1.1 ./Platform.Reflection/AssemblyExtensions.cs

```
using System;
using System.Collections.Concurrent;
using System.Reflection;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Collections.Lists;

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

namespace Platform.Reflection
{
    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

```
using System;
using System.Collections.Generic;
using System.Linq;
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
        ↪  aggressiveInlining)
            where TDelegate : Delegate
        {
            var @delegate = default(TDelegate);
            try
            {
                @delegate = aggressiveInlining ? CompileUsingMethodBuilder<TDelegate>(emitCode)
                ↪  : CompileUsingDynamicMethod<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)]
```

```
35          public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
            ↪  aggressiveInlining)
36              where TDelegate : Delegate
37          {
38              var @delegate = CompileOrDefault<TDelegate>(emitCode, aggressiveInlining);
39              if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
40              {
41                  @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
42              }
43              return @delegate;
44          }
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
            ↪  : Delegate => Compile<TDelegate>(emitCode, false);
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          private static TDelegate CompileUsingDynamicMethod<TDelegate>(Action<ILGenerator>
            ↪  emitCode)
51          {
52              var delegateType = typeof(TDelegate);
53              var invoke = delegateType.GetMethod("Invoke");
54              var returnType = invoke.ReturnType;
55              var parameterTypes = invoke.GetParameters().Select(s => s.ParameterType).ToArray();
56              var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
57              var generator = dynamicMethod.GetILGenerator();
58              emitCode(generator);
59              return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
60          }
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          private static TDelegate CompileUsingMethodBuilder<TDelegate>(Action<ILGenerator>
            ↪  emitCode)
64          {
65              AssemblyName assemblyName = new AssemblyName(GetNewName());
66              var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
                ↪  AssemblyBuilderAccess.Run);
67              var module = assembly.DefineDynamicModule(GetNewName());
68              var type = module.DefineType(GetNewName());
69              var delegateType = typeof(TDelegate);
70              var invoke = delegateType.GetMethod("Invoke");
71              var returnType = invoke.ReturnType;
72              var parameterTypes = invoke.GetParameters().Select(s => s.ParameterType).ToArray();
73              var methodName = GetNewName();
74              MethodBuilder method = type.DefineMethod(methodName, MethodAttributes.Public |
                ↪  MethodAttributes.Static, returnType, parameterTypes);
75              method.SetImplementationFlags(MethodImplAttributes.IL | MethodImplAttributes.Managed
                ↪  | MethodImplAttributes.AggressiveInlining);
76              var generator = method.GetILGenerator();
77              emitCode(generator);
78              var typeInfo = type.CreateTypeInfo();
79              return
                ↪  (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(delegateType);
80          }
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          private static string GetNewName() => Guid.NewGuid().ToString("N");
84      }
85  }
```

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              }
```

```
            return type.GetProperty(propertyName) != null;
        }
    }
}
```

1.4 ./Platform.Reflection/EnsureExtensions.cs

```
using System;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Exceptions.ExtensionRoots;

#pragma warning disable IDE0060 // Remove unused parameter
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    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)]
```

```
64          public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
65          {
66              string messageBuilder() => message;
67              IsSigned<T>(root, messageBuilder);
68          }
69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
            ↪  (string)null);
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
            ↪  messageBuilder)
75          {
76              if (!NumericType<T>.IsNumeric)
77              {
78                  throw new NotSupportedException(messageBuilder());
79              }
80          }
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
84          {
85              string messageBuilder() => message;
86              IsNumeric<T>(root, messageBuilder);
87          }
88
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪  IsNumeric<T>(root, (string)null);
91
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
            ↪  messageBuilder)
94          {
95              if (!NumericType<T>.CanBeNumeric)
96              {
97                  throw new NotSupportedException(messageBuilder());
98              }
99          }
100
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
103         {
104             string messageBuilder() => message;
105             CanBeNumeric<T>(root, messageBuilder);
106         }
107
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪  CanBeNumeric<T>(root, (string)null);
110
111         #endregion
112
113         #region OnDebug
114
115         [Conditional("DEBUG")]
116         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
            ↪  Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
117
118         [Conditional("DEBUG")]
119         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
            ↪  message) => Ensure.Always.IsUnsignedInteger<T>(message);
120
121         [Conditional("DEBUG")]
122         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsUnsignedInteger<T>();
123
124         [Conditional("DEBUG")]
125         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
126
127         [Conditional("DEBUG")]
128         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
            ↪  message) => Ensure.Always.IsSignedInteger<T>(message);
129
130         [Conditional("DEBUG")]
```

```
131         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪ Ensure.Always.IsSignedInteger<T>();
132
133         [Conditional("DEBUG")]
134         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪ messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
135
136         [Conditional("DEBUG")]
137         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
            ↪ Ensure.Always.IsSigned<T>(message);
138
139         [Conditional("DEBUG")]
140         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪ Ensure.Always.IsSigned<T>();
141
142         [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
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator)
```

```
        {
            var type = typeof(TTarget);
            if (type == typeof(short))
            {
                generator.Emit(OpCodes.Conv_I2);
            }
            else if (type == typeof(ushort))
            {
                generator.Emit(OpCodes.Conv_U2);
            }
            else if (type == typeof(sbyte))
            {
                generator.Emit(OpCodes.Conv_I1);
            }
            else if (type == typeof(byte))
            {
                generator.Emit(OpCodes.Conv_U1);
            }
            else if (type == typeof(int))
            {
                generator.Emit(OpCodes.Conv_I4);
            }
            else if (type == typeof(uint))
            {
                generator.Emit(OpCodes.Conv_U4);
            }
            else if (type == typeof(long))
            {
                generator.Emit(OpCodes.Conv_I8);
            }
            else if (type == typeof(ulong))
            {
                generator.Emit(OpCodes.Conv_U8);
            }
            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 CheckedConvert<TSource, TTarget>(this ILGenerator generator)
        {
            var type = typeof(TTarget);
            if (type == typeof(short))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2);
                }
                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);
            }
            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:
                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;
        }
    }

```

```
249         [MethodImpl(MethodImplOptions.AggressiveInlining)]
250         public static void LoadConstant(this ILGenerator generator, int value)
251         {
252             switch (value)
253             {
254                 case -1:
255                     generator.Emit(OpCodes.Ldc_I4_M1);
256                     return;
257                 case 0:
258                     generator.Emit(OpCodes.Ldc_I4_0);
259                     return;
260                 case 1:
261                     generator.Emit(OpCodes.Ldc_I4_1);
262                     return;
263                 case 2:
264                     generator.Emit(OpCodes.Ldc_I4_2);
265                     return;
266                 case 3:
267                     generator.Emit(OpCodes.Ldc_I4_3);
268                     return;
269                 case 4:
270                     generator.Emit(OpCodes.Ldc_I4_4);
271                     return;
272                 case 5:
273                     generator.Emit(OpCodes.Ldc_I4_5);
274                     return;
275                 case 6:
276                     generator.Emit(OpCodes.Ldc_I4_6);
277                     return;
278                 case 7:
279                     generator.Emit(OpCodes.Ldc_I4_7);
280                     return;
281                 case 8:
282                     generator.Emit(OpCodes.Ldc_I4_8);
283                     return;
284                 default:
285                     if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
286                     {
287                         generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
288                     }
289                     else
290                     {
291                         generator.Emit(OpCodes.Ldc_I4, value);
292                     }
293                     return;
294             }
295         }
296 
297         [MethodImpl(MethodImplOptions.AggressiveInlining)]
298         public static void LoadConstant(this ILGenerator generator, short value)
299         {
300             generator.LoadConstant((int)value);
301         }
302 
303         [MethodImpl(MethodImplOptions.AggressiveInlining)]
304         public static void LoadConstant(this ILGenerator generator, ushort value)
305         {
306             generator.LoadConstant((int)value);
307         }
308 
309         [MethodImpl(MethodImplOptions.AggressiveInlining)]
310         public static void LoadConstant(this ILGenerator generator, sbyte value)
311         {
312             generator.LoadConstant((int)value);
313         }
314 
315         [MethodImpl(MethodImplOptions.AggressiveInlining)]
316         public static void LoadConstant(this ILGenerator generator, byte value)
317         {
318             generator.LoadConstant((int)value);
319         }
320 
321         [MethodImpl(MethodImplOptions.AggressiveInlining)]
322         public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
            ↪ LoadConstantOne(generator, typeof(TConstant));
323 
324         [MethodImpl(MethodImplOptions.AggressiveInlining)]
325         public static void LoadConstantOne(this ILGenerator generator, Type constantType)
326         {
327             if (constantType == typeof(float))
```

```
328             {
329                 generator.LoadConstant(1F);
330             }
331             else if (constantType == typeof(double))
332             {
333                 generator.LoadConstant(1D);
334             }
335             else if (constantType == typeof(long))
336             {
337                 generator.LoadConstant(1L);
338             }
339             else if (constantType == typeof(ulong))
340             {
341                 generator.LoadConstant(1UL);
342             }
343             else if (constantType == typeof(int))
344             {
345                 generator.LoadConstant(1);
346             }
347             else if (constantType == typeof(uint))
348             {
349                 generator.LoadConstant(1U);
350             }
351             else if (constantType == typeof(short))
352             {
353                 generator.LoadConstant((short)1);
354             }
355             else if (constantType == typeof(ushort))
356             {
357                 generator.LoadConstant((ushort)1);
358             }
359             else if (constantType == typeof(sbyte))
360             {
361                 generator.LoadConstant((sbyte)1);
362             }
363             else if (constantType == typeof(byte))
364             {
365                 generator.LoadConstant((byte)1);
366             }
367             else
368             {
369                 throw new NotSupportedException();
370             }
371         }
372 
373         [MethodImpl(MethodImplOptions.AggressiveInlining)]
374         public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
            ↪  constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
375 
376         [MethodImpl(MethodImplOptions.AggressiveInlining)]
377         public static void LoadConstant(this ILGenerator generator, Type constantType, object
            ↪  constantValue)
378         {
379             constantValue = Convert.ChangeType(constantValue, constantType);
380             if (constantType == typeof(float))
381             {
382                 generator.LoadConstant((float)constantValue);
383             }
384             else if (constantType == typeof(double))
385             {
386                 generator.LoadConstant((double)constantValue);
387             }
388             else if (constantType == typeof(long))
389             {
390                 generator.LoadConstant((long)constantValue);
391             }
392             else if (constantType == typeof(ulong))
393             {
394                 generator.LoadConstant((ulong)constantValue);
395             }
396             else if (constantType == typeof(int))
397             {
398                 generator.LoadConstant((int)constantValue);
399             }
400             else if (constantType == typeof(uint))
401             {
402                 generator.LoadConstant((uint)constantValue);
403             }
```

```
            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)
            {
                case 0:
                    generator.Emit(OpCodes.Ldarg_0);
                    break;
                case 1:
                    generator.Emit(OpCodes.Ldarg_1);
                    break;
```

```
481                 case 2:
482                     generator.Emit(OpCodes.Ldarg_2);
483                     break;
484                 case 3:
485                     generator.Emit(OpCodes.Ldarg_3);
486                     break;
487                 default:
488                     generator.Emit(OpCodes.Ldarg, argumentIndex);
489                     break;
490             }
491         }
492 
493         [MethodImpl(MethodImplOptions.AggressiveInlining)]
494         public static void LoadArguments(this ILGenerator generator, params int[]
            ↪ argumentIndices)
495         {
496             for (var i = 0; i < argumentIndices.Length; i++)
497             {
498                 generator.LoadArgument(argumentIndices[i]);
499             }
500         }
501 
502         [MethodImpl(MethodImplOptions.AggressiveInlining)]
503         public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
            ↪ generator.Emit(OpCodes.Starg, argumentIndex);
504 
505         [MethodImpl(MethodImplOptions.AggressiveInlining)]
506         public static void CompareGreaterThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Cgt);
507 
508         [MethodImpl(MethodImplOptions.AggressiveInlining)]
509         public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Cgt_Un);
510 
511         [MethodImpl(MethodImplOptions.AggressiveInlining)]
512         public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
513         {
514             if (isSigned)
515             {
516                 generator.CompareGreaterThan();
517             }
518             else
519             {
520                 generator.UnsignedCompareGreaterThan();
521             }
522         }
523 
524         [MethodImpl(MethodImplOptions.AggressiveInlining)]
525         public static void CompareLessThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Clt);
526 
527         [MethodImpl(MethodImplOptions.AggressiveInlining)]
528         public static void UnsignedCompareLessThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Clt_Un);
529 
530         [MethodImpl(MethodImplOptions.AggressiveInlining)]
531         public static void CompareLessThan(this ILGenerator generator, bool isSigned)
532         {
533             if (isSigned)
534             {
535                 generator.CompareLessThan();
536             }
537             else
538             {
539                 generator.UnsignedCompareLessThan();
540             }
541         }
542 
543         [MethodImpl(MethodImplOptions.AggressiveInlining)]
544         public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
            ↪ generator.Emit(OpCodes.Bge, label);
545 
546         [MethodImpl(MethodImplOptions.AggressiveInlining)]
547         public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
            ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
548 
549         [MethodImpl(MethodImplOptions.AggressiveInlining)]
550         public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
            ↪ Label label)
```

```
551         {
552             if (isSigned)
553             {
554                 generator.BranchIfGreaterOrEqual(label);
555             }
556             else
557             {
558                 generator.UnsignedBranchIfGreaterOrEqual(label);
559             }
560         }
561 
562         [MethodImpl(MethodImplOptions.AggressiveInlining)]
563         public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
            ↪ generator.Emit(OpCodes.Ble, label);
564 
565         [MethodImpl(MethodImplOptions.AggressiveInlining)]
566         public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
            ↪ => generator.Emit(OpCodes.Ble_Un, label);
567 
568         [MethodImpl(MethodImplOptions.AggressiveInlining)]
569         public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
            ↪ label)
570         {
571             if (isSigned)
572             {
573                 generator.BranchIfLessOrEqual(label);
574             }
575             else
576             {
577                 generator.UnsignedBranchIfLessOrEqual(label);
578             }
579         }
580 
581         [MethodImpl(MethodImplOptions.AggressiveInlining)]
582         public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
583 
584         [MethodImpl(MethodImplOptions.AggressiveInlining)]
585         public static void Box(this ILGenerator generator, Type boxedType) =>
            ↪ generator.Emit(OpCodes.Box, boxedType);
586 
587         [MethodImpl(MethodImplOptions.AggressiveInlining)]
588         public static void Call(this ILGenerator generator, MethodInfo method) =>
            ↪ generator.Emit(OpCodes.Call, method);
589 
590         [MethodImpl(MethodImplOptions.AggressiveInlining)]
591         public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
592 
593         [MethodImpl(MethodImplOptions.AggressiveInlining)]
594         public static void Unbox<TUnbox>(this ILGenerator generator) =>
            ↪ generator.Unbox(typeof(TUnbox));
595 
596         [MethodImpl(MethodImplOptions.AggressiveInlining)]
597         public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
            ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
598 
599         [MethodImpl(MethodImplOptions.AggressiveInlining)]
600         public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
            ↪ generator.UnboxValue(typeof(TUnbox));
601 
602         [MethodImpl(MethodImplOptions.AggressiveInlining)]
603         public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
            ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
604 
605         [MethodImpl(MethodImplOptions.AggressiveInlining)]
606         public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
            ↪ generator.DeclareLocal(typeof(T));
607 
608         [MethodImpl(MethodImplOptions.AggressiveInlining)]
609         public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪ generator.Emit(OpCodes.Ldloc, local);
610 
611         [MethodImpl(MethodImplOptions.AggressiveInlining)]
612         public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪ generator.Emit(OpCodes.Stloc, local);
613 
614         [MethodImpl(MethodImplOptions.AggressiveInlining)]
615         public static void NewObject(this ILGenerator generator, Type type, params Type[]
            ↪ parameterTypes)
```

```
616         {
617             var allConstructors = type.GetConstructors(BindingFlags.Public |
                ↪ BindingFlags.NonPublic | BindingFlags.Instance
618 #if !NETSTANDARD
619                 | BindingFlags.CreateInstance
620 #endif
621                 );
622             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();
623             if (constructor == null)
624             {
625                 throw new InvalidOperationException("Type " + type + " must have a constructor
                    ↪ that matches parameters [" + string.Join(", ",
                    ↪ parameterTypes.AsEnumerable()) + "]");
626             }
627             generator.NewObject(constructor);
628         }
629
630         [MethodImpl(MethodImplOptions.AggressiveInlining)]
631         public static void NewObject(this ILGenerator generator, ConstructorInfo constructor)
632         {
633             generator.Emit(OpCodes.Newobj, constructor);
634         }
635
636         [MethodImpl(MethodImplOptions.AggressiveInlining)]
637         public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
638
639         [MethodImpl(MethodImplOptions.AggressiveInlining)]
640         public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
            ↪ false, byte? unaligned = null)
641         {
642             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
643             {
644                 throw new ArgumentException("unaligned must be null, 1, 2, or 4");
645             }
646             if (isVolatile)
647             {
648                 generator.Emit(OpCodes.Volatile);
649             }
650             if (unaligned.HasValue)
651             {
652                 generator.Emit(OpCodes.Unaligned, unaligned.Value);
653             }
654             if (type.IsPointer)
655             {
656                 generator.Emit(OpCodes.Ldind_I);
657             }
658             else if (!type.IsValueType)
659             {
660                 generator.Emit(OpCodes.Ldind_Ref);
661             }
662             else if (type == typeof(sbyte))
663             {
664                 generator.Emit(OpCodes.Ldind_I1);
665             }
666             else if (type == typeof(bool))
667             {
668                 generator.Emit(OpCodes.Ldind_I1);
669             }
670             else if (type == typeof(byte))
671             {
672                 generator.Emit(OpCodes.Ldind_U1);
673             }
674             else if (type == typeof(short))
675             {
676                 generator.Emit(OpCodes.Ldind_I2);
677             }
678             else if (type == typeof(ushort))
679             {
680                 generator.Emit(OpCodes.Ldind_U2);
681             }
682             else if (type == typeof(char))
683             {
684                 generator.Emit(OpCodes.Ldind_U2);
685             }
```

```
686             else if (type == typeof(int))
687             {
688                 generator.Emit(OpCodes.Ldind_I4);
689             }
690             else if (type == typeof(uint))
691             {
692                 generator.Emit(OpCodes.Ldind_U4);
693             }
694             else if (type == typeof(long) || type == typeof(ulong))
695             {
696                 generator.Emit(OpCodes.Ldind_I8);
697             }
698             else if (type == typeof(float))
699             {
700                 generator.Emit(OpCodes.Ldind_R4);
701             }
702             else if (type == typeof(double))
703             {
704                 generator.Emit(OpCodes.Ldind_R8);
705             }
706             else
707             {
708                 throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
                    ↪  ", LoadObject may be more appropriate");
709             }
710         }
711 
712         [MethodImpl(MethodImplOptions.AggressiveInlining)]
713         public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪  byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
714 
715         [MethodImpl(MethodImplOptions.AggressiveInlining)]
716         public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
            ↪  = false, byte? unaligned = null)
717         {
718             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
719             {
720                 throw new ArgumentException("unaligned must be null, 1, 2, or 4");
721             }
722             if (isVolatile)
723             {
724                 generator.Emit(OpCodes.Volatile);
725             }
726             if (unaligned.HasValue)
727             {
728                 generator.Emit(OpCodes.Unaligned, unaligned.Value);
729             }
730             if (type.IsPointer)
731             {
732                 generator.Emit(OpCodes.Stind_I);
733             }
734             else if (!type.IsValueType)
735             {
736                 generator.Emit(OpCodes.Stind_Ref);
737             }
738             else if (type == typeof(sbyte) || type == typeof(byte))
739             {
740                 generator.Emit(OpCodes.Stind_I1);
741             }
742             else if (type == typeof(short) || type == typeof(ushort))
743             {
744                 generator.Emit(OpCodes.Stind_I2);
745             }
746             else if (type == typeof(int) || type == typeof(uint))
747             {
748                 generator.Emit(OpCodes.Stind_I4);
749             }
750             else if (type == typeof(long) || type == typeof(ulong))
751             {
752                 generator.Emit(OpCodes.Stind_I8);
753             }
754             else if (type == typeof(float))
755             {
756                 generator.Emit(OpCodes.Stind_R4);
757             }
758             else if (type == typeof(double))
759             {
760                 generator.Emit(OpCodes.Stind_R8);
```

```
761             }
762             else
763             {
764                 throw new InvalidOperationException("StoreIndirect cannot be used with " + type
                    ↪  + ", StoreObject may be more appropriate");
765             }
766         }
767     }
768 }
```

1.7 ./Platform.Reflection/MethodInfoExtensions.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 MethodInfoExtensions
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static byte[] GetILBytes(this MethodInfo methodInfo) =>
            ↪  methodInfo.GetMethodBody().GetILAsByteArray();
12     }
13 }
```

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 {
13     public static class NumericType<T>
14     {
15         public static readonly Type Type;
16         public static readonly Type UnderlyingType;
17         public static readonly Type SignedVersion;
18         public static readonly Type UnsignedVersion;
19         public static readonly bool IsFloatPoint;
20         public static readonly bool IsNumeric;
21         public static readonly bool IsSigned;
22         public static readonly bool CanBeNumeric;
23         public static readonly bool IsNullable;
24         public static readonly int 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;
            }
        }
    }
}
```

1.10 ./Platform.Reflection/PropertyInfoExtensions.cs

```
using System.Reflection;
using System.Runtime.CompilerServices;

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

namespace Platform.Reflection
{
    public static class PropertyInfoExtensions
```

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

1.11 ./Platform.Reflection/TypeExtensions.cs

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

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

namespace Platform.Reflection
{
    public static class TypeExtensions
    {
        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.GetTypeInfo().GetField(name, BindingFlags.Public | BindingFlags.NonPublic |
        ↪  BindingFlags.Static).GetStaticValue<T>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticPropertyValue<T>(this Type type, string name) =>
        ↪  type.GetTypeInfo().GetProperty(name, BindingFlags.Public | BindingFlags.NonPublic |
        ↪  BindingFlags.Static).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.GetTypeInfo().BaseType;

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

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

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

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

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

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

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

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

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

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

        [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);
    }
}
```

1.12 ./Platform.Reflection/Types.cs

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

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

namespace Platform.Reflection
{
    public abstract class Types
    {
        public static ReadOnlyCollection<Type> Collection { get; } = new
        ↪  ReadOnlyCollection<Type>(new Type[0]);
        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);
                    }
                }
            }
        }
    }
}
```

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

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

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

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() { }
    }
}
```

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

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

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

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() { }
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.19 ./Platform.Reflection/Types[T].cs

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

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

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

1.20 ./Platform.Reflection.Tests/CodeGenerationTests.cs

```
using System;
using System.Runtime.CompilerServices;
using Xunit;
using Xunit.Abstractions;
using Platform.Diagnostics;

```

```
namespace Platform.Reflection.Tests
{
    public class CodeGenerationTests
    {
        private readonly ITestOutputHelper _output;

        public CodeGenerationTests(ITestOutputHelper output) => _output = output;

        [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());
        }

        private class MethodsContainer
        {
            public static readonly Func<int> DelegateWithoutAggressiveInlining;
            public static readonly Func<int> DelegateWithAggressiveInlining;

            static MethodsContainer()
            {
                void emitCode(System.Reflection.Emit.ILGenerator generator)
                {
                    generator.LoadConstant(140314);
                    generator.Return();
                };
                DelegateWithoutAggressiveInlining = DelegateHelpers.Compile<Func<int>>(emitCode,
                ↪  aggressiveInlining: false);
                DelegateWithAggressiveInlining = DelegateHelpers.Compile<Func<int>>(emitCode,
                ↪  aggressiveInlining: true);
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static int WrapperForDelegateWithoutAggressiveInlining() =>
            ↪  DelegateWithoutAggressiveInlining();

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static int WrapperForDelegateWithAggressiveInlining() =>
            ↪  DelegateWithAggressiveInlining();
        }

        [Fact]
        public void AggressiveInliningEffectTest()
        {
            const int N = 10000000;

```

```
        int result = 0;

        // Warm up

        for (int i = 0; i < N; i++)
        {
            result = MethodsContainer.DelegateWithoutAggressiveInlining();
        }
        for (int i = 0; i < N; i++)
        {
            result = MethodsContainer.DelegateWithAggressiveInlining();
        }
        for (int i = 0; i < N; i++)
        {
            result = MethodsContainer.WrapperForDelegateWithoutAggressiveInlining();
        }
        for (int i = 0; i < N; i++)
        {
            result = MethodsContainer.WrapperForDelegateWithAggressiveInlining();
        }
        for (int i = 0; i < N; i++)
        {
            result = Function();
        }
        for (int i = 0; i < N; i++)
        {
            result = 140314;
        }

        // Measure
        var ts1 = Performance.Measure(() =>
        {
            for (int i = 0; i < N; i++)
            {
                result = MethodsContainer.DelegateWithoutAggressiveInlining();
            }
        });
        var ts2 = Performance.Measure(() =>
        {
            for (int i = 0; i < N; i++)
            {
                result = MethodsContainer.DelegateWithAggressiveInlining();
            }
        });
        var ts3 = Performance.Measure(() =>
        {
            for (int i = 0; i < N; i++)
            {
                result = MethodsContainer.WrapperForDelegateWithoutAggressiveInlining();
            }
        });
        var ts4 = Performance.Measure(() =>
        {
            for (int i = 0; i < N; i++)
            {
                result = MethodsContainer.WrapperForDelegateWithAggressiveInlining();
            }
        });
        var ts5 = Performance.Measure(() =>
        {
            for (int i = 0; i < N; i++)
            {
                result = Function();
            }
        });
        var ts6 = Performance.Measure(() =>
        {
            for (int i = 0; i < N; i++)
            {
                result = 140314;
            }
        });

        var output = $"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {result}";
        _output.WriteLine(output);

        Assert.True(ts5 < ts1);
        Assert.True(ts5 < ts2);
```

```
            Assert.True(ts5 < ts3);
            Assert.True(ts5 < ts4);
            Assert.True(ts6 < ts1);
            Assert.True(ts6 < ts2);
            Assert.True(ts6 < ts3);
            Assert.True(ts6 < ts4);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static int Function() => 140314;
    }
}
```

1.21 ./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));
        }
    }
}
```

1.22 ./Platform.Reflection.Tests/NumericTypeTests.cs

```
using Xunit;

namespace Platform.Reflection.Tests
{
    public class NumericTypeTests
    {
        [Fact]
        public void UInt64IsNumericTest()
        {
            Assert.True(NumericType<ulong>.IsNumeric);
        }
    }
}
```

Index

./Platform.Reflection.Tests/CodeGenerationTests.cs, 21
./Platform.Reflection.Tests/GetILBytesMethodTests.cs, 24
./Platform.Reflection.Tests/NumericTypeTests.cs, 24
./Platform.Reflection/AssemblyExtensions.cs, 1
./Platform.Reflection/DelegateHelpers.cs, 1
./Platform.Reflection/DynamicExtensions.cs, 2
./Platform.Reflection/EnsureExtensions.cs, 3
./Platform.Reflection/FieldInfoExtensions.cs, 5
./Platform.Reflection/ILGeneratorExtensions.cs, 5
./Platform.Reflection/MethodInfoExtensions.cs, 16
./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 16
./Platform.Reflection/NumericType.cs, 16
./Platform.Reflection/PropertyInfoExtensions.cs, 17
./Platform.Reflection/TypeExtensions.cs, 18
./Platform.Reflection/Types.cs, 19
./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 20
./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 20
./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 20
./Platform.Reflection/Types[T1, T2, T3, T4].cs, 20
./Platform.Reflection/Types[T1, T2, T3].cs, 21
./Platform.Reflection/Types[T1, T2].cs, 21
./Platform.Reflection/Types[T].cs, 21