LinksPlatform's Platform.Reflection Class Library

1.1 ./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
{
    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);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
        ↪  _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
    }
}
```

1.2 ./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
{
    public static class DelegateHelpers
    {
        [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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
        ↪  TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
        ↪  typeMemberMethod)
```

```csharp
                    where TDelegate : Delegate
        {
            var @delegate = CompileOrDefault<TDelegate>(emitCode, typeMemberMethod);
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
                @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
            }
            return @delegate;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
        ↪   : Delegate => Compile<TDelegate>(emitCode, false);

        [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);
        }

        [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 ./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
{
    public static class DynamicExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool HasProperty(this object @object, string propertyName)
        {
            var type = @object.GetType();
            if (type is IDictionary<string, object> dictionary)
            {
                return dictionary.ContainsKey(propertyName);
            }
            return type.GetProperty(propertyName) != null;
        }
    }
}
```

1.4 ./Platform.Reflection/EnsureExtensions.cs

```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

```

```csharp
namespace Platform.Reflection
{
    public static class EnsureExtensions
    {
        #region Always

        [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());
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
        ↪ message)
        {
            string messageBuilder() => message;
            IsUnsignedInteger<T>(root, messageBuilder);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
        ↪ IsUnsignedInteger<T>(root, (string)null);

        [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());
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
        ↪ message)
        {
            string messageBuilder() => message;
            IsSignedInteger<T>(root, messageBuilder);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
        ↪ IsSignedInteger<T>(root, (string)null);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
        ↪ messageBuilder)
        {
            if (!NumericType<T>.IsSigned)
            {
                throw new NotSupportedException(messageBuilder());
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
        {
            string messageBuilder() => message;
            IsSigned<T>(root, messageBuilder);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
        ↪ (string)null);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
        ↪ messageBuilder)
        {
            if (!NumericType<T>.IsNumeric)
```

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

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
    {
        string messageBuilder() => message;
        IsNumeric<T>(root, messageBuilder);
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
        IsNumeric<T>(root, (string)null);

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
        messageBuilder)
    {
        if (!NumericType<T>.CanBeNumeric)
        {
            throw new NotSupportedException(messageBuilder());
        }
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
    {
        string messageBuilder() => message;
        CanBeNumeric<T>(root, messageBuilder);
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
        CanBeNumeric<T>(root, (string)null);

    #endregion

    #region OnDebug

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

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

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

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

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

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

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

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

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

    [Conditional("DEBUG")]
```

```
143         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
      ↪  messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);

144
145         [Conditional("DEBUG")]
146         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
      ↪  Ensure.Always.IsNumeric<T>(message);

147
148         [Conditional("DEBUG")]
149         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
      ↪  Ensure.Always.IsNumeric<T>();

150
151         [Conditional("DEBUG")]
152         public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
      ↪  messageBuilder) => Ensure.Always.CanBeNumeric<T>(messageBuilder);

153
154         [Conditional("DEBUG")]
155         public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
      ↪  => Ensure.Always.CanBeNumeric<T>(message);

156
157         [Conditional("DEBUG")]
158         public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
      ↪  Ensure.Always.CanBeNumeric<T>();

159
160         #endregion
161     }
162 }
```

1.5   ./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      public static class FieldInfoExtensions
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T GetStaticValue<T>(this FieldInfo fieldInfo) =>
      ↪  (T)fieldInfo.GetValue(null);
12     }
13 }
```

1.6   ./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     public static class ILGeneratorExtensions
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void Throw<T>(this ILGenerator generator) =>
      ↪  generator.ThrowException(typeof(T));

15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator) =>
      ↪  UncheckedConvert<TSource, TTarget>(generator, extendSign: false);

17
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator, bool
      ↪  extendSign)
21         {
22             var sourceType = typeof(TSource);
23             var targetType = typeof(TTarget);
24             if (sourceType == targetType)
25             {
26                 return;
27             }
28             if (extendSign)
29             {
30                 if (sourceType == typeof(byte))
31                 {
```

```csharp
                    generator.Emit(OpCodes.Conv_I1);
                }
                if (sourceType == typeof(ushort))
                {
                    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))
            {
                generator.Emit(OpCodes.Conv_U2);
            }
            else if (targetType == typeof(int))
            {
                generator.Emit(OpCodes.Conv_I4);
            }
```

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

        [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))
            {
                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
                {
```

```csharp
                                generator.Emit(OpCodes.Conv_Ovf_I4_Un);
                            }
                        }
                        else if (targetType == typeof(uint))
                        {
                            if (NumericType<TSource>.IsSigned)
                            {
                                generator.Emit(OpCodes.Conv_Ovf_U4);
                            }
                            else
                            {
                                generator.Emit(OpCodes.Conv_Ovf_U4_Un);
                            }
                        }
                        else if (targetType == typeof(long))
                        {
                            if (NumericType<TSource>.IsSigned)
                            {
                                generator.Emit(OpCodes.Conv_Ovf_I8);
                            }
                            else
                            {
                                generator.Emit(OpCodes.Conv_Ovf_I8_Un);
                            }
                        }
                        else if (targetType == typeof(ulong))
                        {
                            if (NumericType<TSource>.IsSigned)
                            {
                                generator.Emit(OpCodes.Conv_Ovf_U8);
                            }
                            else
                            {
                                generator.Emit(OpCodes.Conv_Ovf_U8_Un);
                            }
                        }
                        else if (targetType == typeof(float))
                        {
                            if (NumericType<TSource>.IsSigned)
                            {
                                generator.Emit(OpCodes.Conv_R4);
                            }
                            else
                            {
                                generator.Emit(OpCodes.Conv_R_Un);
                            }
                        }
                        else if (targetType == typeof(double))
                        {
                            generator.Emit(OpCodes.Conv_R8);
                        }
                        else if (targetType == typeof(bool))
                        {
                            generator.ConvertToBoolean<TSource>();
                        }
                        else
                        {
                            throw new NotSupportedException();
                        }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, bool value) =>
            generator.LoadConstant(value ? 1 : 0);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, float value) =>
            generator.Emit(OpCodes.Ldc_R4, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, double value) =>
            generator.Emit(OpCodes.Ldc_R8, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, ulong value) =>
            generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
261        public static void LoadConstant(this ILGenerator generator, long value) =>
      ↪   generator.Emit(OpCodes.Ldc_I8, value);

262
263        [MethodImpl(MethodImplOptions.AggressiveInlining)]
264        public static void LoadConstant(this ILGenerator generator, uint value)
265        {
266            switch (value)
267            {
268                case uint.MaxValue:
269                    generator.Emit(OpCodes.Ldc_I4_M1);
270                    return;
271                case 0:
272                    generator.Emit(OpCodes.Ldc_I4_0);
273                    return;
274                case 1:
275                    generator.Emit(OpCodes.Ldc_I4_1);
276                    return;
277                case 2:
278                    generator.Emit(OpCodes.Ldc_I4_2);
279                    return;
280                case 3:
281                    generator.Emit(OpCodes.Ldc_I4_3);
282                    return;
283                case 4:
284                    generator.Emit(OpCodes.Ldc_I4_4);
285                    return;
286                case 5:
287                    generator.Emit(OpCodes.Ldc_I4_5);
288                    return;
289                case 6:
290                    generator.Emit(OpCodes.Ldc_I4_6);
291                    return;
292                case 7:
293                    generator.Emit(OpCodes.Ldc_I4_7);
294                    return;
295                case 8:
296                    generator.Emit(OpCodes.Ldc_I4_8);
297                    return;
298                default:
299                    if (value <= sbyte.MaxValue)
300                    {
301                        generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
302                    }
303                    else
304                    {
305                        generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
306                    }
307                    return;
308            }
309        }

310
311        [MethodImpl(MethodImplOptions.AggressiveInlining)]
312        public static void LoadConstant(this ILGenerator generator, int value)
313        {
314            switch (value)
315            {
316                case -1:
317                    generator.Emit(OpCodes.Ldc_I4_M1);
318                    return;
319                case 0:
320                    generator.Emit(OpCodes.Ldc_I4_0);
321                    return;
322                case 1:
323                    generator.Emit(OpCodes.Ldc_I4_1);
324                    return;
325                case 2:
326                    generator.Emit(OpCodes.Ldc_I4_2);
327                    return;
328                case 3:
329                    generator.Emit(OpCodes.Ldc_I4_3);
330                    return;
331                case 4:
332                    generator.Emit(OpCodes.Ldc_I4_4);
333                    return;
334                case 5:
335                    generator.Emit(OpCodes.Ldc_I4_5);
336                    return;
337                case 6:
338                    generator.Emit(OpCodes.Ldc_I4_6);
339                    return;
340                case 7:
```

```csharp
                            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;
                }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, short value) =>
            generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, ushort value) =>
            generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, sbyte value) =>
            generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, byte value) =>
            generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
            LoadConstantOne(generator, typeof(TConstant));

        [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))
```

```csharp
                {
                    generator.LoadConstant((byte)1);
                }
                else
                {
                    throw new NotSupportedException();
                }
            }

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

            [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();
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void Increment<TValue>(this ILGenerator generator) =>
            ↪   generator.Increment(typeof(TValue));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void Decrement<TValue>(this ILGenerator generator) =>
            ↪   generator.Decrement(typeof(TValue));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void Increment(this ILGenerator generator, Type valueType)
            {
                generator.LoadConstantOne(valueType);
                generator.Add();
            }
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Decrement(this ILGenerator generator, Type valueType)
        {
            generator.LoadConstantOne(valueType);
            generator.Subtract();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);

        [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;
            }
        }

        [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]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
        ↪  generator.Emit(OpCodes.Starg, argumentIndex);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareGreaterThan(this ILGenerator generator) =>
        ↪  generator.Emit(OpCodes.Cgt);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
        ↪  generator.Emit(OpCodes.Cgt_Un);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
```

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void CompareLessThan(this ILGenerator generator) =>
            ↪   generator.Emit(OpCodes.Clt);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void UnsignedCompareLessThan(this ILGenerator generator) =>
            ↪   generator.Emit(OpCodes.Clt_Un);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void CompareLessThan(this ILGenerator generator, bool isSigned)
            {
                if (isSigned)
                {
                    generator.CompareLessThan();
                }
                else
                {
                    generator.UnsignedCompareLessThan();
                }
            }

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
            ↪   label) => generator.Emit(OpCodes.Bge_Un, label);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
            ↪   Label label)
            {
                if (isSigned)
                {
                    generator.BranchIfGreaterOrEqual(label);
                }
                else
                {
                    generator.UnsignedBranchIfGreaterOrEqual(label);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
            ↪   generator.Emit(OpCodes.Ble, label);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
            ↪   => generator.Emit(OpCodes.Ble_Un, label);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
            ↪   label)
            {
                if (isSigned)
                {
                    generator.BranchIfLessOrEqual(label);
                }
                else
                {
                    generator.UnsignedBranchIfLessOrEqual(label);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Call(this ILGenerator generator, MethodInfo method) =>
        ↪  generator.Emit(OpCodes.Call, method);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Unbox<TUnbox>(this ILGenerator generator) =>
        ↪  generator.Unbox(typeof(TUnbox));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
        ↪  generator.Emit(OpCodes.Unbox, typeToUnbox);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
        ↪  generator.UnboxValue(typeof(TUnbox));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
        ↪  generator.Emit(OpCodes.Unbox_Any, typeToUnbox);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
        ↪  generator.DeclareLocal(typeof(T));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
        ↪  generator.Emit(OpCodes.Ldloc, local);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
        ↪  generator.Emit(OpCodes.Stloc, local);

        [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);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
        ↪  generator.Emit(OpCodes.Newobj, constructor);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
        ↪  byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);

        [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");
```

```csharp
            }
            if (isVolatile)
            {
                generator.Emit(OpCodes.Volatile);
            }
            if (unaligned.HasValue)
            {
                generator.Emit(OpCodes.Unaligned, unaligned.Value);
            }
            if (type.IsPointer)
            {
                generator.Emit(OpCodes.Ldind_I);
            }
            else if (!type.IsValueType)
            {
                generator.Emit(OpCodes.Ldind_Ref);
            }
            else if (type == typeof(sbyte))
            {
                generator.Emit(OpCodes.Ldind_I1);
            }
            else if (type == typeof(bool))
            {
                generator.Emit(OpCodes.Ldind_I1);
            }
            else if (type == typeof(byte))
            {
                generator.Emit(OpCodes.Ldind_U1);
            }
            else if (type == typeof(short))
            {
                generator.Emit(OpCodes.Ldind_I2);
            }
            else if (type == typeof(ushort))
            {
                generator.Emit(OpCodes.Ldind_U2);
            }
            else if (type == typeof(char))
            {
                generator.Emit(OpCodes.Ldind_U2);
            }
            else if (type == typeof(int))
            {
                generator.Emit(OpCodes.Ldind_I4);
            }
            else if (type == typeof(uint))
            {
                generator.Emit(OpCodes.Ldind_U4);
            }
            else if (type == typeof(long) || type == typeof(ulong))
            {
                generator.Emit(OpCodes.Ldind_I8);
            }
            else if (type == typeof(float))
            {
                generator.Emit(OpCodes.Ldind_R4);
            }
            else if (type == typeof(double))
            {
                generator.Emit(OpCodes.Ldind_R8);
            }
            else
            {
                throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
                    ", LoadObject may be more appropriate");
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);

        [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)
            {
```

```
767                         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
768                     }
769                     if (isVolatile)
770                     {
771                         generator.Emit(OpCodes.Volatile);
772                     }
773                     if (unaligned.HasValue)
774                     {
775                         generator.Emit(OpCodes.Unaligned, unaligned.Value);
776                     }
777                     if (type.IsPointer)
778                     {
779                         generator.Emit(OpCodes.Stind_I);
780                     }
781                     else if (!type.IsValueType)
782                     {
783                         generator.Emit(OpCodes.Stind_Ref);
784                     }
785                     else if (type == typeof(sbyte) || type == typeof(byte))
786                     {
787                         generator.Emit(OpCodes.Stind_I1);
788                     }
789                     else if (type == typeof(short) || type == typeof(ushort))
790                     {
791                         generator.Emit(OpCodes.Stind_I2);
792                     }
793                     else if (type == typeof(int) || type == typeof(uint))
794                     {
795                         generator.Emit(OpCodes.Stind_I4);
796                     }
797                     else if (type == typeof(long) || type == typeof(ulong))
798                     {
799                         generator.Emit(OpCodes.Stind_I8);
800                     }
801                     else if (type == typeof(float))
802                     {
803                         generator.Emit(OpCodes.Stind_R4);
804                     }
805                     else if (type == typeof(double))
806                     {
807                         generator.Emit(OpCodes.Stind_R8);
808                     }
809                     else
810                     {
811                         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
812                         ↪  + ", StoreObject may be more appropriate");
813                     }
814                 }
815             }
        }
```

1.7 ./Platform.Reflection/MethodInfoExtensions.cs

```
1   using System;
2   using System.Linq;
3   using System.Reflection;
4   using System.Runtime.CompilerServices;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Reflection
9   {
10      public static class MethodInfoExtensions
11      {
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public static byte[] GetILBytes(this MethodInfo methodInfo) =>
            ↪  methodInfo.GetMethodBody().GetILAsByteArray();
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
            ↪  methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
17      }
18  }
```

1.8 ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Interfaces;
```

```csharp
5
6    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8    namespace Platform.Reflection
9    {
10       public class NotSupportedExceptionDelegateFactory<TDelegate> : IFactory<TDelegate>
11           where TDelegate : Delegate
12       {
13           [MethodImpl(MethodImplOptions.AggressiveInlining)]
14           public TDelegate Create()
15           {
16               var @delegate = DelegateHelpers.CompileOrDefault<TDelegate>(generator =>
17               {
18                   generator.Throw<NotSupportedException>();
19               });
20               if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
21               {
22                   throw new InvalidOperationException("Unable to compile stub delegate.");
23               }
24               return @delegate;
25           }
26       }
27    }
```

1.9 ./Platform.Reflection/NumericType.cs

```csharp
1    using System;
2    using System.Runtime.CompilerServices;
3    using System.Runtime.InteropServices;
4    using Platform.Exceptions;
5
6    // ReSharper disable AssignmentInConditionalExpression
7    // ReSharper disable BuiltInTypeReferenceStyle
8    // ReSharper disable StaticFieldInGenericType
9    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11   namespace Platform.Reflection
12   {
13       public static class NumericType<T>
14       {
15           public static readonly Type Type;
16           public static readonly Type UnderlyingType;
17           public static readonly Type SignedVersion;
18           public static readonly Type UnsignedVersion;
19           public static readonly bool IsFloatPoint;
20           public static readonly bool IsNumeric;
21           public static readonly bool IsSigned;
22           public static readonly bool CanBeNumeric;
23           public static readonly bool IsNullable;
24           public static readonly int BytesSize;
25           public static readonly int BitsSize;
26           public static readonly T MinValue;
27           public static readonly T MaxValue;
28
29           [MethodImpl(MethodImplOptions.AggressiveInlining)]
30           static NumericType()
31           {
32               try
33               {
34                   var type = typeof(T);
35                   var isNullable = type.IsNullable();
36                   var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
37                   var canBeNumeric = underlyingType.CanBeNumeric();
38                   var isNumeric = underlyingType.IsNumeric();
39                   var isSigned = underlyingType.IsSigned();
40                   var isFloatPoint = underlyingType.IsFloatPoint();
41                   var bytesSize = Marshal.SizeOf(underlyingType);
42                   var bitsSize = bytesSize * 8;
43                   GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
44                   GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
                       ↪   out Type unsignedVersion);
45                   Type = type;
46                   IsNullable = isNullable;
47                   UnderlyingType = underlyingType;
48                   CanBeNumeric = canBeNumeric;
49                   IsNumeric = isNumeric;
50                   IsSigned = isSigned;
51                   IsFloatPoint = isFloatPoint;
52                   BytesSize = bytesSize;
53                   BitsSize = bitsSize;
54                   MinValue = minValue;
55                   MaxValue = maxValue;
```

```csharp
                    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 ./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
{
    public static class PropertyInfoExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticValue<T>(this PropertyInfo fieldInfo) =>
        ↪   (T)fieldInfo.GetValue(null);
    }
}
```

1.11 ./Platform.Reflection/TypeBuilderExtensions.cs

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

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

namespace Platform.Reflection
{
    public static class TypeBuilderExtensions
    {
        public static readonly MethodAttributes DefaultStaticMethodAttributes =
        ↪   MethodAttributes.Public | MethodAttributes.Static;
        public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes =
        ↪   MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
        ↪   MethodAttributes.HideBySig;
        public static readonly MethodImplAttributes DefaultMethodImplAttributes =
        ↪   MethodImplAttributes.IL | MethodImplAttributes.Managed |
        ↪   MethodImplAttributes.AggressiveInlining;
```

```csharp
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName,
            ↪  MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes,
            ↪  Action<ILGenerator> emitCode)
18          {
19              typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[]
                ↪  parameterTypes);
20              EmitMethod(type, methodName, methodAttributes, methodImplAttributes, returnType,
                ↪  parameterTypes, emitCode);
21          }
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
            ↪  methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[]
            ↪  parameterTypes, Action<ILGenerator> emitCode)
25          {
26              MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
                ↪  parameterTypes);
27              method.SetImplementationFlags(methodImplAttributes);
28              var generator = method.GetILGenerator();
29              emitCode(generator);
30          }
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName,
            ↪  Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
            ↪  DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public static void EmitFinalVirtualMethod<TDelegate>(this TypeBuilder type, string
            ↪  methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
            ↪  DefaultFinalVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
37      }
38  }
```

1.12   ./Platform.Reflection/TypeExtensions.cs

```csharp
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      public static class TypeExtensions
13      {
14          static public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
            ↪  BindingFlags.NonPublic | BindingFlags.Static;
15          static public readonly string DefaultDelegateMethodName = "Invoke";
16
17          static private readonly HashSet<Type> _canBeNumericTypes;
18          static private readonly HashSet<Type> _isNumericTypes;
19          static private readonly HashSet<Type> _isSignedTypes;
20          static private readonly HashSet<Type> _isFloatPointTypes;
21          static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
22          static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          static TypeExtensions()
26          {
27              _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
                ↪  typeof(DateTime), typeof(TimeSpan) };
28              _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
                ↪  typeof(ulong) };
29              _canBeNumericTypes.UnionWith(_isNumericTypes);
30              _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
                ↪  typeof(long) };
31              _canBeNumericTypes.UnionWith(_isSignedTypes);
32              _isNumericTypes.UnionWith(_isSignedTypes);
33              _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
                ↪  typeof(float) };
34              _canBeNumericTypes.UnionWith(_isFloatPointTypes);
35              _isNumericTypes.UnionWith(_isFloatPointTypes);
36              _isSignedTypes.UnionWith(_isFloatPointTypes);
37              _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticFieldValue<T>(this Type type, string name) =>
        ↪   type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticPropertyValue<T>(this Type type, string name) =>
        ↪   type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
        ↪   genericParameterTypes, Type[] argumentTypes)
        {
            var methods = from m in type.GetMethods()
                          where m.Name == name
                             && m.IsGenericMethodDefinition
                          let typeParams = m.GetGenericArguments()
                          let normalParams = m.GetParameters().Select(x => x.ParameterType)
                          where typeParams.SequenceEqual(genericParameterTypes)
                             && normalParams.SequenceEqual(argumentTypes)
                          select m;
            var method = methods.Single();
            return method;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetBaseType(this Type type) => type.BaseType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Assembly GetAssembly(this Type type) => type.Assembly;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsSubclassOf(this Type type, Type superClass) =>
        ↪   type.IsSubclassOf(superClass);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsValueType(this Type type) => type.IsValueType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsGeneric(this Type type) => type.IsGenericType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
        ↪   type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetUnsignedVersionOrNull(this Type signedType) =>
        ↪   _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetSignedVersionOrNull(this Type unsignedType) =>
        ↪   _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
```

```csharp
110        [MethodImpl(MethodImplOptions.AggressiveInlining)]
111        public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
112
113        [MethodImpl(MethodImplOptions.AggressiveInlining)]
114        public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
115
116        [MethodImpl(MethodImplOptions.AggressiveInlining)]
117        public static Type GetDelegateReturnType(this Type delegateType) =>
     ↪    delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;
118
119        [MethodImpl(MethodImplOptions.AggressiveInlining)]
120        public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
     ↪    delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();
121
122        [MethodImpl(MethodImplOptions.AggressiveInlining)]
123        public static void GetDelegateCharacteristics(this Type delegateType, out Type
     ↪    returnType, out Type[] parameterTypes)
124        {
125            var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
126            returnType = invoke.ReturnType;
127            parameterTypes = invoke.GetParameterTypes();
128        }
129    }
130 }
```

1.13 ./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     public abstract class Types
13     {
14         public static ReadOnlyCollection<Type> Collection { get; } = new
     ↪    ReadOnlyCollection<Type>(System.Array.Empty<Type>());
15         public static Type[] Array => Collection.ToArray();
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected ReadOnlyCollection<Type> ToReadOnlyCollection()
19         {
20             var types = GetType().GetGenericArguments();
21             var result = new List<Type>();
22             AppendTypes(result, types);
23             return new ReadOnlyCollection<Type>(result);
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         private static void AppendTypes(List<Type> container, IList<Type> types)
28         {
29             for (var i = 0; i < types.Count; i++)
30             {
31                 var element = types[i];
32                 if (element != typeof(Types))
33                 {
34                     if (element.IsSubclassOf(typeof(Types)))
35                     {
36                         AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection
     ↪    <Type>>(nameof(Types<object>.Collection)));
37                     }
38                     else
39                     {
40                         container.Add(element);
41                     }
42                 }
43             }
44         }
45     }
46 }
```

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

```csharp
1  using System;
2  using System.Collections.ObjectModel;
```

```
3  using Platform.Collections.Lists;

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

8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
           ↪  T4, T5, T6, T7>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16  }
```

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

```
1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;

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

8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2, T3, T4, T5, T6> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
           ↪  T4, T5, T6>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16  }
```

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

```
1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;

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

8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2, T3, T4, T5> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
           ↪  T4, T5>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16  }
```

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

```
1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;

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

8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2, T3, T4> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
           ↪  T4>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16  }
```

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

```
1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;

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

```csharp
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2, T3> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2,
        ↪  T3>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.19   ./Platform.Reflection/Types[T1, T2].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
{
    public class Types<T1, T2> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1,
        ↪  T2>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.20   ./Platform.Reflection/Types[T].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
{
    public class Types<T> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new
        ↪  Types<T>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```csharp
using System;
using Xunit;

namespace Platform.Reflection.Tests
{
    public class CodeGenerationTests
    {
        [Fact]
        public void EmptyActionCompilationTest()
        {
            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                generator.Return();
            });
            compiledAction();
        }

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

```csharp
        [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),
                CompileUncheckedConverter<sbyte, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<short, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<int, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<short, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<int, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<long, ulong>(extendSign: true)(-1)
            };
            object[] withoutSignExtension = new object[]
```

```
106             {
107                 CompileUncheckedConverter<sbyte, byte>(extendSign: false)(-1),
108                 CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(-1),
109                 CompileUncheckedConverter<short, ushort>(extendSign: false)(-1),
110                 CompileUncheckedConverter<sbyte, uint>(extendSign: false)(-1),
111                 CompileUncheckedConverter<short, uint>(extendSign: false)(-1),
112                 CompileUncheckedConverter<int, uint>(extendSign: false)(-1),
113                 CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(-1),
114                 CompileUncheckedConverter<short, ulong>(extendSign: false)(-1),
115                 CompileUncheckedConverter<int, ulong>(extendSign: false)(-1),
116                 CompileUncheckedConverter<long, ulong>(extendSign: false)(-1)
117             };
118             var i = 0;
119             Assert.Equal((byte)255, (byte)withSignExtension[i]);
120             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
121             Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
122             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
123             Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
124             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
125             Assert.Equal(4294967295, withSignExtension[i]);
126             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
127             Assert.Equal(4294967295, withSignExtension[i]);
128             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
129             Assert.Equal(4294967295, withSignExtension[i]);
130             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
131             Assert.Equal(18446744073709551615, withSignExtension[i]);
132             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
133             Assert.Equal(18446744073709551615, withSignExtension[i]);
134             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
135             Assert.Equal(18446744073709551615, withSignExtension[i]);
136             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
137             Assert.Equal(18446744073709551615, withSignExtension[i]);
138             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
139         }
140
141         [Fact]
142         public void SignedIntegersConversionOfTwoWithSignExtensionTest()
143         {
144             object[] withSignExtension = new object[]
145             {
146                 CompileUncheckedConverter<sbyte, byte>(extendSign: true)(2),
147                 CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(2),
148                 CompileUncheckedConverter<short, ushort>(extendSign: true)(2),
149                 CompileUncheckedConverter<sbyte, uint>(extendSign: true)(2),
150                 CompileUncheckedConverter<short, uint>(extendSign: true)(2),
151                 CompileUncheckedConverter<int, uint>(extendSign: true)(2),
152                 CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(2),
153                 CompileUncheckedConverter<short, ulong>(extendSign: true)(2),
154                 CompileUncheckedConverter<int, ulong>(extendSign: true)(2),
155                 CompileUncheckedConverter<long, ulong>(extendSign: true)(2)
156             };
157             object[] withoutSignExtension = new object[]
158             {
159                 CompileUncheckedConverter<sbyte, byte>(extendSign: false)(2),
160                 CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(2),
161                 CompileUncheckedConverter<short, ushort>(extendSign: false)(2),
162                 CompileUncheckedConverter<sbyte, uint>(extendSign: false)(2),
163                 CompileUncheckedConverter<short, uint>(extendSign: false)(2),
164                 CompileUncheckedConverter<int, uint>(extendSign: false)(2),
165                 CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(2),
166                 CompileUncheckedConverter<short, ulong>(extendSign: false)(2),
167                 CompileUncheckedConverter<int, ulong>(extendSign: false)(2),
168                 CompileUncheckedConverter<long, ulong>(extendSign: false)(2)
169             };
170             for (var i = 0; i < withSignExtension.Length; i++)
171             {
172                 Assert.Equal(2UL, Convert.ToUInt64(withSignExtension[i]));
173                 Assert.Equal(withSignExtension[i], withoutSignExtension[i]);
174             }
175         }
176
177         private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
        ↪  TTarget>(bool extendSign)
178         {
179             return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
180             {
181                 generator.LoadArgument(0);
182                 generator.UncheckedConvert<TSource, TTarget>(extendSign);
```

```
183                    generator.Return();
184                });
185            }
186        }
187    }
```

1.22   ./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   ./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