LinksPlatform's Platform.Reflection Class Library

1.1 ./Platform.Reflection/AssemblyExtensions.cs

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

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

namespace Platform.Reflection
{
    public static class AssemblyExtensions
    {
        private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
        ↪  ConcurrentDictionary<Assembly, Type[]>();

        /// <remarks>
        /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetLoadableTypes(this Assembly assembly)
        {
            Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
            try
            {
                return assembly.GetTypes();
            }
            catch (ReflectionTypeLoadException e)
            {
                return e.Types.ToArray(t => t != null);
            }
        }

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

1.2 ./Platform.Reflection/DelegateHelpers.cs

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Exceptions;

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

namespace Platform.Reflection
{
    public static class DelegateHelpers
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode, bool
        ↪  typeMemberMethod)
            where TDelegate : Delegate
        {
            var @delegate = default(TDelegate);
            try
            {
                @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
                ↪  CompileDynamicMethod<TDelegate>(emitCode);
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
            return @delegate;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
        ↪  typeMemberMethod)
```

```
            where TDelegate : Delegate
        {
            var @delegate = CompileOrDefault<TDelegate>(emitCode, typeMemberMethod);
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
                @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
            }
            return @delegate;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileDynamicMethod<TDelegate>(Action<ILGenerator> emitCode)
        {
            var delegateType = typeof(TDelegate);
            delegateType.GetDelegateCharacteristics(out Type returnType, out Type[]
             ↪  parameterTypes);
            var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
            emitCode(dynamicMethod.GetILGenerator());
            return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileTypeMemberMethod<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 methodName = GetNewName();
            type.EmitStaticMethod<TDelegate>(methodName, emitCode);
            var typeInfo = type.CreateTypeInfo();
            return (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(typeof(TDele⌋
             ↪  gate));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string GetNewName() => Guid.NewGuid().ToString("N");
    }
}
```

1.3 ./Platform.Reflection/DynamicExtensions.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Reflection
{
    public static class DynamicExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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

```

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string> messageBuilder)
        {
            if (!NumericType<T>.IsNumeric)
            {
                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
    }
}
```

1.5 ./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);
    }
}
```

1.6 ./Platform.Reflection/ILGeneratorExtensions.cs

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator) =>
        ↪  UncheckedConvert<TSource, TTarget>(generator, extendSign: false);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator, bool
        ↪  extendSign)
        {
            var sourceType = typeof(TSource);
            var targetType = typeof(TTarget);
            if (sourceType == targetType)
            {
                return;
            }
            if (extendSign)
            {
                if (sourceType == typeof(byte))
                {
                    generator.Emit(OpCodes.Conv_I1);
                }
                if (sourceType == typeof(ushort))
                {
                    generator.Emit(OpCodes.Conv_I2);
                }
            }
            if (NumericType<TSource>.BitsSize > NumericType<TTarget>.BitsSize)
            {
                generator.ConvertToInteger(targetType);
            }
            else
            {
#if NET471
                if (sourceType == typeof(byte) || sourceType == typeof(ushort))
                {
                    if (targetType == typeof(long))
                    {
                        if (extendSign)
                        {
                            generator.Emit(OpCodes.Conv_I8);
                        }
                        else
                        {
                            generator.Emit(OpCodes.Conv_U8);
                        }
                    }
                }
                if (sourceType == typeof(uint) && targetType == typeof(long) && extendSign)
                {
                    generator.Emit(OpCodes.Conv_I8);
                }
#endif
                if (sourceType == typeof(uint) && targetType == typeof(long) && !extendSign)
                {
                    generator.Emit(OpCodes.Conv_U8);
                }
            }
            if (targetType == typeof(float))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_R4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_R_Un);
                }
            }
            else if (targetType == typeof(double))
            {
                generator.Emit(OpCodes.Conv_R8);
            }
        }

        private static void ConvertToInteger(this ILGenerator generator, Type targetType)
        {
            if (targetType == typeof(sbyte))
            {
                generator.Emit(OpCodes.Conv_I1);
            }
            else if (targetType == typeof(byte))
            {
                generator.Emit(OpCodes.Conv_U1);
            }
            else if (targetType == typeof(short))
            {
                generator.Emit(OpCodes.Conv_I2);
            }
            else if (targetType == typeof(ushort))
            {
                generator.Emit(OpCodes.Conv_U2);
            }
            else if (targetType == typeof(int))
            {
                generator.Emit(OpCodes.Conv_I4);
            }
            else if (targetType == typeof(uint))
            {
                generator.Emit(OpCodes.Conv_U4);
            }
            else if (targetType == typeof(long))
            {
                generator.Emit(OpCodes.Conv_I8);
            }
            else if (targetType == typeof(ulong))
            {
                generator.Emit(OpCodes.Conv_U8);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
        {
            var sourceType = typeof(TSource);
            var targetType = typeof(TTarget);
            if (sourceType == targetType)
            {
                return;
            }
            if (targetType == typeof(short))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I2_Un);
                }
            }
            else if (targetType == typeof(ushort))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U2);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U2_Un);
                }
            }
            else if (targetType == typeof(sbyte))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I1);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I1_Un);
                }
            }
            else if (targetType == typeof(byte))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U1);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U1_Un);
                }
            }
            else if (targetType == typeof(int))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I4_Un);
                }
            }
            else if (targetType == typeof(uint))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U4_Un);
                }
            }
            else if (targetType == typeof(long))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_I8);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_I8_Un);
                }
            }
            else if (targetType == typeof(ulong))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_Ovf_U8);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_Ovf_U8_Un);
                }
            }
            else if (targetType == typeof(float))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_R4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_R_Un);
                }
            }
            else if (targetType == typeof(double))
            {
                generator.Emit(OpCodes.Conv_R8);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, double value) =>
        ↪   generator.Emit(OpCodes.Ldc_R8, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, ulong value) =>
        ↪   generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, long value) =>
        ↪   generator.Emit(OpCodes.Ldc_I8, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, uint value)
        {
            switch (value)
            {
```

```csharp
                case uint.MaxValue:
                    generator.Emit(OpCodes.Ldc_I4_M1);
                    return;
                case 0:
                    generator.Emit(OpCodes.Ldc_I4_0);
                    return;
                case 1:
                    generator.Emit(OpCodes.Ldc_I4_1);
                    return;
                case 2:
                    generator.Emit(OpCodes.Ldc_I4_2);
                    return;
                case 3:
                    generator.Emit(OpCodes.Ldc_I4_3);
                    return;
                case 4:
                    generator.Emit(OpCodes.Ldc_I4_4);
                    return;
                case 5:
                    generator.Emit(OpCodes.Ldc_I4_5);
                    return;
                case 6:
                    generator.Emit(OpCodes.Ldc_I4_6);
                    return;
                case 7:
                    generator.Emit(OpCodes.Ldc_I4_7);
                    return;
                case 8:
                    generator.Emit(OpCodes.Ldc_I4_8);
                    return;
                default:
                    if (value <= sbyte.MaxValue)
                    {
                        generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
                    }
                    else
                    {
                        generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
                    }
                    return;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, int value)
        {
            switch (value)
            {
                case -1:
                    generator.Emit(OpCodes.Ldc_I4_M1);
                    return;
                case 0:
                    generator.Emit(OpCodes.Ldc_I4_0);
                    return;
                case 1:
                    generator.Emit(OpCodes.Ldc_I4_1);
                    return;
                case 2:
                    generator.Emit(OpCodes.Ldc_I4_2);
                    return;
                case 3:
                    generator.Emit(OpCodes.Ldc_I4_3);
                    return;
                case 4:
                    generator.Emit(OpCodes.Ldc_I4_4);
                    return;
                case 5:
                    generator.Emit(OpCodes.Ldc_I4_5);
                    return;
                case 6:
                    generator.Emit(OpCodes.Ldc_I4_6);
                    return;
                case 7:
                    generator.Emit(OpCodes.Ldc_I4_7);
                    return;
                case 8:
                    generator.Emit(OpCodes.Ldc_I4_8);
                    return;
                default:
                    if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
                    {
                        generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
                    }
                    else
                    {
                        generator.Emit(OpCodes.Ldc_I4, value);
                    }
                    return;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, short value) =>
         ↪  generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, ushort value) =>
         ↪  generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, sbyte value) =>
         ↪  generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, byte value) =>
         ↪  generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
         ↪  LoadConstantOne(generator, typeof(TConstant));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstantOne(this ILGenerator generator, Type constantType)
        {
            if (constantType == typeof(float))
            {
                generator.LoadConstant(1F);
            }
            else if (constantType == typeof(double))
            {
                generator.LoadConstant(1D);
            }
            else if (constantType == typeof(long))
            {
                generator.LoadConstant(1L);
            }
            else if (constantType == typeof(ulong))
            {
                generator.LoadConstant(1UL);
            }
            else if (constantType == typeof(int))
            {
                generator.LoadConstant(1);
            }
            else if (constantType == typeof(uint))
            {
                generator.LoadConstant(1U);
            }
            else if (constantType == typeof(short))
            {
                generator.LoadConstant((short)1);
            }
            else if (constantType == typeof(ushort))
            {
                generator.LoadConstant((ushort)1);
            }
            else if (constantType == typeof(sbyte))
            {
                generator.LoadConstant((sbyte)1);
            }
            else if (constantType == typeof(byte))
            {
                generator.LoadConstant((byte)1);
            }
            else
            {
                throw new NotSupportedException();
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
         ↪  constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, Type constantType, object
         ↪  constantValue)
        {
            constantValue = Convert.ChangeType(constantValue, constantType);
            if (constantType == typeof(float))
            {
                generator.LoadConstant((float)constantValue);
            }
            else if (constantType == typeof(double))
            {
                generator.LoadConstant((double)constantValue);
            }
            else if (constantType == typeof(long))
            {
                generator.LoadConstant((long)constantValue);
            }
            else if (constantType == typeof(ulong))
            {
                generator.LoadConstant((ulong)constantValue);
            }
            else if (constantType == typeof(int))
            {
                generator.LoadConstant((int)constantValue);
            }
            else if (constantType == typeof(uint))
            {
                generator.LoadConstant((uint)constantValue);
            }
            else if (constantType == typeof(short))
            {
                generator.LoadConstant((short)constantValue);
            }
            else if (constantType == typeof(ushort))
            {
                generator.LoadConstant((ushort)constantValue);
            }
            else if (constantType == typeof(sbyte))
            {
                generator.LoadConstant((sbyte)constantValue);
            }
            else if (constantType == typeof(byte))
            {
                generator.LoadConstant((byte)constantValue);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Increment<TValue>(this ILGenerator generator) =>
         ↪  generator.Increment(typeof(TValue));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Decrement<TValue>(this ILGenerator generator) =>
         ↪  generator.Decrement(typeof(TValue));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Increment(this ILGenerator generator, Type valueType)
        {
            generator.LoadConstantOne(valueType);
            generator.Add();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Decrement(this ILGenerator generator, Type valueType)
        {
            generator.LoadConstantOne(valueType);
            generator.Subtract();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadArgument(this ILGenerator generator, int argumentIndex)
        {
            switch (argumentIndex)
            {
                case 0:
                    generator.Emit(OpCodes.Ldarg_0);
                    break;
                case 1:
                    generator.Emit(OpCodes.Ldarg_1);
                    break;
                case 2:
                    generator.Emit(OpCodes.Ldarg_2);
                    break;
                case 3:
                    generator.Emit(OpCodes.Ldarg_3);
                    break;
                default:
                    generator.Emit(OpCodes.Ldarg, argumentIndex);
                    break;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadArguments(this ILGenerator generator, params int[]
         ↪  argumentIndices)
        {
            for (var i = 0; i < argumentIndices.Length; i++)
            {
                generator.LoadArgument(argumentIndices[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
         ↪  generator.Emit(OpCodes.Starg, argumentIndex);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareGreaterThan(this ILGenerator generator) =>
         ↪  generator.Emit(OpCodes.Cgt);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
         ↪  generator.Emit(OpCodes.Cgt_Un);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
        {
            if (isSigned)
            {
                generator.CompareGreaterThan();
            }
            else
            {
                generator.UnsignedCompareGreaterThan();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareLessThan(this ILGenerator generator) =>
        ↪   generator.Emit(OpCodes.Clt);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnsignedCompareLessThan(this ILGenerator generator) =>
        ↪   generator.Emit(OpCodes.Clt_Un);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CompareLessThan(this ILGenerator generator, bool isSigned)
        {
            if (isSigned)
            {
                generator.CompareLessThan();
            }
            else
            {
                generator.UnsignedCompareLessThan();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
        ↪   generator.Emit(OpCodes.Bge, label);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
        ↪   label) => generator.Emit(OpCodes.Bge_Un, label);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
        ↪   Label label)
        {
            if (isSigned)
            {
                generator.BranchIfGreaterOrEqual(label);
            }
            else
            {
                generator.UnsignedBranchIfGreaterOrEqual(label);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
        ↪   generator.Emit(OpCodes.Ble, label);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
        ↪   => generator.Emit(OpCodes.Ble_Un, label);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
        ↪   label)
        {
            if (isSigned)
            {
                generator.BranchIfLessOrEqual(label);
            }
            else
            {
                generator.UnsignedBranchIfLessOrEqual(label);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Box(this ILGenerator generator, Type boxedType) =>
        ↪   generator.Emit(OpCodes.Box, boxedType);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Call(this ILGenerator generator, MethodInfo method) =>
        ↪   generator.Emit(OpCodes.Call, method);
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Unbox<TUnbox>(this ILGenerator generator) => generator.Unbox(typeof(TUnbox));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Unbox(this ILGenerator generator, Type typeToUnbox) => generator.Emit(OpCodes.Unbox, typeToUnbox);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnboxValue<TUnbox>(this ILGenerator generator) => generator.UnboxValue(typeof(TUnbox));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) => generator.Emit(OpCodes.Unbox_Any, typeToUnbox);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) => generator.DeclareLocal(typeof(T));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadLocal(this ILGenerator generator, LocalBuilder local) => generator.Emit(OpCodes.Ldloc, local);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreLocal(this ILGenerator generator, LocalBuilder local) => generator.Emit(OpCodes.Stloc, local);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void NewObject(this ILGenerator generator, Type type, params Type[] parameterTypes)
        {
            var allConstructors = type.GetConstructors(BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Instance
#if !NETSTANDARD
                | BindingFlags.CreateInstance
#endif
                );
            var constructor = allConstructors.Where(c => c.GetParameters().Length == parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType == parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
            if (constructor == null)
            {
                throw new InvalidOperationException("Type " + type + " must have a constructor that matches parameters [" + string.Join(", ", parameterTypes.AsEnumerable()) + "]");
            }
            generator.NewObject(constructor);
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile = false, byte? unaligned = null)
        {
            if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
            {
                throw new ArgumentException("unaligned must be null, 1, 2, or 4");
            }
            if (isVolatile)
            {
                generator.Emit(OpCodes.Volatile);
            }
            if (unaligned.HasValue)
            {
                generator.Emit(OpCodes.Unaligned, unaligned.Value);
            }
            if (type.IsPointer)
            {
                generator.Emit(OpCodes.Ldind_I);
            }
            else if (!type.IsValueType)
            {
                generator.Emit(OpCodes.Ldind_Ref);
            }
            else if (type == typeof(sbyte))
            {
                generator.Emit(OpCodes.Ldind_I1);
            }
            else if (type == typeof(bool))
            {
                generator.Emit(OpCodes.Ldind_I1);
            }
            else if (type == typeof(byte))
            {
                generator.Emit(OpCodes.Ldind_U1);
            }
            else if (type == typeof(short))
            {
                generator.Emit(OpCodes.Ldind_I2);
            }
            else if (type == typeof(ushort))
            {
                generator.Emit(OpCodes.Ldind_U2);
            }
            else if (type == typeof(char))
            {
                generator.Emit(OpCodes.Ldind_U2);
            }
            else if (type == typeof(int))
            {
                generator.Emit(OpCodes.Ldind_I4);
            }
            else if (type == typeof(uint))
            {
                generator.Emit(OpCodes.Ldind_U4);
            }
            else if (type == typeof(long) || type == typeof(ulong))
            {
                generator.Emit(OpCodes.Ldind_I8);
            }
            else if (type == typeof(float))
            {
                generator.Emit(OpCodes.Ldind_R4);
            }
            else if (type == typeof(double))
            {
                generator.Emit(OpCodes.Ldind_R8);
            }
            else
            {
                throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
                ↪  ", LoadObject may be more appropriate");
            }
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
        ↪  = false, byte? unaligned = null)
        {
            if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
            {
                throw new ArgumentException("unaligned must be null, 1, 2, or 4");
            }
            if (isVolatile)
            {
                generator.Emit(OpCodes.Volatile);
            }
            if (unaligned.HasValue)
            {
                    generator.Emit(OpCodes.Unaligned, unaligned.Value);
                }
                if (type.IsPointer)
                {
                    generator.Emit(OpCodes.Stind_I);
                }
                else if (!type.IsValueType)
                {
                    generator.Emit(OpCodes.Stind_Ref);
                }
                else if (type == typeof(sbyte) || type == typeof(byte))
                {
                    generator.Emit(OpCodes.Stind_I1);
                }
                else if (type == typeof(short) || type == typeof(ushort))
                {
                    generator.Emit(OpCodes.Stind_I2);
                }
                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");
                }
            }
        }
}
```

1.7 ./Platform.Reflection/MethodInfoExtensions.cs

```
using System;
using System.Linq;
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 MethodInfoExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static byte[] GetILBytes(this MethodInfo methodInfo) =>
        ↪  methodInfo.GetMethodBody().GetILAsByteArray();

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

1.8 ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs

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

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

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

1.9 ./Platform.Reflection/NumericType.cs

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

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

namespace Platform.Reflection
{
    public static class NumericType<T>
    {
        public static readonly Type Type;
        public static readonly Type UnderlyingType;
        public static readonly Type SignedVersion;
        public static readonly Type UnsignedVersion;
        public static readonly bool IsFloatPoint;
        public static readonly bool IsNumeric;
        public static readonly bool IsSigned;
        public static readonly bool CanBeNumeric;
        public static readonly bool IsNullable;
        public static readonly int BytesSize;
        public static readonly int BitsSize;
        public static readonly T MinValue;
        public static readonly T MaxValue;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static NumericType()
        {
            try
            {
                var type = typeof(T);
                var isNullable = type.IsNullable();
                var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
                var canBeNumeric = underlyingType.CanBeNumeric();
                var isNumeric = underlyingType.IsNumeric();
                var isSigned = underlyingType.IsSigned();
                var isFloatPoint = underlyingType.IsFloatPoint();
                var bytesSize = Marshal.SizeOf(underlyingType);
                var bitsSize = bytesSize * 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;
                BytesSize = bytesSize;
                BitsSize = bitsSize;
                MinValue = minValue;
                MaxValue = maxValue;
                SignedVersion = signedVersion;
                UnsignedVersion = unsignedVersion;
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
        {
            if (type == typeof(bool))
            {
                minValue = (T)(object)false;
                maxValue = (T)(object)true;
            }
            else
            {
                minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
                maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
        ↪  signedVersion, out Type unsignedVersion)
        {
            if (isSigned)
            {
                signedVersion = type;
                unsignedVersion = type.GetUnsignedVersionOrNull();
            }
            else
            {
                signedVersion = type.GetSignedVersionOrNull();
                unsignedVersion = type;
            }
        }
    }
}
```

1.10 ./Platform.Reflection/PropertyInfoExtensions.cs

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

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

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

1.11 ./Platform.Reflection/TypeBuilderExtensions.cs

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

using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;

namespace Platform.Reflection
{
    public static class TypeBuilderExtensions
    {
        public static readonly MethodAttributes DefaultStaticMethodAttributes =
        ↪  MethodAttributes.Public | MethodAttributes.Static;
        public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes =
        ↪  MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
        ↪  MethodAttributes.HideBySig;
        public static readonly MethodImplAttributes DefaultMethodImplAttributes =
        ↪  MethodImplAttributes.IL | MethodImplAttributes.Managed |
        ↪  MethodImplAttributes.AggressiveInlining;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName,
        ↪  MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes,
        ↪  Action<ILGenerator> emitCode)
        {
            typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[]
            ↪  parameterTypes);
            EmitMethod(type, methodName, methodAttributes, methodImplAttributes, returnType,
            ↪  parameterTypes, emitCode);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
         ↪  methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[]
         ↪  parameterTypes, Action<ILGenerator> emitCode)
        {
            MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
             ↪  parameterTypes);
            method.SetImplementationFlags(methodImplAttributes);
            var generator = method.GetILGenerator();
            emitCode(generator);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName,
         ↪  Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
         ↪  DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitFinalVirtualMethod<TDelegate>(this TypeBuilder type, string
         ↪  methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
         ↪  DefaultFinalVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
    }
}
```

1.12 ./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 public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
         ↪  BindingFlags.NonPublic | BindingFlags.Static;
        static public readonly string DefaultDelegateMethodName = "Invoke";

        static private readonly HashSet<Type> _canBeNumericTypes;
        static private readonly HashSet<Type> _isNumericTypes;
        static private readonly HashSet<Type> _isSignedTypes;
        static private readonly HashSet<Type> _isFloatPointTypes;
        static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
        static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticFieldValue<T>(this Type type, string name) =>
        ↪   type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticPropertyValue<T>(this Type type, string name) =>
        ↪   type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.13 ./Platform.Reflection/Types.cs

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void AppendTypes(List<Type> container, IList<Type> types)
        {
            for (var i = 0; i < types.Count; i++)
            {
                var element = types[i];
                if (element != typeof(Types))
                {
                    if (element.IsSubclassOf(typeof(Types)))
                    {
                        AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection
                         ↪  <Type>>(nameof(Types<object>.Collection)));
                    }
                    else
                    {
                        container.Add(element);
                    }
                }
            }
        }
    }
}
```

1.14 ./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
#pragma warning disable CA1819 // Properties should not return arrays

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() { }
    }
}
```

1.15 ./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
#pragma warning disable CA1819 // Properties should not return arrays

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() { }
    }
}
```

1.16 ./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
#pragma warning disable CA1819 // Properties should not return arrays

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() { }
    }
}
```

1.17 ./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
#pragma warning disable CA1819 // Properties should not return arrays

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() { }
    }
}
```

1.18 ./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
#pragma warning disable CA1819 // Properties should not return arrays

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() { }
    }
}
```

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

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

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

namespace Platform.Reflection
{
    public 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() { }
    }
}
```

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

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

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

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

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

```csharp
using System;
using Xunit;

namespace Platform.Reflection.Tests
{
    public class CodeGenerationTests
    {
        [Fact]
        public void EmptyActionCompilationTest()
        {
            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                generator.Return();
            });
            compiledAction();
        }

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

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

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

            [Fact]
            public void ConversionWithSignExtensionTest()
            {
                object[] withSignExtension = new object[]
                {
                    CompileUncheckedConverter<byte, sbyte>(extendSign: true)(128),
                    CompileUncheckedConverter<byte, short>(extendSign: true)(128),
                    CompileUncheckedConverter<ushort, short>(extendSign: true)(32768),
                    CompileUncheckedConverter<byte, int>(extendSign: true)(128),
                    CompileUncheckedConverter<ushort, int>(extendSign: true)(32768),
                    CompileUncheckedConverter<uint, int>(extendSign: true)(2147483648),
                    CompileUncheckedConverter<byte, long>(extendSign: true)(128),
                    CompileUncheckedConverter<ushort, long>(extendSign: true)(32768),
                    CompileUncheckedConverter<uint, long>(extendSign: true)(2147483648),
                    CompileUncheckedConverter<ulong, long>(extendSign: true)(9223372036854775808)
                };
                object[] withoutSignExtension = new object[]
                {
                    CompileUncheckedConverter<byte, sbyte>(extendSign: false)(128),
                    CompileUncheckedConverter<byte, short>(extendSign: false)(128),
                    CompileUncheckedConverter<ushort, short>(extendSign: false)(32768),
                    CompileUncheckedConverter<byte, int>(extendSign: false)(128),
                    CompileUncheckedConverter<ushort, int>(extendSign: false)(32768),
                    CompileUncheckedConverter<uint, int>(extendSign: false)(2147483648),
                    CompileUncheckedConverter<byte, long>(extendSign: false)(128),
                    CompileUncheckedConverter<ushort, long>(extendSign: false)(32768),
                    CompileUncheckedConverter<uint, long>(extendSign: false)(2147483648),
                    CompileUncheckedConverter<ulong, long>(extendSign: false)(9223372036854775808)
                };
                var i = 0;
                Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
                Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
                Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
                Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
                Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
                Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
                Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
                Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
                Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
                Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            }

            private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
             ↪  TTarget>(bool extendSign)
            {
                return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
                {
                    generator.LoadArgument(0);
                    generator.UncheckedConvert<TSource, TTarget>(extendSign);
                    generator.Return();
                });
            }
        }
    }
```

1.22 ./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));
        }
    }
}
```

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

```
using Xunit;

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

Index