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          }

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

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          }

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          }

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

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

```
32                     generator.Emit(OpCodes.Conv_I1);
33                 }
34                 if (sourceType == typeof(ushort))
35                 {
36                     generator.Emit(OpCodes.Conv_I2);
37                 }
38             }
39             if (NumericType<TSource>.BitsSize > NumericType<TTarget>.BitsSize)
40             {
41                 generator.ConvertToInteger(targetType);
42             }
43             else
44             {
45 #if NETFRAMEWORK
46                 if (sourceType == typeof(byte) || sourceType == typeof(ushort))
47                 {
48                     if (targetType == typeof(long))
49                     {
50                         if (extendSign)
51                         {
52                             generator.Emit(OpCodes.Conv_I8);
53                         }
54                         else
55                         {
56                             generator.Emit(OpCodes.Conv_U8);
57                         }
58                     }
59                 }
60                 if (sourceType == typeof(uint) && targetType == typeof(long) && extendSign)
61                 {
62                     generator.Emit(OpCodes.Conv_I8);
63                 }
64 #endif
65                 if (sourceType == typeof(uint) && targetType == typeof(long) && !extendSign)
66                 {
67                     generator.Emit(OpCodes.Conv_U8);
68                 }
69             }
70             if (targetType == typeof(float))
71             {
72                 if (NumericType<TSource>.IsSigned)
73                 {
74                     generator.Emit(OpCodes.Conv_R4);
75                 }
76                 else
77                 {
78                     generator.Emit(OpCodes.Conv_R_Un);
79                 }
80             }
81             else if (targetType == typeof(double))
82             {
83                 generator.Emit(OpCodes.Conv_R8);
84             }
85             else if (targetType == typeof(bool))
86             {
87                 generator.ConvertToBoolean<TSource>();
88             }
89         }
90
91         [MethodImpl(MethodImplOptions.AggressiveInlining)]
92         private static void ConvertToBoolean<TSource>(this ILGenerator generator)
93         {
94             generator.LoadConstant<TSource>(default);
95             var sourceType = typeof(TSource);
96             if (sourceType == typeof(float) || sourceType == typeof(double))
97             {
98                 generator.Emit(OpCodes.Ceq);
99                 // Inversion of the first Ceq instruction
100                generator.LoadConstant<int>(0);
101                generator.Emit(OpCodes.Ceq);
102            }
103            else
104            {
105                generator.Emit(OpCodes.Cgt_Un);
106            }
107        }
108
109        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        private static void ConvertToInteger(this ILGenerator generator, Type targetType)
        {
            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);
            }
            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);
            }
        }

        [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);
                }
            }
```

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

```csharp
        }

        [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:
                    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);
```

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

```csharp
                    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))
            {
                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
```

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

```csharp
                {
                    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);

            [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)]
```

```csharp
        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);

        [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()) + "]");
            }
```

```
696                }
697                generator.NewObject(constructor);
698            }
699
700            [MethodImpl(MethodImplOptions.AggressiveInlining)]
701            public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
                 ↪   generator.Emit(OpCodes.Newobj, constructor);
702
703            [MethodImpl(MethodImplOptions.AggressiveInlining)]
704            public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
                 ↪   byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
705
706            [MethodImpl(MethodImplOptions.AggressiveInlining)]
707            public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
                 ↪   false, byte? unaligned = null)
708            {
709                if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
710                {
711                    throw new ArgumentException("unaligned must be null, 1, 2, or 4");
712                }
713                if (isVolatile)
714                {
715                    generator.Emit(OpCodes.Volatile);
716                }
717                if (unaligned.HasValue)
718                {
719                    generator.Emit(OpCodes.Unaligned, unaligned.Value);
720                }
721                if (type.IsPointer)
722                {
723                    generator.Emit(OpCodes.Ldind_I);
724                }
725                else if (!type.IsValueType)
726                {
727                    generator.Emit(OpCodes.Ldind_Ref);
728                }
729                else if (type == typeof(sbyte))
730                {
731                    generator.Emit(OpCodes.Ldind_I1);
732                }
733                else if (type == typeof(bool))
734                {
735                    generator.Emit(OpCodes.Ldind_I1);
736                }
737                else if (type == typeof(byte))
738                {
739                    generator.Emit(OpCodes.Ldind_U1);
740                }
741                else if (type == typeof(short))
742                {
743                    generator.Emit(OpCodes.Ldind_I2);
744                }
745                else if (type == typeof(ushort))
746                {
747                    generator.Emit(OpCodes.Ldind_U2);
748                }
749                else if (type == typeof(char))
750                {
751                    generator.Emit(OpCodes.Ldind_U2);
752                }
753                else if (type == typeof(int))
754                {
755                    generator.Emit(OpCodes.Ldind_I4);
756                }
757                else if (type == typeof(uint))
758                {
759                    generator.Emit(OpCodes.Ldind_U4);
760                }
761                else if (type == typeof(long) || type == typeof(ulong))
762                {
763                    generator.Emit(OpCodes.Ldind_I8);
764                }
765                else if (type == typeof(float))
766                {
767                    generator.Emit(OpCodes.Ldind_R4);
768                }
769                else if (type == typeof(double))
770                {
```

```
771                    generator.Emit(OpCodes.Ldind_R8);
772                }
773                else
774                {
775                    throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
                        ↪ ", LoadObject may be more appropriate");
776                }
777            }
778
779            [MethodImpl(MethodImplOptions.AggressiveInlining)]
780            public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
                ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
781
782            [MethodImpl(MethodImplOptions.AggressiveInlining)]
783            public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
                ↪ = false, byte? unaligned = null)
784            {
785                if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
786                {
787                    throw new ArgumentException("unaligned must be null, 1, 2, or 4");
788                }
789                if (isVolatile)
790                {
791                    generator.Emit(OpCodes.Volatile);
792                }
793                if (unaligned.HasValue)
794                {
795                    generator.Emit(OpCodes.Unaligned, unaligned.Value);
796                }
797                if (type.IsPointer)
798                {
799                    generator.Emit(OpCodes.Stind_I);
800                }
801                else if (!type.IsValueType)
802                {
803                    generator.Emit(OpCodes.Stind_Ref);
804                }
805                else if (type == typeof(sbyte) || type == typeof(byte))
806                {
807                    generator.Emit(OpCodes.Stind_I1);
808                }
809                else if (type == typeof(short) || type == typeof(ushort))
810                {
811                    generator.Emit(OpCodes.Stind_I2);
812                }
813                else if (type == typeof(int) || type == typeof(uint))
814                {
815                    generator.Emit(OpCodes.Stind_I4);
816                }
817                else if (type == typeof(long) || type == typeof(ulong))
818                {
819                    generator.Emit(OpCodes.Stind_I8);
820                }
821                else if (type == typeof(float))
822                {
823                    generator.Emit(OpCodes.Stind_R4);
824                }
825                else if (type == typeof(double))
826                {
827                    generator.Emit(OpCodes.Stind_R8);
828                }
829                else
830                {
831                    throw new InvalidOperationException("StoreIndirect cannot be used with " + type
                        ↪ + ", StoreObject may be more appropriate");
832                }
833            }
834        }
835    }
```

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
```

```csharp
namespace Platform.Reflection
{
    public static class MethodInfoExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static byte[] GetILBytes(this MethodInfo methodInfo) =>
            methodInfo.GetMethodBody().GetILAsByteArray();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
            methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
    }
}
```

1.8   ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs

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

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

namespace Platform.Reflection
{
    public class NotSupportedExceptionDelegateFactory<TDelegate> : IFactory<TDelegate>
        where TDelegate : Delegate
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TDelegate Create()
        {
            var @delegate = DelegateHelpers.CompileOrDefault<TDelegate>(generator =>
            {
                generator.Throw<NotSupportedException>();
            });
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
                throw new InvalidOperationException("Unable to compile stub delegate.");
            }
            return @delegate;
        }
    }
}
```

1.9   ./Platform.Reflection/NumericType.cs

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

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

namespace Platform.Reflection
{
    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 BytesSize;
        public static readonly int BitsSize;
        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;
```

```
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;
56                    SignedVersion = signedVersion;
57                    UnsignedVersion = unsignedVersion;
58                }
59                catch (Exception exception)
60                {
61                    exception.Ignore();
62                }
63            }
64
65            [MethodImpl(MethodImplOptions.AggressiveInlining)]
66            private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
67            {
68                if (type == typeof(bool))
69                {
70                    minValue = (T)(object)false;
71                    maxValue = (T)(object)true;
72                }
73                else
74                {
75                    minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
76                    maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
77                }
78            }
79
80            [MethodImpl(MethodImplOptions.AggressiveInlining)]
81            private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
   ↪    signedVersion, out Type unsignedVersion)
82            {
83                if (isSigned)
84                {
85                    signedVersion = type;
86                    unsignedVersion = type.GetUnsignedVersionOrNull();
87                }
88                else
89                {
90                    signedVersion = type.GetSignedVersionOrNull();
91                    unsignedVersion = type;
92                }
93            }
94        }
95    }
```

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  }
```

```csharp
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 DefaultFinalVirtualMethodAttributes =
           ↪   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 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;
```

```csharp
        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();
            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;
```

```csharp
        [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);

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetDelegateReturnType(this Type delegateType) =>
        ↪   delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
        ↪   delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetDelegateCharacteristics(this Type delegateType, out Type
        ↪   returnType, out Type[] parameterTypes)
        {
            var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
            returnType = invoke.ReturnType;
            parameterTypes = invoke.GetParameterTypes();
        }
    }
}
```

1.13 ./Platform.Reflection/Types.cs

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

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

namespace Platform.Reflection
{
    public abstract class Types
    {
        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];
```

```
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

```
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, 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;
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, 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;
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

```
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

```
 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

```
 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

```
 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
```

```csharp
    {
        [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);
        }

        [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 ConversionWithSignExtensionTest()
        {
            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++]);
```

```
85            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
86            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
87        }
88
89        private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
   ↪      TTarget>(bool extendSign)
90        {
91            return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
92            {
93                generator.LoadArgument(0);
94                generator.UncheckedConvert<TSource, TTarget>(extendSign);
95                generator.Return();
96            });
97        }
98    }
99 }
```

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