LinksPlatform's Platform.Reflection Class Library

./Platform.Reflection/AssemblyExtensions.cs

```
using System;
using System.Collections.Concurrent;
using System.Reflection;
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>
        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);
            }
        }

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

./Platform.Reflection/DelegateHelpers.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Reflection.Emit;
using Platform.Exceptions;

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

namespace Platform.Reflection
{
    public static class DelegateHelpers
    {
        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;
        }

        public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
        ↪  TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);

        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
        ↪  aggressiveInlining)
            where TDelegate : Delegate
        {
            var @delegate = CompileOrDefault<TDelegate>(emitCode, aggressiveInlining);
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
```

```
                @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
            }
            return @delegate;
        }

        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
        ↪  : Delegate => Compile<TDelegate>(emitCode, false);

        private static TDelegate CompileUsingDynamicMethod<TDelegate>(Action<ILGenerator>
        ↪  emitCode)
        {
            var delegateType = typeof(TDelegate);
            var invoke = delegateType.GetMethod("Invoke");
            var returnType = invoke.ReturnType;
            var parameterTypes = invoke.GetParameters().Select(s => s.ParameterType).ToArray();
            var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
            var generator = dynamicMethod.GetILGenerator();
            emitCode(generator);
            return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
        }

        private static TDelegate CompileUsingMethodBuilder<TDelegate>(Action<ILGenerator>
        ↪  emitCode)
        {
            AssemblyName assemblyName = new AssemblyName(GetNewName());
            var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
            ↪  AssemblyBuilderAccess.Run);
            var module = assembly.DefineDynamicModule(GetNewName());
            var type = module.DefineType(GetNewName());
            var delegateType = typeof(TDelegate);
            var invoke = delegateType.GetMethod("Invoke");
            var returnType = invoke.ReturnType;
            var parameterTypes = invoke.GetParameters().Select(s => s.ParameterType).ToArray();
            var methodName = GetNewName();
            MethodBuilder method = type.DefineMethod(methodName, MethodAttributes.Public |
            ↪  MethodAttributes.Static, returnType, parameterTypes);
            method.SetImplementationFlags(MethodImplAttributes.IL | MethodImplAttributes.Managed
            ↪  | MethodImplAttributes.AggressiveInlining);
            var generator = method.GetILGenerator();
            emitCode(generator);
            var typeInfo = type.CreateTypeInfo();
            return
            ↪  (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(delegateType);
        }

        private static string GetNewName() => Guid.NewGuid().ToString("N");
    }
}
```

./Platform.Reflection/DynamicExtensions.cs

```
using System.Collections.Generic;

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

namespace Platform.Reflection
{
    public static class DynamicExtensions
    {
        public static bool HasProperty(this object @object, string propertyName)
        {
            var type = @object.GetType();
            if (type is IDictionary<string, object> dictionary)
            {
                return dictionary.ContainsKey(propertyName);
            }
            return type.GetProperty(propertyName) != null;
        }
    }
}
```

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

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

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

```
        {
            if (!NumericType<T>.IsNumeric)
            {
                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")]
        public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
        ↪  messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);

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

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

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

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

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

        #endregion
    }
}
```

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

./Platform.Reflection/ILGeneratorExtensions.cs

```
using System;
using System.Linq;
using System.Reflection;
using System.Reflection.Emit;

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

namespace Platform.Reflection
{
    public static class ILGeneratorExtensions
    {
        public static void Throw<T>(this ILGenerator generator) =>
        ↪  generator.ThrowException(typeof(T));

        public static void ConvertTo<T>(this ILGenerator generator)
        {
            var type = typeof(T);
            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);
```

```
32              }
33              else
34              {
35                  throw new NotSupportedException();
36              }
37          }
38
39          public static void LoadConstant(this ILGenerator generator, bool value) =>
            ↪  generator.LoadConstant(value ? 1 : 0);
40
41          public static void LoadConstant(this ILGenerator generator, float value) =>
            ↪  generator.Emit(OpCodes.Ldc_R4, value);
42
43          public static void LoadConstant(this ILGenerator generator, double value) =>
            ↪  generator.Emit(OpCodes.Ldc_R8, value);
44
45          public static void LoadConstant(this ILGenerator generator, ulong value) =>
            ↪  generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
46
47          public static void LoadConstant(this ILGenerator generator, long value) =>
            ↪  generator.Emit(OpCodes.Ldc_I8, value);
48
49          public static void LoadConstant(this ILGenerator generator, uint value)
50          {
51              switch (value)
52              {
53                  case uint.MaxValue:
54                      generator.Emit(OpCodes.Ldc_I4_M1);
55                      return;
56                  case 0:
57                      generator.Emit(OpCodes.Ldc_I4_0);
58                      return;
59                  case 1:
60                      generator.Emit(OpCodes.Ldc_I4_1);
61                      return;
62                  case 2:
63                      generator.Emit(OpCodes.Ldc_I4_2);
64                      return;
65                  case 3:
66                      generator.Emit(OpCodes.Ldc_I4_3);
67                      return;
68                  case 4:
69                      generator.Emit(OpCodes.Ldc_I4_4);
70                      return;
71                  case 5:
72                      generator.Emit(OpCodes.Ldc_I4_5);
73                      return;
74                  case 6:
75                      generator.Emit(OpCodes.Ldc_I4_6);
76                      return;
77                  case 7:
78                      generator.Emit(OpCodes.Ldc_I4_7);
79                      return;
80                  case 8:
81                      generator.Emit(OpCodes.Ldc_I4_8);
82                      return;
83                  default:
84                      if (value <= sbyte.MaxValue)
85                      {
86                          generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
87                      }
88                      else
89                      {
90                          generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
91                      }
92                      return;
93              }
94          }
95
96          public static void LoadConstant(this ILGenerator generator, int value)
97          {
98              switch (value)
99              {
100                 case -1:
101                     generator.Emit(OpCodes.Ldc_I4_M1);
102                     return;
103                 case 0:
104                     generator.Emit(OpCodes.Ldc_I4_0);
105                     return;
106                 case 1:
```

```
                    generator.Emit(OpCodes.Ldc_I4_1);
                    return;
                case 2:
                    generator.Emit(OpCodes.Ldc_I4_2);
                    return;
                case 3:
                    generator.Emit(OpCodes.Ldc_I4_3);
                    return;
                case 4:
                    generator.Emit(OpCodes.Ldc_I4_4);
                    return;
                case 5:
                    generator.Emit(OpCodes.Ldc_I4_5);
                    return;
                case 6:
                    generator.Emit(OpCodes.Ldc_I4_6);
                    return;
                case 7:
                    generator.Emit(OpCodes.Ldc_I4_7);
                    return;
                case 8:
                    generator.Emit(OpCodes.Ldc_I4_8);
                    return;
                default:
                    if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
                    {
                        generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
                    }
                    else
                    {
                        generator.Emit(OpCodes.Ldc_I4, value);
                    }
                    return;
            }
        }

        public static void LoadConstant(this ILGenerator generator, short value)
        {
            generator.LoadConstant((int)value);
        }

        public static void LoadConstant(this ILGenerator generator, ushort value)
        {
            generator.LoadConstant((int)value);
        }

        public static void LoadConstant(this ILGenerator generator, sbyte value)
        {
            generator.LoadConstant((int)value);
        }

        public static void LoadConstant(this ILGenerator generator, byte value)
        {
            generator.LoadConstant((int)value);
        }

        public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
        ↪  LoadConstantOne(generator, typeof(TConstant));

        public static void LoadConstantOne(this ILGenerator generator, Type constantType)
        {
            if (constantType == typeof(float))
            {
                generator.LoadConstant(1F);
            }
            else if (constantType == typeof(double))
            {
                generator.LoadConstant(1D);
            }
            else if (constantType == typeof(long))
            {
                generator.LoadConstant(1L);
            }
            else if (constantType == typeof(ulong))
            {
                generator.LoadConstant(1UL);
            }
            else if (constantType == typeof(int))
            {
                generator.LoadConstant(1);
```

```
186             }
187             else if (constantType == typeof(uint))
188             {
189                 generator.LoadConstant(1U);
190             }
191             else if (constantType == typeof(short))
192             {
193                 generator.LoadConstant((short)1);
194             }
195             else if (constantType == typeof(ushort))
196             {
197                 generator.LoadConstant((ushort)1);
198             }
199             else if (constantType == typeof(sbyte))
200             {
201                 generator.LoadConstant((sbyte)1);
202             }
203             else if (constantType == typeof(byte))
204             {
205                 generator.LoadConstant((byte)1);
206             }
207             else
208             {
209                 throw new NotSupportedException();
210             }
211         }
212 
213         public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
            ↪ constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
214 
215         public static void LoadConstant(this ILGenerator generator, Type constantType, object
            ↪ constantValue)
216         {
217             constantValue = Convert.ChangeType(constantValue, constantType);
218             if (constantType == typeof(float))
219             {
220                 generator.LoadConstant((float)constantValue);
221             }
222             else if (constantType == typeof(double))
223             {
224                 generator.LoadConstant((double)constantValue);
225             }
226             else if (constantType == typeof(long))
227             {
228                 generator.LoadConstant((long)constantValue);
229             }
230             else if (constantType == typeof(ulong))
231             {
232                 generator.LoadConstant((ulong)constantValue);
233             }
234             else if (constantType == typeof(int))
235             {
236                 generator.LoadConstant((int)constantValue);
237             }
238             else if (constantType == typeof(uint))
239             {
240                 generator.LoadConstant((uint)constantValue);
241             }
242             else if (constantType == typeof(short))
243             {
244                 generator.LoadConstant((short)constantValue);
245             }
246             else if (constantType == typeof(ushort))
247             {
248                 generator.LoadConstant((ushort)constantValue);
249             }
250             else if (constantType == typeof(sbyte))
251             {
252                 generator.LoadConstant((sbyte)constantValue);
253             }
254             else if (constantType == typeof(byte))
255             {
256                 generator.LoadConstant((byte)constantValue);
257             }
258             else
259             {
260                 throw new NotSupportedException();
261             }
```

```
262         }
263
264         public static void Increment<TValue>(this ILGenerator generator) =>
            ↪ generator.Increment(typeof(TValue));
265
266         public static void Decrement<TValue>(this ILGenerator generator) =>
            ↪ generator.Decrement(typeof(TValue));
267
268         public static void Increment(this ILGenerator generator, Type valueType)
269         {
270             generator.LoadConstantOne(valueType);
271             generator.Add();
272         }
273
274         public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
275
276         public static void Decrement(this ILGenerator generator, Type valueType)
277         {
278             generator.LoadConstantOne(valueType);
279             generator.Subtract();
280         }
281
282         public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
283
284         public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
285
286         public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
287
288         public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
289
290         public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
291
292         public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
293
294         public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
295
296         public static void LoadArgument(this ILGenerator generator, int argumentIndex)
297         {
298             switch (argumentIndex)
299             {
300                 case 0:
301                     generator.Emit(OpCodes.Ldarg_0);
302                     break;
303                 case 1:
304                     generator.Emit(OpCodes.Ldarg_1);
305                     break;
306                 case 2:
307                     generator.Emit(OpCodes.Ldarg_2);
308                     break;
309                 case 3:
310                     generator.Emit(OpCodes.Ldarg_3);
311                     break;
312                 default:
313                     generator.Emit(OpCodes.Ldarg, argumentIndex);
314                     break;
315             }
316         }
317
318         public static void LoadArguments(this ILGenerator generator, params int[]
            ↪ argumentIndices)
319         {
320             for (var i = 0; i < argumentIndices.Length; i++)
321             {
322                 generator.LoadArgument(argumentIndices[i]);
323             }
324         }
325
326         public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
            ↪ generator.Emit(OpCodes.Starg, argumentIndex);
327
328         public static void CompareGreaterThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Cgt);
329
330         public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Cgt_Un);
331
332         public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
333         {
334             if (isSigned)
```

```
335             {
336                 generator.CompareGreaterThan();
337             }
338             else
339             {
340                 generator.UnsignedCompareGreaterThan();
341             }
342         }
343 
344         public static void CompareLessThan(this ILGenerator generator) =>
        ↪  generator.Emit(OpCodes.Clt);
345 
346         public static void UnsignedCompareLessThan(this ILGenerator generator) =>
        ↪  generator.Emit(OpCodes.Clt_Un);
347 
348         public static void CompareLessThan(this ILGenerator generator, bool isSigned)
349         {
350             if (isSigned)
351             {
352                 generator.CompareLessThan();
353             }
354             else
355             {
356                 generator.UnsignedCompareLessThan();
357             }
358         }
359 
360         public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
        ↪  generator.Emit(OpCodes.Bge, label);
361 
362         public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
        ↪  label) => generator.Emit(OpCodes.Bge_Un, label);
363 
364         public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
        ↪  Label label)
365         {
366             if (isSigned)
367             {
368                 generator.BranchIfGreaterOrEqual(label);
369             }
370             else
371             {
372                 generator.UnsignedBranchIfGreaterOrEqual(label);
373             }
374         }
375 
376         public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
        ↪  generator.Emit(OpCodes.Ble, label);
377 
378         public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
        ↪  => generator.Emit(OpCodes.Ble_Un, label);
379 
380         public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
        ↪  label)
381         {
382             if (isSigned)
383             {
384                 generator.BranchIfLessOrEqual(label);
385             }
386             else
387             {
388                 generator.UnsignedBranchIfLessOrEqual(label);
389             }
390         }
391 
392         public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
393 
394         public static void Box(this ILGenerator generator, Type boxedType) =>
        ↪  generator.Emit(OpCodes.Box, boxedType);
395 
396         public static void Call(this ILGenerator generator, MethodInfo method) =>
        ↪  generator.Emit(OpCodes.Call, method);
397 
398         public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
399 
400         public static void Unbox<TUnbox>(this ILGenerator generator) =>
        ↪  generator.Unbox(typeof(TUnbox));
401 
```

```
402         public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
            ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
403
404         public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
            ↪ generator.UnboxValue(typeof(TUnbox));
405
406         public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
            ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
407
408         public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
            ↪ generator.DeclareLocal(typeof(T));
409
410         public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪ generator.Emit(OpCodes.Ldloc, local);
411
412         public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪ generator.Emit(OpCodes.Stloc, local);
413
414         public static void NewObject(this ILGenerator generator, Type type, params Type[]
            ↪ parameterTypes)
415         {
416             var allConstructors = type.GetConstructors(BindingFlags.Public |
                ↪ BindingFlags.NonPublic | BindingFlags.Instance
417 #if !NETSTANDARD
418                 | BindingFlags.CreateInstance
419 #endif
420                 );
421             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();
422             if (constructor == null)
423             {
424                 throw new InvalidOperationException("Type " + type + " must have a constructor
                    ↪ that matches parameters [" + string.Join(", ",
                    ↪ parameterTypes.AsEnumerable()) + "]");
425             }
426             generator.NewObject(constructor);
427         }
428
429         public static void NewObject(this ILGenerator generator, ConstructorInfo constructor)
430         {
431             generator.Emit(OpCodes.Newobj, constructor);
432         }
433
434         public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
435
436         public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
            ↪ false, byte? unaligned = null)
437         {
438             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
439             {
440                 throw new ArgumentException("unaligned must be null, 1, 2, or 4");
441             }
442             if (isVolatile)
443             {
444                 generator.Emit(OpCodes.Volatile);
445             }
446             if (unaligned.HasValue)
447             {
448                 generator.Emit(OpCodes.Unaligned, unaligned.Value);
449             }
450             if (type.IsPointer)
451             {
452                 generator.Emit(OpCodes.Ldind_I);
453             }
454             else if (!type.IsValueType)
455             {
456                 generator.Emit(OpCodes.Ldind_Ref);
457             }
458             else if (type == typeof(sbyte))
459             {
460                 generator.Emit(OpCodes.Ldind_I1);
461             }
462             else if (type == typeof(bool))
463             {
464                 generator.Emit(OpCodes.Ldind_I1);
```

```
465             }
466             else if (type == typeof(byte))
467             {
468                 generator.Emit(OpCodes.Ldind_U1);
469             }
470             else if (type == typeof(short))
471             {
472                 generator.Emit(OpCodes.Ldind_I2);
473             }
474             else if (type == typeof(ushort))
475             {
476                 generator.Emit(OpCodes.Ldind_U2);
477             }
478             else if (type == typeof(char))
479             {
480                 generator.Emit(OpCodes.Ldind_U2);
481             }
482             else if (type == typeof(int))
483             {
484                 generator.Emit(OpCodes.Ldind_I4);
485             }
486             else if (type == typeof(uint))
487             {
488                 generator.Emit(OpCodes.Ldind_U4);
489             }
490             else if (type == typeof(long) || type == typeof(ulong))
491             {
492                 generator.Emit(OpCodes.Ldind_I8);
493             }
494             else if (type == typeof(float))
495             {
496                 generator.Emit(OpCodes.Ldind_R4);
497             }
498             else if (type == typeof(double))
499             {
500                 generator.Emit(OpCodes.Ldind_R8);
501             }
502             else
503             {
504                 throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
                    ↪  ", LoadObject may be more appropriate");
505             }
506         }
507 
508         public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪  byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
509 
510         public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
            ↪  = false, byte? unaligned = null)
511         {
512             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
513             {
514                 throw new ArgumentException("unaligned must be null, 1, 2, or 4");
515             }
516             if (isVolatile)
517             {
518                 generator.Emit(OpCodes.Volatile);
519             }
520             if (unaligned.HasValue)
521             {
522                 generator.Emit(OpCodes.Unaligned, unaligned.Value);
523             }
524             if (type.IsPointer)
525             {
526                 generator.Emit(OpCodes.Stind_I);
527             }
528             else if (!type.IsValueType)
529             {
530                 generator.Emit(OpCodes.Stind_Ref);
531             }
532             else if (type == typeof(sbyte) || type == typeof(byte))
533             {
534                 generator.Emit(OpCodes.Stind_I1);
535             }
536             else if (type == typeof(short) || type == typeof(ushort))
537             {
538                 generator.Emit(OpCodes.Stind_I2);
539             }
```

```
            else if (type == typeof(int) || type == typeof(uint))
            {
                generator.Emit(OpCodes.Stind_I4);
            }
            else if (type == typeof(long) || type == typeof(ulong))
            {
                generator.Emit(OpCodes.Stind_I8);
            }
            else if (type == typeof(float))
            {
                generator.Emit(OpCodes.Stind_R4);
            }
            else if (type == typeof(double))
            {
                generator.Emit(OpCodes.Stind_R8);
            }
            else
            {
                throw new InvalidOperationException("StoreIndirect cannot be used with " + type
                ↪  + ", StoreObject may be more appropriate");
            }
        }
    }
}
```

./Platform.Reflection/MethodInfoExtensions.cs

```
using System.Reflection;

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

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

./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs

```
using System;
using System.Collections.Generic;
using Platform.Interfaces;

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

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

./Platform.Reflection/NumericType.cs

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

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

namespace Platform.Reflection
{
    public static class NumericType<T>
```

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

        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();
            }
        }

        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));
            }
        }

        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;
            }
        }
    }
}
```

./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);
    }
}
```

./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;

        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);
    }
}
```

./Platform.Reflection/Types.cs

```
using System;
using System.Collections.Generic;
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 abstract class Types
    {
        public static ReadOnlyCollection<Type> Collection { get; } = new
        ↪ ReadOnlyCollection<Type>(new Type[0]);
```

```
        public static Type[] Array => Collection.ToArray();

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

        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);
                    }
                }
            }
        }
    }
```

./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() { }
    }
}
```

./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() { }
    }
}
```

./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() { }
    }
}
```

./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() { }
    }
}
```

./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() { }
    }
}
```

./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() { }
    }
}
```

./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() { }
    }
}
```

./Platform.Reflection.Tests/CodeGenerationTests.cs

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

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;
    }
}
```

./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));
        }
    }
}
```

./Platform.Reflection.Tests/NumericTypeTests.cs

```
using Xunit;

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

Index