

# LinksPlatform's Platform.Reflection Class Library

## 1.1 ./Platform.Reflection/AssemblyExtensions.cs

```
1 using System;
2 using System.Collections.Concurrent;
3 using System.Reflection;
4 using System.Runtime.CompilerServices;
5 using Platform.Exceptions;
6 using Platform.Collections.Lists;
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 AssemblyExtensions
13     {
14         private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
15             ↳ ConcurrentDictionary<Assembly, Type[]>();
16
17         /// <remarks>
18         /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
19         /// </remarks>
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public static Type[] GetLoadableTypes(this Assembly assembly)
22         {
23             Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
24             try
25             {
26                 return assembly.GetTypes();
27             }
28             catch (ReflectionTypeLoadException e)
29             {
30                 return e.Types.ToArray(t => t != null);
31             }
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
36             ↳ _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
37     }
38 }
```

## 1.2 ./Platform.Reflection/DelegateHelpers.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Reflection;
4 using System.Reflection.Emit;
5 using System.Runtime.CompilerServices;
6 using Platform.Exceptions;
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 DelegateHelpers
13     {
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode, bool
16             ↳ typeMemberMethod)
17             where TDelegate : Delegate
18         {
19             var @delegate = default(TDelegate);
20             try
21             {
22                 @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
23                     ↳ CompileDynamicMethod<TDelegate>(emitCode);
24             }
25             catch (Exception exception)
26             {
27                 exception.Ignore();
28             }
29             return @delegate;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
34             ↳ TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
38             ↳ typeMemberMethod)
39         {
40             var @delegate = default(TDelegate);
41             try
42             {
43                 @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
44                     ↳ CompileDynamicMethod<TDelegate>(emitCode);
45             }
46             catch (Exception exception)
47             {
48                 exception.Ignore();
49             }
50             return @delegate;
51         }
52     }
53 }
```

```

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

```

### 1.3 ./Platform.Reflection/DynamicExtensions.cs

```

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

```

### 1.4 ./Platform.Reflection/EnsureExtensions.cs

```

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

```

```

10 namespace Platform.Reflection
11 {
12     public static class EnsureExtensions
13     {
14         #region Always
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root,
18         ↪ Func<string> messageBuilder)
19         {
20             if (!NumericType<T>.IsNumeric || NumericType<T>.IsSigned ||
21             ↪ NumericType<T>.IsFloatPoint)
22             {
23                 throw new NotSupportedException(messageBuilder());
24             }
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
29         ↪ message)
30         {
31             string messageBuilder() => message;
32             IsUnsignedInteger<T>(root, messageBuilder());
33         }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
37         ↪ IsUnsignedInteger<T>(root, (string)null);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
41         ↪ messageBuilder)
42         {
43             if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
44             ↪ NumericType<T>.IsFloatPoint)
45             {
46                 throw new NotSupportedException(messageBuilder());
47             }
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
52         ↪ message)
53         {
54             string messageBuilder() => message;
55             IsSignedInteger<T>(root, messageBuilder());
56         }
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
60         ↪ IsSignedInteger<T>(root, (string)null);
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
64         ↪ messageBuilder)
65         {
66             if (!NumericType<T>.IsSigned)
67             {
68                 throw new NotSupportedException(messageBuilder());
69             }
70         }
71
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
74         {
75             string messageBuilder() => message;
76             IsSigned<T>(root, messageBuilder());
77         }
78
79         [MethodImpl(MethodImplOptions.AggressiveInlining)]
80         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
81         ↪ (string)null);
82
83         [MethodImpl(MethodImplOptions.AggressiveInlining)]
84         public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
85         ↪ messageBuilder)
86         {
87             if (!NumericType<T>.IsNumeric)

```

```

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

```

```

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

```

```

110     }
111 }
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 private static void ConvertToInteger(this ILGenerator generator, Type targetType)
115 {
116     if (targetType == typeof(sbyte))
117     {
118         generator.Emit(OpCodes.Conv_I1);
119     }
120     else if (targetType == typeof(byte))
121     {
122         generator.Emit(OpCodes.Conv_U1);
123     }
124     else if (targetType == typeof(short))
125     {
126         generator.Emit(OpCodes.Conv_I2);
127     }
128     else if (targetType == typeof(ushort))
129     {
130         generator.Emit(OpCodes.Conv_U2);
131     }
132     else if (targetType == typeof(int))
133     {
134         generator.Emit(OpCodes.Conv_I4);
135     }
136     else if (targetType == typeof(uint))
137     {
138         generator.Emit(OpCodes.Conv_U4);
139     }
140     else if (targetType == typeof(long))
141     {
142         generator.Emit(OpCodes.Conv_I8);
143     }
144     else if (targetType == typeof(ulong))
145     {
146         generator.Emit(OpCodes.Conv_U8);
147     }
148 }
149
150 [MethodImpl(MethodImplOptions.AggressiveInlining)]
151 public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
152 {
153     var sourceType = typeof(TSource);
154     var targetType = typeof(TTarget);
155     if (sourceType == targetType)
156     {
157         return;
158     }
159     if (targetType == typeof(short))
160     {
161         if (NumericType<TSource>.IsSigned)
162         {
163             generator.Emit(OpCodes.Conv_Ovf_I2);
164         }
165         else
166         {
167             generator.Emit(OpCodes.Conv_Ovf_I2_Un);
168         }
169     }
170     else if (targetType == typeof(ushort))
171     {
172         if (NumericType<TSource>.IsSigned)
173         {
174             generator.Emit(OpCodes.Conv_Ovf_U2);
175         }
176         else
177         {
178             generator.Emit(OpCodes.Conv_Ovf_U2_Un);
179         }
180     }
181     else if (targetType == typeof(sbyte))
182     {
183         if (NumericType<TSource>.IsSigned)
184         {
185             generator.Emit(OpCodes.Conv_Ovf_I1);
186         }
187         else

```

```

188     {
189         generator.Emit(OpCodes.Conv_Ovf_I1_Un);
190     }
191 }
192 else if (targetType == typeof(byte))
193 {
194     if (NumericType<TSource>.IsSigned)
195     {
196         generator.Emit(OpCodes.Conv_Ovf_U1);
197     }
198     else
199     {
200         generator.Emit(OpCodes.Conv_Ovf_U1_Un);
201     }
202 }
203 else if (targetType == typeof(int))
204 {
205     if (NumericType<TSource>.IsSigned)
206     {
207         generator.Emit(OpCodes.Conv_Ovf_I4);
208     }
209     else
210     {
211         generator.Emit(OpCodes.Conv_Ovf_I4_Un);
212     }
213 }
214 else if (targetType == typeof(uint))
215 {
216     if (NumericType<TSource>.IsSigned)
217     {
218         generator.Emit(OpCodes.Conv_Ovf_U4);
219     }
220     else
221     {
222         generator.Emit(OpCodes.Conv_Ovf_U4_Un);
223     }
224 }
225 else if (targetType == typeof(long))
226 {
227     if (NumericType<TSource>.IsSigned)
228     {
229         generator.Emit(OpCodes.Conv_Ovf_I8);
230     }
231     else
232     {
233         generator.Emit(OpCodes.Conv_Ovf_I8_Un);
234     }
235 }
236 else if (targetType == typeof(ulong))
237 {
238     if (NumericType<TSource>.IsSigned)
239     {
240         generator.Emit(OpCodes.Conv_Ovf_U8);
241     }
242     else
243     {
244         generator.Emit(OpCodes.Conv_Ovf_U8_Un);
245     }
246 }
247 else if (targetType == typeof(float))
248 {
249     if (NumericType<TSource>.IsSigned)
250     {
251         generator.Emit(OpCodes.Conv_R4);
252     }
253     else
254     {
255         generator.Emit(OpCodes.Conv_R_Un);
256     }
257 }
258 else if (targetType == typeof(double))
259 {
260     generator.Emit(OpCodes.Conv_R8);
261 }
262 else if (targetType == typeof(bool))
263 {
264     generator.ConvertToBoolean<TSource>();
265 }

```



```

266     else
267     {
268         throw new NotSupportedException();
269     }
270 }
271
272 [MethodImpl(MethodImplOptions.AggressiveInlining)]
273 public static void LoadConstant(this ILGenerator generator, bool value) =>
274     ↪ generator.LoadConstant(value ? 1 : 0);
275
276 [MethodImpl(MethodImplOptions.AggressiveInlining)]
277 public static void LoadConstant(this ILGenerator generator, float value) =>
278     ↪ generator.Emit(OpCodes.Ldc_R4, value);
279
280 [MethodImpl(MethodImplOptions.AggressiveInlining)]
281 public static void LoadConstant(this ILGenerator generator, double value) =>
282     ↪ generator.Emit(OpCodes.Ldc_R8, value);
283
284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
285 public static void LoadConstant(this ILGenerator generator, long value) =>
286     ↪ generator.Emit(OpCodes.Ldc_I8, value);
287
288 [MethodImpl(MethodImplOptions.AggressiveInlining)]
289 public static void LoadConstant(this ILGenerator generator, uint value)
290 {
291     switch (value)
292     {
293         case uint.MaxValue:
294             generator.Emit(OpCodes.Ldc_I4_M1);
295             return;
296         case 0:
297             generator.Emit(OpCodes.Ldc_I4_0);
298             return;
299         case 1:
300             generator.Emit(OpCodes.Ldc_I4_1);
301             return;
302         case 2:
303             generator.Emit(OpCodes.Ldc_I4_2);
304             return;
305         case 3:
306             generator.Emit(OpCodes.Ldc_I4_3);
307             return;
308         case 4:
309             generator.Emit(OpCodes.Ldc_I4_4);
310             return;
311         case 5:
312             generator.Emit(OpCodes.Ldc_I4_5);
313             return;
314         case 6:
315             generator.Emit(OpCodes.Ldc_I4_6);
316             return;
317         case 7:
318             generator.Emit(OpCodes.Ldc_I4_7);
319             return;
320         case 8:
321             generator.Emit(OpCodes.Ldc_I4_8);
322             return;
323         default:
324             if (value <= sbyte.MaxValue)
325             {
326                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
327             }
328             else
329             {
330                 generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
331             }
332             return;
333     }
334 }
335
336 [MethodImpl(MethodImplOptions.AggressiveInlining)]
337 public static void LoadConstant(this ILGenerator generator, int value)
338 {
339     switch (value)
340     {
341         case -1:

```

```

341         generator.Emit(OpCodes.Ldc_I4_M1);
342         return;
343     case 0:
344         generator.Emit(OpCodes.Ldc_I4_0);
345         return;
346     case 1:
347         generator.Emit(OpCodes.Ldc_I4_1);
348         return;
349     case 2:
350         generator.Emit(OpCodes.Ldc_I4_2);
351         return;
352     case 3:
353         generator.Emit(OpCodes.Ldc_I4_3);
354         return;
355     case 4:
356         generator.Emit(OpCodes.Ldc_I4_4);
357         return;
358     case 5:
359         generator.Emit(OpCodes.Ldc_I4_5);
360         return;
361     case 6:
362         generator.Emit(OpCodes.Ldc_I4_6);
363         return;
364     case 7:
365         generator.Emit(OpCodes.Ldc_I4_7);
366         return;
367     case 8:
368         generator.Emit(OpCodes.Ldc_I4_8);
369         return;
370     default:
371         if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
372         {
373             generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
374         }
375         else
376         {
377             generator.Emit(OpCodes.Ldc_I4, value);
378         }
379         return;
380     }
381 }
382
383 [MethodImpl(MethodImplOptions.AggressiveInlining)]
384 public static void LoadConstant(this ILGenerator generator, short value) =>
385     ↪ generator.LoadConstant((int)value);
386
387 [MethodImpl(MethodImplOptions.AggressiveInlining)]
388 public static void LoadConstant(this ILGenerator generator, ushort value) =>
389     ↪ generator.LoadConstant((int)value);
390
391 [MethodImpl(MethodImplOptions.AggressiveInlining)]
392 public static void LoadConstant(this ILGenerator generator, sbyte value) =>
393     ↪ generator.LoadConstant((int)value);
394
395 [MethodImpl(MethodImplOptions.AggressiveInlining)]
396 public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
397     ↪ LoadConstantOne(generator, typeof(TConstant));
398
399 [MethodImpl(MethodImplOptions.AggressiveInlining)]
400 public static void LoadConstantOne(this ILGenerator generator, Type constantType)
401 {
402     if (constantType == typeof(float))
403     {
404         generator.LoadConstant(1F);
405     }
406     else if (constantType == typeof(double))
407     {
408         generator.LoadConstant(1D);
409     }
410     else if (constantType == typeof(long))
411     {
412         generator.LoadConstant(1L);
413     }
414     else if (constantType == typeof(ulong))
415     {
416         generator.LoadConstant(1UL);

```

```

416     }
417     else if (constantType == typeof(int))
418     {
419         generator.LoadConstant(1);
420     }
421     else if (constantType == typeof(uint))
422     {
423         generator.LoadConstant(1U);
424     }
425     else if (constantType == typeof(short))
426     {
427         generator.LoadConstant((short)1);
428     }
429     else if (constantType == typeof(ushort))
430     {
431         generator.LoadConstant((ushort)1);
432     }
433     else if (constantType == typeof(sbyte))
434     {
435         generator.LoadConstant((sbyte)1);
436     }
437     else if (constantType == typeof(byte))
438     {
439         generator.LoadConstant((byte)1);
440     }
441     else
442     {
443         throw new NotSupportedException();
444     }
445 }
446
447 [MethodImpl(MethodImplOptions.AggressiveInlining)]
448 public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
    ↪ constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
449
450 [MethodImpl(MethodImplOptions.AggressiveInlining)]
451 public static void LoadConstant(this ILGenerator generator, Type constantType, object
    ↪ constantValue)
452 {
453     constantValue = Convert.ChangeType(constantValue, constantType);
454     if (constantType == typeof(float))
455     {
456         generator.LoadConstant((float)constantValue);
457     }
458     else if (constantType == typeof(double))
459     {
460         generator.LoadConstant((double)constantValue);
461     }
462     else if (constantType == typeof(long))
463     {
464         generator.LoadConstant((long)constantValue);
465     }
466     else if (constantType == typeof(ulong))
467     {
468         generator.LoadConstant((ulong)constantValue);
469     }
470     else if (constantType == typeof(int))
471     {
472         generator.LoadConstant((int)constantValue);
473     }
474     else if (constantType == typeof(uint))
475     {
476         generator.LoadConstant((uint)constantValue);
477     }
478     else if (constantType == typeof(short))
479     {
480         generator.LoadConstant((short)constantValue);
481     }
482     else if (constantType == typeof(ushort))
483     {
484         generator.LoadConstant((ushort)constantValue);
485     }
486     else if (constantType == typeof(sbyte))
487     {
488         generator.LoadConstant((sbyte)constantValue);
489     }
490     else if (constantType == typeof(byte))
491     {

```

```

492         generator.LoadConstant((byte) constantValue);
493     }
494     else
495     {
496         throw new NotSupportedException();
497     }
498 }
499
500 [MethodImpl(MethodImplOptions.AggressiveInlining)]
501 public static void Increment<TValue>(this ILGenerator generator) =>
502     ↪ generator.Increment(typeof(TValue));
503
504 [MethodImpl(MethodImplOptions.AggressiveInlining)]
505 public static void Decrement<TValue>(this ILGenerator generator) =>
506     ↪ generator.Decrement(typeof(TValue));
507
508 [MethodImpl(MethodImplOptions.AggressiveInlining)]
509 public static void Increment(this ILGenerator generator, Type valueType)
510 {
511     generator.LoadConstantOne(valueType);
512     generator.Add();
513 }
514
515 [MethodImpl(MethodImplOptions.AggressiveInlining)]
516 public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
517
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 public static void Decrement(this ILGenerator generator, Type valueType)
520 {
521     generator.LoadConstantOne(valueType);
522     generator.Subtract();
523 }
524
525 [MethodImpl(MethodImplOptions.AggressiveInlining)]
526 public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
527
528 [MethodImpl(MethodImplOptions.AggressiveInlining)]
529 public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
530
531 [MethodImpl(MethodImplOptions.AggressiveInlining)]
532 public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
533
534 [MethodImpl(MethodImplOptions.AggressiveInlining)]
535 public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
536
537 [MethodImpl(MethodImplOptions.AggressiveInlining)]
538 public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
539
540 [MethodImpl(MethodImplOptions.AggressiveInlining)]
541 public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
542
543 [MethodImpl(MethodImplOptions.AggressiveInlining)]
544 public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
545
546 [MethodImpl(MethodImplOptions.AggressiveInlining)]
547 public static void LoadArgument(this ILGenerator generator, int argumentIndex)
548 {
549     switch (argumentIndex)
550     {
551         case 0:
552             generator.Emit(OpCodes.Ldarg_0);
553             break;
554         case 1:
555             generator.Emit(OpCodes.Ldarg_1);
556             break;
557         case 2:
558             generator.Emit(OpCodes.Ldarg_2);
559             break;
560         case 3:
561             generator.Emit(OpCodes.Ldarg_3);
562             break;
563         default:
564             generator.Emit(OpCodes.Ldarg, argumentIndex);
565             break;
566     }
567 }
568
569 [MethodImpl(MethodImplOptions.AggressiveInlining)]
570 public static void LoadArguments(this ILGenerator generator, params int[]
571     ↪ argumentIndices)

```

```

569 {
570     for (var i = 0; i < argumentIndices.Length; i++)
571     {
572         generator.LoadArgument(argumentIndices[i]);
573     }
574 }
575
576 [MethodImpl(MethodImplOptions.AggressiveInlining)]
577 public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
578     ↪ generator.Emit(OpCodes.Starg, argumentIndex);
579
580 [MethodImpl(MethodImplOptions.AggressiveInlining)]
581 public static void CompareGreaterThan(this ILGenerator generator) =>
582     ↪ generator.Emit(OpCodes.Cgt);
583
584 [MethodImpl(MethodImplOptions.AggressiveInlining)]
585 public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
586     ↪ generator.Emit(OpCodes.Cgt_Un);
587
588 [MethodImpl(MethodImplOptions.AggressiveInlining)]
589 public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
590 {
591     if (isSigned)
592     {
593         generator.CompareGreaterThan();
594     }
595     else
596     {
597         generator.UnsignedCompareGreaterThan();
598     }
599 }
600
601 [MethodImpl(MethodImplOptions.AggressiveInlining)]
602 public static void CompareLessThan(this ILGenerator generator) =>
603     ↪ generator.Emit(OpCodes.Clt);
604
605 [MethodImpl(MethodImplOptions.AggressiveInlining)]
606 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
607     ↪ generator.Emit(OpCodes.Clt_Un);
608
609 [MethodImpl(MethodImplOptions.AggressiveInlining)]
610 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
611 {
612     if (isSigned)
613     {
614         generator.CompareLessThan();
615     }
616     else
617     {
618         generator.UnsignedCompareLessThan();
619     }
620 }
621
622 [MethodImpl(MethodImplOptions.AggressiveInlining)]
623 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
624     ↪ generator.Emit(OpCodes.Bge, label);
625
626 [MethodImpl(MethodImplOptions.AggressiveInlining)]
627 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
628     ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
629
630 [MethodImpl(MethodImplOptions.AggressiveInlining)]
631 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
632     ↪ Label label)
633 {
634     if (isSigned)
635     {
636         generator.BranchIfGreaterOrEqual(label);
637     }
638     else
639     {
640         generator.UnsignedBranchIfGreaterOrEqual(label);
641     }
642 }
643
644 [MethodImpl(MethodImplOptions.AggressiveInlining)]
645 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
646     ↪ generator.Emit(OpCodes.Ble, label);

```

```

638 [MethodImpl(MethodImplOptions.AggressiveInlining)]
639 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
640     => generator.Emit(OpCodes.Ble_Un, label);
641
642 [MethodImpl(MethodImplOptions.AggressiveInlining)]
643 public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
644     => label)
645 {
646     if (isSigned)
647     {
648         generator.BranchIfLessOrEqual(label);
649     }
650     else
651     {
652         generator.UnsignedBranchIfLessOrEqual(label);
653     }
654 }
655
656 [MethodImpl(MethodImplOptions.AggressiveInlining)]
657 public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
658
659 [MethodImpl(MethodImplOptions.AggressiveInlining)]
660 public static void Box(this ILGenerator generator, Type boxedType) =>
661     => generator.Emit(OpCodes.Box, boxedType);
662
663 [MethodImpl(MethodImplOptions.AggressiveInlining)]
664 public static void Call(this ILGenerator generator, MethodInfo method) =>
665     => generator.Emit(OpCodes.Call, method);
666
667 [MethodImpl(MethodImplOptions.AggressiveInlining)]
668 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
669
670 [MethodImpl(MethodImplOptions.AggressiveInlining)]
671 public static void Unbox<TUnbox>(this ILGenerator generator) =>
672     => generator.Unbox(typeof(TUnbox));
673
674 [MethodImpl(MethodImplOptions.AggressiveInlining)]
675 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
676     => generator.Emit(OpCodes.Unbox, typeToUnbox);
677
678 [MethodImpl(MethodImplOptions.AggressiveInlining)]
679 public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
680     => generator.UnboxValue(typeof(TUnbox));
681
682 [MethodImpl(MethodImplOptions.AggressiveInlining)]
683 public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
684     => generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
685
686 [MethodImpl(MethodImplOptions.AggressiveInlining)]
687 public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
688     => generator.DeclareLocal(typeof(T));
689
690 [MethodImpl(MethodImplOptions.AggressiveInlining)]
691 public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
692     => generator.Emit(OpCodes.Ldloc, local);
693
694 [MethodImpl(MethodImplOptions.AggressiveInlining)]
695 public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
696     => generator.Emit(OpCodes.Stloc, local);
697
698 [MethodImpl(MethodImplOptions.AggressiveInlining)]
699 public static void NewObject(this ILGenerator generator, Type type, params Type[]
700     => parameterTypes)
701 {
702     var allConstructors = type.GetConstructors(BindingFlags.Public |
703         => BindingFlags.NonPublic | BindingFlags.Instance
704         | BindingFlags.CreateInstance
705     );
706     var constructor = allConstructors.Where(c => c.GetParameters().Length ==
707         => parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
708         => parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
709     if (constructor == null)
710     {
711

```

```

        throw new InvalidOperationException("Type " + type + " must have a constructor
        ↪ that matches parameters [" + string.Join(", ",
        ↪ parameterTypes.AsEnumerable()) + "]");
    }
    generator.NewObject(constructor);
}

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

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

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
    ↪ false, byte? unaligned = null)
{
    if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
    {
        throw new ArgumentException("unaligned must be null, 1, 2, or 4");
    }
    if (isVolatile)
    {
        generator.Emit(OpCodes.Volatile);
    }
    if (unaligned.HasValue)
    {
        generator.Emit(OpCodes.Unaligned, unaligned.Value);
    }
    if (type.IsPointer)
    {
        generator.Emit(OpCodes.Ldind_I);
    }
    else if (!type.IsValueType)
    {
        generator.Emit(OpCodes.Ldind_Ref);
    }
    else if (type == typeof(sbyte))
    {
        generator.Emit(OpCodes.Ldind_I1);
    }
    else if (type == typeof(bool))
    {
        generator.Emit(OpCodes.Ldind_I1);
    }
    else if (type == typeof(byte))
    {
        generator.Emit(OpCodes.Ldind_U1);
    }
    else if (type == typeof(short))
    {
        generator.Emit(OpCodes.Ldind_I2);
    }
    else if (type == typeof(ushort))
    {
        generator.Emit(OpCodes.Ldind_U2);
    }
    else if (type == typeof(char))
    {
        generator.Emit(OpCodes.Ldind_U2);
    }
    else if (type == typeof(int))
    {
        generator.Emit(OpCodes.Ldind_I4);
    }
    else if (type == typeof(uint))
    {
        generator.Emit(OpCodes.Ldind_U4);
    }
    else if (type == typeof(long) || type == typeof(ulong))
    {
        generator.Emit(OpCodes.Ldind_I8);
    }
    else if (type == typeof(float))
    {

```

```

771         generator.Emit(OpCodes.Ldind_R4);
772     }
773     else if (type == typeof(double))
774     {
775         generator.Emit(OpCodes.Ldind_R8);
776     }
777     else
778     {
779         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
780             ↪ " ", LoadObject may be more appropriate");
781     }
782 }
783 [MethodImpl(MethodImplOptions.AggressiveInlining)]
784 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
785     ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
786 [MethodImpl(MethodImplOptions.AggressiveInlining)]
787 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
788     ↪ = false, byte? unaligned = null)
789 {
790     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
791     {
792         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
793     }
794     if (isVolatile)
795     {
796         generator.Emit(OpCodes.Volatile);
797     }
798     if (unaligned.HasValue)
799     {
800         generator.Emit(OpCodes.Unaligned, unaligned.Value);
801     }
802     if (type.IsPointer)
803     {
804         generator.Emit(OpCodes.Stind_I);
805     }
806     else if (!type.IsValueType)
807     {
808         generator.Emit(OpCodes.Stind_Ref);
809     }
810     else if (type == typeof(sbyte) || type == typeof(byte))
811     {
812         generator.Emit(OpCodes.Stind_I1);
813     }
814     else if (type == typeof(short) || type == typeof(ushort))
815     {
816         generator.Emit(OpCodes.Stind_I2);
817     }
818     else if (type == typeof(int) || type == typeof(uint))
819     {
820         generator.Emit(OpCodes.Stind_I4);
821     }
822     else if (type == typeof(long) || type == typeof(ulong))
823     {
824         generator.Emit(OpCodes.Stind_I8);
825     }
826     else if (type == typeof(float))
827     {
828         generator.Emit(OpCodes.Stind_R4);
829     }
830     else if (type == typeof(double))
831     {
832         generator.Emit(OpCodes.Stind_R8);
833     }
834     else
835     {
836         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
837             ↪ + " ", StoreObject may be more appropriate");
838     }
839 }

```

## 1.7 ./Platform.Reflection/MethodInfoExtensions.cs

```

1 using System;
2 using System.Linq;
3 using System.Reflection;

```



```

4 using System.Runtime.CompilerServices;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Reflection
9 {
10     public static class MethodInfoExtensions
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static byte[] GetILBytes(this MethodInfo methodInfo) =>
14             ↪ methodInfo.GetMethodBody().GetILAsByteArray();
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
18             ↪ methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
19     }
20 }

```

## 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 BytesSize;
25         public static readonly int BitsSize;
26         public static readonly T MinValue;
27         public static readonly T MaxValue;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         static NumericType()
31         {
32             try

```

```

33     {
34         var type = typeof(T);
35         var isNullable = type.IsNullable();
36         var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
37         var canBeNumeric = underlyingType.CanBeNumeric();
38         var isNumeric = underlyingType.IsNumeric();
39         var isSigned = underlyingType.IsSigned();
40         var isFloatPoint = underlyingType.IsFloatPoint();
41         var bytesSize = Marshal.SizeOf(underlyingType);
42         var bitsSize = bytesSize * 8;
43         GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
44         GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
45             ↪ out Type unsignedVersion);
46         Type = type;
47         IsNullable = isNullable;
48         UnderlyingType = underlyingType;
49         CanBeNumeric = canBeNumeric;
50         IsNumeric = isNumeric;
51         IsSigned = isSigned;
52         IsFloatPoint = isFloatPoint;
53         BytesSize = bytesSize;
54         BitsSize = bitsSize;
55         MinValue = minValue;
56         MaxValue = maxValue;
57         SignedVersion = signedVersion;
58         UnsignedVersion = unsignedVersion;
59     }
60     catch (Exception exception)
61     {
62         exception.Ignore();
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
82     ↪ signedVersion, out Type unsignedVersion)
83 {
84     if (isSigned)
85     {
86         signedVersion = type;
87         unsignedVersion = type.GetUnsignedVersionOrNull();
88     }
89     else
90     {
91         signedVersion = type.GetSignedVersionOrNull();
92         unsignedVersion = type;
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) =>
12             ↪ (T)fieldInfo.GetValue(null);
13     }
14 }

```

```
13 }
```

### 1.11 ./Platform.Reflection/TypeBuilderExtensions.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  using System;
4  using System.Reflection;
5  using System.Reflection.Emit;
6  using System.Runtime.CompilerServices;
7
8  namespace Platform.Reflection
9  {
10     public static class TypeBuilderExtensions
11     {
12         public static readonly MethodAttributes DefaultStaticMethodAttributes =
13             ↪ MethodAttributes.Public | MethodAttributes.Static;
14         public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes =
15             ↪ MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
16             ↪ MethodAttributes.HideBySig;
17         public static readonly MethodImplAttributes DefaultMethodImplAttributes =
18             ↪ MethodImplAttributes.IL | MethodImplAttributes.Managed |
19             ↪ MethodImplAttributes.AggressiveInlining;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName,
23             ↪ MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes,
24             ↪ Action<ILGenerator> emitCode)
25         {
26             typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[]
27                 ↪ parameterTypes);
28             EmitMethod(type, methodName, methodAttributes, methodImplAttributes, returnType,
29                 ↪ parameterTypes, emitCode);
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
34             ↪ methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[]
35             ↪ parameterTypes, Action<ILGenerator> emitCode)
36         {
37             MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
38                 ↪ parameterTypes);
39             method.SetImplementationFlags(methodImplAttributes);
40             var generator = method.GetILGenerator();
41             emitCode(generator);
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName,
46             ↪ Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
47             ↪ DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static void EmitFinalVirtualMethod<TDelegate>(this TypeBuilder type, string
51             ↪ methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
52             ↪ DefaultFinalVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
53     }
54 }
```

### 1.12 ./Platform.Reflection/TypeExtensions.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Reflection;
5  using System.Runtime.CompilerServices;
6  using Platform.Collections;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class TypeExtensions
13     {
14         static public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
15             ↪ BindingFlags.NonPublic | BindingFlags.Static;
16         static public readonly string DefaultDelegateMethodName = "Invoke";
17
18         static private readonly HashSet<Type> _canBeNumericTypes;
19         static private readonly HashSet<Type> _isNumericTypes;
20         static private readonly HashSet<Type> _isSignedTypes;
```

```

20 static private readonly HashSet<Type> _isFloatPointTypes;
21 static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
22 static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;
23
24 [MethodImpl(MethodImplOptions.AggressiveInlining)]
25 static TypeExtensions()
26 {
27     _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
28         ↳ typeof(DateTime), typeof(TimeSpan) };
29     _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
30         ↳ typeof(ulong) };
31     _canBeNumericTypes.UnionWith(_isNumericTypes);
32     _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
33         ↳ typeof(long) };
34     _canBeNumericTypes.UnionWith(_isSignedTypes);
35     _isNumericTypes.UnionWith(_isSignedTypes);
36     _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
37         ↳ typeof(float) };
38     _canBeNumericTypes.UnionWith(_isFloatPointTypes);
39     _isNumericTypes.UnionWith(_isFloatPointTypes);
40     _isSignedTypes.UnionWith(_isFloatPointTypes);
41     unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
42     {
43         { typeof(sbyte), typeof(byte) },
44         { typeof(short), typeof(ushort) },
45         { typeof(int), typeof(uint) },
46         { typeof(long), typeof(ulong) },
47     };
48     signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
49     {
50         { typeof(byte), typeof(sbyte) },
51         { typeof(ushort), typeof(short) },
52         { typeof(uint), typeof(int) },
53         { typeof(ulong), typeof(long) },
54     };
55 }
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public static T GetStaticFieldValue<T>(this Type type, string name) =>
62     ↳ type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public static T GetStaticPropertyValue<T>(this Type type, string name) =>
66     ↳ type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
70     ↳ genericParameterTypes, Type[] argumentTypes)
71 {
72     var methods = from m in type.GetMethods()
73         where m.Name == name
74         && m.IsGenericMethodDefinition
75         let typeParams = m.GetGenericArguments()
76         let normalParams = m.GetParameters().Select(x => x.ParameterType)
77         where typeParams.SequenceEqual(genericParameterTypes)
78         && normalParams.SequenceEqual(argumentTypes)
79         select m;
80     var method = methods.Single();
81     return method;
82 }
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public static Type GetBaseType(this Type type) => type.BaseType;
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public static Assembly GetAssembly(this Type type) => type.Assembly;
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public static bool IsSubclassOf(this Type type, Type superClass) =>
92     ↳ type.IsSubclassOf(superClass);
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 public static bool IsValueType(this Type type) => type.IsValueType;
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

90     public static bool IsGeneric(this Type type) => type.IsGenericType;
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
94         ↪ type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
98
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100    public static Type GetUnsignedVersionOrNull(this Type signedType) =>
101        ↪ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
102
103    [MethodImpl(MethodImplOptions.AggressiveInlining)]
104    public static Type GetSignedVersionOrNull(this Type unsignedType) =>
105        ↪ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
106
107    [MethodImpl(MethodImplOptions.AggressiveInlining)]
108    public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
109
110    [MethodImpl(MethodImplOptions.AggressiveInlining)]
111    public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
112
113    [MethodImpl(MethodImplOptions.AggressiveInlining)]
114    public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
115
116    [MethodImpl(MethodImplOptions.AggressiveInlining)]
117    public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
118
119    [MethodImpl(MethodImplOptions.AggressiveInlining)]
120    public static Type GetDelegateReturnType(this Type delegateType) =>
121        ↪ delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;
122
123    [MethodImpl(MethodImplOptions.AggressiveInlining)]
124    public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
125        ↪ delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();
126
127    [MethodImpl(MethodImplOptions.AggressiveInlining)]
128    public static void GetDelegateCharacteristics(this Type delegateType, out Type
129        ↪ returnType, out Type[] parameterTypes)
130    {
131        var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
132        returnType = invoke.ReturnType;
133        parameterTypes = invoke.GetParameterTypes();
134    }
135 }

```

### 1.13 ./Platform.Reflection/Types.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Collections.ObjectModel;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8  #pragma warning disable CA1819 // Properties should not return arrays
9
10 namespace Platform.Reflection
11 {
12     public abstract class Types
13     {
14         public static ReadOnlyCollection<Type> Collection { get; } = new
15             ↪ ReadOnlyCollection<Type>(System.Array.Empty<Type>());
16         public static Type[] Array => Collection.ToArray();
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected ReadOnlyCollection<Type> ToReadOnlyCollection()
20         {
21             var types = GetType().GetGenericArguments();
22             var result = new List<Type>();
23             AppendTypes(result, types);
24             return new ReadOnlyCollection<Type>(result);
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         private static void AppendTypes(List<Type> container, IList<Type> types)
29         {
30             for (var i = 0; i < types.Count; i++)

```

```

30     {
31         var element = types[i];
32         if (element != typeof(Types))
33         {
34             if (element.IsSubclassOf(typeof(Types)))
35             {
36                 AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<
37                     ↳ <Type>>(nameof(Types<object>.Collection))));
38             }
39             else
40             {
41                 container.Add(element);
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,
13             ↳ T4, T5, T6, T7>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

#### 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,
13             ↳ T4, T5, T6>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

#### 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,
13             ↳ T4, T5>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

### 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,
13             ↪ T4>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }
```

### 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,
13             ↪ T3>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }
```

### 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,
13             ↪ T2>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }
```

### 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
13             ↪ Types<T>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }
```

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

```

1  using System;
2  using Xunit;
3
4  namespace Platform.Reflection.Tests
5  {
6      public class CodeGenerationTests
7      {
8          [Fact]
9          public void EmptyActionCompilationTest()
10         {
11             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
12             {
13                 generator.Return();
14             });
15             compiledAction();
16         }
17
18         [Fact]
19         public void FailedActionCompilationTest()
20         {
21             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
22             {
23                 throw new NotImplementedException();
24             });
25             Assert.Throws<NotSupportedException>(compiledAction);
26         }
27
28         [Fact]
29         public void ConstantLoadingTest()
30         {
31             CheckConstantLoading<byte>(8);
32             CheckConstantLoading<uint>(8);
33             CheckConstantLoading<ushort>(8);
34             CheckConstantLoading<ulong>(8);
35         }
36
37         private void CheckConstantLoading<T>(T value)
38         {
39             var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
40             {
41                 generator.LoadConstant(value);
42                 generator.Return();
43             });
44             Assert.Equal(value, compiledFunction());
45         }
46
47         [Fact]
48         public void UnsignedIntegersConversionWithSignExtensionTest()
49         {
50             object[] withSignExtension = new object[]
51             {
52                 CompileUncheckedConverter<byte, sbyte>(extendSign: true)(128),
53                 CompileUncheckedConverter<byte, short>(extendSign: true)(128),
54                 CompileUncheckedConverter<ushort, short>(extendSign: true)(32768),
55                 CompileUncheckedConverter<byte, int>(extendSign: true)(128),
56                 CompileUncheckedConverter<ushort, int>(extendSign: true)(32768),
57                 CompileUncheckedConverter<uint, int>(extendSign: true)(2147483648),
58                 CompileUncheckedConverter<byte, long>(extendSign: true)(128),
59                 CompileUncheckedConverter<ushort, long>(extendSign: true)(32768),
60                 CompileUncheckedConverter<uint, long>(extendSign: true)(2147483648),
61                 CompileUncheckedConverter<ulong, long>(extendSign: true)(9223372036854775808)
62             };
63             object[] withoutSignExtension = new object[]
64             {
65                 CompileUncheckedConverter<byte, sbyte>(extendSign: false)(128),
66                 CompileUncheckedConverter<byte, short>(extendSign: false)(128),
67                 CompileUncheckedConverter<ushort, short>(extendSign: false)(32768),
68                 CompileUncheckedConverter<byte, int>(extendSign: false)(128),
69                 CompileUncheckedConverter<ushort, int>(extendSign: false)(32768),
70                 CompileUncheckedConverter<uint, int>(extendSign: false)(2147483648),
71                 CompileUncheckedConverter<byte, long>(extendSign: false)(128),
72                 CompileUncheckedConverter<ushort, long>(extendSign: false)(32768),
73                 CompileUncheckedConverter<uint, long>(extendSign: false)(2147483648),
74                 CompileUncheckedConverter<ulong, long>(extendSign: false)(9223372036854775808)
75             };
76             var i = 0;
77             Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);

```



```

78     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
79     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
80     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
81     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
82     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
83     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
84     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
85     Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
86     Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
87 }
88
89 [Fact]
90 public void SignedIntegersConversionOfMinusOneWithSignExtensionTest()
91 {
92     object[] withSignExtension = new object[]
93     {
94         CompileUncheckedConverter<sbyte, byte>(extendSign: true)(-1),
95         CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(-1),
96         CompileUncheckedConverter<short, ushort>(extendSign: true)(-1),
97         CompileUncheckedConverter<sbyte, uint>(extendSign: true)(-1),
98         CompileUncheckedConverter<short, uint>(extendSign: true)(-1),
99         CompileUncheckedConverter<int, uint>(extendSign: true)(-1),
100        CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(-1),
101        CompileUncheckedConverter<short, ulong>(extendSign: true)(-1),
102        CompileUncheckedConverter<int, ulong>(extendSign: true)(-1),
103        CompileUncheckedConverter<long, ulong>(extendSign: true)(-1)
104    };
105    object[] withoutSignExtension = new object[]
106    {
107        CompileUncheckedConverter<sbyte, byte>(extendSign: false)(-1),
108        CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(-1),
109        CompileUncheckedConverter<short, ushort>(extendSign: false)(-1),
110        CompileUncheckedConverter<sbyte, uint>(extendSign: false)(-1),
111        CompileUncheckedConverter<short, uint>(extendSign: false)(-1),
112        CompileUncheckedConverter<int, uint>(extendSign: false)(-1),
113        CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(-1),
114        CompileUncheckedConverter<short, ulong>(extendSign: false)(-1),
115        CompileUncheckedConverter<int, ulong>(extendSign: false)(-1),
116        CompileUncheckedConverter<long, ulong>(extendSign: false)(-1)
117    };
118    var i = 0;
119    Assert.Equal((byte)255, (byte)withSignExtension[i]);
120    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
121    Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
122    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
123    Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
124    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
125    Assert.Equal(4294967295, withSignExtension[i]);
126    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
127    Assert.Equal(4294967295, withSignExtension[i]);
128    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
129    Assert.Equal(4294967295, withSignExtension[i]);
130    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
131    Assert.Equal(18446744073709551615, withSignExtension[i]);
132    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
133    Assert.Equal(18446744073709551615, withSignExtension[i]);
134    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
135    Assert.Equal(18446744073709551615, withSignExtension[i]);
136    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
137    Assert.Equal(18446744073709551615, withSignExtension[i]);
138    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
139 }
140
141 [Fact]
142 public void SignedIntegersConversionOfTwoWithSignExtensionTest()
143 {
144     object[] withSignExtension = new object[]
145     {
146         CompileUncheckedConverter<sbyte, byte>(extendSign: true)(2),
147         CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(2),
148         CompileUncheckedConverter<short, ushort>(extendSign: true)(2),
149         CompileUncheckedConverter<sbyte, uint>(extendSign: true)(2),
150         CompileUncheckedConverter<short, uint>(extendSign: true)(2),
151         CompileUncheckedConverter<int, uint>(extendSign: true)(2),
152         CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(2),
153         CompileUncheckedConverter<short, ulong>(extendSign: true)(2),
154         CompileUncheckedConverter<int, ulong>(extendSign: true)(2),
155         CompileUncheckedConverter<long, ulong>(extendSign: true)(2)

```

```

156     };
157     object[] withoutSignExtension = new object[]
158     {
159         CompileUncheckedConverter<sbyte, byte>(extendSign: false)(2),
160         CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(2),
161         CompileUncheckedConverter<short, ushort>(extendSign: false)(2),
162         CompileUncheckedConverter<sbyte, uint>(extendSign: false)(2),
163         CompileUncheckedConverter<short, uint>(extendSign: false)(2),
164         CompileUncheckedConverter<int, uint>(extendSign: false)(2),
165         CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(2),
166         CompileUncheckedConverter<short, ulong>(extendSign: false)(2),
167         CompileUncheckedConverter<int, ulong>(extendSign: false)(2),
168         CompileUncheckedConverter<long, ulong>(extendSign: false)(2)
169     };
170     for (var i = 0; i < withSignExtension.Length; i++)
171     {
172         Assert.Equal(2UL, Convert.ToUInt64(withSignExtension[i]));
173         Assert.Equal(withSignExtension[i], withoutSignExtension[i]);
174     }
175 }
176
177 private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
178     ↪ TTarget>(bool extendSign)
179 {
180     return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
181     {
182         generator.LoadArgument(0);
183         generator.UncheckedConvert<TSource, TTarget>(extendSign);
184         generator.Return();
185     });
186 }
187 }

```

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

```

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

```

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

```

1  using Xunit;
2
3  namespace Platform.Reflection.Tests
4  {
5      public class NumericTypeTests
6      {
7          [Fact]
8          public void UInt64IsNumericTest()
9          {

```

```
10         Assert.True(NumericType<ulong>.IsNumeric);
11     }
12 }
13 }
```

## Index

- ./Platform.Reflection.Tests/CodeGenerationTests.cs, 23
- ./Platform.Reflection.Tests/GetILBytesMethodTests.cs, 26
- ./Platform.Reflection.Tests/NumericTypeTests.cs, 26
- ./Platform.Reflection/AssemblyExtensions.cs, 1
- ./Platform.Reflection/DelegateHelpers.cs, 1
- ./Platform.Reflection/DynamicExtensions.cs, 2
- ./Platform.Reflection/EnsureExtensions.cs, 2
- ./Platform.Reflection/FieldInfoExtensions.cs, 5
- ./Platform.Reflection/ILGeneratorExtensions.cs, 5
- ./Platform.Reflection/MethodInfoExtensions.cs, 16
- ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 17
- ./Platform.Reflection/NumericType.cs, 17
- ./Platform.Reflection/PropertyInfoExtensions.cs, 18
- ./Platform.Reflection/TypeBuilderExtensions.cs, 19
- ./Platform.Reflection/TypeExtensions.cs, 19
- ./Platform.Reflection/Types.cs, 21
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3, T4].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3].cs, 23
- ./Platform.Reflection/Types[T1, T2].cs, 23
- ./Platform.Reflection/Types[T].cs, 23