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)
```

```
35              where TDelegate : Delegate
36          {
37              var @delegate = CompileOrDefault<TDelegate>(emitCode, typeMemberMethod);
38              if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
39              {
40                  @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
41              }
42              return @delegate;
43          }
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
        ↪   : Delegate => Compile<TDelegate>(emitCode, false);
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          public static TDelegate CompileDynamicMethod<TDelegate>(Action<ILGenerator> emitCode)
50          {
51              var delegateType = typeof(TDelegate);
52              delegateType.GetDelegateCharacteristics(out Type returnType, out Type[]
        ↪   parameterTypes);
53              var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
54              emitCode(dynamicMethod.GetILGenerator());
55              return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
56          }
57
58          [MethodImpl(MethodImplOptions.AggressiveInlining)]
59          public static TDelegate CompileTypeMemberMethod<TDelegate>(Action<ILGenerator> emitCode)
60          {
61              AssemblyName assemblyName = new AssemblyName(GetNewName());
62              var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
        ↪   AssemblyBuilderAccess.Run);
63              var module = assembly.DefineDynamicModule(GetNewName());
64              var type = module.DefineType(GetNewName());
65              var methodName = GetNewName();
66              type.EmitStaticMethod<TDelegate>(methodName, emitCode);
67              var typeInfo = type.CreateTypeInfo();
68              return (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(typeof(TDele
        ↪   gate));
69          }
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          private static string GetNewName() => Guid.NewGuid().ToString("N");
73      }
74  }
```

1.3 ./Platform.Reflection/DynamicExtensions.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Reflection
7  {
8      public static class DynamicExtensions
9      {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public static bool HasProperty(this object @object, string propertyName)
12          {
13              var type = @object.GetType();
14              if (type is IDictionary<string, object> dictionary)
15              {
16                  return dictionary.ContainsKey(propertyName);
17              }
18              return type.GetProperty(propertyName) != null;
19          }
20      }
21  }
```

1.4 ./Platform.Reflection/EnsureExtensions.cs

```
1  using System;
2  using System.Diagnostics;
3  using System.Runtime.CompilerServices;
4  using Platform.Exceptions;
5  using Platform.Exceptions.ExtensionRoots;
6
7  #pragma warning disable IDE0060 // Remove unused parameter
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
```

```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
17         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator) =>
       ↪  UncheckedConvert<TSource, TTarget>(generator, extendSign: false);

18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19
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                 {
```

```
                                    generator.Emit(OpCodes.Conv_I1);
                                }
                                if (sourceType == typeof(ushort))
                                {
                                    generator.Emit(OpCodes.Conv_I2);
                                }
                        }
                        if (NumericType<TSource>.BitsLength > NumericType<TTarget>.BitsLength)
                        {
                                if (targetType == typeof(short))
                                {
                                    generator.Emit(OpCodes.Conv_I2);
                                }
                                else if (targetType == typeof(ushort))
                                {
                                    generator.Emit(OpCodes.Conv_U2);
                                }
                                else if (targetType == typeof(sbyte))
                                {
                                    generator.Emit(OpCodes.Conv_I1);
                                }
                                else if (targetType == typeof(byte))
                                {
                                    generator.Emit(OpCodes.Conv_U1);
                                }
                                else if (targetType == typeof(int))
                                {
                                    generator.Emit(OpCodes.Conv_I4);
                                }
                                else if (targetType == typeof(uint))
                                {
                                    generator.Emit(OpCodes.Conv_U4);
                                }
                                else if (targetType == typeof(long))
                                {
                                    generator.Emit(OpCodes.Conv_I8);
                                }
                                else if (targetType == typeof(ulong))
                                {
                                    generator.Emit(OpCodes.Conv_U8);
                                }
                        }
                        else
                        {
                                if (!extendSign)
                                {
                                    if (sourceType == typeof(uint) && targetType == typeof(long))
                                    {
                                        generator.Emit(OpCodes.Conv_U8);
                                    }
                                }
                        }
                        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);
                        }
                }

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
                {
                        var type = typeof(TTarget);
                        if (type == typeof(short))
                        {
                                if (NumericType<TSource>.IsSigned)
                                {
                                    generator.Emit(OpCodes.Conv_Ovf_I2);
```

```csharp
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2_Un);
                }
            }
            else if (type == typeof(ushort))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U2);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U2_Un);
                }
            }
            else if (type == typeof(sbyte))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I1);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I1_Un);
                }
            }
            else if (type == typeof(byte))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U1);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U1_Un);
                }
            }
            else if (type == typeof(int))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I4_Un);
                }
            }
            else if (type == typeof(uint))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U4_Un);
                }
            }
            else if (type == typeof(long))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I8);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I8_Un);
                }
            }
            else if (type == typeof(ulong))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U8);
                }
```

```csharp
                    else
                    {
                        generator.Emit(OpCodes.Conv_Ovf_U8_Un);
                    }
                }
                else if (type == typeof(float))
                {
                    if (NumericType<TSource>.IsSigned)
                    {
                        generator.Emit(OpCodes.Conv_R4);
                    }
                    else
                    {
                        generator.Emit(OpCodes.Conv_R_Un);
                    }
                }
                else if (type == typeof(double))
                {
                    generator.Emit(OpCodes.Conv_R8);
                }
                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)]
            public static void LoadConstant(this ILGenerator generator, long value) =>
                generator.Emit(OpCodes.Ldc_I8, value);

            [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:
```

```csharp
                        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;
            }
        }

        [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;
            }
        }

        [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)]
```

```csharp
        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))
            {
                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))
```

```csharp
                {
                    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();
        }

        [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)
```

```csharp
                {
                    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)
            {
                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);
```

```csharp
[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));

[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);
```

```csharp
        [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");
            }
            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);
```

```csharp
            }
            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)
            {
                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))
            {
```

```
769             generator.Emit(OpCodes.Stind_R4);
770         }
771         else if (type == typeof(double))
772         {
773             generator.Emit(OpCodes.Stind_R8);
774         }
775         else
776         {
777             throw new InvalidOperationException("StoreIndirect cannot be used with " + type
        ↪  + ", StoreObject may be more appropriate");
778         }
779     }
780     }
781 }
```

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;
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

```
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 {
```

```csharp
public static class NumericType<T>
{
    public static readonly Type Type;
    public static readonly Type UnderlyingType;
    public static readonly Type SignedVersion;
    public static readonly Type UnsignedVersion;
    public static readonly bool IsFloatPoint;
    public static readonly bool IsNumeric;
    public static readonly bool IsSigned;
    public static readonly bool CanBeNumeric;
    public static readonly bool IsNullable;
    public static readonly int BitsLength;
    public static readonly T MinValue;
    public static readonly T MaxValue;

    [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 bitsLength = Marshal.SizeOf(underlyingType) * 8;
            GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
            GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
                out Type unsignedVersion);
            Type = type;
            IsNullable = isNullable;
            UnderlyingType = underlyingType;
            CanBeNumeric = canBeNumeric;
            IsNumeric = isNumeric;
            IsSigned = isSigned;
            IsFloatPoint = isFloatPoint;
            BitsLength = bitsLength;
            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;
        }
    }
```

```
91        }
92   }
```

1.10 ./Platform.Reflection/PropertyInfoExtensions.cs

```
1  using System.Reflection;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Reflection
7  {
8      public static class PropertyInfoExtensions
9      {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public static T GetStaticValue<T>(this PropertyInfo fieldInfo) =>
             ↪  (T)fieldInfo.GetValue(null);
12      }
13  }
```

1.11 ./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     public static class TypeBuilderExtensions
11     {
12         public static readonly MethodAttributes DefaultStaticMethodAttributes =
            ↪  MethodAttributes.Public | MethodAttributes.Static;
13         public static readonly MethodAttributes DefaultVirtualMethodAttributes =
            ↪  MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
            ↪  MethodAttributes.HideBySig;
14         public static readonly MethodImplAttributes DefaultMethodImplAttributes =
            ↪  MethodImplAttributes.IL | MethodImplAttributes.Managed |
            ↪  MethodImplAttributes.AggressiveInlining;
15
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 EmitVirtualMethod<TDelegate>(this TypeBuilder type, string
            ↪  methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
            ↪  DefaultVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
37     }
38  }
```

1.12 ./Platform.Reflection/TypeExtensions.cs

```
1  using System;
2  using System.Collections.Generic;
```

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

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

namespace Platform.Reflection
{
    public static class TypeExtensions
    {
        static public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
            BindingFlags.NonPublic | BindingFlags.Static;
        static public readonly string DefaultDelegateMethodName = "Invoke";

        static private readonly HashSet<Type> _canBeNumericTypes;
        static private readonly HashSet<Type> _isNumericTypes;
        static private readonly HashSet<Type> _isSignedTypes;
        static private readonly HashSet<Type> _isFloatPointTypes;
        static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
        static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static TypeExtensions()
        {
            _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
                typeof(DateTime), typeof(TimeSpan) };
            _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
                typeof(ulong) };
            _canBeNumericTypes.UnionWith(_isNumericTypes);
            _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
                typeof(long) };
            _canBeNumericTypes.UnionWith(_isSignedTypes);
            _isNumericTypes.UnionWith(_isSignedTypes);
            _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
                typeof(float) };
            _canBeNumericTypes.UnionWith(_isFloatPointTypes);
            _isNumericTypes.UnionWith(_isFloatPointTypes);
            _isSignedTypes.UnionWith(_isFloatPointTypes);
            _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
            {
                { typeof(sbyte), typeof(byte) },
                { typeof(short), typeof(ushort) },
                { typeof(int), typeof(uint) },
                { typeof(long), typeof(ulong) },
            };
            _signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
            {
                { typeof(byte), typeof(sbyte)},
                { typeof(ushort), typeof(short) },
                { typeof(uint), typeof(int) },
                { typeof(ulong), typeof(long) },
            };
        }

        [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();
```

```
74            return method;
75        }
76
77        [MethodImpl(MethodImplOptions.AggressiveInlining)]
78        public static Type GetBaseType(this Type type) => type.BaseType;
79
80        [MethodImpl(MethodImplOptions.AggressiveInlining)]
81        public static Assembly GetAssembly(this Type type) => type.Assembly;
82
83        [MethodImpl(MethodImplOptions.AggressiveInlining)]
84        public static bool IsSubclassOf(this Type type, Type superClass) =>
      ↪    type.IsSubclassOf(superClass);
85
86        [MethodImpl(MethodImplOptions.AggressiveInlining)]
87        public static bool IsValueType(this Type type) => type.IsValueType;
88
89        [MethodImpl(MethodImplOptions.AggressiveInlining)]
90        public static bool IsGeneric(this Type type) => type.IsGenericType;
91
92        [MethodImpl(MethodImplOptions.AggressiveInlining)]
93        public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
      ↪    type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
94
95        [MethodImpl(MethodImplOptions.AggressiveInlining)]
96        public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
97
98        [MethodImpl(MethodImplOptions.AggressiveInlining)]
99        public static Type GetUnsignedVersionOrNull(this Type signedType) =>
      ↪    _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
100
101        [MethodImpl(MethodImplOptions.AggressiveInlining)]
102        public static Type GetSignedVersionOrNull(this Type unsignedType) =>
      ↪    _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
103
104        [MethodImpl(MethodImplOptions.AggressiveInlining)]
105        public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
106
107        [MethodImpl(MethodImplOptions.AggressiveInlining)]
108        public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
109
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

```
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     {
```

```csharp
             public static ReadOnlyCollection<Type> Collection { get; } = new
             ↪  ReadOnlyCollection<Type>(System.Array.Empty<Type>());
             public static Type[] Array => Collection.ToArray();

             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             protected ReadOnlyCollection<Type> ToReadOnlyCollection()
             {
                 var types = GetType().GetGenericArguments();
                 var result = new List<Type>();
                 AppendTypes(result, types);
                 return new ReadOnlyCollection<Type>(result);
             }

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

1.14 ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].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, T3, T4, T5, T6, T7> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
        ↪  T4, T5, T6, T7>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

1.15 ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].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, T3, T4, T5, T6> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
        ↪  T4, T5, T6>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

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

```csharp
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      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

```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      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

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

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

1.20 ./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        public class Types<T> : Types
11        {
12            public new static ReadOnlyCollection<Type> Collection { get; } = new
              ↪   Types<T>().ToReadOnlyCollection();
13            public new static Type[] Array => Collection.ToArray();
14            private Types() { }
15        }
16    }
```

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

```
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
18           [Fact]
19           public void FailedActionCompilationTest()
20           {
21               var compiledAction = DelegateHelpers.Compile<Action>(generator =>
22               {
23                   throw new NotImplementedException();
24               });
25               Assert.Throws<NotSupportedException>(compiledAction);
26           }
27
28           [Fact]
29           public void ConstantLoadingTest()
30           {
31               CheckConstantLoading<byte>(8);
32               CheckConstantLoading<uint>(8);
33               CheckConstantLoading<ushort>(8);
34               CheckConstantLoading<ulong>(8);
35           }
36
37           private void CheckConstantLoading<T>(T value)
38           {
39               var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
40               {
41                   generator.LoadConstant(value);
42                   generator.Return();
43               });
44               Assert.Equal(value, compiledFunction());
45           }
46
47           [Fact]
48           public void ConversionWithSignExtensionTest()
49           {
50               object[] withSignExtension = new object[]
51               {
52                   CompileUncheckedConverter<byte, sbyte>(extendSign: true)(128),
53                   CompileUncheckedConverter<byte, short>(extendSign: true)(128),
54                   CompileUncheckedConverter<ushort, short>(extendSign: true)(32768),
55                   CompileUncheckedConverter<byte, int>(extendSign: true)(128),
56                   CompileUncheckedConverter<ushort, int>(extendSign: true)(32768),
57                   CompileUncheckedConverter<uint, int>(extendSign: true)(2147483648),
58                   CompileUncheckedConverter<byte, long>(extendSign: true)(128),
59                   CompileUncheckedConverter<ushort, long>(extendSign: true)(32768),
60                   CompileUnReckedConverter<uint, long>(extendSign: true)(2147483648),
61                   CompileUncheckedConverter<ulong, long>(extendSign: true)(9223372036854775808)
62               };
63               object[] withoutSignExtension = new object[]
64               {
65                   CompileUncheckedConverter<byte, sbyte>(extendSign: false)(128),
66                   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++]);
                }

                private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
                    TTarget>(bool extendSign)
                {
                    return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
                    {
                        generator.LoadArgument(0);
                        generator.UncheckedConvert<TSource, TTarget>(extendSign);
                        generator.Return();
                    });
                }
            }
        }
    }
}
```

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

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

namespace Platform.Reflection.Tests
{
    public static class GetILBytesMethodTests
    {
        [Fact]
        public static void ILBytesForDelegateAreAvailableTest()
        {
            var function = new Func<object, int>(argument => 0);
            var bytes = function.GetMethodInfo().GetILBytes();
            Assert.False(bytes.IsNullOrEmpty());
        }

        [Fact]
        public static void ILBytesForDifferentDelegatesAreTheSameTest()
        {
            var firstFunction = new Func<object, int>(argument => 0);
            var secondFunction = new Func<object, int>(argument => 0);
            Assert.False(firstFunction == secondFunction);
            var firstFunctionBytes = firstFunction.GetMethodInfo().GetILBytes();
            Assert.False(firstFunctionBytes.IsNullOrEmpty());
            var secondFunctionBytes = secondFunction.GetMethodInfo().GetILBytes();
            Assert.False(secondFunctionBytes.IsNullOrEmpty());
            Assert.True(firstFunctionBytes.EqualTo(secondFunctionBytes));
        }
    }
}
```

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

```
using Xunit;

namespace Platform.Reflection.Tests
{
    public class NumericTypeTests
    {
        [Fact]
        public void UInt64IsNumericTest()
        {
```

```csharp
            Assert.True(NumericType<ulong>.IsNumeric);
        }
    }
}
```

Index