LinksPlatform's Platform.Reflection Class Library

./Platform.Reflection/AssemblyExtensions.cs

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

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

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

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

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

./Platform.Reflection/DynamicExtensions.cs

```
using System.Collections.Generic;

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

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

./Platform.Reflection/EnsureExtensions.cs

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

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

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

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

```csharp
            if (!NumericType<T>.IsNumeric || NumericType<T>.IsSigned || NumericType<T>.IsFloatPoint)
            {
                throw new NotSupportedException(messageBuilder());
            }
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        #endregion

        #region OnDebug

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        #endregion
    }
}
```

./Platform.Reflection/FieldInfoExtensions.cs

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

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

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

./Platform.Reflection/MethodInfoExtensions.cs

```
using System.Reflection;

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

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

./Platform.Reflection/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
            {
                Type = typeof(T);
                IsNullable = Type.IsNullable();
                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);
                CanBeNumeric = canBeNumeric;
                IsNumeric = isNumeric;
                IsSigned = isSigned;
                IsFloatPoint = isFloatPoint;
                BitsLength = bitsLength;
                MinValue = minValue;
                MaxValue = maxValue;
                SignedVersion = signedVersion;
                UnsignedVersion = unsignedVersion;
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
        }

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

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

./Platform.Reflection/PropertyInfoExtensions.cs

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

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

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

./Platform.Reflection/TypeExtensions.cs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        public static Type GetUnsignedVersionOrNull(this Type signedType) =>
        ↪  _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);

        public static Type GetSignedVersionOrNull(this Type unsignedType) =>
        ↪  _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);

        public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);

        public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);

        public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);

        public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
    }
}
```

./Platform.Reflection/Types.cs

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

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

namespace Platform.Reflection
{
    public abstract class Types
    {
        protected ReadOnlyCollection<Type> ToReadOnlyCollection()
        {
            var types = GetType().GetGenericArguments();
            var result = new List<Type>();
            AppendTypes(result, types);
            return new ReadOnlyCollection<Type>(result);
        }

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

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

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

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

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

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

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

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

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

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

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

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

./Platform.Reflection.Tests/GetILBytesMethodTests.cs

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

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

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

./Platform.Reflection.Tests/NumericTypeTests.cs

```
using Xunit;

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

Index