

## LinksPlatform's Platform.Reflection Class Library

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

```
1 using System;
2 using System.Collections.Concurrent;
3 using System.Reflection;
4 using Platform.Exceptions;
5 using Platform.Collections.Lists;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Reflection
10 {
11     public static class AssemblyExtensions
12     {
13         private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
14             ↳ ConcurrentDictionary<Assembly, Type[]>();
15
16         /// <remarks>
17         /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
18         /// </remarks>
19         public static Type[] GetLoadableTypes(this Assembly assembly)
20         {
21             Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
22             try
23             {
24                 return assembly.GetTypes();
25             }
26             catch (ReflectionTypeLoadException e)
27             {
28                 return e.Types.ToArray(t => t != null);
29             }
30         }
31
32         public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
33             ↳ _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
34     }
35 }
```

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

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

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

```
1 using System;
2 using System.Diagnostics;
3 using System.Runtime.CompilerServices;
4 using Platform.Exceptions;
5 using Platform.Exceptions.ExtensionRoots;
6
7 #pragma warning disable IDE0060 // Remove unused parameter
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class EnsureExtensions
13     {
14         #region Always
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root,
18             ↳ Func<string> messageBuilder)
19         {
20         }
```

```

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

```

```

86     IsNumeric<T>(root, messageBuilder);
87 }
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
    ↳ IsNumeric<T>(root, (string)null);
91
92 [MethodImpl(MethodImplOptions.AggressiveInlining)]
93 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
    ↳ messageBuilder)
94 {
95     if (!NumericType<T>.CanBeNumeric)
96     {
97         throw new NotSupportedException(messageBuilder());
98     }
99 }
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
103 {
104     string messageBuilder() => message;
105     CanBeNumeric<T>(root, messageBuilder);
106 }
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
    ↳ CanBeNumeric<T>(root, (string)null);
110
111 #endregion
112
113 #region OnDebug
114
115 [Conditional("DEBUG")]
116 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
    ↳ Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
117
118 [Conditional("DEBUG")]
119 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
    ↳ message) => Ensure.Always.IsUnsignedInteger<T>(message);
120
121 [Conditional("DEBUG")]
122 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
    ↳ Ensure.Always.IsUnsignedInteger<T>();
123
124 [Conditional("DEBUG")]
125 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
    ↳ messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
126
127 [Conditional("DEBUG")]
128 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
    ↳ message) => Ensure.Always.IsSignedInteger<T>(message);
129
130 [Conditional("DEBUG")]
131 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
    ↳ Ensure.Always.IsSignedInteger<T>();
132
133 [Conditional("DEBUG")]
134 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
    ↳ messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
135
136 [Conditional("DEBUG")]
137 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
    ↳ Ensure.Always.IsSigned<T>(message);
138
139 [Conditional("DEBUG")]
140 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
    ↳ Ensure.Always.IsSigned<T>();
141
142 [Conditional("DEBUG")]
143 public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
    ↳ messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
144
145 [Conditional("DEBUG")]
146 public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
    ↳ Ensure.Always.IsNumeric<T>(message);
147
148 [Conditional("DEBUG")]

```

```

149     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
150         ↪ Ensure.Always.IsNumeric<T>();
151
152     [Conditional("DEBUG")]
153     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
154         ↪ messageBuilder) => Ensure.Always.CanBeNumeric<T>(messageBuilder);
155
156     [Conditional("DEBUG")]
157     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
158         ↪ => Ensure.Always.CanBeNumeric<T>(message);
159
160     [Conditional("DEBUG")]
161     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
162         ↪ Ensure.Always.CanBeNumeric<T>();
163
164     #endregion
165 }
166 }

```

#### ./Platform.Reflection/FieldInfoExtensions.cs

```

1  using System.Reflection;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Reflection
7  {
8      public static class FieldInfoExtensions
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T GetStaticValue<T>(this FieldInfo fieldInfo) =>
12             ↪ (T)fieldInfo.GetValue(null);
13     }
14 }

```

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

```

1  using System.Reflection;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Reflection
6  {
7      public static class MethodInfoExtensions
8      {
9         public static byte[] GetILBytes(this MethodInfo methodInfo) =>
10             ↪ methodInfo.GetMethodBody().GetILAsByteArray();
11     }
12 }

```

#### ./Platform.Reflection/NumericType.cs

```

1  using System;
2  using System.Runtime.InteropServices;
3  using Platform.Exceptions;
4
5  // ReSharper disable AssignmentInConditionalExpression
6  // ReSharper disable BuiltInTypeReferenceStyle
7  // ReSharper disable StaticFieldInGenericType
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class NumericType<T>
13     {
14         public static readonly Type Type;
15         public static readonly Type UnderlyingType;
16         public static readonly Type SignedVersion;
17         public static readonly Type UnsignedVersion;
18         public static readonly bool IsFloatPoint;
19         public static readonly bool IsNumeric;
20         public static readonly bool IsSigned;
21         public static readonly bool CanBeNumeric;
22         public static readonly bool IsNullable;
23         public static readonly int BitsLength;
24         public static readonly T MinValue;
25         public static readonly T MaxValue;
26
27         static NumericType()
28         {
29             try
30             {

```

```

31     Type = typeof(T);
32     IsNullable = Type.IsNullable();
33     UnderlyingType = IsNullable ? Nullable.GetUnderlyingType(Type) : Type;
34     var canBeNumeric = UnderlyingType.CanBeNumeric();
35     var isNumeric = UnderlyingType.IsNumeric();
36     var isSigned = UnderlyingType.IsSigned();
37     var isFloatPoint = UnderlyingType.IsFloatPoint();
38     var bitsLength = Marshal.SizeOf(UnderlyingType) * 8;
39     GetMinAndMaxValues(UnderlyingType, out T minValue, out T maxValue);
40     GetSignedAndUnsignedVersions(UnderlyingType, isSigned, out Type signedVersion,
    ↪ out Type unsignedVersion);
41     CanBeNumeric = canBeNumeric;
42     IsNumeric = isNumeric;
43     IsSigned = isSigned;
44     IsFloatPoint = isFloatPoint;
45     BitsLength = bitsLength;
46     MinValue = minValue;
47     MaxValue = maxValue;
48     SignedVersion = signedVersion;
49     UnsignedVersion = unsignedVersion;
50 }
51 catch (Exception exception)
52 {
53     exception.Ignore();
54 }
55 }
56
57 private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
58 {
59     if (type == typeof(bool))
60     {
61         minValue = (T)(object>false;
62         maxValue = (T)(object>true;
63     }
64     else
65     {
66         minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
67         maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
68     }
69 }
70
71 private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
    ↪ signedVersion, out Type unsignedVersion)
72 {
73     if (isSigned)
74     {
75         signedVersion = type;
76         unsignedVersion = type.GetUnsignedVersionOrNull();
77     }
78     else
79     {
80         signedVersion = type.GetSignedVersionOrNull();
81         unsignedVersion = type;
82     }
83 }
84 }
85 }

```

./Platform.Reflection/PropertyInfoExtensions.cs

```

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

```

./Platform.Reflection/TypeExtensions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Reflection;

```

```

5 using System.Runtime.CompilerServices;
6 using Platform.Collections;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class TypeExtensions
13     {
14         static private readonly HashSet<Type> _canBeNumericTypes;
15         static private readonly HashSet<Type> _isNumericTypes;
16         static private readonly HashSet<Type> _isSignedTypes;
17         static private readonly HashSet<Type> _isFloatPointTypes;
18         static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
19         static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;
20
21         static TypeExtensions()
22         {
23             _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
24                 ↳ typeof(DateTime), typeof(TimeSpan) };
25             _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
26                 ↳ typeof(ulong) };
27             _canBeNumericTypes.UnionWith(_isNumericTypes);
28             _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
29                 ↳ typeof(long) };
30             _canBeNumericTypes.UnionWith(_isSignedTypes);
31             _isNumericTypes.UnionWith(_isSignedTypes);
32             _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
33                 ↳ typeof(float) };
34             _canBeNumericTypes.UnionWith(_isFloatPointTypes);
35             _isNumericTypes.UnionWith(_isFloatPointTypes);
36             _isSignedTypes.UnionWith(_isFloatPointTypes);
37             _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
38             {
39                 { typeof(sbyte), typeof(byte) },
40                 { typeof(short), typeof(ushort) },
41                 { typeof(int), typeof(uint) },
42                 { typeof(long), typeof(ulong) },
43             };
44             _signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
45             {
46                 { typeof(byte), typeof(sbyte) },
47                 { typeof(ushort), typeof(short) },
48                 { typeof(uint), typeof(int) },
49                 { typeof(ulong), typeof(long) },
50             };
51         }
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public static T GetStaticFieldValue<T>(this Type type, string name) =>
58             ↳ type.GetTypeInfo().GetField(name, BindingFlags.Public | BindingFlags.NonPublic |
59             ↳ BindingFlags.Static).GetStaticValue<T>();
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         public static T GetStaticPropertyValue<T>(this Type type, string name) =>
63             ↳ type.GetTypeInfo().GetProperty(name, BindingFlags.Public | BindingFlags.NonPublic |
64             ↳ BindingFlags.Static).GetStaticValue<T>();
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
68             ↳ genericParameterTypes, Type[] argumentTypes)
69         {
70             var methods = from m in type.GetMethods()
71                 where m.Name == name
72                 && m.IsGenericMethodDefinition
73                 let typeParams = m.GetGenericArguments()
74                 let normalParams = m.GetParameters().Select(x => x.ParameterType)
75                 where typeParams.SequenceEqual(genericParameterTypes)
76                 && normalParams.SequenceEqual(argumentTypes)
77                 select m;
78             var method = methods.Single();
79             return method;
80         }
81
82         [MethodImpl(MethodImplOptions.AggressiveInlining)]
83         public static Type GetBaseType(this Type type) => type.GetTypeInfo().BaseType;
84     }
85 }

```

```

75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     public static Assembly GetAssembly(this Type type) => type.GetTypeInfo().Assembly;
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     public static bool IsSubclassOf(this Type type, Type superClass) =>
80         type.GetTypeInfo().IsSubclassOf(superClass);
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     public static bool IsValueType(this Type type) => type.GetTypeInfo().IsValueType;
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static bool IsGeneric(this Type type) => type.GetTypeInfo().IsGenericType;
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
90         type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
94
95     public static Type GetUnsignedVersionOrNull(this Type signedType) =>
96         _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
97
98     public static Type GetSignedVersionOrNull(this Type unsignedType) =>
99         _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
100
101     public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
102
103     public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
104
105     public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
106     public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
107 }

```

# ./Platform.Reflection/Types.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Collections.ObjectModel;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Reflection
8  {
9      public abstract class Types
10     {
11         protected ReadOnlyCollection<Type> ToReadOnlyCollection()
12         {
13             var types = GetType().GetGenericArguments();
14             var result = new List<Type>();
15             AppendTypes(result, types);
16             return new ReadOnlyCollection<Type>(result);
17         }
18
19         private static void AppendTypes(List<Type> container, IList<Type> types)
20         {
21             for (var i = 0; i < types.Count; i++)
22             {
23                 var element = types[i];
24                 if (element != typeof(Types))
25                 {
26                     if (element.IsSubclassOf(typeof(Types)))
27                     {
28                         AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<Type>>(nameof(Types<object>.Collection)));
29                     }
30                     else
31                     {
32                         container.Add(element);
33                     }
34                 }
35             }
36         }
37     }
38 }

```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Collections.ObjectModel;
4 using Platform.Collections.Lists;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2> : Types
11     {
12         public static ReadOnlyCollection<Type> Collection { get; } = new Types<T1,
13             ↪ T2>().ToReadOnlyCollection();
14         public static Type[] Array => ((IList<Type>)Collection).ToArray();
15         private Types() { }
16     }
17 }
```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Collections.ObjectModel;
4 using Platform.Collections.Lists;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2, T3> : Types
11     {
12         public static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2,
13             ↪ T3>().ToReadOnlyCollection();
14         public static Type[] Array => ((IList<Type>)Collection).ToArray();
15         private Types() { }
16     }
17 }
```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Collections.ObjectModel;
4 using Platform.Collections.Lists;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2, T3, T4> : Types
11     {
12         public static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
13             ↪ T4>().ToReadOnlyCollection();
14         public static Type[] Array => ((IList<Type>)Collection).ToArray();
15         private Types() { }
16     }
17 }
```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Collections.ObjectModel;
4 using Platform.Collections.Lists;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2, T3, T4, T5> : Types
11     {
12         public static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4,
13             ↪ T5>().ToReadOnlyCollection();
14         public static Type[] Array => ((IList<Type>)Collection).ToArray();
15         private Types() { }
16     }
17 }
```



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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Collections.ObjectModel;
4 using Platform.Collections.Lists;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2, T3, T4, T5, T6> : Types
11     {
12         public static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4,
13             ↪ T5, T6>().ToReadOnlyCollection();
14         public static Type[] Array => ((IList<Type>)Collection).ToArray();
15         private Types() { }
16     }
17 }
```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Collections.ObjectModel;
4 using Platform.Collections.Lists;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Reflection
9 {
10     public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
11     {
12         public static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4,
13             ↪ T5, T6, T7>().ToReadOnlyCollection();
14         public static Type[] Array => ((IList<Type>)Collection).ToArray();
15         private Types() { }
16     }
17 }
```

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

```
1 using System;
2 using Platform.Collections.Lists;
3 using System.Collections.Generic;
4 using System.Collections.ObjectModel;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Reflection
9 {
10     public class Types<T> : Types
11     {
12         public static ReadOnlyCollection<Type> Collection { get; } = new
13             ↪ Types<T>().ToReadOnlyCollection();
14         public static Type[] Array => ((IList<Type>)Collection).ToArray();
15         private Types() { }
16     }
17 }
```

./Platform.Reflection.Tests/GetILBytesMethodTests.cs

```
1 using System;
2 using System.Reflection;
3 using Xunit;
4 using Platform.Collections;
5 using Platform.Collections.Lists;
6
7 namespace Platform.Reflection.Tests
8 {
9     public static class GetILBytesMethodTests
10     {
11         [Fact]
12         public static void ILBytesForDelegateAreAvailableTest()
13         {
14             var function = new Func<object, int>(argument => 0);
15             var bytes = function.GetMethodInfo().GetILBytes();
16             Assert.False(bytes.IsNullOrEmpty());
17         }
18
19         [Fact]
20         public static void ILBytesForDifferentDelegatesAreTheSameTest()
21         {
22         }
```

```

22     var firstFunction = new Func<object, int>(argument => 0);
23     var secondFunction = new Func<object, int>(argument => 0);
24     Assert.False(firstFunction == secondFunction);
25     var firstFunctionBytes = firstFunction.GetMethodInfo().GetILBytes();
26     Assert.False(firstFunctionBytes.IsNullOrEmpty());
27     var secondFunctionBytes = secondFunction.GetMethodInfo().GetILBytes();
28     Assert.False(secondFunctionBytes.IsNullOrEmpty());
29     Assert.True(firstFunctionBytes.EqualTo(secondFunctionBytes));
30 }
31 }
32 }

```

./Platform.Reflection.Tests/NumericTypeTests.cs

```

1  using Xunit;
2
3  namespace Platform.Reflection.Tests
4  {
5      public class NumericTypeTests
6      {
7          [Fact]
8          public void UInt64IsNumericTest()
9          {
10             Assert.True(NumericType<ulong>.IsNumeric);
11          }
12      }
13 }

```

## Index

- ./Platform.Reflection.Tests/GetILBytesMethodTests.cs, 9
- ./Platform.Reflection.Tests/NumericTypeTests.cs, 10
- ./Platform.Reflection/AssemblyExtensions.cs, 1
- ./Platform.Reflection/DynamicExtensions.cs, 1
- ./Platform.Reflection/EnsureExtensions.cs, 1
- ./Platform.Reflection/FieldInfoExtensions.cs, 4
- ./Platform.Reflection/MethodInfoExtensions.cs, 4
- ./Platform.Reflection/NumericType.cs, 4
- ./Platform.Reflection/PropertyInfoExtensions.cs, 5
- ./Platform.Reflection/TypeExtensions.cs, 5
- ./Platform.Reflection/Types.cs, 7
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 9
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 8
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 8
- ./Platform.Reflection/Types[T1, T2, T3, T4].cs, 8
- ./Platform.Reflection/Types[T1, T2, T3].cs, 8
- ./Platform.Reflection/Types[T1, T2].cs, 7
- ./Platform.Reflection/Types[T].cs, 9