./AssemblyExtensions.cs

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

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

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

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

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

./CachedTypeInfo.cs

```
using System;
using System.Runtime.InteropServices;
using Platform.Exceptions;

// ReSharper disable AssignmentInConditionalExpression
// ReSharper disable BuiltInTypeReferenceStyle
// ReSharper disable StaticFieldInGenericType
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public class CachedTypeInfo<T>
    {
        public static readonly bool IsSupported;
        public static readonly Type Type;
        public static readonly Type UnderlyingType;
        public static readonly Type SignedVersion;
        public static readonly Type UnsignedVersion;
        public static readonly bool IsFloatPoint;
        public static readonly bool IsNumeric;
        public static readonly bool IsSigned;
        public static readonly bool CanBeNumeric;
        public static readonly bool IsNullable;
        public static readonly int BitsLength;
        public static readonly T MinValue;
        public static readonly T MaxValue;

        static CachedTypeInfo()
        {
            try
            {
                Type = typeof(T);
                IsNullable = Type.IsNullable();
                UnderlyingType = IsNullable ? Nullable.GetUnderlyingType(Type) : Type;
                var canBeNumeric = UnderlyingType.CanBeNumeric();
                var isNumeric = UnderlyingType.IsNumeric();
                var isSigned = UnderlyingType.IsSigned();
                var isFloatPoint = UnderlyingType.IsFloatPoint();
                var bitsLength = Marshal.SizeOf(UnderlyingType) * 8;
                GetMinAndMaxValues(UnderlyingType, out T minValue, out T maxValue);
```

```
                GetSignedAndUnsignedVersions(UnderlyingType, isSigned, out Type signedVersion,
                ↪  out Type unsignedVersion);
                IsSupported = true;
                CanBeNumeric = canBeNumeric;
                IsNumeric = isNumeric;
                IsSigned = isSigned;
                IsFloatPoint = isFloatPoint;
                BitsLength = bitsLength;
                MinValue = minValue;
                MaxValue = maxValue;
                SignedVersion = signedVersion;
                UnsignedVersion = unsignedVersion;
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
        }

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

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

./DynamicExtensions.cs

```
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;
        }
    }
}
```

./EnsureExtensions.cs

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

#pragma warning disable IDE0060 // Remove unused parameter
```

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

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

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

```
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) =>
            ↪  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 (!CachedTypeInfo<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>();
```

```

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

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

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

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

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

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

        #endregion
    }
}
```

./FieldInfoExtensions.cs

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

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

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

./MethodInfoExtensions.cs

```
using System.Reflection;

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

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

./PropertyInfoExtensions.cs

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

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

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

./TypeExtensions.cs

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

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

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

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

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

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

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

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

```
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public static Type GetBaseType(this Type type) => type.GetTypeInfo().BaseType;
75
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) =>
            ↪  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) =>
            ↪  type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
90
91          [MethodImpl(MethodImplOptions.AggressiveInlining)]
92          public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
93
94          public static Type GetUnsignedVersionOrNull(this Type signedType) =>
            ↪  _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
95
96          public static Type GetSignedVersionOrNull(this Type unsignedType) =>
            ↪  _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
97
98          public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
99
100         public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
101
102         public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
103
104         public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
105     }
106 }
```

./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     }
```

```
}
```

./Types[T1, T2].cs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

./Types[T].cs

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

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

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

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