

LinksPlatform's Platform.Reflection Class Library

./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 }
```

./CachedTypeInfo.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 class CachedTypeInfo<T>
13     {
14         public static readonly bool IsSupported;
15         public static readonly Type Type;
16         public static readonly Type UnderlyingType;
17         public static readonly Type SignedVersion;
18         public static readonly Type UnsignedVersion;
19         public static readonly bool IsFloatPoint;
20         public static readonly bool IsNumeric;
21         public static readonly bool IsSigned;
22         public static readonly bool CanBeNumeric;
23         public static readonly bool IsNullable;
24         public static readonly int BitsLength;
25         public static readonly T MinValue;
26         public static readonly T MaxValue;
27
28         static CachedTypeInfo()
29         {
30             try
31             {
32                 Type = typeof(T);
33                 IsNullable = Type.IsNullable();
34                 UnderlyingType = IsNullable ? Nullable.GetUnderlyingType(Type) : Type;
35                 var canBeNumeric = UnderlyingType.CanBeNumeric();
36                 var isNumeric = UnderlyingType.IsNumeric();
37                 var isSigned = UnderlyingType.IsSigned();
38                 var isFloatPoint = UnderlyingType.IsFloatPoint();
39                 var bitsLength = Marshal.SizeOf(UnderlyingType) * 8;
40                 GetMinAndMaxValues(UnderlyingType, out T minValue, out T maxValue);
41             }
42             catch { }
43         }
44     }
45 }
```

```

41         GetSignedAndUnsignedVersions(UnderlyingType, isSigned, out Type signedVersion,
42             ↪ out Type unsignedVersion);
43         IsSupported = true;
44         CanBeNumeric = canBeNumeric;
45         IsNumeric = isNumeric;
46         IsSigned = isSigned;
47         IsFloatPoint = isFloatPoint;
48         BitsLength = bitsLength;
49         MinValue = minValue;
50         MaxValue = maxValue;
51         SignedVersion = signedVersion;
52         UnsignedVersion = unsignedVersion;
53     }
54     catch (Exception exception)
55     {
56         exception.Ignore();
57     }
58 }
59 private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
60 {
61     if (type == typeof(bool))
62     {
63         minValue = (T)(object>false;
64         maxValue = (T)(object>true;
65     }
66     else
67     {
68         minValue = type.GetStaticFieldValue<T>("MinValue");
69         maxValue = type.GetStaticFieldValue<T>("MaxValue");
70     }
71 }
72 private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
73     ↪ signedVersion, out Type unsignedVersion)
74 {
75     if (isSigned)
76     {
77         signedVersion = type;
78         unsignedVersion = type.GetUnsignedVersionOrNull();
79     }
80     else
81     {
82         signedVersion = type.GetSignedVersionOrNull();
83         unsignedVersion = type;
84     }
85 }
86 }
87 }

```

./DynamicExtensions.cs

```

1  using System.Collections.Generic;
2  using System.Dynamic;
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 DynamicExtensions
9      {
10         public static bool HasProperty(this object @object, string propertyName)
11         {
12             var type = @object.GetType();
13             if (type is IDictionary<string, object> dictionary)
14             {
15                 return dictionary.ContainsKey(propertyName);
16             }
17             return type.GetProperty(propertyName) != null;
18         }
19     }
20 }

```

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

```

```

75 {
76     if (!CachedTypeInfo<T>.IsNumeric)
77     {
78         throw new NotSupportedException(messageBuilder());
79     }
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) =>
91     => IsNumeric<T>(root, (string)null);
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
95     => messageBuilder)
96 {
97     if (!CachedTypeInfo<T>.CanBeNumeric)
98     {
99         throw new NotSupportedException(messageBuilder());
100     }
101 }
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
105 {
106     string messageBuilder() => message;
107     CanBeNumeric<T>(root, messageBuilder());
108 }
109
110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
111 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
112     => CanBeNumeric<T>(root, (string)null);
113
114 #endregion
115
116 #region OnDebug
117
118 [Conditional("DEBUG")]
119 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
120     => Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
121
122 [Conditional("DEBUG")]
123 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
124     => message) => Ensure.Always.IsUnsignedInteger<T>(message);
125
126 [Conditional("DEBUG")]
127 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
128     => Ensure.Always.IsUnsignedInteger<T>();
129
130 [Conditional("DEBUG")]
131 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
132     => messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
133
134 [Conditional("DEBUG")]
135 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
136     => message) => Ensure.Always.IsSignedInteger<T>(message);
137
138 [Conditional("DEBUG")]
139 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
140     => Ensure.Always.IsSignedInteger<T>();
141
142 [Conditional("DEBUG")]
143 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
144     => messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
145
146 [Conditional("DEBUG")]
147 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
148     => Ensure.Always.IsSigned<T>(message);
149
150 [Conditional("DEBUG")]
151 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
152     => Ensure.Always.IsSigned<T>();

```

```

141     [Conditional("DEBUG")]
142     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
143         ↪ messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
144
145     [Conditional("DEBUG")]
146     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
147         ↪ Ensure.Always.IsNumeric<T>(message);
148
149     [Conditional("DEBUG")]
150     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
151         ↪ Ensure.Always.IsNumeric<T>();
152
153     [Conditional("DEBUG")]
154     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
155         ↪ messageBuilder) => Ensure.Always.CanBeNumeric<T>(messageBuilder);
156
157     [Conditional("DEBUG")]
158     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
159         ↪ => Ensure.Always.CanBeNumeric<T>(message);
160
161     [Conditional("DEBUG")]
162     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
163         ↪ Ensure.Always.CanBeNumeric<T>();
164
165     #endregion
166 }

```

./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 }

```

./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 }

```

./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) =>
12             ↪ (T)fieldInfo.GetValue(null);
13     }
14 }

```

./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.Static).GetStaticValue<T>();
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public static T GetStaticPropertyValue<T>(this Type type, string name) =>
62             ↳ type.GetTypeInfo().GetProperty(name, BindingFlags.Static).GetStaticValue<T>();
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
66             ↳ genericParameterTypes, Type[] argumentTypes)
67         {
68             var methods = from m in type.GetMethods()
69                             where m.Name == name
70                                 && m.IsGenericMethodDefinition
71                                 let typeParams = m.GetGenericArguments()
72                                 let normalParams = m.GetParameters().Select(x => x.ParameterType)
73                                 where typeParams.SequenceEqual(genericParameterTypes)
74                                 && normalParams.SequenceEqual(argumentTypes)
75                                 select m;
76             var method = methods.Single();
77             return method;
78         }
79     }
80 }
```

```

72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     public static Type GetBaseType(this Type type) => type.GetTypeInfo().BaseType;
74
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
107     public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
108 }

```

./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 }
```

```
./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 }
```

```
./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 }
```

```
./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 }
```

```
./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 }
```



```
./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 }
```

```
./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 }
```

```
./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 }
```

Index

./AssemblyExtensions.cs, 1
./CachedTypeInfo.cs, 1
./DynamicExtensions.cs, 2
./EnsureExtensions.cs, 2
./FieldInfoExtensions.cs, 5
./MethodInfoExtensions.cs, 5
./PropertyInfoExtensions.cs, 5
./TypeExtensions.cs, 5
./Types.cs, 7
./Types[T1, T2, T3, T4, T5, T6, T7].cs, 9
./Types[T1, T2, T3, T4, T5, T6].cs, 8
./Types[T1, T2, T3, T4, T5].cs, 8
./Types[T1, T2, T3, T4].cs, 8
./Types[T1, T2, T3].cs, 8
./Types[T1, T2].cs, 8
./Types[T].cs, 9