LinksPlatform's Platform.Reflection Class Library

./Platform.Reflection/AssemblyExtensions.cs

```csharp
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

```csharp
using System.Collections.Generic;
using System.Dynamic;

#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

```csharp
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 (!Type<T>.IsNumeric || Type<T>.IsSigned || Type<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 (!Type<T>.IsNumeric || !Type<T>.IsSigned || Type<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 (!Type<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 (!Type<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);
        }
```

```csharp
        [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 (!Type<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>();
```

```
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.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   }
```

```csharp
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,
                        out Type unsignedVersion);
42                  IsSupported = true;
43                  CanBeNumeric = canBeNumeric;
44                  IsNumeric = isNumeric;
45                  IsSigned = isSigned;
46                  IsFloatPoint = isFloatPoint;
47                  BitsLength = bitsLength;
48                  MinValue = minValue;
49                  MaxValue = maxValue;
50                  SignedVersion = signedVersion;
51                  UnsignedVersion = unsignedVersion;
52              }
53              catch (Exception exception)
54              {
55                  exception.Ignore();
56              }
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
73          private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
                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 }
```

./Platform.Reflection/TypeExtensions.cs

```csharp
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),
             ↪  typeof(DateTime), typeof(TimeSpan) };
24             _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
             ↪  typeof(ulong) };
25             _canBeNumericTypes.UnionWith(_isNumericTypes);
26             _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
             ↪  typeof(long) };
27             _canBeNumericTypes.UnionWith(_isSignedTypes);
28             _isNumericTypes.UnionWith(_isSignedTypes);
29             _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
             ↪  typeof(float) };
30             _canBeNumericTypes.UnionWith(_isFloatPointTypes);
31             _isNumericTypes.UnionWith(_isFloatPointTypes);
32             _isSignedTypes.UnionWith(_isFloatPointTypes);
33             _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
34             {
35                 { typeof(sbyte), typeof(byte) },
36                 { typeof(short), typeof(ushort) },
37                 { typeof(int), typeof(uint) },
38                 { typeof(long), typeof(ulong) },
39             };
40             _signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
41             {
42                 { typeof(byte), typeof(sbyte)},
43                 { typeof(ushort), typeof(short) },
44                 { typeof(uint), typeof(int) },
45                 { typeof(ulong), typeof(long) },
46             };
47         }
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public static T GetStaticFieldValue<T>(this Type type, string name) =>
             ↪  type.GetTypeInfo().GetField(name, BindingFlags.Public | BindingFlags.NonPublic |
             ↪  BindingFlags.Static).GetStaticValue<T>();
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public static T GetStaticPropertyValue<T>(this Type type, string name) =>
             ↪  type.GetTypeInfo().GetProperty(name, BindingFlags.Public | BindingFlags.NonPublic |
             ↪  BindingFlags.Static).GetStaticValue<T>();
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        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

```csharp
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))
```

```
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,
   ↪  T2>().ToReadOnlyCollection();
13         public static Type[] Array => ((IList<Type>)Collection).ToArray();
14         private Types() { }
15     }
16 }
```

./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,
   ↪  T3>().ToReadOnlyCollection();
13         public static Type[] Array => ((IList<Type>)Collection).ToArray();
14         private Types() { }
15     }
16 }
```

./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,
   ↪  T4>().ToReadOnlyCollection();
13         public static Type[] Array => ((IList<Type>)Collection).ToArray();
14         private Types() { }
15     }
16 }
```

./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
```

```csharp
#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

```csharp
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

```csharp
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

```csharp
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() { }
    }
}
```

Index