1.1 ./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);
    }
}
```

1.2 ./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");
    }
}
```

1.3 ./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;
        }
    }
}
```

1.4 ./Platform.Reflection/EnsureExtensions.cs

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

#pragma warning disable IDE0060 // Remove unused parameter
```

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

namespace Platform.Reflection
{
    public static class EnsureExtensions
    {
        #region Always

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root,
        ↪  Func<string> messageBuilder)
        {
            if (!NumericType<T>.IsNumeric || NumericType<T>.IsSigned ||
            ↪  NumericType<T>.IsFloatPoint)
            {
                throw new NotSupportedException(messageBuilder());
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
        ↪  message)
        {
            string messageBuilder() => message;
            IsUnsignedInteger<T>(root, messageBuilder);
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
        ↪  messageBuilder)
        {
            if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
            ↪  NumericType<T>.IsFloatPoint)
            {
                throw new NotSupportedException(messageBuilder());
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
        ↪  message)
        {
            string messageBuilder() => message;
            IsSignedInteger<T>(root, messageBuilder);
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
        ↪  messageBuilder)
        {
            if (!NumericType<T>.IsSigned)
            {
                throw new NotSupportedException(messageBuilder());
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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>();
```

```
141
142         [Conditional("DEBUG")]
143         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
144
145         [Conditional("DEBUG")]
146         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
            ↪  Ensure.Always.IsNumeric<T>(message);
147
148         [Conditional("DEBUG")]
149         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsNumeric<T>();
150
151         [Conditional("DEBUG")]
152         public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.CanBeNumeric<T>(messageBuilder);
153
154         [Conditional("DEBUG")]
155         public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
            ↪  => Ensure.Always.CanBeNumeric<T>(message);
156
157         [Conditional("DEBUG")]
158         public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.CanBeNumeric<T>();
159
160         #endregion
161     }
162 }
```

1.5 ./Platform.Reflection/FieldInfoExtensions.cs

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

1.6 ./Platform.Reflection/ILGeneratorExtensions.cs

```
1  using System;
2  using System.Linq;
3  using System.Reflection;
4  using System.Reflection.Emit;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Reflection
9  {
10     public static class ILGeneratorExtensions
11     {
12         public static void Throw<T>(this ILGenerator generator) =>
           ↪  generator.ThrowException(typeof(T));
13
14         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator)
15         {
16             var type = typeof(TTarget);
17             if (type == typeof(short))
18             {
19                 generator.Emit(OpCodes.Conv_I2);
20             }
21             else if (type == typeof(ushort))
22             {
23                 generator.Emit(OpCodes.Conv_U2);
24             }
25             else if (type == typeof(sbyte))
26             {
27                 generator.Emit(OpCodes.Conv_I1);
28             }
29             else if (type == typeof(byte))
30             {
31                 generator.Emit(OpCodes.Conv_U1);
```

```
            }
            else if (type == typeof(int))
            {
                generator.Emit(OpCodes.Conv_I4);
            }
            else if (type == typeof(uint))
            {
                generator.Emit(OpCodes.Conv_U4);
            }
            else if (type == typeof(long))
            {
                generator.Emit(OpCodes.Conv_I8);
            }
            else if (type == typeof(ulong))
            {
                generator.Emit(OpCodes.Conv_U8);
            }
            else if (type == typeof(float))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_R4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_R_Un);
                }
            }
            else if (type == typeof(double))
            {
                generator.Emit(OpCodes.Conv_R8);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
        {
            var type = typeof(TTarget);
            if (type == typeof(short))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2_Un);
                }
            }
            else if (type == typeof(ushort))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U2);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U2_Un);
                }
            }
            else if (type == typeof(sbyte))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I1);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I1_Un);
                }
            }
            else if (type == typeof(byte))
            {
                if (NumericType<TSource>.IsSigned)
                {
```

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

        public static void LoadConstant(this ILGenerator generator, bool value) =>
        ↪  generator.LoadConstant(value ? 1 : 0);

        public static void LoadConstant(this ILGenerator generator, float value) =>
        ↪  generator.Emit(OpCodes.Ldc_R4, value);

```

```
186         public static void LoadConstant(this ILGenerator generator, double value) =>
            ↪ generator.Emit(OpCodes.Ldc_R8, value);
187
188         public static void LoadConstant(this ILGenerator generator, ulong value) =>
            ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
189
190         public static void LoadConstant(this ILGenerator generator, long value) =>
            ↪ generator.Emit(OpCodes.Ldc_I8, value);
191
192         public static void LoadConstant(this ILGenerator generator, uint value)
193         {
194             switch (value)
195             {
196                 case uint.MaxValue:
197                     generator.Emit(OpCodes.Ldc_I4_M1);
198                     return;
199                 case 0:
200                     generator.Emit(OpCodes.Ldc_I4_0);
201                     return;
202                 case 1:
203                     generator.Emit(OpCodes.Ldc_I4_1);
204                     return;
205                 case 2:
206                     generator.Emit(OpCodes.Ldc_I4_2);
207                     return;
208                 case 3:
209                     generator.Emit(OpCodes.Ldc_I4_3);
210                     return;
211                 case 4:
212                     generator.Emit(OpCodes.Ldc_I4_4);
213                     return;
214                 case 5:
215                     generator.Emit(OpCodes.Ldc_I4_5);
216                     return;
217                 case 6:
218                     generator.Emit(OpCodes.Ldc_I4_6);
219                     return;
220                 case 7:
221                     generator.Emit(OpCodes.Ldc_I4_7);
222                     return;
223                 case 8:
224                     generator.Emit(OpCodes.Ldc_I4_8);
225                     return;
226                 default:
227                     if (value <= sbyte.MaxValue)
228                     {
229                         generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
230                     }
231                     else
232                     {
233                         generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
234                     }
235                     return;
236             }
237         }
238
239         public static void LoadConstant(this ILGenerator generator, int value)
240         {
241             switch (value)
242             {
243                 case -1:
244                     generator.Emit(OpCodes.Ldc_I4_M1);
245                     return;
246                 case 0:
247                     generator.Emit(OpCodes.Ldc_I4_0);
248                     return;
249                 case 1:
250                     generator.Emit(OpCodes.Ldc_I4_1);
251                     return;
252                 case 2:
253                     generator.Emit(OpCodes.Ldc_I4_2);
254                     return;
255                 case 3:
256                     generator.Emit(OpCodes.Ldc_I4_3);
257                     return;
258                 case 4:
259                     generator.Emit(OpCodes.Ldc_I4_4);
260                     return;
261                 case 5:
262                     generator.Emit(OpCodes.Ldc_I4_5);
263                     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);
            }
            else if (constantType == typeof(uint))
            {
                generator.LoadConstant(1U);
            }
            else if (constantType == typeof(short))
            {
                generator.LoadConstant((short)1);
            }
            else if (constantType == typeof(ushort))
            {
                generator.LoadConstant((ushort)1);
            }
```

```
342             else if (constantType == typeof(sbyte))
343             {
344                 generator.LoadConstant((sbyte)1);
345             }
346             else if (constantType == typeof(byte))
347             {
348                 generator.LoadConstant((byte)1);
349             }
350             else
351             {
352                 throw new NotSupportedException();
353             }
354         }
355 
356         public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
        ↪  constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
357 
358         public static void LoadConstant(this ILGenerator generator, Type constantType, object
        ↪  constantValue)
359         {
360             constantValue = Convert.ChangeType(constantValue, constantType);
361             if (constantType == typeof(float))
362             {
363                 generator.LoadConstant((float)constantValue);
364             }
365             else if (constantType == typeof(double))
366             {
367                 generator.LoadConstant((double)constantValue);
368             }
369             else if (constantType == typeof(long))
370             {
371                 generator.LoadConstant((long)constantValue);
372             }
373             else if (constantType == typeof(ulong))
374             {
375                 generator.LoadConstant((ulong)constantValue);
376             }
377             else if (constantType == typeof(int))
378             {
379                 generator.LoadConstant((int)constantValue);
380             }
381             else if (constantType == typeof(uint))
382             {
383                 generator.LoadConstant((uint)constantValue);
384             }
385             else if (constantType == typeof(short))
386             {
387                 generator.LoadConstant((short)constantValue);
388             }
389             else if (constantType == typeof(ushort))
390             {
391                 generator.LoadConstant((ushort)constantValue);
392             }
393             else if (constantType == typeof(sbyte))
394             {
395                 generator.LoadConstant((sbyte)constantValue);
396             }
397             else if (constantType == typeof(byte))
398             {
399                 generator.LoadConstant((byte)constantValue);
400             }
401             else
402             {
403                 throw new NotSupportedException();
404             }
405         }
406 
407         public static void Increment<TValue>(this ILGenerator generator) =>
        ↪  generator.Increment(typeof(TValue));
408 
409         public static void Decrement<TValue>(this ILGenerator generator) =>
        ↪  generator.Decrement(typeof(TValue));
410 
411         public static void Increment(this ILGenerator generator, Type valueType)
412         {
413             generator.LoadConstantOne(valueType);
414             generator.Add();
415         }
```

```
416
417         public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
418
419         public static void Decrement(this ILGenerator generator, Type valueType)
420         {
421             generator.LoadConstantOne(valueType);
422             generator.Subtract();
423         }
424
425         public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
426
427         public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
428
429         public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
430
431         public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
432
433         public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
434
435         public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
436
437         public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
438
439         public static void LoadArgument(this ILGenerator generator, int argumentIndex)
440         {
441             switch (argumentIndex)
442             {
443                 case 0:
444                     generator.Emit(OpCodes.Ldarg_0);
445                     break;
446                 case 1:
447                     generator.Emit(OpCodes.Ldarg_1);
448                     break;
449                 case 2:
450                     generator.Emit(OpCodes.Ldarg_2);
451                     break;
452                 case 3:
453                     generator.Emit(OpCodes.Ldarg_3);
454                     break;
455                 default:
456                     generator.Emit(OpCodes.Ldarg, argumentIndex);
457                     break;
458             }
459         }
460
461         public static void LoadArguments(this ILGenerator generator, params int[]
    ↪    argumentIndices)
462         {
463             for (var i = 0; i < argumentIndices.Length; i++)
464             {
465                 generator.LoadArgument(argumentIndices[i]);
466             }
467         }
468
469         public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
    ↪    generator.Emit(OpCodes.Starg, argumentIndex);
470
471         public static void CompareGreaterThan(this ILGenerator generator) =>
    ↪    generator.Emit(OpCodes.Cgt);
472
473         public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
    ↪    generator.Emit(OpCodes.Cgt_Un);
474
475         public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
476         {
477             if (isSigned)
478             {
479                 generator.CompareGreaterThan();
480             }
481             else
482             {
483                 generator.UnsignedCompareGreaterThan();
484             }
485         }
486
487         public static void CompareLessThan(this ILGenerator generator) =>
    ↪    generator.Emit(OpCodes.Clt);
488
489         public static void UnsignedCompareLessThan(this ILGenerator generator) =>
    ↪    generator.Emit(OpCodes.Clt_Un);
```

```
490
491         public static void CompareLessThan(this ILGenerator generator, bool isSigned)
492         {
493             if (isSigned)
494             {
495                 generator.CompareLessThan();
496             }
497             else
498             {
499                 generator.UnsignedCompareLessThan();
500             }
501         }
502
503         public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
            ↪  generator.Emit(OpCodes.Bge, label);
504
505         public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
            ↪  label) => generator.Emit(OpCodes.Bge_Un, label);
506
507         public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
            ↪  Label label)
508         {
509             if (isSigned)
510             {
511                 generator.BranchIfGreaterOrEqual(label);
512             }
513             else
514             {
515                 generator.UnsignedBranchIfGreaterOrEqual(label);
516             }
517         }
518
519         public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
            ↪  generator.Emit(OpCodes.Ble, label);
520
521         public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
            ↪  => generator.Emit(OpCodes.Ble_Un, label);
522
523         public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
            ↪  label)
524         {
525             if (isSigned)
526             {
527                 generator.BranchIfLessOrEqual(label);
528             }
529             else
530             {
531                 generator.UnsignedBranchIfLessOrEqual(label);
532             }
533         }
534
535         public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
536
537         public static void Box(this ILGenerator generator, Type boxedType) =>
            ↪  generator.Emit(OpCodes.Box, boxedType);
538
539         public static void Call(this ILGenerator generator, MethodInfo method) =>
            ↪  generator.Emit(OpCodes.Call, method);
540
541         public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
542
543         public static void Unbox<TUnbox>(this ILGenerator generator) =>
            ↪  generator.Unbox(typeof(TUnbox));
544
545         public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
            ↪  generator.Emit(OpCodes.Unbox, typeToUnbox);
546
547         public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
            ↪  generator.UnboxValue(typeof(TUnbox));
548
549         public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
            ↪  generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
550
551         public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
            ↪  generator.DeclareLocal(typeof(T));
552
553         public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪  generator.Emit(OpCodes.Ldloc, local);
```

```
554
555         public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪ generator.Emit(OpCodes.Stloc, local);
556
557         public static void NewObject(this ILGenerator generator, Type type, params Type[]
            ↪ parameterTypes)
558         {
559             var allConstructors = type.GetConstructors(BindingFlags.Public |
                ↪ BindingFlags.NonPublic | BindingFlags.Instance
560 #if !NETSTANDARD
561                 | BindingFlags.CreateInstance
562 #endif
563                 );
564             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();
565             if (constructor == null)
566             {
567                 throw new InvalidOperationException("Type " + type + " must have a constructor
                    ↪ that matches parameters [" + string.Join(", ",
                    ↪ parameterTypes.AsEnumerable()) + "]");
568             }
569             generator.NewObject(constructor);
570         }
571
572         public static void NewObject(this ILGenerator generator, ConstructorInfo constructor)
573         {
574             generator.Emit(OpCodes.Newobj, constructor);
575         }
576
577         public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
578
579         public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
            ↪ false, byte? unaligned = null)
580         {
581             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
582             {
583                 throw new ArgumentException("unaligned must be null, 1, 2, or 4");
584             }
585             if (isVolatile)
586             {
587                 generator.Emit(OpCodes.Volatile);
588             }
589             if (unaligned.HasValue)
590             {
591                 generator.Emit(OpCodes.Unaligned, unaligned.Value);
592             }
593             if (type.IsPointer)
594             {
595                 generator.Emit(OpCodes.Ldind_I);
596             }
597             else if (!type.IsValueType)
598             {
599                 generator.Emit(OpCodes.Ldind_Ref);
600             }
601             else if (type == typeof(sbyte))
602             {
603                 generator.Emit(OpCodes.Ldind_I1);
604             }
605             else if (type == typeof(bool))
606             {
607                 generator.Emit(OpCodes.Ldind_I1);
608             }
609             else if (type == typeof(byte))
610             {
611                 generator.Emit(OpCodes.Ldind_U1);
612             }
613             else if (type == typeof(short))
614             {
615                 generator.Emit(OpCodes.Ldind_I2);
616             }
617             else if (type == typeof(ushort))
618             {
619                 generator.Emit(OpCodes.Ldind_U2);
620             }
621             else if (type == typeof(char))
```

```
622                 {
623                     generator.Emit(OpCodes.Ldind_U2);
624                 }
625                 else if (type == typeof(int))
626                 {
627                     generator.Emit(OpCodes.Ldind_I4);
628                 }
629                 else if (type == typeof(uint))
630                 {
631                     generator.Emit(OpCodes.Ldind_U4);
632                 }
633                 else if (type == typeof(long) || type == typeof(ulong))
634                 {
635                     generator.Emit(OpCodes.Ldind_I8);
636                 }
637                 else if (type == typeof(float))
638                 {
639                     generator.Emit(OpCodes.Ldind_R4);
640                 }
641                 else if (type == typeof(double))
642                 {
643                     generator.Emit(OpCodes.Ldind_R8);
644                 }
645                 else
646                 {
647                     throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
                        ↪  ", LoadObject may be more appropriate");
648                 }
649             }
650 
651             public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
                ↪  byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
652 
653             public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
                ↪  = false, byte? unaligned = null)
654             {
655                 if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
656                 {
657                     throw new ArgumentException("unaligned must be null, 1, 2, or 4");
658                 }
659                 if (isVolatile)
660                 {
661                     generator.Emit(OpCodes.Volatile);
662                 }
663                 if (unaligned.HasValue)
664                 {
665                     generator.Emit(OpCodes.Unaligned, unaligned.Value);
666                 }
667                 if (type.IsPointer)
668                 {
669                     generator.Emit(OpCodes.Stind_I);
670                 }
671                 else if (!type.IsValueType)
672                 {
673                     generator.Emit(OpCodes.Stind_Ref);
674                 }
675                 else if (type == typeof(sbyte) || type == typeof(byte))
676                 {
677                     generator.Emit(OpCodes.Stind_I1);
678                 }
679                 else if (type == typeof(short) || type == typeof(ushort))
680                 {
681                     generator.Emit(OpCodes.Stind_I2);
682                 }
683                 else if (type == typeof(int) || type == typeof(uint))
684                 {
685                     generator.Emit(OpCodes.Stind_I4);
686                 }
687                 else if (type == typeof(long) || type == typeof(ulong))
688                 {
689                     generator.Emit(OpCodes.Stind_I8);
690                 }
691                 else if (type == typeof(float))
692                 {
693                     generator.Emit(OpCodes.Stind_R4);
694                 }
695                 else if (type == typeof(double))
696                 {
```

```
                generator.Emit(OpCodes.Stind_R8);
            }
            else
            {
                throw new InvalidOperationException("StoreIndirect cannot be used with " + type
                ↪  + ", StoreObject may be more appropriate");
            }
        }
    }
}
```

1.7 ./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();
    }
}
```

1.8 ./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;
        }
    }
}
```

1.9 ./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;
            }
        }
    }
}
```

1.10 ./Platform.Reflection/PropertyInfoExtensions.cs

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

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

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

1.11 ./Platform.Reflection/TypeExtensions.cs

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

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

namespace Platform.Reflection
{
    public static class TypeExtensions
    {
        static private readonly HashSet<Type> _canBeNumericTypes;
        static private readonly HashSet<Type> _isNumericTypes;
        static private readonly HashSet<Type> _isSignedTypes;
        static private readonly HashSet<Type> _isFloatPointTypes;
        static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
        static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
        ↪  genericParameterTypes, Type[] argumentTypes)
        {
            var methods = from m in type.GetMethods()
                          where m.Name == name
                             && m.IsGenericMethodDefinition
                          let typeParams = m.GetGenericArguments()
                          let normalParams = m.GetParameters().Select(x => x.ParameterType)
                          where typeParams.SequenceEqual(genericParameterTypes)
                             && normalParams.SequenceEqual(argumentTypes)
                          select m;
            var method = methods.Single();
```

```
            return method;
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

1.12 ./Platform.Reflection/Types.cs

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using 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);
                    }
                }
            }
        }
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Reflection
{
    public class Types<T1, T2, T3, T4> : Types
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [Fact]
        public void EmptyActionCompilationTest()
```

```
        {
            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                generator.Return();
            });
            compiledAction();
        }

        [Fact]
        public void FailedActionCompilationTest()
        {
            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                throw new NotImplementedException();
            });
            Assert.Throws<NotSupportedException>(compiledAction);
        }

        [Fact]
        public void ConstantLoadingTest()
        {
            CheckConstantLoading<byte>(8);
            CheckConstantLoading<uint>(8);
            CheckConstantLoading<ushort>(8);
            CheckConstantLoading<ulong>(8);
        }

        private void CheckConstantLoading<T>(T value)
        {
            var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
            {
                generator.LoadConstant(value);
                generator.Return();
            });
            Assert.Equal(value, compiledFunction());
        }

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

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

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

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

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

            int result = 0;

            // Warm up

            for (int i = 0; i < N; i++)
            {
                result = MethodsContainer.DelegateWithoutAggressiveInlining();
            }
            for (int i = 0; i < N; i++)
            {
```

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

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

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

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

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

```
    }
}
```

1.21 ./Platform.Reflection.Tests/GetILBytesMethodTests.cs

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

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

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

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

```
using Xunit;

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

Index