MethodBase.Invoke Method

Reference

Definition

Namespace: System.Reflection

Assembly: System.Runtime.dll

Invokes the method or constructor reflected by this `MethodInfo` instance.

Overloads

Invoke(Object, Object[])	Invokes the method or constructor represented by the current instance, using the specified parameters.
Invoke(Object, BindingFlags, Binder, Object[], CultureInfo)	When overridden in a derived class, invokes the reflected method or constructor with the given parameters.

Invoke(Object, Object[])

Invokes the method or constructor represented by the current instance, using the specified parameters.

C#

```
public object? Invoke (object? obj, object?[]? parameters);
```

Parameters

obj Object

The object on which to invoke the method or constructor. If a method is static, this argument is ignored. If a constructor is static, this argument must be `null` or an instance of the class that defines the constructor.

parameters Object[]

An argument list for the invoked method or constructor. This is an array of objects with the same number, order, and type as the parameters of the method or constructor to be invoked. If there are no parameters, `parameters` should be `null`.

If the method or constructor represented by this instance takes a `ref` parameter (`ByRef` in Visual Basic), no special attribute is required for that parameter in order to invoke the method or constructor using this function. Any object in this array that is not explicitly initialized with a value will contain the default value for that object type. For reference-type elements, this value is `null`. For value-type elements, the default value is 0, 0.0, or `false`, depending on the specific element type.

Returns

Object

An object containing the return value of the invoked method, or `null` in the case of a constructor.

Exceptions

TargetException

The `obj` parameter is `null` and the method is not static.

-or-

The method is not declared or inherited by the class of `obj`.

-or-

A static constructor is invoked, and `obj` is neither `null` nor an instance of the class that declared the constructor.

Note: In .NET for Windows Store apps or the Portable Class Library, catch Exception instead.

ArgumentException

The elements of the `parameters` array do not match the signature of the method or constructor reflected by this instance.

TargetInvocationException

The invoked method or constructor throws an exception.

-or-

The current instance is a DynamicMethod that contains unverifiable code. See the "Verification" section in Remarks for DynamicMethod.

TargetParameterCountException

The `parameters` array does not have the correct number of arguments.

MethodAccessException

The caller does not have permission to execute the method or constructor that is represented by the current instance.

Note: In .NET for Windows Store apps or the Portable Class Library, catch the base class exception, MemberAccessException, instead.

InvalidOperationException

The type that declares the method is an open generic type. That is, the ContainsGenericParameters property returns `true` for the declaring type.

NotSupportedException

The current instance is a MethodBuilder.

Examples

The following code example demonstrates dynamic method lookup using reflection. Note that you cannot use the MethodInfo object from the base class to invoke the overridden method in the derived class, because late binding cannot resolve overrides.

C#

```
using System;
using System.Reflection;

public class MagicClass
{
    private int magicBaseValue;
```

```
    public MagicClass()
    {
        magicBaseValue = 9;
    }

    public int ItsMagic(int preMagic)
    {
        return preMagic * magicBaseValue;
    }
}

public class TestMethodInfo
{
    public static void Main()
    {
        // Get the constructor and create an instance of MagicClass

        Type magicType = Type.GetType("MagicClass");
        ConstructorInfo magicConstructor =
magicType.GetConstructor(Type.EmptyTypes);
        object magicClassObject = magicConstructor.Invoke(new object[]
{});

        // Get the ItsMagic method and invoke with a parameter value of
100

        MethodInfo magicMethod = magicType.GetMethod("ItsMagic");
        object magicValue = magicMethod.Invoke(magicClassObject, new ob-
ject[]{100});

        Console.WriteLine("MethodInfo.Invoke() Example\n");
        Console.WriteLine("MagicClass.ItsMagic() returned: {0}",
magicValue);
    }
}

// The example program gives the following output:
//
// MethodInfo.Invoke() Example
//
// MagicClass.ItsMagic() returned: 900
```

Remarks

This is a convenience method that calls the Invoke(Object, BindingFlags, Binder, Object[], CultureInfo) method overload, passing Default for `invokeAttr` and `null` for

`binder` and `culture`.

If the invoked method throws an exception, the Exception.GetBaseException method returns the originating exception.

To invoke a static method using its MethodInfo object, pass `null` for `obj`.

> **Note**
>
> If this method overload is used to invoke an instance constructor, the object supplied for `obj` is reinitialized; that is, all instance initializers are executed. The return value is `null`. If a class constructor is invoked, the class is reinitialized; that is, all class initializers are executed. The return value is `null`.

> **Note**
>
> Starting with .NET Framework 2.0, this method can be used to access non-public members if the caller has been granted **ReflectionPermission** with the **ReflectionPermissionFlag.RestrictedMemberAccess** flag and if the grant set of the non-public members is restricted to the caller's grant set, or a subset thereof. (See **Security Considerations for Reflection**.) To use this functionality, your application should target .NET Framework 3.5 or later.

If a parameter of the reflected method is a value type, and the corresponding argument in `parameters` is `null`, the runtime passes a zero-initialized instance of the value type.

See also

- BindingFlags
- Missing
- InvokeMember(String, BindingFlags, Binder, Object, Object[], ParameterModifier[], CultureInfo, String[])

Applies to

▼ .NET 8 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8
.NET Framework	1.1, 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0
Xamarin.iOS	10.8
Xamarin.Mac	3.0

Invoke(Object, BindingFlags, Binder, Object[], CultureInfo)

When overridden in a derived class, invokes the reflected method or constructor with the given parameters.

C#

```
public abstract object? Invoke (object? obj,
System.Reflection.BindingFlags invokeAttr, System.Reflection.Binder?
binder, object?[]? parameters, System.Globalization.CultureInfo? cul-
ture);
```

Parameters

`obj` Object

The object on which to invoke the method or constructor. If a method is static, this argument is ignored. If a constructor is static, this argument must be `null` or an instance of the class that defines the constructor.

`invokeAttr` BindingFlags

A bitmask that is a combination of 0 or more bit flags from BindingFlags. If `binder` is `null`, this parameter is assigned the value Default; thus, whatever you pass in is

ignored.

`binder` Binder

An object that enables the binding, coercion of argument types, invocation of members, and retrieval of `MemberInfo` objects via reflection. If `binder` is `null`, the default binder is used.

`parameters` Object[]

An argument list for the invoked method or constructor. This is an array of objects with the same number, order, and type as the parameters of the method or constructor to be invoked. If there are no parameters, this should be `null`.

If the method or constructor represented by this instance takes a ByRef parameter, there is no special attribute required for that parameter in order to invoke the method or constructor using this function. Any object in this array that is not explicitly initialized with a value will contain the default value for that object type. For reference-type elements, this value is `null`. For value-type elements, this value is 0, 0.0, or `false`, depending on the specific element type.

`culture` CultureInfo

An instance of `CultureInfo` used to govern the coercion of types. If this is `null`, the CultureInfo for the current thread is used. (This is necessary to convert a string that represents 1000 to a Double value, for example, since 1000 is represented differently by different cultures.)

Returns

Object

An `Object` containing the return value of the invoked method, or `null` in the case of a constructor, or `null` if the method's return type is `void`. Before calling the method or constructor, `Invoke` checks to see if the user has access permission and verifies that the parameters are valid.

Exceptions

TargetException

The `obj` parameter is `null` and the method is not static.

-or-

The method is not declared or inherited by the class of `obj`.

-or-

A static constructor is invoked, and `obj` is neither `null` nor an instance of the class that declared the constructor.

ArgumentException

The type of the `parameters` parameter does not match the signature of the method or constructor reflected by this instance.

TargetParameterCountException

The `parameters` array does not have the correct number of arguments.

TargetInvocationException

The invoked method or constructor throws an exception.

MethodAccessException

The caller does not have permission to execute the method or constructor that is represented by the current instance.

InvalidOperationException

The type that declares the method is an open generic type. That is, the ContainsGenericParameters property returns `true` for the declaring type.

Examples

The following example demonstrates all members of the System.Reflection.Binder class using an overload of Type.InvokeMember. The private method `CanConvertFrom` finds compatible types for a given type. For another example of invoking members in a custom binding scenario, see Dynamically Loading and Using Types.

C#

```
using System;
using System.Reflection;
using System.Globalization;

public class MyBinder : Binder
{
    public MyBinder() : base()
    {
    }
    private class BinderState
    {
        public object[] args;
    }
    public override FieldInfo BindToField(
        BindingFlags bindingAttr,
        FieldInfo[] match,
        object value,
        CultureInfo culture
        )
    {
        if(match == null)
            throw new ArgumentNullException("match");
        // Get a field for which the value parameter can be converted to
the specified field type.
        for(int i = 0; i < match.Length; i++)
            if(ChangeType(value, match[i].FieldType, culture) != null)
                return match[i];
        return null;
    }
    public override MethodBase BindToMethod(
        BindingFlags bindingAttr,
        MethodBase[] match,
        ref object[] args,
        ParameterModifier[] modifiers,
        CultureInfo culture,
        string[] names,
        out object state
        )
    {
        // Store the arguments to the method in a state object.
        BinderState myBinderState = new BinderState();
        object[] arguments = new Object[args.Length];
        args.CopyTo(arguments, 0);
        myBinderState.args = arguments;
        state = myBinderState;
        if(match == null)
            throw new ArgumentNullException();
        // Find a method that has the same parameters as those of the
args parameter.
```

```
        for(int i = 0; i < match.Length; i++)
        {
            // Count the number of parameters that match.
            int count = 0;
            ParameterInfo[] parameters = match[i].GetParameters();
            // Go on to the next method if the number of parameters do
not match.
            if(args.Length != parameters.Length)
                continue;
            // Match each of the parameters that the user expects the
method to have.
            for(int j = 0; j < args.Length; j++)
            {
                // If the names parameter is not null, then reorder args.
                if(names != null)
                {
                    if(names.Length != args.Length)
                        throw new ArgumentException("names and args must
have the same number of elements.");
                    for(int k = 0; k < names.Length; k++)
                        if(String.Compare(parameters[j].Name,
names[k].ToString()) == 0)
                            args[j] = myBinderState.args[k];
                }
                // Determine whether the types specified by the user can
be converted to the parameter type.
                if(ChangeType(args[j], parameters[j].ParameterType, cul-
ture) != null)
                    count += 1;
                else
                    break;
            }
            // Determine whether the method has been found.
            if(count == args.Length)
                return match[i];
        }
        return null;
    }
    public override object ChangeType(
        object value,
        Type myChangeType,
        CultureInfo culture
        )
    {
        // Determine whether the value parameter can be converted to a
value of type myType.
        if(CanConvertFrom(value.GetType(), myChangeType))
            // Return the converted object.
            return Convert.ChangeType(value, myChangeType);
        else
```

```
            // Return null.
            return null;
    }
    public override void ReorderArgumentArray(
        ref object[] args,
        object state
        )
    {
        // Return the args that had been reordered by BindToMethod.
        ((BinderState)state).args.CopyTo(args, 0);
    }
    public override MethodBase SelectMethod(
        BindingFlags bindingAttr,
        MethodBase[] match,
        Type[] types,
        ParameterModifier[] modifiers
        )
    {
        if(match == null)
            throw new ArgumentNullException("match");
        for(int i = 0; i < match.Length; i++)
        {
            // Count the number of parameters that match.
            int count = 0;
            ParameterInfo[] parameters = match[i].GetParameters();
            // Go on to the next method if the number of parameters do
not match.
            if(types.Length != parameters.Length)
                continue;
            // Match each of the parameters that the user expects the
method to have.
            for(int j = 0; j < types.Length; j++)
                // Determine whether the types specified by the user can
be converted to parameter type.
                if(CanConvertFrom(types[j], parameters[j].ParameterType))
                    count += 1;
                else
                    break;
            // Determine whether the method has been found.
            if(count == types.Length)
                return match[i];
        }
        return null;
    }
    public override PropertyInfo SelectProperty(
        BindingFlags bindingAttr,
        PropertyInfo[] match,
        Type returnType,
        Type[] indexes,
        ParameterModifier[] modifiers
```

```
        )
    {
        if(match == null)
            throw new ArgumentNullException("match");
        for(int i = 0; i < match.Length; i++)
        {
            // Count the number of indexes that match.
            int count = 0;
            ParameterInfo[] parameters = match[i].GetIndexParameters();
            // Go on to the next property if the number of indexes do not
match.
            if(indexes.Length != parameters.Length)
                continue;
            // Match each of the indexes that the user expects the prop-
erty to have.
            for(int j = 0; j < indexes.Length; j++)
                // Determine whether the types specified by the user can
be converted to index type.
                if(CanConvertFrom(indexes[j], parame-
ters[j].ParameterType))
                    count += 1;
                else
                    break;
            // Determine whether the property has been found.
            if(count == indexes.Length)
                // Determine whether the return type can be converted to
the properties type.
                if(CanConvertFrom(returnType, match[i].PropertyType))
                    return match[i];
                else
                    continue;
        }
        return null;
    }
    // Determines whether type1 can be converted to type2. Check only for
primitive types.
    private bool CanConvertFrom(Type type1, Type type2)
    {
        if(type1.IsPrimitive && type2.IsPrimitive)
        {
            TypeCode typeCode1 = Type.GetTypeCode(type1);
            TypeCode typeCode2 = Type.GetTypeCode(type2);
            // If both type1 and type2 have the same type, return true.
            if(typeCode1 == typeCode2)
                return true;
            // Possible conversions from Char follow.
            if(typeCode1 == TypeCode.Char)
                switch(typeCode2)
                {
                    case TypeCode.UInt16 : return true;
```

```
            case TypeCode.UInt32 : return true;
            case TypeCode.Int32  : return true;
            case TypeCode.UInt64 : return true;
            case TypeCode.Int64  : return true;
            case TypeCode.Single : return true;
            case TypeCode.Double : return true;
            default              : return false;
        }
    // Possible conversions from Byte follow.
    if(typeCode1 == TypeCode.Byte)
        switch(typeCode2)
        {
            case TypeCode.Char   : return true;
            case TypeCode.UInt16 : return true;
            case TypeCode.Int16  : return true;
            case TypeCode.UInt32 : return true;
            case TypeCode.Int32  : return true;
            case TypeCode.UInt64 : return true;
            case TypeCode.Int64  : return true;
            case TypeCode.Single : return true;
            case TypeCode.Double : return true;
            default              : return false;
        }
    // Possible conversions from SByte follow.
    if(typeCode1 == TypeCode.SByte)
        switch(typeCode2)
        {
            case TypeCode.Int16  : return true;
            case TypeCode.Int32  : return true;
            case TypeCode.Int64  : return true;
            case TypeCode.Single : return true;
            case TypeCode.Double : return true;
            default              : return false;
        }
    // Possible conversions from UInt16 follow.
    if(typeCode1 == TypeCode.UInt16)
        switch(typeCode2)
        {
            case TypeCode.UInt32 : return true;
            case TypeCode.Int32  : return true;
            case TypeCode.UInt64 : return true;
            case TypeCode.Int64  : return true;
            case TypeCode.Single : return true;
            case TypeCode.Double : return true;
            default              : return false;
        }
    // Possible conversions from Int16 follow.
    if(typeCode1 == TypeCode.Int16)
        switch(typeCode2)
        {
```

```
                case TypeCode.Int32  : return true;
                case TypeCode.Int64  : return true;
                case TypeCode.Single : return true;
                case TypeCode.Double : return true;
                default              : return false;
            }
        // Possible conversions from UInt32 follow.
        if(typeCode1 == TypeCode.UInt32)
            switch(typeCode2)
            {
                case TypeCode.UInt64 : return true;
                case TypeCode.Int64  : return true;
                case TypeCode.Single : return true;
                case TypeCode.Double : return true;
                default              : return false;
            }
        // Possible conversions from Int32 follow.
        if(typeCode1 == TypeCode.Int32)
            switch(typeCode2)
            {
                case TypeCode.Int64  : return true;
                case TypeCode.Single : return true;
                case TypeCode.Double : return true;
                default              : return false;
            }
        // Possible conversions from UInt64 follow.
        if(typeCode1 == TypeCode.UInt64)
            switch(typeCode2)
            {
                case TypeCode.Single : return true;
                case TypeCode.Double : return true;
                default              : return false;
            }
        // Possible conversions from Int64 follow.
        if(typeCode1 == TypeCode.Int64)
            switch(typeCode2)
            {
                case TypeCode.Single : return true;
                case TypeCode.Double : return true;
                default              : return false;
            }
        // Possible conversions from Single follow.
        if(typeCode1 == TypeCode.Single)
            switch(typeCode2)
            {
                case TypeCode.Double : return true;
                default              : return false;
            }
    }
    return false;
```

```
    }
}
public class MyClass1
{
    public short myFieldB;
    public int myFieldA;
    public void MyMethod(long i, char k)
    {
        Console.WriteLine("\nThis is MyMethod(long i, char k)");
    }
    public void MyMethod(long i, long j)
    {
        Console.WriteLine("\nThis is MyMethod(long i, long j)");
    }
}
public class Binder_Example
{
    public static void Main()
    {
        // Get the type of MyClass1.
        Type myType = typeof(MyClass1);
        // Get the instance of MyClass1.
        MyClass1 myInstance = new MyClass1();
        Console.WriteLine("\nDisplaying the results of using the MyBinder
binder.\n");
        // Get the method information for MyMethod.
        MethodInfo myMethod = myType.GetMethod("MyMethod",
BindingFlags.Public | BindingFlags.Instance,
            new MyBinder(), new Type[] {typeof(short), typeof(short)},
null);
        Console.WriteLine(myMethod);
        // Invoke MyMethod.
        myMethod.Invoke(myInstance, BindingFlags.InvokeMethod, new
MyBinder(), new Object[] {(int)32, (int)32}, CultureInfo.CurrentCulture);
    }
}
```

Remarks

This method dynamically invokes the method reflected by this instance on `obj`, and passes along the specified parameters. If the method is static, the `obj` parameter is ignored. For non-static methods, `obj` should be an instance of a class that inherits or declares the method and must be the same type as this class. If the method has no parameters, the value of `parameters` should be `null`. Otherwise, the number, type, and order of elements in `parameters` should be identical to the number, type, and

order of parameters for the method reflected by this instance.

You may not omit optional parameters in calls to `Invoke`. To invoke a method and omit optional parameters, call `Type.InvokeMember` instead.

> **Note**
>
> If this method overload is used to invoke an instance constructor, the object supplied for `obj` is reinitialized; that is, all instance initializers are executed. The return value is `null`. If a class constructor is invoked, the class is reinitialized; that is, all class initializers are executed. The return value is `null`.

For pass-by-value primitive parameters, normal widening is performed (Int16 -> Int32, for example). For pass-by-value reference parameters, normal reference widening is allowed (derived class to base class, and base class to interface type). However, for pass-by-reference primitive parameters, the types must match exactly. For pass-by-reference reference parameters, the normal widening still applies.

For example, if the method reflected by this instance is declared as `public boolean Compare(String a, String b)`, then `parameters` should be an array of `Objects` with length 2 such that `parameters[0] = new Object("SomeString1") and parameters[1] = new Object("SomeString2")`.

If a parameter of the current method is a value type, and the corresponding argument in `parameters` is `null`, the runtime passes a zero-initialized instance of the value type.

Reflection uses dynamic method lookup when invoking virtual methods. For example, suppose that class B inherits from class A and both implement a virtual method named M. Now suppose that you have a `MethodInfo` object that represents M on class A. If you use the `Invoke` method to invoke M on an object of type B, then reflection will use the implementation given by class B. Even if the object of type B is cast to A, the implementation given by class B is used (see code sample below).

On the other hand, if the method is non-virtual, then reflection will use the implementation given by the type from which the `MethodInfo` was obtained, regardless of the type of the object passed as the target.

Access restrictions are ignored for fully trusted code. That is, private constructors, methods, fields, and properties can be accessed and invoked via reflection whenever the code is fully trusted.

If the invoked method throws an exception, the Exception.GetBaseException method returns the originating exception.

> **Note**
>
> Starting with .NET Framework 2.0, this method can be used to access non-public members if the caller has been granted **ReflectionPermission** with the **ReflectionPermissionFlag.RestrictedMemberAccess** flag and if the grant set of the non-public members is restricted to the caller's grant set, or a subset thereof. (See **Security Considerations for Reflection**.) To use this functionality, your application should target .NET Framework 3.5 or later.

See also

- InvokeMember(String, BindingFlags, Binder, Object, Object[], ParameterModifier[], CultureInfo, String[])
- Dynamically Loading and Using Types

Applies to

▼ .NET 8 and other versions

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8
.NET Framework	1.1, 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1
Xamarin.iOS	10.8
Xamarin.Mac	3.0