LinksPlatform's Platform.Reflection Class Library

./Platform.Reflection/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);
    }
}
```

./Platform.Reflection/DelegateHelpers.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection.Emit;
using Platform.Exceptions;

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

namespace Platform.Reflection
{
    public static class DelegateHelpers
    {
        public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode)
            where TDelegate : Delegate
        {
            var @delegate = default(TDelegate);
            try
            {
                var delegateType = typeof(TDelegate);
                var invoke = delegateType.GetMethod("Invoke");
                var returnType = invoke.ReturnType;
                var parameterTypes = invoke.GetParameters().Select(s =>
                ↪  s.ParameterType).ToArray();
                var dynamicMethod = new DynamicMethod(Guid.NewGuid().ToString(), returnType,
                ↪  parameterTypes);
                var generator = dynamicMethod.GetILGenerator();
                emitCode(generator);
                @delegate = (TDelegate)dynamicMethod.CreateDelegate(delegateType);
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
            return @delegate;
        }

        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode)
            where TDelegate : Delegate
        {
            var @delegate = CompileOrDefault<TDelegate>(emitCode);
```

```
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
                @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
            }
            return @delegate;
        }
    }
}
```

./Platform.Reflection/DynamicExtensions.cs

```
using System.Collections.Generic;

#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

```
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 (!NumericType<T>.IsNumeric || NumericType<T>.IsSigned ||
            ↪  NumericType<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 (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
            ↪  NumericType<T>.IsFloatPoint)
            {
                throw new NotSupportedException(messageBuilder());
            }
```

```
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 (!NumericType<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 (!NumericType<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 (!NumericType<T>.CanBeNumeric)
96              {
97                  throw new NotSupportedException(messageBuilder());
98              }
99          }
100
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
103         {
104             string messageBuilder() => message;
105             CanBeNumeric<T>(root, messageBuilder);
106         }
107
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪  CanBeNumeric<T>(root, (string)null);
110
111         #endregion
```

```
112
113         #region OnDebug
114
115         [Conditional("DEBUG")]
116         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
            ↪  Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
117
118         [Conditional("DEBUG")]
119         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
            ↪  message) => Ensure.Always.IsUnsignedInteger<T>(message);
120
121         [Conditional("DEBUG")]
122         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsUnsignedInteger<T>();
123
124         [Conditional("DEBUG")]
125         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
126
127         [Conditional("DEBUG")]
128         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
            ↪  message) => Ensure.Always.IsSignedInteger<T>(message);
129
130         [Conditional("DEBUG")]
131         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsSignedInteger<T>();
132
133         [Conditional("DEBUG")]
134         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
135
136         [Conditional("DEBUG")]
137         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
            ↪  Ensure.Always.IsSigned<T>(message);
138
139         [Conditional("DEBUG")]
140         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsSigned<T>();
141
142         [Conditional("DEBUG")]
143         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
144
145         [Conditional("DEBUG")]
146         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
            ↪  Ensure.Always.IsNumeric<T>(message);
147
148         [Conditional("DEBUG")]
149         public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsNumeric<T>();
150
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)]
```

```
        public static T GetStaticValue<T>(this FieldInfo fieldInfo) =>
        ↪  (T)fieldInfo.GetValue(null);
    }
}
```

./Platform.Reflection/ILGeneratorExtensions.cs

```
using System;
using System.Linq;
using System.Reflection;
using System.Reflection.Emit;

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

namespace Platform.Reflection
{
    public static class ILGeneratorExtensions
    {
        public static void Throw<T>(this ILGenerator generator) =>
        ↪  generator.ThrowException(typeof(T));

        public static void ConvertTo<T>(this ILGenerator generator)
        {
            var type = typeof(T);
            if (type == typeof(short))
            {
                generator.Emit(OpCodes.Conv_I2);
            }
            else if (type == typeof(ushort))
            {
                generator.Emit(OpCodes.Conv_U2);
            }
            else if (type == typeof(sbyte))
            {
                generator.Emit(OpCodes.Conv_I1);
            }
            else if (type == typeof(byte))
            {
                generator.Emit(OpCodes.Conv_U1);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        public static void LoadConstant(this ILGenerator generator, bool value) =>
        ↪  generator.LoadConstant(value ? 1 : 0);

        public static void LoadConstant(this ILGenerator generator, float value) =>
        ↪  generator.Emit(OpCodes.Ldc_R4, value);

        public static void LoadConstant(this ILGenerator generator, double value) =>
        ↪  generator.Emit(OpCodes.Ldc_R8, value);

        public static void LoadConstant(this ILGenerator generator, ulong value) =>
        ↪  generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));

        public static void LoadConstant(this ILGenerator generator, long value) =>
        ↪  generator.Emit(OpCodes.Ldc_I8, value);

        public static void LoadConstant(this ILGenerator generator, uint value)
        {
            switch (value)
            {
                case uint.MaxValue: generator.Emit(OpCodes.Ldc_I4_M1, value); return;
                case 0: generator.Emit(OpCodes.Ldc_I4_0, value); return;
                case 1: generator.Emit(OpCodes.Ldc_I4_1, value); return;
                case 2: generator.Emit(OpCodes.Ldc_I4_2, value); return;
                case 3: generator.Emit(OpCodes.Ldc_I4_3, value); return;
                case 4: generator.Emit(OpCodes.Ldc_I4_4, value); return;
                case 5: generator.Emit(OpCodes.Ldc_I4_5, value); return;
                case 6: generator.Emit(OpCodes.Ldc_I4_6, value); return;
                case 7: generator.Emit(OpCodes.Ldc_I4_7, value); return;
                case 8: generator.Emit(OpCodes.Ldc_I4_8, value); return;
            }
            if (value <= sbyte.MaxValue)
            {
                generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
```

```
            }
            else
            {
                generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
            }
        }

        public static void LoadConstant(this ILGenerator generator, int value)
        {
            switch (value)
            {
                case -1: generator.Emit(OpCodes.Ldc_I4_M1, value); return;
                case 0: generator.Emit(OpCodes.Ldc_I4_0, value); return;
                case 1: generator.Emit(OpCodes.Ldc_I4_1, value); return;
                case 2: generator.Emit(OpCodes.Ldc_I4_2, value); return;
                case 3: generator.Emit(OpCodes.Ldc_I4_3, value); return;
                case 4: generator.Emit(OpCodes.Ldc_I4_4, value); return;
                case 5: generator.Emit(OpCodes.Ldc_I4_5, value); return;
                case 6: generator.Emit(OpCodes.Ldc_I4_6, value); return;
                case 7: generator.Emit(OpCodes.Ldc_I4_7, value); return;
                case 8: generator.Emit(OpCodes.Ldc_I4_8, value); return;
            }
            if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
            {
                generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
            }
            else
            {
                generator.Emit(OpCodes.Ldc_I4, value);
            }
        }

        public static void LoadConstant(this ILGenerator generator, short value)
        {
            generator.LoadConstant((int)value);
            generator.ConvertTo<short>();
        }

        public static void LoadConstant(this ILGenerator generator, ushort value)
        {
            generator.LoadConstant((int)value);
            generator.ConvertTo<ushort>();
        }

        public static void LoadConstant(this ILGenerator generator, sbyte value)
        {
            generator.LoadConstant((int)value);
            generator.ConvertTo<sbyte>();
        }

        public static void LoadConstant(this ILGenerator generator, byte value)
        {
            generator.LoadConstant((int)value);
            generator.ConvertTo<byte>();
        }

        public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
        ↪  LoadConstantOne(generator, typeof(TConstant));

        public static void LoadConstantOne(this ILGenerator generator, Type constantType)
        {
            if (constantType == typeof(float))
            {
                generator.LoadConstant(1F);
            }
            else if (constantType == typeof(double))
            {
                generator.LoadConstant(1D);
            }
            else if (constantType == typeof(long))
            {
                generator.LoadConstant(1L);
            }
            else if (constantType == typeof(ulong))
            {
                generator.LoadConstant(1UL);
            }
            else if (constantType == typeof(int))
```

```
144             {
145                 generator.LoadConstant(1);
146             }
147             else if (constantType == typeof(uint))
148             {
149                 generator.LoadConstant(1U);
150             }
151             else if (constantType == typeof(short))
152             {
153                 generator.LoadConstant((short)1);
154             }
155             else if (constantType == typeof(ushort))
156             {
157                 generator.LoadConstant((ushort)1);
158             }
159             else if (constantType == typeof(sbyte))
160             {
161                 generator.LoadConstant((sbyte)1);
162             }
163             else if (constantType == typeof(byte))
164             {
165                 generator.LoadConstant((byte)1);
166             }
167             else
168             {
169                 throw new NotSupportedException();
170             }
171         }
172 
173         public static void LoadConstant<TConstant>(this ILGenerator generator, object
            ↪  constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
174 
175         public static void LoadConstant(this ILGenerator generator, Type constantType, object
            ↪  constantValue)
176         {
177             if (constantType == typeof(float))
178             {
179                 generator.LoadConstant((float)constantValue);
180             }
181             else if (constantType == typeof(double))
182             {
183                 generator.LoadConstant((double)constantValue);
184             }
185             else if (constantType == typeof(long))
186             {
187                 generator.LoadConstant((long)constantValue);
188             }
189             else if (constantType == typeof(ulong))
190             {
191                 generator.LoadConstant((ulong)constantValue);
192             }
193             else if (constantType == typeof(int))
194             {
195                 generator.LoadConstant((int)constantValue);
196             }
197             else if (constantType == typeof(uint))
198             {
199                 generator.LoadConstant((uint)constantValue);
200             }
201             else if (constantType == typeof(short))
202             {
203                 generator.LoadConstant((short)constantValue);
204             }
205             else if (constantType == typeof(ushort))
206             {
207                 generator.LoadConstant((ushort)constantValue);
208             }
209             else if (constantType == typeof(sbyte))
210             {
211                 generator.LoadConstant((sbyte)constantValue);
212             }
213             else if (constantType == typeof(byte))
214             {
215                 generator.LoadConstant((byte)constantValue);
216             }
217             else
218             {
219                 throw new NotSupportedException();
```

```
220             }
221         }
222 
223         public static void Increment<TValue>(this ILGenerator generator) =>
            ↪ generator.Increment(typeof(TValue));
224 
225         public static void Decrement<TValue>(this ILGenerator generator) =>
            ↪ generator.Decrement(typeof(TValue));
226 
227         public static void Increment(this ILGenerator generator, Type valueType)
228         {
229             generator.LoadConstantOne(valueType);
230             generator.Add();
231         }
232 
233         public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
234 
235         public static void Decrement(this ILGenerator generator, Type valueType)
236         {
237             generator.LoadConstantOne(valueType);
238             generator.Subtract();
239         }
240 
241         public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
242 
243         public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
244 
245         public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
246 
247         public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
248 
249         public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
250 
251         public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
252 
253         public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
254 
255         public static void LoadArgument(this ILGenerator generator, int argumentIndex)
256         {
257             if (argumentIndex == 0)
258             {
259                 generator.Emit(OpCodes.Ldarg_0);
260             }
261             else if (argumentIndex == 1)
262             {
263                 generator.Emit(OpCodes.Ldarg_1);
264             }
265             else if (argumentIndex == 2)
266             {
267                 generator.Emit(OpCodes.Ldarg_2);
268             }
269             else if (argumentIndex == 3)
270             {
271                 generator.Emit(OpCodes.Ldarg_3);
272             }
273             else
274             {
275                 generator.Emit(OpCodes.Ldarg, argumentIndex);
276             }
277         }
278 
279         public static void LoadArguments(this ILGenerator generator, params int[]
            ↪ argumentIndices)
280         {
281             for (var i = 0; i < argumentIndices.Length; i++)
282             {
283                 generator.LoadArgument(argumentIndices[i]);
284             }
285         }
286 
287         public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
            ↪ generator.Emit(OpCodes.Starg, argumentIndex);
288 
289         public static void CompareGreaterThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Cgt);
290 
291         public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
            ↪ generator.Emit(OpCodes.Cgt_Un);
292 
```

```
293         public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
294         {
295             if (isSigned)
296             {
297                 generator.CompareGreaterThan();
298             }
299             else
300             {
301                 generator.UnsignedCompareGreaterThan();
302             }
303         }
304 
305         public static void CompareLessThan(this ILGenerator generator) =>
            ↪  generator.Emit(OpCodes.Clt);
306 
307         public static void UnsignedCompareLessThan(this ILGenerator generator) =>
            ↪  generator.Emit(OpCodes.Clt_Un);
308 
309         public static void CompareLessThan(this ILGenerator generator, bool isSigned)
310         {
311             if (isSigned)
312             {
313                 generator.CompareLessThan();
314             }
315             else
316             {
317                 generator.UnsignedCompareLessThan();
318             }
319         }
320 
321         public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
            ↪  generator.Emit(OpCodes.Bge, label);
322 
323         public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
            ↪  label) => generator.Emit(OpCodes.Bge_Un, label);
324 
325         public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
            ↪  Label label)
326         {
327             if (isSigned)
328             {
329                 generator.BranchIfGreaterOrEqual(label);
330             }
331             else
332             {
333                 generator.UnsignedBranchIfGreaterOrEqual(label);
334             }
335         }
336 
337         public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
            ↪  generator.Emit(OpCodes.Ble, label);
338 
339         public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
            ↪  => generator.Emit(OpCodes.Ble_Un, label);
340 
341         public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
            ↪  label)
342         {
343             if (isSigned)
344             {
345                 generator.BranchIfLessOrEqual(label);
346             }
347             else
348             {
349                 generator.UnsignedBranchIfLessOrEqual(label);
350             }
351         }
352 
353         public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
354 
355         public static void Box(this ILGenerator generator, Type boxedType) =>
            ↪  generator.Emit(OpCodes.Box, boxedType);
356 
357         public static void Call(this ILGenerator generator, MethodInfo method) =>
            ↪  generator.Emit(OpCodes.Call, method);
358 
359         public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
360 
```

```
361         public static void Unbox<TUnbox>(this ILGenerator generator) =>
            ↪ generator.Unbox(typeof(TUnbox));
362
363         public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
            ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
364
365         public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
            ↪ generator.UnboxValue(typeof(TUnbox));
366
367         public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
            ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
368
369         public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
            ↪ generator.DeclareLocal(typeof(T));
370
371         public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪ generator.Emit(OpCodes.Ldloc, local);
372
373         public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
            ↪ generator.Emit(OpCodes.Stloc, local);
374
375         public static void NewObject(this ILGenerator generator, Type type, params Type[]
            ↪ parameterTypes)
376         {
377             var allConstructors = type.GetConstructors(BindingFlags.Public |
                ↪ BindingFlags.NonPublic | BindingFlags.Instance
378 #if !NETSTANDARD
379                 | BindingFlags.CreateInstance
380 #endif
381                 );
382             var constructor = allConstructors.Where(c => c.GetParameters().Length ==
                ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
                ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
383             if (constructor == null)
384             {
385                 throw new InvalidOperationException("Type " + type + " must have a constructor
                    ↪ that matches parameters [" + string.Join(", ",
                    ↪ parameterTypes.AsEnumerable()) + "]");
386             }
387             generator.NewObject(constructor);
388         }
389
390         public static void NewObject(this ILGenerator generator, ConstructorInfo constructor)
391         {
392             generator.Emit(OpCodes.Newobj, constructor);
393         }
394
395         public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
396
397         public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
            ↪ false, byte? unaligned = null)
398         {
399             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
400             {
401                 throw new ArgumentException("unaligned must be null, 1, 2, or 4");
402             }
403             if (isVolatile)
404             {
405                 generator.Emit(OpCodes.Volatile);
406             }
407             if (unaligned.HasValue)
408             {
409                 generator.Emit(OpCodes.Unaligned, unaligned.Value);
410             }
411             if (type.IsPointer)
412             {
413                 generator.Emit(OpCodes.Ldind_I);
414             }
415             else if (!type.IsValueType)
416             {
417                 generator.Emit(OpCodes.Ldind_Ref);
418             }
419             else if (type == typeof(sbyte))
420             {
421                 generator.Emit(OpCodes.Ldind_I1);
422             }
```

```
423             else if (type == typeof(bool))
424             {
425                 generator.Emit(OpCodes.Ldind_I1);
426             }
427             else if (type == typeof(byte))
428             {
429                 generator.Emit(OpCodes.Ldind_U1);
430             }
431             else if (type == typeof(short))
432             {
433                 generator.Emit(OpCodes.Ldind_I2);
434             }
435             else if (type == typeof(ushort))
436             {
437                 generator.Emit(OpCodes.Ldind_U2);
438             }
439             else if (type == typeof(char))
440             {
441                 generator.Emit(OpCodes.Ldind_U2);
442             }
443             else if (type == typeof(int))
444             {
445                 generator.Emit(OpCodes.Ldind_I4);
446             }
447             else if (type == typeof(uint))
448             {
449                 generator.Emit(OpCodes.Ldind_U4);
450             }
451             else if (type == typeof(long) || type == typeof(ulong))
452             {
453                 generator.Emit(OpCodes.Ldind_I8);
454             }
455             else if (type == typeof(float))
456             {
457                 generator.Emit(OpCodes.Ldind_R4);
458             }
459             else if (type == typeof(double))
460             {
461                 generator.Emit(OpCodes.Ldind_R8);
462             }
463             else
464             {
465                 throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
                    ↪  ", LoadObject may be more appropriate");
466             }
467         }
468 
469         public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
            ↪  byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
470 
471         public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
            ↪  = false, byte? unaligned = null)
472         {
473             if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
474             {
475                 throw new ArgumentException("unaligned must be null, 1, 2, or 4");
476             }
477             if (isVolatile)
478             {
479                 generator.Emit(OpCodes.Volatile);
480             }
481             if (unaligned.HasValue)
482             {
483                 generator.Emit(OpCodes.Unaligned, unaligned.Value);
484             }
485             if (type.IsPointer)
486             {
487                 generator.Emit(OpCodes.Stind_I);
488             }
489             else if (!type.IsValueType)
490             {
491                 generator.Emit(OpCodes.Stind_Ref);
492             }
493             else if (type == typeof(sbyte) || type == typeof(byte))
494             {
495                 generator.Emit(OpCodes.Stind_I1);
496             }
497             else if (type == typeof(short) || type == typeof(ushort))
```

```
            {
                generator.Emit(OpCodes.Stind_I2);
            }
            else if (type == typeof(int) || type == typeof(uint))
            {
                generator.Emit(OpCodes.Stind_I4);
            }
            else if (type == typeof(long) || type == typeof(ulong))
            {
                generator.Emit(OpCodes.Stind_I8);
            }
            else if (type == typeof(float))
            {
                generator.Emit(OpCodes.Stind_R4);
            }
            else if (type == typeof(double))
            {
                generator.Emit(OpCodes.Stind_R8);
            }
            else
            {
                throw new InvalidOperationException("StoreIndirect cannot be used with " + type + ", StoreObject may be more appropriate");
            }
        }
    }
}
```

./Platform.Reflection/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();
    }
}
```

./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs

```
using System;
using System.Collections.Generic;
using Platform.Interfaces;

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

namespace Platform.Reflection
{
    public class NotSupportedExceptionDelegateFactory<TDelegate> : IFactory<TDelegate>
        where TDelegate : Delegate
    {
        public TDelegate Create()
        {
            var @delegate = DelegateHelpers.CompileOrDefault<TDelegate>(generator =>
            {
                generator.Throw<NotSupportedException>();
            });
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
                throw new InvalidOperationException("Unable to compile stub delegate.");
            }
            return @delegate;
        }
    }
}
```

./Platform.Reflection/NumericType.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
```

```
9
10  namespace Platform.Reflection
11  {
12      public static class NumericType<T>
13      {
14          public static readonly Type Type;
15          public static readonly Type UnderlyingType;
16          public static readonly Type SignedVersion;
17          public static readonly Type UnsignedVersion;
18          public static readonly bool IsFloatPoint;
19          public static readonly bool IsNumeric;
20          public static readonly bool IsSigned;
21          public static readonly bool CanBeNumeric;
22          public static readonly bool IsNullable;
23          public static readonly int BitsLength;
24          public static readonly T MinValue;
25          public static readonly T MaxValue;
26
27          static NumericType()
28          {
29              try
30              {
31                  var type = typeof(T);
32                  var isNullable = type.IsNullable();
33                  var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
34                  var canBeNumeric = underlyingType.CanBeNumeric();
35                  var isNumeric = underlyingType.IsNumeric();
36                  var isSigned = underlyingType.IsSigned();
37                  var isFloatPoint = underlyingType.IsFloatPoint();
38                  var bitsLength = Marshal.SizeOf(underlyingType) * 8;
39                  GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
40                  GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
                    ↪  out Type unsignedVersion);
41                  Type = type;
42                  IsNullable = isNullable;
43                  UnderlyingType = underlyingType;
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>(nameof(int.MinValue));
70                  maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
71              }
72          }
73
74          private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
            ↪  signedVersion, out Type unsignedVersion)
75          {
76              if (isSigned)
77              {
78                  signedVersion = type;
79                  unsignedVersion = type.GetUnsignedVersionOrNull();
80              }
81              else
82              {
83                  signedVersion = type.GetSignedVersionOrNull();
84                  unsignedVersion = type;
85              }
86          }
```

```
    }
}
```

./Platform.Reflection/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);
    }
}
```

./Platform.Reflection/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.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)]
59          public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
            ↪  genericParameterTypes, Type[] argumentTypes)
60          {
61              var methods = from m in type.GetMethods()
62                            where m.Name == name
63                               && m.IsGenericMethodDefinition
64                            let typeParams = m.GetGenericArguments()
65                            let normalParams = m.GetParameters().Select(x => x.ParameterType)
66                            where typeParams.SequenceEqual(genericParameterTypes)
67                               && normalParams.SequenceEqual(argumentTypes)
68                            select m;
69              var method = methods.Single();
70              return method;
71          }
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          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95          public static Type GetUnsignedVersionOrNull(this Type signedType) =>
            ↪  _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
96
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          public static Type GetSignedVersionOrNull(this Type unsignedType) =>
            ↪  _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
99
100         [MethodImpl(MethodImplOptions.AggressiveInlining)]
101         public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
102
103         [MethodImpl(MethodImplOptions.AggressiveInlining)]
104         public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
105
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
111     }
112 }
```

./Platform.Reflection/Types.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 abstract class Types
11     {
```

```
        public static ReadOnlyCollection<Type> Collection { get; } = new
        ↪  ReadOnlyCollection<Type>(new Type[0]);
        public static Type[] Array => Collection.ToArray();

        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))
                {
                    if (element.IsSubclassOf(typeof(Types)))
                    {
                        AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection
                        ↪  <Type>>(nameof(Types<object>.Collection)));
                    }
                    else
                    {
                        container.Add(element);
                    }
                }
            }
        }
    }
}
```

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

```
using System;
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 new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1,
        ↪  T2>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
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 new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2,
        ↪  T3>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
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 new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
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 new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4, T5>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
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 new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4, T5, T6>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

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

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

```
using System;
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<T> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
```

```
    }
}
```

./Platform.Reflection.Tests/CodeGenerationTests.cs

```
using System;
using Xunit;

namespace Platform.Reflection.Tests
{
    public static class CodeGenerationTests
    {
        [Fact]
        public static void EmptyActionCompilationTest()
        {
            Action compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                generator.Return();
            });
            compiledAction();
        }

        [Fact]
        public static void FailedActionCompilationTest()
        {
            Action compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                throw new NotImplementedException();
            });
            Assert.Throws<NotSupportedException>(compiledAction);
        }
    }
}
```

./Platform.Reflection.Tests/GetILBytesMethodTests.cs

```
using System;
using System.Reflection;
using Xunit;
using Platform.Collections;
using Platform.Collections.Lists;

namespace Platform.Reflection.Tests
{
    public static class GetILBytesMethodTests
    {
        [Fact]
        public static void ILBytesForDelegateAreAvailableTest()
        {
            var function = new Func<object, int>(argument => 0);
            var bytes = function.GetMethodInfo().GetILBytes();
            Assert.False(bytes.IsNullOrEmpty());
        }

        [Fact]
        public static void ILBytesForDifferentDelegatesAreTheSameTest()
        {
            var firstFunction = new Func<object, int>(argument => 0);
            var secondFunction = new Func<object, int>(argument => 0);
            Assert.False(firstFunction == secondFunction);
            var firstFunctionBytes = firstFunction.GetMethodInfo().GetILBytes();
            Assert.False(firstFunctionBytes.IsNullOrEmpty());
            var secondFunctionBytes = secondFunction.GetMethodInfo().GetILBytes();
            Assert.False(secondFunctionBytes.IsNullOrEmpty());
            Assert.True(firstFunctionBytes.EqualTo(secondFunctionBytes));
        }
    }
}
```

./Platform.Reflection.Tests/NumericTypeTests.cs

```
using Xunit;

namespace Platform.Reflection.Tests
{
    public class NumericTypeTests
    {
        [Fact]
        public void UInt64IsNumericTest()
        {
            Assert.True(NumericType<ulong>.IsNumeric);
        }
```

```
12          }
13  }
```

Index