

## 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  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 }
```

### ./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)
```

```

18 {
19     if (!Type<T>.IsNumeric || Type<T>.IsSigned || Type<T>.IsFloatPoint)
20     {
21         throw new NotSupportedException(messageBuilder());
22     }
23 }
24
25 [MethodImpl(MethodImplOptions.AggressiveInlining)]
26 public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
    ↳ message)
27 {
28     string messageBuilder() => message;
29     IsUnsignedInteger<T>(root, messageBuilder());
30 }
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
    ↳ IsUnsignedInteger<T>(root, (string)null);
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
    ↳ messageBuilder)
37 {
38     if (!Type<T>.IsNumeric || !Type<T>.IsSigned || Type<T>.IsFloatPoint)
39     {
40         throw new NotSupportedException(messageBuilder());
41     }
42 }
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
    ↳ message)
46 {
47     string messageBuilder() => message;
48     IsSignedInteger<T>(root, messageBuilder());
49 }
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
    ↳ IsSignedInteger<T>(root, (string)null);
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
    ↳ messageBuilder)
56 {
57     if (!Type<T>.IsSigned)
58     {
59         throw new NotSupportedException(messageBuilder());
60     }
61 }
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
65 {
66     string messageBuilder() => message;
67     IsSigned<T>(root, messageBuilder());
68 }
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
    ↳ (string)null);
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
    ↳ messageBuilder)
75 {
76     if (!Type<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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
90     ↳ IsNumeric<T>(root, (string)null);
91
92 [MethodImpl(MethodImplOptions.AggressiveInlining)]
93 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
94     ↳ messageBuilder)
95 {
96     if (!Type<T>.CanBeNumeric)
97     {
98         throw new NotSupportedException(messageBuilder());
99     }
100 }
101
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
104 {
105     string messageBuilder() => message;
106     CanBeNumeric<T>(root, messageBuilder);
107 }
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
111     ↳ CanBeNumeric<T>(root, (string)null);
112
113 #endregion
114
115 #region OnDebug
116
117 [Conditional("DEBUG")]
118 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
119     ↳ Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
120
121 [Conditional("DEBUG")]
122 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
123     ↳ message) => Ensure.Always.IsUnsignedInteger<T>(message);
124
125 [Conditional("DEBUG")]
126 public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
127     ↳ Ensure.Always.IsUnsignedInteger<T>();
128
129 [Conditional("DEBUG")]
130 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
131     ↳ messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
132
133 [Conditional("DEBUG")]
134 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
135     ↳ message) => Ensure.Always.IsSignedInteger<T>(message);
136
137 [Conditional("DEBUG")]
138 public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
139     ↳ Ensure.Always.IsSignedInteger<T>();
140
141 [Conditional("DEBUG")]
142 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
143     ↳ messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
144
145 [Conditional("DEBUG")]
146 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
147     ↳ Ensure.Always.IsSigned<T>(message);
148
149 [Conditional("DEBUG")]
150 public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
151     ↳ Ensure.Always.IsSigned<T>();
152
153 [Conditional("DEBUG")]
154 public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
155     ↳ messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
156
157 [Conditional("DEBUG")]
158 public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
159     ↳ Ensure.Always.IsNumeric<T>(message);
160
161 [Conditional("DEBUG")]
162 public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
163     ↳ Ensure.Always.IsNumeric<T>();

```

```

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

```

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

```

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

```

#### ./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) =>
            ↳ methodInfo.GetMethodBody().GetILAsByteArray();
10     }
11 }

```

#### ./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/Type.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 Type<T>
13     {
14         public static readonly bool IsSupported;
15         public static readonly Type TheType;
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 Type()
29     {
30         try
31         {
32             TheType = typeof(T);
33             IsNullable = TheType.IsNullable();
34             UnderlyingType = IsNullable ? Nullable.GetUnderlyingType(TheType) : TheType;
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             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
60     private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
61     {
62         if (type == typeof(bool))
63         {
64             minValue = (T)(object>false;
65             maxValue = (T)(object>true;
66         }
67         else
68         {
69             minValue = type.GetStaticFieldValue<T>("MinValue");
70             maxValue = type.GetStaticFieldValue<T>("MaxValue");
71         }
72     }
73
74     private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
75     ↪ signedVersion, out Type unsignedVersion)
76     {
77         if (isSigned)
78         {
79             signedVersion = type;
80             unsignedVersion = type.GetUnsignedVersionOrNull();
81         }
82         else
83         {
84             signedVersion = type.GetSignedVersionOrNull();
85             unsignedVersion = type;
86         }
87     }
88 }

```

./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     }
86 }

```

```

76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 public static Assembly GetAssembly(this Type type) => type.GetTypeInfo().Assembly;
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 public static bool IsSubclassOf(this Type type, Type superClass) =>
81     ↳ type.GetTypeInfo().IsSubclassOf(superClass);
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 public static bool IsValueType(this Type type) => type.GetTypeInfo().IsValueType;
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 public static bool IsGeneric(this Type type) => type.GetTypeInfo().IsGenericType;
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
91     ↳ type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
95
96 public static Type GetUnsignedVersionOrNull(this Type signedType) =>
97     ↳ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
98
99 public static Type GetSignedVersionOrNull(this Type unsignedType) =>
100     ↳ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
101
102 public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
103
104 public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
105
106 public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
107
108 public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
109 }
110 }

```

## ./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/TypeTests.cs

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

## Index

- ./Platform.Reflection.Tests/TypeTests.cs, 9
- ./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/PropertyInfoExtensions.cs, 4
- ./Platform.Reflection/Type.cs, 4
- ./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