
27             var secondFunctionBytes = secondFunction.GetMethodInfo().GetILBytes();
28             Assert.False(secondFunctionBytes.IsNullOrEmpty());
29             Assert.True(firstFunctionBytes.EqualTo(secondFunctionBytes));
30         }
31     }
32 }
```

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

```
1  using Xunit;
2
3  namespace Platform.Reflection.Tests
4  {
5      public class NumericTypeTests
6      {
7          [Fact]
8          public void UInt64IsNumericTest()
9          {
10             Assert.True(NumericType<ulong>.IsNumeric);
11         }
12     }
13 }
```

Index