Contents

Interceptor actions get all of the information about the context of what they are intercepting from the single interceptor argument passed into them. This argument will obviously be different for different contexts; i.e. a *set* versus a *get* action, a change to an employee versus a company, a change to the FirstName property instead of the LastName property. Because of this there are many possible implementations of what the single argument passed into any interceptor action might contain. However, all of these implementations implement a single primary interface: *IPropertyInterceptorArgs*.

Every interceptor action shown previously provides an example of this. In each case, a single argument of type *PropertyInterceptorArgs<Employee, String of type IPropertyInterceptorArgs>* was passed into each of the interceptor methods.

In fact, the type of the 'args' instance that is actually be passed into each of these methods at runtime is an instance of a subtype of the argument type declared in the methods signature. For any interceptor action defined on a DevForce entity, the **actual** args passed into the action will be a concrete implementation of one of the following classes.

DataEntityPropertyGetInterceptorArgs<TInstance, TValue>

DataEntityPropertySetInterceptorArgs<TInstance, TValue>

NavigationEntityPropertyGetInterceptorArgs<TInstance, TValue>

NavigationEntityPropertySetInterceptorArgs<TInstance, TValue>

The boldfaced characters above indicate whether we are providing interception to a get or a set property, as well as whether we are intercepting a data or a navigation property.

In general, you can write an interception method with an argument type that is **any** base class of the actual argument type defined for that interceptor. If you do use a base class, then you may need to perform runtime casts in order to access some of the additional properties provided by the specific subclass passed in at runtime. These subclassed properties will be discussed later.

The entire inheritance hierarchy for property interceptor arguments is shown below:

Assembly Where Defined	*Property Interceptor Arguments*
IdeaBlade.Core	*IPropertyInterceptorArgs*
	IPropertyInterceptorArgs<TInstance, TValue>
	PropertyInterceptorArgs<TInstance, TValue>
IdeaBlade.EntityModel	*DataEntityPropertyInterceptorArgs<TInstance, TValue>*
	DataEntityPropertyGetInterceptorArgs<TInstance, TValue>
	DataEntityPropertySetInterceptorArgs<TInstance, TValue>
	NavigationEntityPropertyInterceptorArgs<TInstance, TValue>
	NavigationEntityPropertyGetInterceptorArgs<TInstance, TValue>
	NavigationEntityPropertySetInterceptorArgs<TInstance, TValue>

The generic <TInstance> argument will always be the type that the intercepted method will operate on, known elsewhere in this document and the interceptor API as the "TargetType". The <TValue> argument will be the type of the property being intercepted. i.e. 'String' for the 'LastName' property.

Note that the interceptor arguments defined to operate on DevForce entities break into multiple subclasses with additional associated interfaces based on two primary criteria.

1. Is it a 'get' or a 'set' interceptor?
    1. 'get' interceptor args implement **IEntityPropertyGetInterceptorArgs**
    2. 'set' interceptor args implement **IEntityPropertySetInterceptorArgs**
1. Does it involve a 'DataEntityProperty' or a 'NavigationEntityProperty'?.
    1. 'DataEntityProperty' args implement **IDataEntityPropertyInterceptorArgs**
    2. 'NavigationEntityProperty' args implement **INavigationEntityPropertyInterceptorArgs**

The API for each of the interfaces above is discussed below.

IPropertyInterceptorArgs

The root of all property interceptor arguments is the IPropertyInterceptorArgs interface. Its properties will be available to all interceptors.

```
public interface IPropertyInterceptorArgs {
  Object Instance { get; }
  Object Value { get; set; }
 bool Cancel { get; set; }
  Action<Exception> ExceptionAction { get; set; }
 object Tag { get; set; }
 object Context { get; }
}
```

```
Public Interface IPropertyInterceptorArgs
 ReadOnly Property Instance() As Object
 Property Value() As Object
 Property Cancel() As Boolean
 Property ExceptionAction() As Action(Of Exception)
 Property Tag() As Object
 ReadOnly Property Context() As Object
End Interface
```

In general the most useful of these properties will be the 'Instance' and 'Value' properties.

The 'Instance' property will always contain the 'parent' object whose property is being intercepted. The 'Value' will always be the value that is being either retrieved or set.

The 'Cancel' property allows you to stop the execution of the property interceptor chain at any point by setting the 'Cancel' property to 'true.

The 'ExceptionAction' property allows you to set up an action that will be performed whenever an exception occurs anywhere after this point in the chain of interceptors.

The 'Tag' property is intended as a general purpose grab bag for the developer to use for his/her own purposes.

The 'Context' property is used for internal purposes and should be ignored.

An example of using the ExceptionAction and Cancel is shown below:

Code Listing 8. Order.LogExceptionsAndCancelSets(). Demoed in MainDemo.TestLoggingOfExceptionsAndAutoCancellationOfSets().

```
/// Do not let any setters throw an exception. Instead, eat them,
/// log them, and cancel the remainder of the Set operation.
[IbCore.BeforeSet(Order = -1)]
public void LogExceptionsAndCancelSets(
  IbCore.IPropertyInterceptorArgs args) {
  args.ExceptionAction = (e) => {
    Common.LogManager.LogAnAction(string.Format(
     "Here in {0}", e.Message));
    args.Cancel = true;
  };
}
```

```
 ''' <remarks>
 ''' Do not let any setters throw an exception. Instead, eat them,
 ''' log them, and cancel the remainder of the Set operation.
 ''' </remarks>
<IbCore.BeforeSet(Order:=-1)> _
Public Sub LogExceptionsAndCancelSets(ByVal args _
 As IbCore.IPropertyInterceptorArgs)
  args.ExceptionAction = Sub(e)
                         Common.LogManager.LogAnAction( _
                          String.Format("Here in: {0}", e.Message))
                         args.Cancel = True
                       End Sub
End Sub
```

Note that we applied an explicit Order value less than 0 for this interceptor. Assuming that none of the property-specific interceptors have an explicit Order defined, their Order value defaults to zero, this interceptor will run first for all properties of the type on which it's defined.

> In our samples, this interceptor happens to be defined on the Order type. Please note that there is no relationship between that fact and the use of the *Order* parameter in the BeforeSet attribute. Two different things!

Generic IPropertyInterceptorArgs

The following is a generic version of IPropertyInterceptorArgs where both the Instance and Value properties are now strongly typed; otherwise it is identical to IPropertyInterceptorArgs.

```
public interface IPropertyInterceptorArgs<TInstance, TValue> :
  IdeaBlade.Core.IPropertyInterceptorArgs {
  TInstance Instance { get; }
  TValue Value { get; set; }
 bool Cancel { get; set; }
  Action<Exception> ExceptionAction { get; set; }
 object Tag { get; set; }
 object Context { get; }
}
```

```
Public Interface IPropertyInterceptorArgs(Of TInstance, TValue)
 Inherits IdeaBlade.Core.IPropertyInterceptorArgs
 ReadOnly Property Instance() As TInstance
 Property Value() As TValue
 Property Cancel() As Boolean
 Property ExceptionAction() As Action(Of Exception)
 Property Tag() As Object
 ReadOnly Property Context() As Object
End Interface
```

IEntity PropertyInterceptorArgs and subclasses

Whereas the interfaces above can be used to intercept any property on any object, the argument interfaces below are for use only with DevForce specific entities and complex objects. Each interface below provides additional contextual data to any interceptor actions defined to operate on DevForce entities.

The most basic of these is simply the idea that each property on a DevForce entity has a corresponding "EntityProperty" (discussed elsewhere in this guide).

```
public interface IEntityPropertyInterceptorArgs :
  IbCore.IPropertyInterceptorArgs {
    IbEm.EntityProperty EntityProperty { get; }
}
```

```
Public Interface IEntityPropertyInterceptorArgs
 Inherits IbCore.IPropertyInterceptorArgs
 ReadOnly Property EntityProperty() As IbEm.EntityProperty
End Interface
```

An example is shown below:

Code Listing 9. Customer.AfterSetAnyUsingIEntityPropertyInterceptorArgs().

```
[IbCore.AfterSet]
public void AfterSetAnyUsingIEntityPropertyInterceptorArgs(
  IbCore.IPropertyInterceptorArgs args) {
 // Cast the IPropertyInterceptorArgs to the entity-specific version, then
 // values information available on the EntityProperty contained thereby.
 var entityPropertyArgs = args as IbEm.IEntityPropertyInterceptorArgs;
 if (entityPropertyArgs != null) {
    Common.LogManager.LogAnAction(string.Format(
     "Property [Customer.{0}] was set to the value: [{1}]",
    entityPropertyArgs.EntityProperty.Name, args.Value.ToString()));
  }
}
```

```
<AfterSet()> _
Public Sub AfterSetAny(ByVal args As IPropertyInterceptorArgs)
 Dim entityPropertyArgs = TryCast(args, IEntityPropertyInterceptorArgs)
 If entityPropertyArgs IsNot Nothing Then
```

```
    Log("The " + entityPropertyArgs.EntityProperty.Name + _
      " was set to the value:= " + args.Value.ToString())
  End If
End Sub
```

The next two interfaces provide additional context based on whether the interceptor action being performed is a 'get' operation or a 'set' operation.

For a get operation, IdeaBlade entities have a concept of possibly multiple versions, i.e. an original, current, or proposed version, of an entity at any single point in time. It may be useful to know which 'version' is being retrieved during the current action. Note that the version cannot be changed.

```
public interface IEntityPropertyGetInterceptorArgs :
  IEntityPropertyInterceptorArgs {
    IbEm.EntityVersion EntityVersion { get; }
}
```

```
Public Interface IEntityPropertyGetInterceptorArgs
  Inherits IEntityPropertyInterceptorArgs
  ReadOnly Property EntityVersion() As IbEm.EntityVersion
End Interface
```

The DevForce EntityProperty is an abstract class with two concrete subclasses; a DataEntityProperty and a NavigationEntityProperty (discussed elsewhere in this guide). The next two IEntityPropertyInterceptorArgs subinterfaces allow access to instances of one or the other of these depending on whether the property being intercepted is a data or a navigation property.

```
public interface IDataEntityPropertyInterceptorArgs :
  IEntityPropertyInterceptorArgs {
    IbEm.DataEntityProperty DataEntityProperty { get; }
}
```

```
Public Interface IDataEntityPropertyInterceptorArgs
  Inherits IEntityPropertyInterceptorArgs
  ReadOnly Property DataEntityProperty() As  _
    IbEm.DataEntityProperty
End Interface
```

```
public interface INavigationEntityPropertyInterceptorArgs :
  IEntityPropertyInterceptorArgs {
    IbEm.NavigationEntityProperty NavigationEntityProperty { get; }
}
```

```
Public Interface INavigationEntityPropertyInterceptorArgs
  Inherits IEntityPropertyInterceptorArgs
  ReadOnly Property NavigationEntityProperty() As  _
    IbEm.NavigationEntityProperty
End Interface
```

IPropertyInterceptorArgs Type Coercion

One of the first issues that a developer will encounter with writing interceptor actions that handle more than one property is that it becomes difficult or impossible to use a concrete subtype as the argument to the interceptor.

For example, imagine that we wanted to write a single action that handled two or more very different properties each of a different type:

This could be written as follows:

Code Listing 10. Employee.ActionToBePerformedAgainstDifferentTypesV1().

```
[BeforeSet(EntityPropertyNames.HireDate)] // hire date is of type datetime
[BeforeSet(EntityPropertyNames.FirstName)] // firstname is of type string
public void ActionToBePerformedAgainstDifferentTypesV1(
  IbCore.IPropertyInterceptorArgs args) {
  var emp = (Employee)args.Instance;
  var entityProperty = ((IbEm.IDataEntityPropertyInterceptorArgs)
    args).EntityProperty;
  //.. do some action with emp and entityProperty
}
```

```
 ' hire date is of type datetime
 ' firstname is of type string
<IbCore.BeforeSet(EntityPropertyNames.HireDate)> _
<IbCore.BeforeSet(EntityPropertyNames.FirstName)> _
```

```
Public Sub ActionToBePerformedAgainstDifferentTypesV1(ByVal args As  _
  IbCore.IPropertyInterceptorArgs)
 Dim emp = DirectCast(args.Instance, Employee)
 Dim entityProperty = DirectCast(args,  _
    IbEm.IDataEntityPropertyInterceptorArgs).EntityProperty
 '.. do some action with emp and entityProperty
End Sub
```

But ideally we would prefer to write it like this, in order to avoid performing a lot of superfluous casts:

Code Listing 11. Employee.ActionToBePerformedAgainstDifferentTypesV2().

```
[IbCore.BeforeSet(EntityPropertyNames.HireDate)]
// hire date is of type datetime
[IbCore.BeforeSet(EntityPropertyNames.FirstName)]
// firstname is of type string
public void ActionToBePerformedAgainstDifferentTypesV2(
    IbEm.DataEntityPropertySetInterceptorArgs<Employee, Object> args) {
 // no casting
 var emp = args.Instance;
  var entityProperty = args.DataEntityProperty;
 //.. do some action with emp and entityProperty
}
```

```
 ' hire date is of type datetime
 ' firstname is of type string
<IbCore.BeforeSet(EntityPropertyNames.HireDate)> _
<IbCore.BeforeSet(EntityPropertyNames.FirstName)> _
Public Sub ActionToBePerformedAgainstDifferentTypesV2( _
 ByVal args As IbEm.DataEntityPropertySetInterceptorArgs( _
   Of Employee, [Object]))
 ' no casting
 Dim emp = args.Instance
 Dim entityProperty = args.DataEntityProperty
 '.. do some action with emp and entityProperty
End Sub
```

The problem is that, according to the rules of inheritance, the two concrete classes that this method will be called with:

Type 1: DataEntityPropertySetInterceptorArgs<Employee, String>

Type 2: DataEntityPropertySetInterceptorArgs<Employee, DateTime>

…do not inherit from:

Type 3: DataEntityPropertySetInterceptorArgs<Employee, Object>

In fact, the only class or interface that they do share is:

IPropertyInterceptorArgs

So in order to allow this construction, DevForce needs to “coerce” each of ‘Type1’ and ‘Type2” into ‘Type3” for the duration of the method call.

Because DevForce does do this, any of the following arguments are also valid:

Type 4: DataEntityPropertySetInterceptorArgs<Entity, Object>

Type 5: DataEntityPropertySetInterceptorArgs<Object, Object>

Type 5: PropertyInterceptorArgs<Employee, Object>

… **etc.**

The basic rule for the type coercion facility is that any concrete type can be specified if its generic version is a subtype of the generic version of the actual argument type that will be passed in.