Interception Hands-On Lab for Enterprise Library

patterns & practices
proven practices for predictable results

This walkthrough should act as your guide to learn about interception and practice how to leverage its capabilities in various application contexts.

This hands-on lab includes the following 10 labs:

- Lab 1: Set Up Interception by Using the Stand-alone Interception API
- Lab 2: Use a Unity Container to Perform Interception
- Lab 3: Use Interface Interception
- Lab 4: Set Up Interception Using the Configuration File
- Lab 5: Create a Custom Interception Behavior
- Lab 6: Set Up Policy Injection Using Attributes
- Lab 7: Set Up Policy Injection Using the Streamlined Interception Configuration API
- Lab 8: Set Up Policy Injection Using the Configuration File
- Lab 9: Implement a Custom Call Handler
- Lab 10: Use the Policy Injection Application Block to Perform Interception

Authors

These Hands-On Labs were produced by the following individuals:

- Program Management: Grigori Melnik and Madalyn Parker (Microsoft Corporation)
- Development/Testing: Fernando Simonazzi (Clarius Consulting), Chris Tavares (Microsoft Corporation), Mariano Grande (Digit Factory), and Naveen Pitipornvivat (Adecco)
- Documentation: Fernando Simonazzi (Clarius Consulting), Grigori Melnik, Alex Homer, Nelly Delgado and RoAnn Corbisier (Microsoft Corporation)

Lab 1: Set Up Interception by Using the Stand-alone Interception API

Estimated time to complete this lab: **15 minutes**

In this lab, you will practice updating a class to be interceptable with different interception mechanisms and creating intercepted instances of the interceptable class. Interception is explicitly indicated by using the methods on the **Intercept** class.

In this lab, you will update a simple Windows Forms application that uses a **BankAccount** object to perform updates for withdrawals and deposits to the account and to check the account balance. You will change the way the **BankAccount** instance is obtained, and use the **TransparentProxyInterceptor** and the **VirtualMethodInterceptor** as interception mechanisms. The **TransparentProxyInterceptor** interceptor can be used to intercept existing instances of classes which derive from **MarshalByRefObject** or which implement an interface. The **VirtualMethodInterceptor** interceptor can be used to intercept virtual methods on new instances of non-sealed classes. For information about the interception, see the topic "Interception with Unity" in the Unity 3 documentation.

Procedures

This Hands-On Lab includes the following tasks:

- Task 1: Add references to the necessary assemblies.
- Task 2: Make an interception behavior available.
- Task 3: Make the target class interceptable by the TransparentProxyInterceptor.
- Task 4: Wrap an existing instance to intercept it using the TransparentProxyInterceptor.
- Task 5: Make the target class interceptable by the VirtualMethodInterceptor.
- Task 6: Create a new intercepted instance using the VirtualMethodInterceptor.

Task 1: Add references to the necessary assemblies

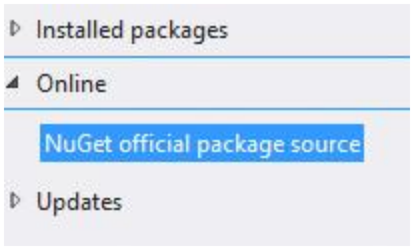
This lab demonstrates how to set-up your application to perform basic interception. You will use the **Microsoft.Practices.Unity.Interception.dll** assembly, which provides the basic mechanisms to set-up and provide interception.

To begin this exercise open the solution file Lab01\Before\InterceptionHOL.sln.

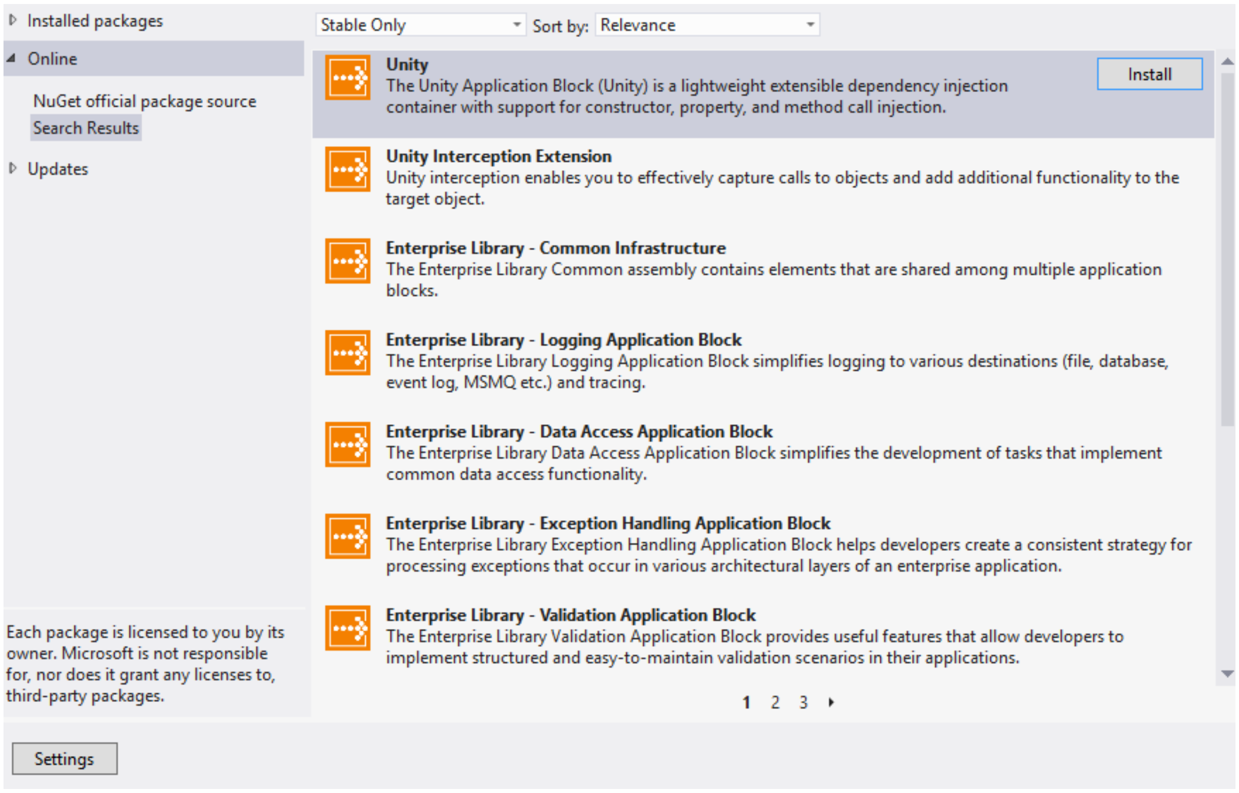
To add a reference to the Microsoft.Practices.Unity.Interception.dll assembly

1. Select the **IntereptionHOL** project. Select the **Project | Manage NuGet Packages** menu command.

2. Select the “Online” option to view NuGet packages available online.



3. Search for **EntLib6** in the search bar. Select **Unity Interception Extension** and click install.



Unity
The Unity Application Block (Unity) is a lightweight extensible dependency injection container with support for constructor, property, and method call injection.

Unity Interception Extension
Unity interception enables you to effectively capture calls to objects and add additional functionality to the target object.

Enterprise Library - Common Infrastructure
The Enterprise Library Common assembly contains elements that are shared among multiple application blocks.

Enterprise Library - Logging Application Block
The Enterprise Library Logging Application Block simplifies logging to various destinations (file, database, event log, MSMQ etc.) and tracing.

Enterprise Library - Data Access Application Block
The Enterprise Library Data Access Application Block simplifies the development of tasks that implement common data access functionality.

Enterprise Library - Exception Handling Application Block
The Enterprise Library Exception Handling Application Block helps developers create a consistent strategy for processing exceptions that occur in various architectural layers of an enterprise application.

Enterprise Library - Validation Application Block
The Enterprise Library Validation Application Block provides useful features that allow developers to implement structured and easy-to-maintain validation scenarios in their applications.

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

4. Click **Accept** on the License Acceptance window that pops up.

Task 2: Make an interception behavior available

Interception relies on **IInterceptionBehavior** objects to perform actions on the intercepted method calls. In this lab you will use a custom interception behavior.

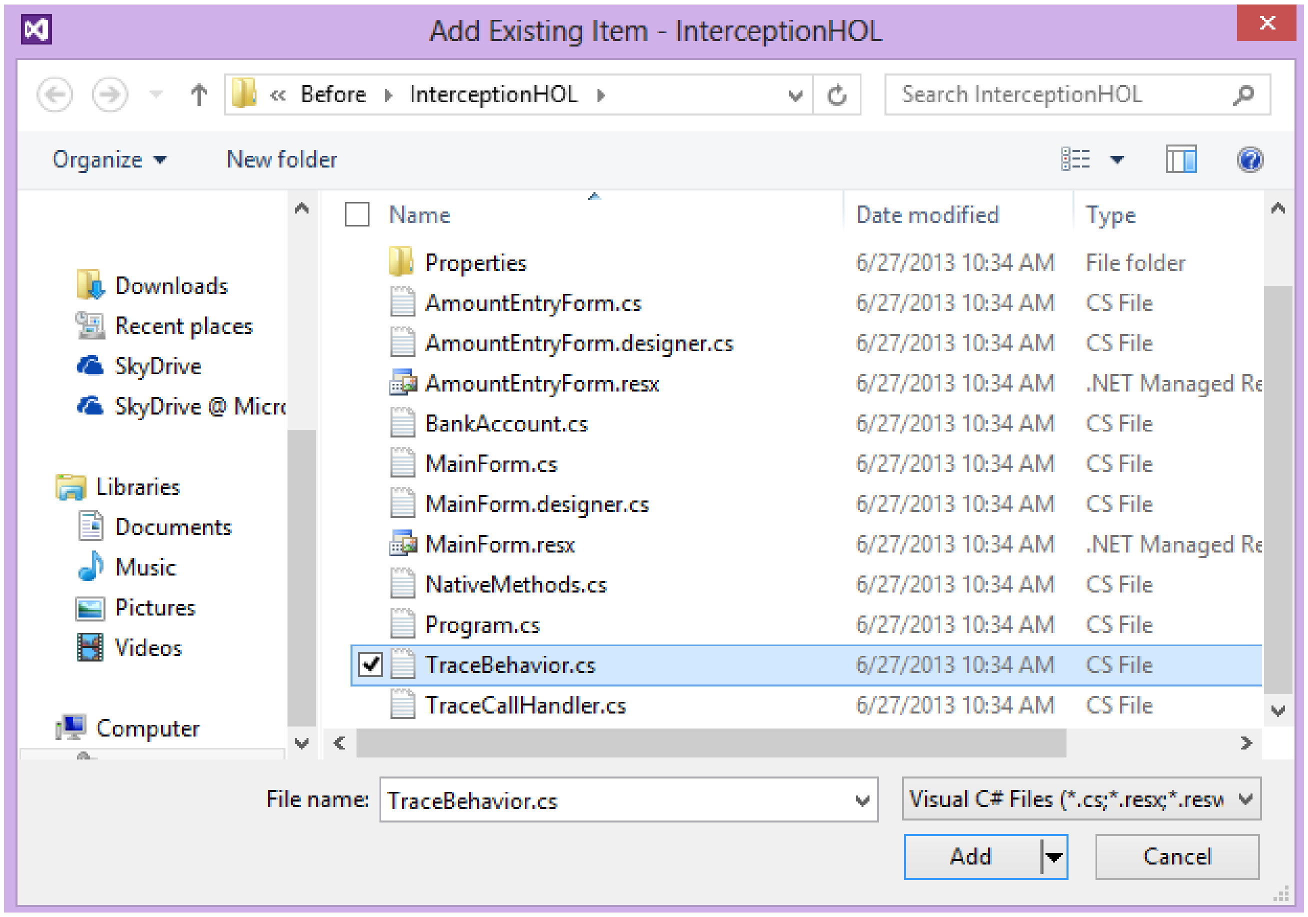
Unity ships with the **PolicyInjectionBehavior** that supports a specific type of interception, policy injection.

The custom interception behavior used in this lab logs entries before and after invoking the remainder of the behaviors pipeline. Whether or not the intercepted method actually is invoked depends on the next behaviors in the pipeline. The second entry indicates whether an exception will be thrown as the result of the intercepted method. The interception behavior uses a standard **TraceSource** object which must be configured when the call handler is created. The configuration file for the lab application (**app.config**) already contains the **system.diagnostics** configuration for the **TraceSource** instances required by the next few labs. For more information see the TraceSource Class on MSDN.

To use the Hands-On Lab interception behavior add the TraceBehavior.cs file to the InterceptionHOL project

1. Right-click the InterceptionHOL project in Solution Explorer, click **Add**, and click **Existing Item**.

2. Add the existing behavior file. Select the **TraceBehavior.cs** file and then click **Add**.



	Name	Date modified	Type
	Properties	6/27/2013 10:34 AM	File folder
	AmountEntryForm.cs	6/27/2013 10:34 AM	CS File
	AmountEntryForm.designer.cs	6/27/2013 10:34 AM	CS File
	AmountEntryForm.resx	6/27/2013 10:34 AM	.NET Managed Re
	BankAccount.cs	6/27/2013 10:34 AM	CS File
	MainForm.cs	6/27/2013 10:34 AM	CS File
	MainForm.designer.cs	6/27/2013 10:34 AM	CS File
	MainForm.resx	6/27/2013 10:34 AM	.NET Managed Re
	NativeMethods.cs	6/27/2013 10:34 AM	CS File
	Program.cs	6/27/2013 10:34 AM	CS File
✓	TraceBehavior.cs	6/27/2013 10:34 AM	CS File
	TraceCallHandler.cs	6/27/2013 10:34 AM	CS File

File name: TraceBehavior.cs

Visual C# Files (*.cs;*.resx;*.resw

Task 3: Make the target class interceptable by the TransparentProxyInterceptor

Unity's interceptors have specific requirements to make a class interceptable. This lab uses the **TransparentProxyInterceptor** first, so the target class will be updated to inherit from the **MarshalByRefObject** class. For more information about the requirements of the built-in interceptors, including the **TransparentProxyInterceptor**, see the topic Interception with Unity in the Unity 3 documentation.

To make the BankAccount class interceptable

Update the **BankAccount** class definition to make it inherit from **MarshalByRefObject** by adding the bold and highlighted code shown in the following excerpt.

```
public sealed class BankAccount : MarshalByRefObject
{
  ...
}
```

Task 4: Wrap an existing instance to intercept it using the TransparentProxyInterceptor

In this task the instance of the **BankAccount** class will be wrapped by using the **Intercept.ThroughProxy** method.

To wrap an existing instance

1. Add **using** directives to the **MainForm.cs** file in the InterceptionHOL project to make the required types available without full name qualification.

```
using System.Diagnostics;
using Microsoft.Practices.Unity.InterceptionExtension;
```

2. Invoke the **Intercept.ThroughProxy** method before storing the reference to the **BankAccount** instance in the constructor of the **MainForm** class. Provide a new instance of the **TransparentProxyInterceptor** as the interceptor and a new, properly configured instance of the custom behavior **TraceBehavior**, by adding the bold and highlighted code shown in the following excerpt.

```
public MainForm()
{
  InitializeComponent();
  PopulateUserList();
  bankAccount =
         Intercept.ThroughProxy(
             new BusinessLogic.BankAccount(),
             new TransparentProxyInterceptor(),
             new[] { new TraceBehavior(new TraceSource("interception"))
});
}
```

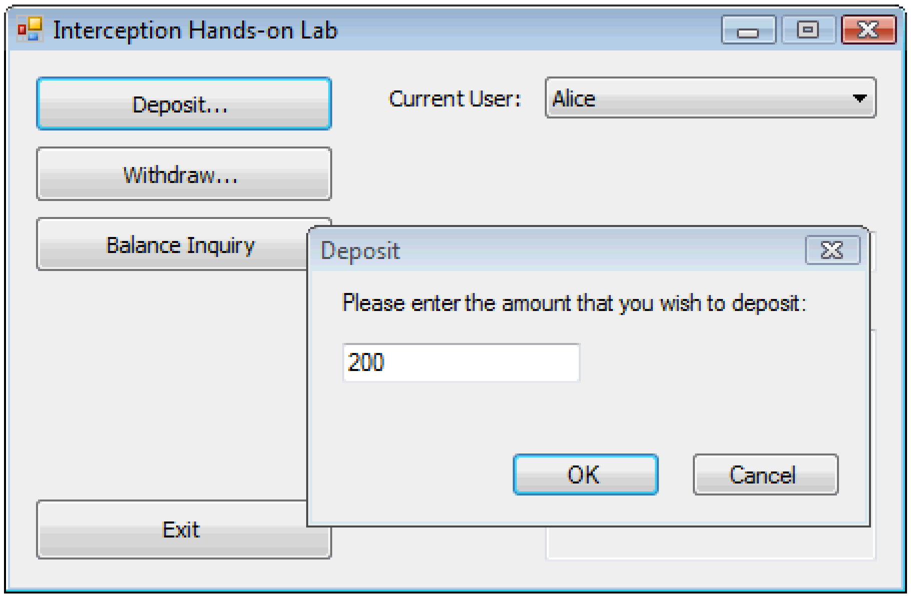
Verification

In this section, you will validate that the specified interception behavior (logging each call to a method in the **BankAccount** type) is actually invoked.

To validate that the interception behavior is invoked

1. Build and run the InterceptionHOL project.

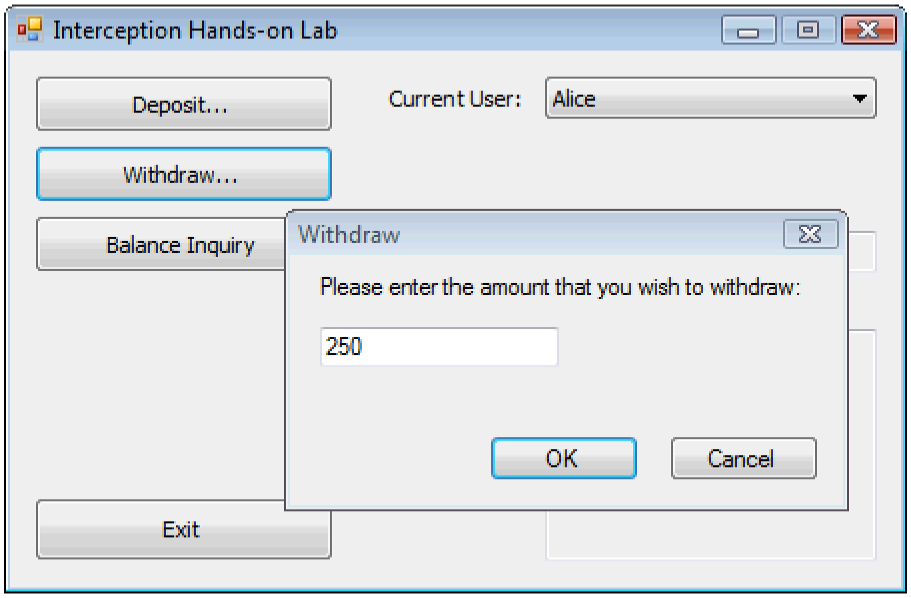
2. Perform a deposit with the application, as shown in the following figure.



Please enter the amount that you wish to deposit:

200

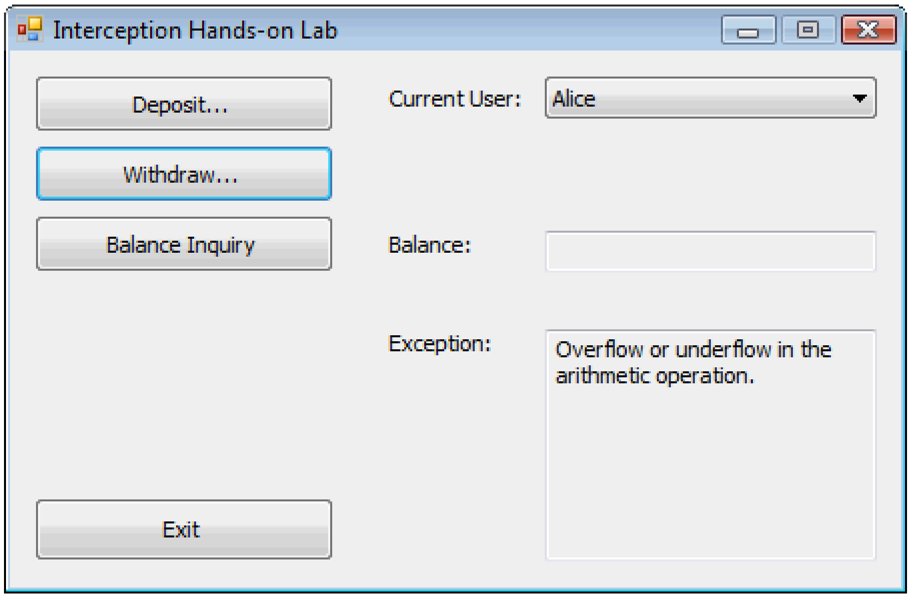
3. Make a withdrawal.



Please enter the amount that you wish to withdraw:

250

4. Withdraw an amount higher than the current balance to cause an exception to be thrown.

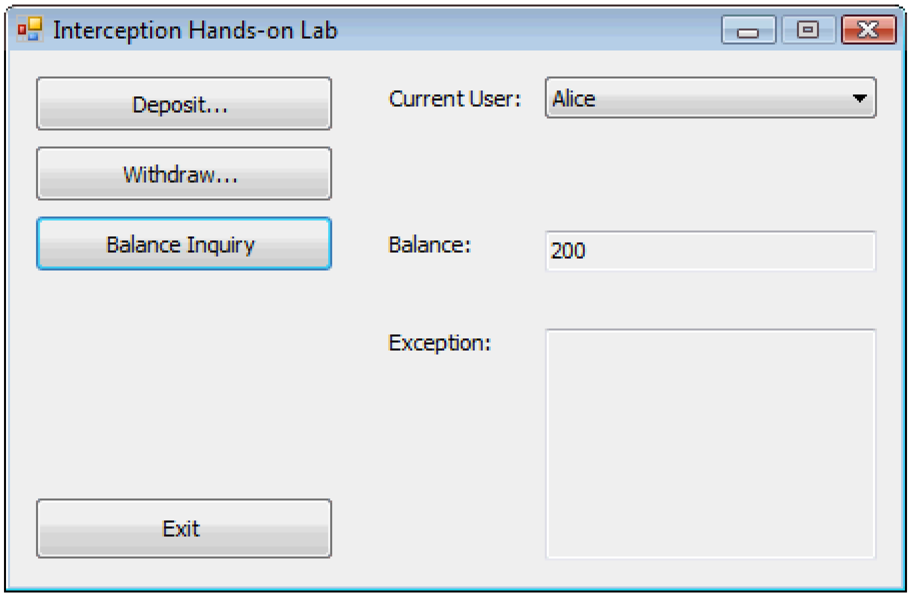


Current User: Alice

Balance:

Exception: Overflow or underflow in the arithmetic operation.

5. Perform a balance inquiry.



Current User: Alice

Balance: 200

Exception:

6. Open the interception.log file in the application's folder; it should contain a log of all the methods invoked on the **BankAccount** object used by the application. The contents of the log file should look like the following.

```
interception Information: 0 : Invoking Void Deposit(System.Decimal)
    DateTime=2008-11-27T16:18:12.1030778Z
interception Information: 0 : Successfully finished Void
Deposit(System.Decimal)
    DateTime=2008-11-27T16:18:12.1250778Z
interception Information: 0 : Invoking Void Withdraw(System.Decimal)
    DateTime=2008-11-27T16:18:44.7880778Z
```

```
interception Information: 0 : Finished Void Withdraw(System.Decimal)
with exception ArithmeticException: Overflow or underflow in the
arithmetic operation.
    DateTime=2008-11-27T16:18:44.9720778Z
interception Information: 0 : Invoking System.Decimal
GetCurrentBalance()
    DateTime=2008-11-27T16:19:24.2230778Z
interception Information: 0 : Successfully finished System.Decimal
GetCurrentBalance()
    DateTime=2008-11-27T16:19:24.2250778Z
```

Task 5: Make the target class interceptable by the VirtualMethodInterceptor

To make the BankAccount class interceptable

Update its definition to make it non-sealed and make its methods virtual by using the bold and highlighted code shown in the following excerpt. Remove the existing inheritance definition.

```
public ~~sealed~~ class BankAccount ~~: MarshalByRefObject~~
{
  private decimal balance;

  public virtual decimal GetCurrentBalance()
  {
    return balance;
  }

  public virtual void Deposit(decimal depositAmount)
  {
    balance += depositAmount;
  }

  public virtual void Withdraw(decimal withdrawAmount)
  {
    if (withdrawAmount > balance)
    {
      throw new ArithmeticException();
    }
    balance -= withdrawAmount;
  }
}
```

Task 6: Create a new intercepted instance using the VirtualMethodInterceptor

In this task an intercepted instance of a type derived from the **BankAccount** class is created using the **Intercept.NewInstance** method. Virtual method interception requires that you replace the original class with a new derived class before creating an instance. The **NewInstance** method creates the required derived class at runtime and then uses it to create a new instance.

To create a new intercepted instance

Invoke the **Intercept.NewInstance** method to create an intercepted instance of the **BankAccount** class in the constructor of the **MainForm** class. Provide a new instance of the **TransparentProxyInterceptor** as the interceptor and a new, properly configured instance of the custom behavior **TraceBehavior**, by using the bold and highlighted code shown in the following excerpt.

```
public MainForm()
{
  InitializeComponent();
  PopulateUserList();
  bankAccount =
        Intercept.NewInstance<BusinessLogic.BankAccount>(
            new VirtualMethodInterceptor(),
            new[] { new TraceBehavior(new TraceSource("interception")) });
}
```

Verification

The verification steps for virtual method interception are the same as the transparent proxy interception verification. The interception mechanism is different, but the pipeline of interception behaviors defines what happens when a method is intercepted. The different interception mechanisms intercept different methods.

To verify you have completed the lab correctly, you can use the solution provided in the Lab01\After folder.

Lab 2: Use a Unity Container to Perform Interception

Estimated time to complete this lab: **10 minutes**

In this lab, you will set up a Unity container to resolve intercepted instances, instead of using the **Intercept** class to create new instances or wrap existing ones. Using a container to perform interception is more powerful than using the stand-alone API. The container can set up interception on resolved dependencies and is not limited to just the top-level objects. In addition, it only needs to be configured once in order to apply interception to all instances encountered that must be resolved.

In this lab you will update the application to set up a container and resolve the **BankAccount** instance it uses. For information about the Unity container, see the topic Interception with Unity in the Unity 3 documentation.

Preparation

Continue working in the solution from the previous lab or open the solution file Lab02\Before\InterceptionHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Update the MainForm Class to Use a Unity Container to resolve an Intercepted Instance.

Task 1: Update the MainForm Class to Use a Unity Container to resolve an Intercepted Instance

To use and set up the container to perform interception

1. Add a **using** directive to the MainForm.cs file in the **InterceptionHOL** project to make the required types available without full name qualification. Add the following code to the beginning of the MainForm.cs file.

```
using Microsoft.Practices.Unity;
```

2. Add a **container** field to the **MainForm** class by using the bold and highlighted code shown in the following excerpt.

```
public partial class MainForm : Form
{
  private BusinessLogic.BankAccount bankAccount;
  private IUnityContainer container;
```

3. Add a new method with name **InitializeContainer** to initialize the value of the **container** variable as shown in the following code.

```
private void InitializeContainer()
{
  container = new UnityContainer();
}
```

4. Add the **Interception** extension to the newly created container as shown bold and highlighted in the following code.

```
private void InitializeContainer()
{
  container = new UnityContainer();
  container.AddNewExtension<Interception>();
}
```

5. Configure the container to perform interception when resolving the **BankAccount** class using the **RegisterType** method. Indicate the interception mechanism to use with an **Interceptor** object and the interception behavior to use with an **InterceptionBehavior** object by using the bold and highlighted code shown in the following excerpt.

```
private void InitializeContainer()
{
  container = new UnityContainer();
  container.AddNewExtension<Interception>();
  container.RegisterType<BusinessLogic.BankAccount>(
        new Interceptor<VirtualMethodInterceptor>(),
        new InterceptionBehavior(new TraceBehavior(
                                    new TraceSource("interception"))));
}
```

6. Update the constructor on the **MainForm** class to invoke the new **InitializeContainer** method by using the bold and highlighted code shown in the following excerpt.

```
public MainForm()
{
  InitializeComponent();
  InitializeContainer();
  PopulateUserList();
  bankAccount =
        Intercept.NewInstance<BusinessLogic.BankAccount>(
            new VirtualMethodInterceptor(),
            new[] { new TraceBehavior(new TraceSource("interception"))
});
}
```

7. Update the constructor on the **MainForm** class to resolve the **BankAccount** instance from the new container by using the bold and highlighted code shown in the following excerpt.

```
public MainForm()
```

```
{
    InitializeComponent();
    InitializeContainer();
    PopulateUserList();
    bankAccount = container.Resolve<BusinessLogic.BankAccount>();
}
```

8. Update **exitButton_Click** event handler method to dispose the container by using the bold and highlighted code shown in the following excerpt.

```
private void exitButton_Click(object sender, EventArgs e)
{
    container.Dispose();
    Application.Exit();
}
```

Verification

The verification steps are the same as the Lab 1 verification. This simple application can only illustrate the process for setting up interception. It cannot illustrate the benefits of using interception in a sophisticated real-life container. Applications that already use containers or that require more sophisticated interception functionality greatly benefit from using the interception approach described in this exercise.

To verify you have completed the lab correctly, you can use the solution provided in the Lab02\After folder.

Lab 3: Use Interface Interception

Estimated time to complete this lab: **15 minutes.**

In this lab, you will practice using the **InterfaceInterceptor** class. When using interface interception, the intercepted class must implement an interface and instances must be resolved through the interface, which requires setting up a type mapping in the container. For information about the different kinds of interceptors provided, see Interception with Unity in the Unity 3 documentation.

Preparation

Continue working in the solution from the previous lab or open the solution file Lab03\Before\InterceptionHOL.sln.

Procedures

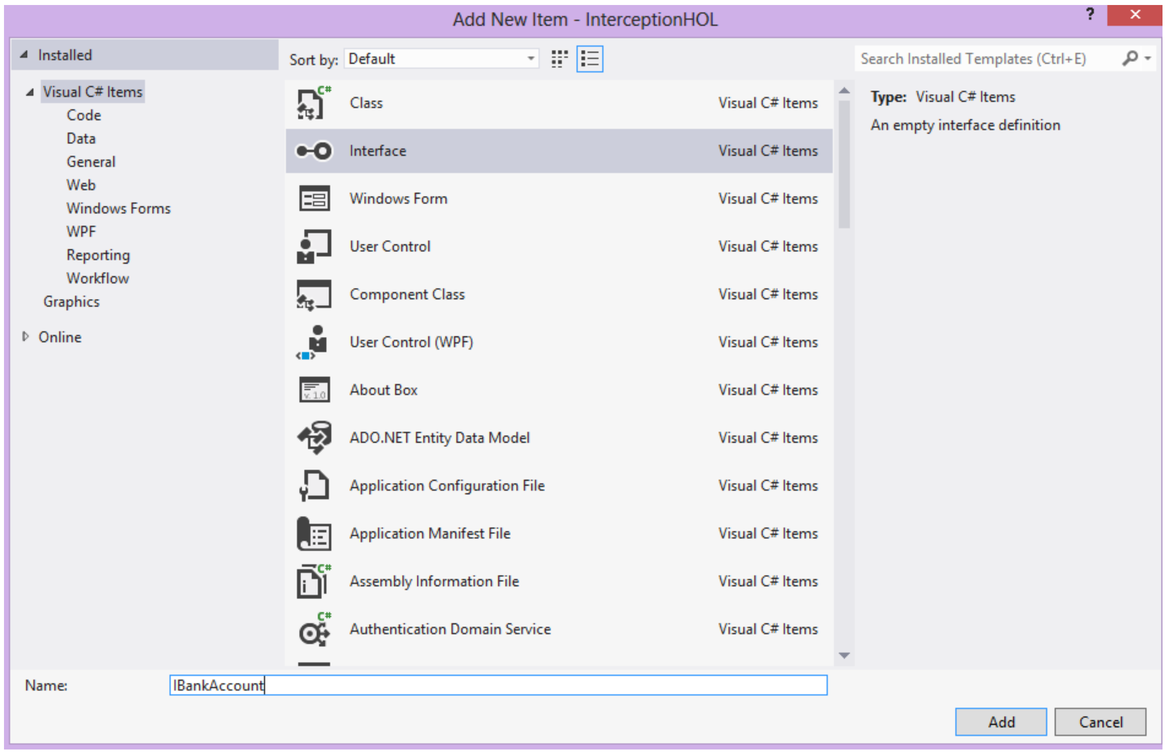
This lab includes the following tasks:

- Task 1: Create an **IBankAccount** interface.
- Task 2: Update the **BankAccount** class to implement the **IBankAccount** interface.
- Task 3: Replace references to **BankAccount** in **MainForm** with references to **IBankAccount.**
- Task 4: Update the Container Initialization Code.

Task 1: Create an IBankAccount Interface

To create the IBankAccount interface

1. Add the **IBankAccount** interface to the project. Right-click the **InterceptionHOL** project in Solution Explorer, point to **Add**, click **New Item**, select the **Interface** entry from the list of available templates, enter **IBankAccount** as the name, and then click **Add**.


Add New Item - InterceptionHOL
Installed
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
Workflow
Graphics
Online
Sort by: Default
Class — Visual C# Items
Interface — Visual C# Items
Windows Form — Visual C# Items
User Control — Visual C# Items
Component Class — Visual C# Items
User Control (WPF) — Visual C# Items
About Box — Visual C# Items
ADO.NET Entity Data Model — Visual C# Items
Application Configuration File — Visual C# Items
Application Manifest File — Visual C# Items
Assembly Information File — Visual C# Items
Authentication Domain Service — Visual C# Items
Search Installed Templates (Ctrl+E)
Add
Cancel

Type: Visual C# Items

An empty interface definition

Name: IBankAccount

2. Update the definition of the interface, as shown in the following bold and highlighted code.

```
namespace InterceptionHOL.BusinessLogic
{
  public interface IBankAccount
  {
    void Deposit(decimal depositAmount);
    decimal GetCurrentBalance();
    void Withdraw(decimal withdrawAmount);
  }
}
```

Task 2: Update the BankAccount Class to Implement the IBankAccount Interface

Because the **BankAccount** class already provides implementations for the methods in the new **IBankAccount** interface, you need only to change the class so it implements the **IBankAccount** interface, as shown in the following bold and highlighted code.

```
public class BankAccount : IBankAccount
{
  ...
}
```

Task 3: Replace References to BankAccount in MainForm with References to IBankAccount

Change the type for the **bankAccount** field and the type used in the **Resolve** call from **BankAccount** to **IBankAccount**, as shown in the following bold and highlighted code.

```
public partial class MainForm : Form
{
  private BusinessLogic.IBankAccount bankAccount;
  private IUnityContainer container;

  public MainForm()
  {
    InitializeComponent();
    InitializeContainer();
    PopulateUserList();
    bankAccount = container.Resolve<BusinessLogic.IBankAccount>();
  }
  ...
}
```

Task 4: Update the Container Initialization Code

Besides updating the interception mechanism, the container must be configured to map the new interface to the **BankAccount** class. To set up and use the container to perform interface interception, update the **RegisterType** call in the **InitializeContainer** method to map the **IBankAccount** interface to the **BankAccount** class and to use the **InterfaceInterceptor** as shown in the following bold and highlighted code.

```
private void InitializeContainer()
{
  container = new UnityContainer();
  container.AddNewExtension<Interception>();
  container.RegisterType<BusinessLogic.IBankAccount, BusinessLogic.BankAccount>(
        new Interceptor<InterfaceInterceptor>(),
        new InterceptionBehavior(new TraceBehavior(
            new TraceSource("interception"))));
}
```

Verification

The verification steps are the same as in the Lab 1 verification. The only difference is that the interception mechanism has changed.

To verify you have completed the lab correctly, you can use the solution provided in the Lab03\After folder.

Lab 4: Set Up Interception Using the Configuration File

Estimated time to complete this lab: **20 minutes.**

In this lab, you will practice setting up interception using a configuration file. For information about using configuration files to set up containers, see Configuration Files for Interception in the Unity 3 documentation. For information about the schema for the interception configuration elements, see Interception Configuration Schema Elements in the Unity 3 documentation.

Preparation

Continue working in the solution from the previous lab or open the solution file Lab04\Before\InterceptionHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Use configuration to set up the container.
- Task 2: Add container setup configuration to the configuration file.

Task 1: Use Configuration to Set Up the Container

A container can be set up, partially or entirely, using information retrieved from a configuration file, either the running application's configuration file or a different one. Advanced scenarios require accessing the information from the configuration file using the .NET Framework **ConfigurationManager** class. In simpler scenarios you simply need to invoke an extension method on the container to load configuration from the default locations. For information about the **ConfigurationManager** class, see ConfigurationManager Class on MSDN. For information about how to load configuration into a container, see Using Design-Time Configuration in the Unity 3 documentation.

In this lab, only the information from the configuration file is used to set up the container.

To use configuration from the configuration file to set up the container

1. Add a **using** directive to the MainForm.cs file in the **InterceptionHOL** project to make the required types available without full name qualification.

```
using Microsoft.Practices.Unity.Configuration;
```

2. Replace the container-initialization code to apply configuration from the configuration file, as shown in the following bold and highlighted code.

```
private void InitializeContainer()
{
  container = new UnityContainer();
  container.LoadConfiguration();
  //...
}
```

Task 2: Add Container Setup Configuration to the Configuration File

For information about Unity's configuration schema, see The Unity Configuration Schema in the Unity 3 documentation.

To add the necessary container set-up configuration to the app.config configuration file

1. Add the following bold and highlighted code to add the **unity** configuration section declaration.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="unity"
type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
Microsoft.Practices.Unity.Configuration"/>
  </configSections>
```

2. Add the following bold and highlighted code to add a new element for the **unity** configuration section as a child of the main **configuration** element. Specify the **xmlns** attribute to enable IntelliSense support in the VisualStudio XML editor.

```
  </system.diagnostics>
  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  </unity>
</configuration>
```

3. Add the following bold and highlighted code to add **assembly** and **namespace** entries for the types that will be used to configure the container. Although they are not required, they are often used to make Unity's configuration less verbose.

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <assembly name="InterceptionHOL"/>
  <namespace name="InterceptionHOL"/>
  <namespace name="InterceptionHOL.BusinessLogic"/>
  <namespace name="System.Diagnostics"/>
</unity>
```

4. Add the following bold and highlighted code to add a **sectionExtension** element to the **unity** section for the **InterceptionConfigurationExtension**. This extends the configuration schema with new element names and type aliases that are specific to interception.

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <assembly name="InterceptionHOL"/>
  <namespace name="InterceptionHOL"/>
  <namespace name="InterceptionHOL.BusinessLogic"/>
  <namespace name="System.Diagnostics"/>

  <sectionExtension
type="Microsoft.Practices.Unity.InterceptionExtension.Configuration.Inte
rceptionConfigurationExtension,
Microsoft.Practices.Unity.Interception.Configuration"/>
</unity>
```

5. Add the following bold and highlighted code to add a **container** element to the **unity** section for the default container. A container element with no **name** is considered the default. The Unity configuration section can be used to configure several containers provided that each container is provided a different name to store their configuration.

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <assembly name="InterceptionHOL"/>
  <namespace name="InterceptionHOL"/>
  <namespace name="InterceptionHOL.BusinessLogic"/>
  <namespace name="System.Diagnostics"/>

  <sectionExtension
type="Microsoft.Practices.Unity.InterceptionExtension.Configuration.Inte
rceptionConfigurationExtension,
Microsoft.Practices.Unity.Interception.Configuration"/>

  <container>
  </container>
</unity>
```

6. Add the following bold and highlighted code to add an **extension** element for the **Interception Extension** to the default container's configuration, using the **interception** type alias defined by the section extension. This is equivalent to invoking the **AddNewExtension** method in a container.

```
<container>
  <extension type="Interception"/>
</container>
```

7. Add the following bold and highlighted code to add a **register** element to map the **IBankAccount** interface to the **BankAccount** class. This is equivalent to invoking the **RegisterType** method in a container.

```
<container>
  <extension type="Interception"/>

  <register type="IBankAccount" mapTo="BankAccount">
  </register>
</container>
```

8. Add the following bold and highlighted code to add an **interceptor** element to indicate that interface interception should be used and an **interceptionBehavior** element to indicate that the

TraceBehavior should be invoked when intercepting the registered type. This is equivalent to supplying **Interceptor** and **InterceptionBehavior** objects in calls to the **RegisterType** method in a container.

```
<container>
  <extension type="Interception"/>

  <register type="IBankAccount" mapTo="BankAccount">
    <interceptor type="InterfaceInterceptor"/>
    <interceptionBehavior type="TraceBehavior"/>
  </register>
</container>
```

9. Add the following bold and highlighted code to add **register** elements to instruct the container on how to resolve the **TraceBehavior** indicated by the **interceptionBehavior** element and the **TraceSource** required by it. When setting up interception through API calls in Lab 1 a properly configured instance of the **TraceBehavior** was supplied, but when using the configuration file the instances need to be described so the container can build them.

```
<container>
  <extension type="Interception"/>

  <register type="IBankAccount" mapTo="BankAccount">
    <interceptor type="InterfaceInterceptor"/>
    <interceptionBehavior type="TraceBehavior"/>
  </register>

  <register type="TraceBehavior">
    <constructor>
      <param name="source" dependencyName="interception"/>
    </constructor>
  </register>

  <register type="TraceSource" name="interception">
    <constructor>
      <param name="name" value="interception"/>
    </constructor>
  </register>

</container>
```

10. You can compare the final configuration file you built with this lab with the version available at Lab04\After\InterceptionHOL\app.config to verify the changes.

Verification

The verification steps are the same as in the previous three labs verification. All that changed was the way the container is configured.

To verify you have completed the lab correctly, you can use the solution provided in the Lab04\After folder.

Lab 5: Create a Custom Interception Behavior

Estimated time to complete this lab: **20** minutes

In this lab, you will add a custom interception behavior to the application. This behavior will do a validation check on the amounts being deposited or withdrawn from the account.

Preparation

Continue working in the solution from the previous lab or open the solution file Lab05\Before\InterceptionHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Add the behavior code.
- Task 2: Add the behavior to the interception configuration.

Task 1: Add the behavior code

To create the behavior class

1. A behavior is simply an object that implements the **IInterceptionBehavior** interface. Create a new class named **AmountValidationBehavior**. To do this, right-click on the **InterceptionHOL** project in Solution Explorer, choose **Add**, and click **Class**. Enter **AmountValidationBehavior** as the name.
2. Add the following using statements at the top of the file to make required types available.

```
using System.Windows.Forms;
using Microsoft.Practices.Unity.InterceptionExtension;
```

3. Change the class definition so that it is public and implements the **IInterceptionBehavior** interface.

```
public class AmountValidationBehavior : IInterceptionBehavior
{

}
```

4. Add a private field to store the value of the maximum amount for parameters of the call to methods of the intercepted type.

```
public class AmountValidationBehavior : IInterceptionBehavior
```

```
{
  private decimal maxAmount;
}
```

5. Add the following implementations of the methods of the behavior to the new class.

```
public AmountValidationBehavior(decimal maxAmount)
{
  this.maxAmount = maxAmount;
}

public IMethodReturn Invoke(IMethodInvocation input,
                     GetNextInterceptionBehaviorDelegate getNext)
{
  if(input.Inputs.Count > 0)
  {
    foreach(var inputValue in input.Inputs)
    {
      if(inputValue is Decimal)
      {
        if((Decimal)inputValue > maxAmount)
        {
          MessageBox.Show(
            string.Format("Amount of {0} is beyond max limit of {1}",
              inputValue, maxAmount),
            "Limit Exceeded");
          return input.CreateExceptionMethodReturn(
            new InvalidOperationException("Limit Exceeded"));
        }
      }
    }
  }
  return getNext()(input, getNext);
}

public IEnumerable<Type> GetRequiredInterfaces()
{
  return Enumerable.Empty<Type>();
}

public bool WillExecute
{
  get { return true; }
}
```

This implementation goes through each parameter of each method called. If that parameter is of type **Decimal**, it checks to make sure the value is not higher than the amount passed in the constructor.

Task 2: Add the behavior to the interception configuration

The project is using the configuration file to set up interception. In this task we will edit the configuration to include this behavior in addition to the existing **TraceBehavior**.

To edit the configuration

1. Open the **app.config** file. Edit the registration for the **BankAccount** type to include the new behavior by adding the highlighted line.

```
<register type="IBankAccount" mapTo="BankAccount">
  <interceptor type="InterfaceInterceptor"/>
  <interceptionBehavior type="TraceBehavior"/>
  <interceptionBehavior type="AmountValidationBehavior"/>
</register>
```

2. Add a registration for the **AmountValidationBehavior** so that the **maxAmount** value is set to 125. Add a new **register** element to the container section to do this.

```
<container>
  ...
  <register type="AmountValidationBehavior">
    <constructor>
      <param name="maxAmount" value="125" />
    </constructor>
  </register>
</container>
```

Verification

To verify the tasks were completed successfully, run the application. The log files should still be generated as usual. Any deposits or withdrawals over 125 should result in a message box and an exception message in the log file.

To verify you have completed the lab correctly, you can use the solution provided in the Lab05\After folder.

Lab 6: Set Up Policy Injection Using Attributes

Estimated time to complete this lab: **10 minutes**

In this lab, you will practice using the policy injection behavior, and specifying policy injection by using attributes.

In this lab, you will update the simple Windows Forms application used in the previous labs and modify its behavior by using policy injection. Policy injection builds on the core interception features. It provides mechanisms to modify intercepted methods' behavior by consolidating **call handlers** specified through **policies** on a per-method basis. These call handlers are used by the **PolicyInjectionBehavior** object, which is invoked when a method on an intercepted object is invoked and the behavior has been specified for the intercepted object. For more information about Policy Injection, see Using Interception and Policy Injection in the Unity 3 documentation.

Preparation

Open the solution file Lab06\Before\InterceptionHOL.sln.

Procedures

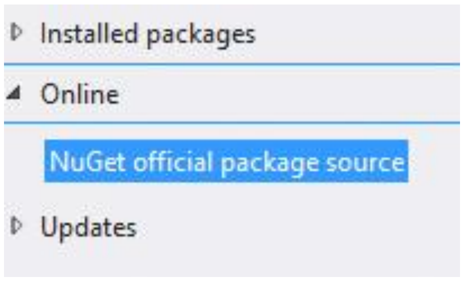
This lab includes the following tasks:

- Task 1: Add references to the necessary assemblies.
- Task 2: Make a call handler available.
- Task 3: Add call handler attributes to the target class's methods.
- Task 4: Set up the container to perform Policy Injection when intercepting.

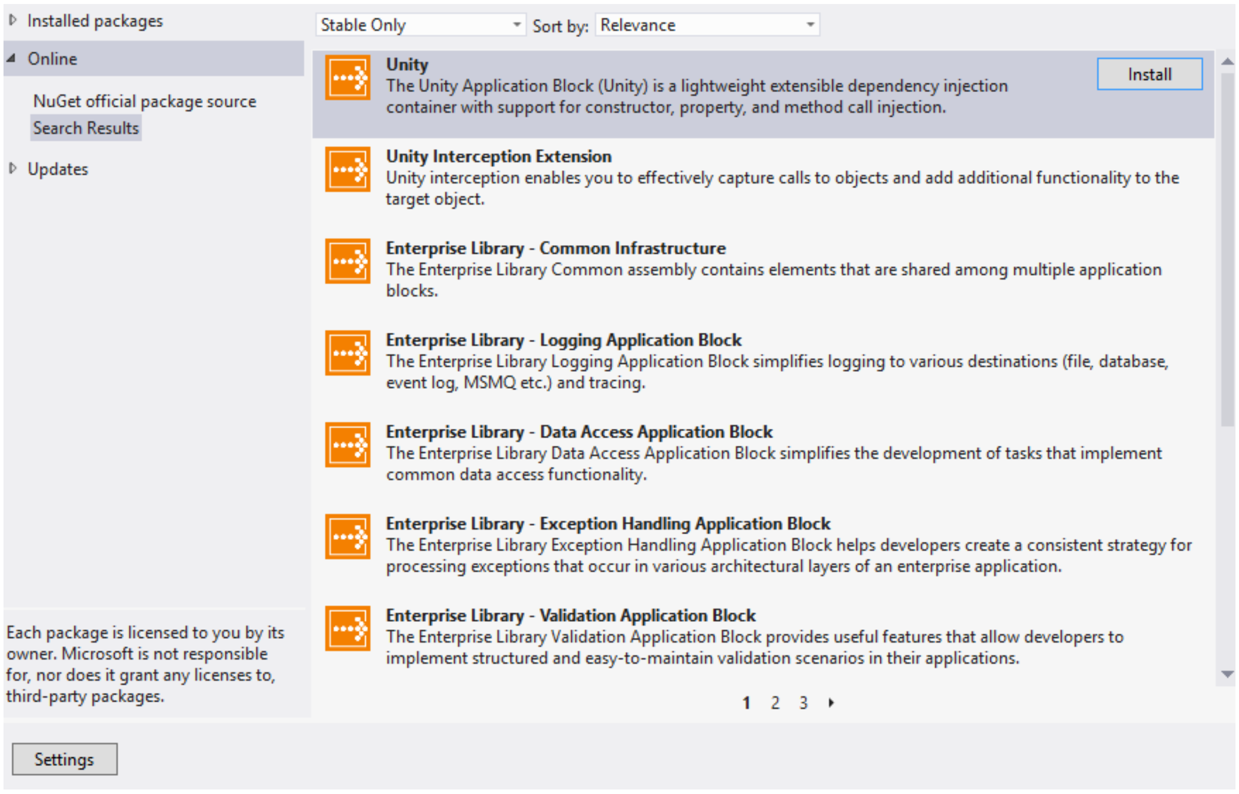
Task 1: Add References to the Necessary Assemblies

To add references

1. Select the **IntereptionHOL** project. Select the **Project | Manage NuGet Packages** menu command.
2. Select the "Online" option to view NuGet packages available online.



3. Search for **EntLib6** in the search bar. Select **Unity Interception Extension** and click install.



Unity
The Unity Application Block (Unity) is a lightweight extensible dependency injection container with support for constructor, property, and method call injection.

Unity Interception Extension
Unity interception enables you to effectively capture calls to objects and add additional functionality to the target object.

Enterprise Library - Common Infrastructure
The Enterprise Library Common assembly contains elements that are shared among multiple application blocks.

Enterprise Library - Logging Application Block
The Enterprise Library Logging Application Block simplifies logging to various destinations (file, database, event log, MSMQ etc.) and tracing.

Enterprise Library - Data Access Application Block
The Enterprise Library Data Access Application Block simplifies the development of tasks that implement common data access functionality.

Enterprise Library - Exception Handling Application Block
The Enterprise Library Exception Handling Application Block helps developers create a consistent strategy for processing exceptions that occur in various architectural layers of an enterprise application.

Enterprise Library - Validation Application Block
The Enterprise Library Validation Application Block provides useful features that allow developers to implement structured and easy-to-maintain validation scenarios in their applications.

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

4. Click **Accept** on the License Acceptance window that pops up.

Task 2: Make a Call Handler Available

Policy injection relies on implementations of the **ICallHandler** interface to perform actions on the intercepted method calls. These concrete **ICallHandler** implementations are used by the **PolicyInjectionBehavior** object shipped with Unity when intercepting a method. Unity does not include any concrete implementations of the **ICallHandler** interface. You can easily create your own call handler implementations that perform the interception actions you need, or use third-party handlers. For the purposes of this lab, you will use a custom call handler without delving into its implementation; Lab 9 addresses the creation of a custom call handler in detail.

The custom call handler used in this lab is similar to the custom interception behavior used in the previous labs. In fact, call handlers are very similar to interception behaviors and the main difference between them is how they are used: while behaviors act when any method from the intercepted object is invoked, call handlers only act when the associated matching rules match the invoked method.

To use the supplied call handler and its associated attribute

Add the TraceCallHandler.cs file to the **InterceptionHOL** project. As you did in Lab 1, right-click the **InterceptionHOL** project in the Solution Explorer, click **Add**, click **Existing Item**, select the TraceCallHandler.cs file, and then click **Add**.

Task 3: Add Call Handler Attributes to the Target Class's Methods

Annotating methods with call handler attributes is one way to indicate what actions to perform when an intercepted method is called during policy injection. Alternative approaches will be described in the following labs.

To add the attributes for the call handler provided in this lab, annotate each method in the **BankAccount** class with the **[TraceCallHandler("interception")]** attribute by adding the following bold and highlighted code. This causes all intercepted methods to invoke the custom call handler and log to the interception.log file (based on the configuration for the **interception TraceSource** in the configuration file).

```
public sealed class BankAccount : MarshalByRefObject
{
  private decimal balance;

  [TraceCallHandler("interception")]
  public decimal GetCurrentBalance()
  {
    return balance;
  }

  [TraceCallHandler("interception")]
  public void Deposit(decimal depositAmount)
  {
    balance += depositAmount;
  }

  [TraceCallHandler("interception")]
  public void Withdraw(decimal withdrawAmount)
  {
    if (withdrawAmount > balance)
    {
      throw new ArithmeticException();
    }
    balance -= withdrawAmount;
  }
}
```

Task 4: Set Up the Container to Perform Policy Injection When Intercepting

The final task in this lab is to configure the container to intercept resolved **BankAccount** instances by using the **PolicyInjectionBehavior** to perform policy injection.

To set up the container to perform interception

1. Add a **using** directive to the MainForm.cs file in the **InterceptionHOL** project. This makes the required types available without full name qualification.

   ```
   using Microsoft.Practices.Unity.InterceptionExtension;
   ```

2. Add the **Interception** extension to the container to enable the interception features by adding the following bold and highlighted code.

   ```
   private void InitializeContainer()
   {
     container = new UnityContainer();
     container.AddNewExtension<Interception>();
   }
   ```

3. Configure the container to perform interception when resolving the **BankAccount** class by using the **RegisterType** method. Indicate the interception mechanism to use with an **Interceptor** object and the **PolicyInjectionBehavior** with an **InterceptionBehavior** object. Add the following bold and highlighted code.

   ```
   private void InitializeContainer()
   {
     container = new UnityContainer();
     container.AddNewExtension<Interception>();
     container.RegisterType<BusinessLogic.BankAccount>(
           new InterceptionBehavior<PolicyInjectionBehavior>(),
           new Interceptor<TransparentProxyInterceptor>());
   }
   ```

Verification

In this section, you will validate that the specified interception behavior (logging each call to a method in the **BankAccount** type) is actually invoked.

To validate that the interception behavior is invoked

1. Build and run the **InterceptionHOL** project.
2. Perform some operations with the application as you did in Lab 1. Try to withdraw an amount higher than the current balance to cause an exception to be thrown.
3. Open the interception.log file in the application's folder. It should contain a log of all the methods invoked on the **BankAccount** object used by the application.

To verify you have completed the lab correctly, you can use the solution provided in the Lab06\After folder.

Lab 7: Set Up Policy Injection Using the Streamlined Interception Configuration API

Estimated time to complete this lab: **10 minutes**

In this lab, you will practice explicitly configuring policy injection to use rules and handlers at the container level, instead of annotating the intercepted types' methods with call handler attributes.

Although you can use both attributes and rules, or a combination of the two, in the container to specify policy injection, this lab will only use rules.

To configure policy injection in a container, you must define policies. These policies are regular objects that are resolved from the container by the **PolicyInjectionBehavior** when necessary and the general-purpose mechanisms used to set up a container, such as the **RegisterInstance** and **RegisterType** methods, can be used to make these policies available. However, the **Interception** extension provides a streamlined API that simplifies the definition of policies. For more information, see Registering Policy Injection Components in the Unity 3 documentation.

Policies consist of two parts: matching rules and call handlers. The matching rules determine whether the policy applies to a particular method. A policy applies when the checks from all its matching rules are satisfied. The call handlers from all the matching policies and those specified through attributes are merged into each intercepted method's handler pipeline.

The streamlined API provides three different ways you can specify matching rules and call handlers:

- A constrained version of the general purpose API used to specify injection.
- A constrained version of the general purpose API to register pre-created instances.
- A way to reference a rule or handler defined by using a general-purpose mechanism.

You can mix and match any of the above approaches. This lab only demonstrates the first option. For a description of all the different ways to specify matching rules and call handlers, see Run-Time Configuration in the Unity 3 documentation.

Specifying policies is not enough to actually perform policy injection on a resolved object, even if the rules specified for the policies match the resolved object's methods. You still must specify an interceptor object to indicate that the object is to be intercepted and you must indicate that the **PolicyInjectionBehavior** is to be used when intercepting the resolved object's methods.

Specifying policy injection using policies is more powerful than using attributes, because policies can target different methods from different types simultaneously. However, using attributes is usually simpler. Using attributes binds the policy injection specification to the intercepted method through code. Using policies decouples the intercepted class from the handlers, particularly when a configuration file is used to specify interception, as demonstrated in Lab 8: Set Up Policy Injection Using the Configuration File.

Preparation

Continue working in the solution from the previous lab or open the solution file Lab07\Before\InterceptionHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Remove the call handler attributes from the **BankAccount** class.
- Task 2: Add a policy for update methods.
- Task 3: Add a policy for query methods.

Task 1: Remove the Call Handler Attributes from the BankAccount Class

In this task you will remove the call handler attributes. The specification of what actions to perform when intercepting a **BankAccount** will reside elsewhere.

To remove the call handler attributes

1. Open the BankAccount.cs file from the **InterceptionHOL** project.
2. Remove the bold highlighted lines shown in the following code.

```
public sealed class BankAccount : MarshalByRefObject
{
  private decimal balance;

  ~~[TraceCallHandler("interception")]~~
  public decimal GetCurrentBalance()
  {
    return balance;
  }

  ~~[TraceCallHandler("interception")]~~
  public void Deposit(decimal depositAmount)
  {
    balance += depositAmount;
  }

  ~~[TraceCallHandler("interception")]~~
  public void Withdraw(decimal withdrawAmount)
  {
    if (withdrawAmount > balance)
    {
      throw new ArithmeticException();
    }
    balance -= withdrawAmount;
  }
}
```

Task 2: Add a Policy for Update Methods

Set up a policy targeting the **Deposit** and **Withdraw** methods from the **BankAccount**. Use the **TypeMatchingRule** targeting the **BankAccount** type and the **MemberNameMatchingRule** configured with the desired method names, and use a **TraceCallHandler** using the **interception-updates** trace source.

To add the policy

1. Open MainForm.cs.
2. Add a **using** directive for the System.Diagnostics namespace.

```
using System.Diagnostics;
```

3. Add the policy named **policy-updates** to the container through the **Interception** extension. Add the bold highlighted lines shown in the following code excerpt.

```
private void InitializeContainer()
{
  container = new UnityContainer();
  container.AddNewExtension<Interception>();
  container.RegisterType<BusinessLogic.BankAccount>(
        new InterceptionBehavior<PolicyInjectionBehavior>(),
        new Interceptor<TransparentProxyInterceptor>());
  container.Configure<Interception>()
    .AddPolicy("policy-updates");
}
```

4. Add a type matching rule to the policy using the **AddMatchingRule<T>** method. Add the bold highlighted lines shown in the following code shows to use injection for defining a policy.

```
private void InitializeContainer()
{
  container = new UnityContainer();
  container.AddNewExtension<Interception>();
  container.RegisterType<BusinessLogic.BankAccount>(
        new InterceptionBehavior<PolicyInjectionBehavior>(),
        new Interceptor<TransparentProxyInterceptor>());
  container.Configure<Interception>()
    .AddPolicy("policy-updates")
        .AddMatchingRule<TypeMatchingRule>(
            new InjectionConstructor(
                new
InjectionParameter(typeof(BusinessLogic.BankAccount))));
}
```

5. Add a member name matching rule to the policy, chaining a new call to the **AddMatchingRule<T>** method to the one added in the previous step. Add the bold highlighted lines shown in the following code excerpt.

```
private void InitializeContainer()
{
  container = new UnityContainer();
  container.AddNewExtension<Interception>();
  container.RegisterType<BusinessLogic.BankAccount>(
        new InterceptionBehavior<PolicyInjectionBehavior>(),
        new Interceptor<TransparentProxyInterceptor>());
  container.Configure<Interception>()
    .AddPolicy("policy-updates")
        .AddMatchingRule<TypeMatchingRule>(
            new InjectionConstructor(
                new
InjectionParameter(typeof(BusinessLogic.BankAccount))))
        .AddMatchingRule<MemberNameMatchingRule>(
            new InjectionConstructor(
                new InjectionParameter(new string[]
                                       { "Deposit", "Withdraw" })));
}
```

6. Add the call handler to the policy by using the **AddCallHandler<T>** method. Using injection for the streamlined API, the **TraceCallHandler** is defined by specifying a constructor call. Add the bold highlighted lines shown in the following code excerpt.

```
private void InitializeContainer()
{
  container = new UnityContainer();
  container.AddNewExtension<Interception>();
  container.RegisterType<BusinessLogic.BankAccount>(
        new InterceptionBehavior<PolicyInjectionBehavior>(),
        new Interceptor<TransparentProxyInterceptor>());
  container.Configure<Interception>()
    .AddPolicy("policy-updates")
        .AddMatchingRule<TypeMatchingRule>(
            new InjectionConstructor(
                new
InjectionParameter(typeof(BusinessLogic.BankAccount))))
        .AddMatchingRule<MemberNameMatchingRule>(
            new InjectionConstructor(
                new InjectionParameter(new string[]
                                       { "Deposit", "Withdraw" })))
        .AddCallHandler<TraceCallHandler>(
            new ContainerControlledLifetimeManager(),
            new InjectionConstructor(
                new TraceSource("interception-updates")));
}
```

In this task, a **ContainerControlledLifetimeManager** is specified for the call handler, which causes the same call handler instance to be shared by all of the methods.

Task 3: Add a Policy for Query Methods

Set up a policy targeting the **GetCurrentBalance** method from the **BankAccount** with a **TraceCallHandler** using the **interception-queries** trace source as in the previous task. In this task you will use the **Interception** property before defining a new policy to terminate the definition of the previous policy. Add the bold highlighted lines shown in the following code excerpt.

```
private void InitializeContainer()
{
  container = new UnityContainer();
  container.AddNewExtension<Interception>();
  container.RegisterType<BusinessLogic.BankAccount>(
        new InterceptionBehavior<PolicyInjectionBehavior>(),
        new Interceptor<TransparentProxyInterceptor>());
  container.Configure<Interception>()
    .AddPolicy("policy-updates")
       .AddMatchingRule<TypeMatchingRule>(
            new InjectionConstructor(
                new InjectionParameter(typeof(BusinessLogic.BankAccount))))
        .AddMatchingRule<MemberNameMatchingRule>(
            new InjectionConstructor(
                new InjectionParameter(new string[]
                                       { "Deposit", "Withdraw" })))
        .AddCallHandler<TraceCallHandler>(
            new ContainerControlledLifetimeManager(),
            new InjectionConstructor(
                new TraceSource("interception-updates")))
    .Interception
    .AddPolicy("policy-query")
        .AddMatchingRule<TypeMatchingRule>(
            new InjectionConstructor(
                new InjectionParameter(typeof(BusinessLogic.BankAccount))))
        .AddMatchingRule<MemberNameMatchingRule>(
            new InjectionConstructor("GetCurrentBalance"))
        .AddCallHandler<TraceCallHandler>(
            new ContainerControlledLifetimeManager(),
            new InjectionConstructor(
                new TraceSource("interception-queries")));
}
```

Verification

In this section, you will validate that the interception behavior specified with the different policies is actually invoked.

To validate that the interception behavior is invoked:

1. Build and run the InterceptionHOL project.
2. Perform some operations with the application. Try to withdraw an amount higher than the current balance to cause an exception to be thrown as you did in Lab 1.
3. Open the interception-updates.log and interception-queries.log files in the application's folder. They should contain a log of all the methods invoked on the **BankAccount** object used by the application.

To verify you have completed the lab correctly, you can use the solution provided in the Lab07\After folder.

Lab 8: Set Up Policy Injection Using the Configuration File

Estimated time to complete this lab: **15 minutes**

In this lab, you will practice setting up policy injection by using a configuration file. Specifying policies in the configuration file is quite similar to specifying them by using the streamlined API. Because policy injection requires interception, some of the changes required in the configuration file are similar to those performed in Lab 4. For information about the schema for the policy-injection-specific configuration elements, see Interception Configuration Schema Elements in the Unity 3 documentation.

Preparation

Open the solution file Lab08\Before\InterceptionHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Add support for interception in the configuration file.
- Task 2: Indicate the interceptor and interception behavior for the **BankAccount** type.
- Task 3: Specify the policies for the container.

Task 1: Add Support for Interception in the Configuration File

As in the previous configuration-related lab, you must enable interception support for the container.

To enable interception support for the container

1. Open the **app.config** file from the **InterceptionHOL** project.
2. Add the interception extension to the container. Insert the following bold and highlighted lines of code into the configuration file.

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <alias alias="TraceSource" type="System.Diagnostics.TraceSource,
System, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"/>
  <alias alias="BankAccount"
type="InterceptionHOL.BusinessLogic.BankAccount, InterceptionHOL"/>

  <sectionExtension
type="Microsoft.Practices.Unity.InterceptionExtension.Configuration.Inte
rceptionConfigurationExtension,
Microsoft.Practices.Unity.Interception.Configuration"/>
```

```
  <container>
    <extension type="Interception"/>

    <register type="BankAccount">
    </register>
  </container>
</unity>
```

Task 2: Indicate the interceptor and interception behavior for the BankAccount type

As in the previous configuration-related lab, you must opt-in for interception when resolving an instance of **BankAccount**. In this task set the interception behavior to the **PolicyInjectionBehavior**. Add the following bold and highlighted lines of code into the configuration file.

```
<container>
  <extension type="Interception"/>

  <register type="BankAccount">
    <interceptionBehavior type="PolicyInjectionBehavior"/>
    <interceptor type="TransparentProxyInterceptor"/>
  </register>
</container>
```

Task 3: Specify the Policies for the Container

In order to specify policies through the configuration file you must indicate how different objects should be created. This is same requirement applies when specifying policies with the streamlined API. For details on how injection is specified through a configuration file see Specifying Values for Injection in the Unity 3 documentation.

To validate that the interception behavior is invoked:

1. Add the **interception** element to the container as shown in the following bold and highlighted lines of code. This element is enabled by adding the interception section extension.

```
<container>
  <extension type="Interception"/>

  <register type="BankAccount">
    <interceptionBehavior type="PolicyInjectionBehavior"/>
    <interceptor type="TransparentProxyInterceptor"/>
  </register>

  <interception>
  </interception>
</container>
```

2. Add a **policy** element to the **interception** element with name **"policy-updates"** as shown in the following bold and highlighted lines of code.

```
<container>
  <extension type="Interception"/>

  <register type="BankAccount">
    <interceptionBehavior type="PolicyInjectionBehavior"/>
    <interceptor type="TransparentProxyInterceptor"/>
  </register>

  <interception>
    <policy name="policy-updates">
    </policy>
  </interception>
</container
```

3. Add **matchingRule** elements for the type and member name matching rules as shown in the following bold and highlighted lines of code. These elements require configuration which is specified through the standard **constructor** element.

```
<container>
  <extension type="Interception"/>

  <register type="BankAccount">
    <interceptionBehavior type="PolicyInjectionBehavior"/>
    <interceptor type="TransparentProxyInterceptor"/>
  </register>

  <interception>
    <policy name="policy-updates">
      <matchingRule name="updates-rule1" type="TypeMatchingRule">
        <constructor>
          <param name="typeName"
                 value="InterceptionHOL.BusinessLogic.BankAccount" />
        </constructor>
      </matchingRule>
      <matchingRule name="updates-rule2" type="MemberNameMatchingRule">
        <constructor>
          <param name="namesToMatch">
            <array type="string[]">
              <value value="Deposit" />
              <value value="Withdraw" />
            </array>
          </param>
        </constructor>
      </matchingRule>
    </policy>
  </interception>
```

```
</container>
```

4. Add a **callHandler** element for custom call handler as shown in the following bold and highlighted lines of code. In this example, the **lifetime** element is used in addition to the **constructor** element.

```
<container>
  <extension type="Interception"/>

  <register type="BankAccount">
    <interceptionBehavior type="PolicyInjectionBehavior"/>
    <interceptor type="TransparentProxyInterceptor"/>
  </register>

  <interception>
    <policy name="policy-updates">
      <matchingRule name="updates-rule1" type="TypeMatchingRule">
        <constructor>
          <param name="typeName"
                 value="InterceptionHOL.BusinessLogic.BankAccount" />
        </constructor>
      </matchingRule>
      <matchingRule name="updates-rule2" type="MemberNameMatchingRule">
        <constructor>
          <param name="namesToMatch">
            <array type="string[]">
              <value value="Deposit" />
              <value value="Withdraw" />
            </array>
          </param>
        </constructor>
      </matchingRule>
      <callHandler name="updates-handler1"
                   type="InterceptionHOL.TraceCallHandler,
InterceptionHOL">
        <lifetime type="singleton" />
        <constructor>
          <param name="source" dependencyName="interception-updates"/>
        </constructor>
      </callHandler>
    </policy>
  </interception>
</container
```

5. Add a **policy** element for the query methods as shown in the following bold and highlighted lines of code.

```
<container>
  <extension type="Interception"/>
  <register type="BankAccount">
    <interceptionBehavior type="PolicyInjectionBehavior"/>
    <interceptor type="TransparentProxyInterceptor"/>
  </register>

  <interception>
    <policy name="policy-updates">
      <matchingRule name="updates-rule1" type="TypeMatchingRule">
        <constructor>
          <param name="typeName"
                 value="InterceptionHOL.BusinessLogic.BankAccount" />
        </constructor>
      </matchingRule>
      <matchingRule name="updates-rule2" type="MemberNameMatchingRule">
        <constructor>
          <param name="namesToMatch">
            <array type="string[]">
              <value value="Deposit" />
              <value value="Withdraw" />
            </array>
          </param>
        </constructor>
      </matchingRule>
      <callHandler name="updates-handler1"
                   type="InterceptionHOL.TraceCallHandler,
InterceptionHOL">
        <lifetime type="singleton" />
        <constructor>
          <param name="source" dependencyName="interception-updates"/>
        </constructor>
      </callHandler>
    </policy>
    <policy name="policy-queries">
      <matchingRule name="queries-rule1" type="TypeMatchingRule">
        <constructor>
          <param name="typeName"
                 value="InterceptionHOL.BusinessLogic.BankAccount" />
        </constructor>
      </matchingRule>
      <matchingRule name="queries-rule2" type="MemberNameMatchingRule">
        <constructor>
          <param name="nameToMatch" value="GetCurrentBalance" />
        </constructor>
      </matchingRule>
      <callHandler name="queries-handler1"
                   type="InterceptionHOL.TraceCallHandler,
InterceptionHOL">
        <lifetime type="singleton" />
        <constructor>
```

```
          <param name="source" dependencyName="interception-queries"/>
        </constructor>
      </callHandler>
    </policy>
  </interception>
</container>
```

6. Configure the container to resolve the **TraceSource** objects required by the call handlers.

```
<container>
  <extension type="Interception"/>

  <register type="BankAccount">
    <interceptionBehavior type="PolicyInjectionBehavior"/>
    <interceptor type="TransparentProxyInterceptor"/>
  </register>

  <register name="interception-updates" type="TraceSource">
    <constructor>
      <param name="name" value="interception-updates" />
    </constructor>
  </register>
  <register name="interception-queries" type="TraceSource">
    <constructor>
      <param name="name" value="interception-queries" />
    </constructor>
  </register>

  <interception>
```

Verification

The verification steps are the same as in Lab 1. The only difference is that the container configuration has changed.

To verify you have completed the lab correctly, you can use the solution provided in the Lab08\After folder.

Lab 9: Implement a Custom Call Handler

Estimated time to complete this lab: **20 minutes**

Purpose

In this lab you will create a new call handler and add it to a policy. The new call handler will perform a very simple role-based access check on the intercepted methods by using entries in the standard **appSettings** configuration to establish which users belong to which roles.

Preparation

Open the solution file Lab09\Before\InterceptionHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Create the call handler class.
- Task 2: Set up a new policy in the configuration file using the new call handler.
- Task 3: Specify the order of call handlers in the configuration file.

Task 1: Create the Call Handler Class

A call handler is simply a class that implements the **ICallHandler** interface defined in the **Microsoft.Practices.Unity.Interception** assembly. For information about how this interface should be implemented, see Creating Interception Policy Injection Call Handlers in the Unity 3 documentation.

To create the call handler class

1. Add a new class with the name **AccessCheckCallHandler** to the InterceptionHOL project by adding the following bold and highlighted lines of code. Call handlers, as well as matching rules, can reside in any assembly. For the sake of convenience, you are using a single project for both the call handler and the intercepted type.

2. Add the following **using** directives to the new **AccessCheckCallHandler.cs** file in the InterceptionHOL project to make the required types available without full name qualification.

```
using System.Security.Principal;
using System.Threading;
using Microsoft.Practices.Unity.InterceptionExtension;
using Microsoft.Practices.Unity.Utility;
```

3. Update the **AccessCheckCallHandler** class to be publicly accessible and to implement the **ICallHandler** interface by adding the following bold and highlighted lines of code.

```
public class AccessCheckCallHandler : ICallHandler
```

```
{
```

4. Define a field to hold the allowed roles and a constructor to initialize it.

```
public class AccessCheckCallHandler : ICallHandler
{
  private string[] allowedRoles;

  public AccessCheckCallHandler(params string[] allowedRoles)
  {
    Guard.ArgumentNotNull(allowedRoles, "allowedRoles");
    this.allowedRoles = allowedRoles;
  }
}
```

5. Implement the **Order** property required by the **ICallHandler** interface by adding the following lines of code.

```
private int order;
public int Order
{
  get
  {
    return order;
  }
  set
  {
    this.order = value;
  }
}
```

6. Implement the **Invoke** method required by the **ICallHandler** interface by adding the following code. The method checks whether the current thread's principal belongs to one of the roles allowed by the call handler's configuration. If the check fails, the intercepted method is not invoked and an exception is thrown.

```
public IMethodReturn Invoke(
                     IMethodInvocation input,
                     GetNextHandlerDelegate getNext)
{
  if (this.allowedRoles.Length > 0)
  {
    IPrincipal currentPrincipal = Thread.CurrentPrincipal;

    if (currentPrincipal != null)
    {
      bool allowed = false;
      foreach (string role in this.allowedRoles)
      {
        if (allowed = currentPrincipal.IsInRole(role))
        {
          break;
        }
      }

      if (!allowed)
      {
        // short circuit the call
        return input.CreateExceptionMethodReturn(
            new UnauthorizedAccessException(
                "User not allowed to invoke the method"));
      }
    }
  }

    return getNext()(input, getNext);
}
```

There are two call handler–specific aspects to consider in this implementation:

- Instead of throwing an exception, an exception method return is created and returned.
- Call handlers must explicitly invoke the next step in the call handler chain by invoking the delegate that results from invoking the **getNext** delegate parameter.

Task 2: Set Up a New Policy in the Configuration File Using the New Call Handler

Set up a new policy to target only the **Withdraw** method in the **BankAccount** class. Use the **TypeMatchingRule** and **MemberNameMatchingRule** matching rules to add the new **AccessCheckCallHandler** allowing only the **Teller** role. Make this the first entry in the **<interception>** element in the configuration file. After adding this policy there will be two separate policies targeting the **Withdraw** method that will supply call handlers to its handler pipeline. Add the following bold and highlighted lines of code.

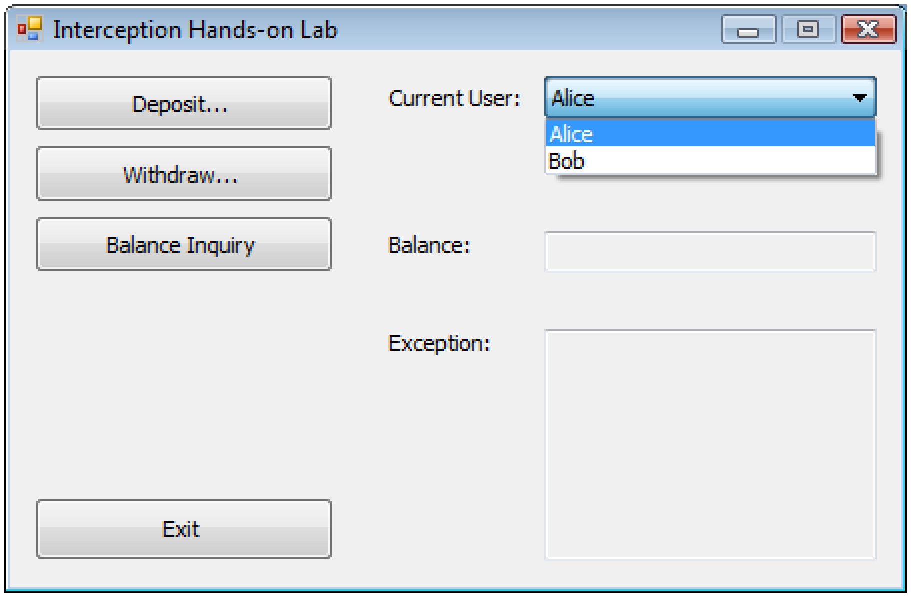
```
<interception>
  <policy name="policy-withdraw">
    <matchingRule name="withdraw-rule1" type="TypeMatchingRule">
      <constructor>
        <param name="typeName">
          <value value="InterceptionHOL.BusinessLogic.BankAccount" />
        </param>
      </constructor>
    </matchingRule>
    <matchingRule name="withdraw-rule2" type="MemberNameMatchingRule">
      <constructor>
        <param name="namesToMatch">
```

```
          <array type="string[]">
            <value value="Withdraw" />
          </array>
        </param>
      </constructor>
    </matchingRule>
    <callHandler name="withdraw-handler1"
                 type="InterceptionHOL.AccessCheckCallHandler, InterceptionHOL">
      <lifetime type="singleton" />
      <constructor>
        <param name="allowedRoles">
          <array type="string[]">
            <value value="Teller"/>
          </array>
        </param>
      </constructor>
    </callHandler>
  </policy>
  <policy name="policy-updates">
    ...
  </policy>
  <policy name="policy-queries">
    ...
  </policy>
</interception
```

Verification

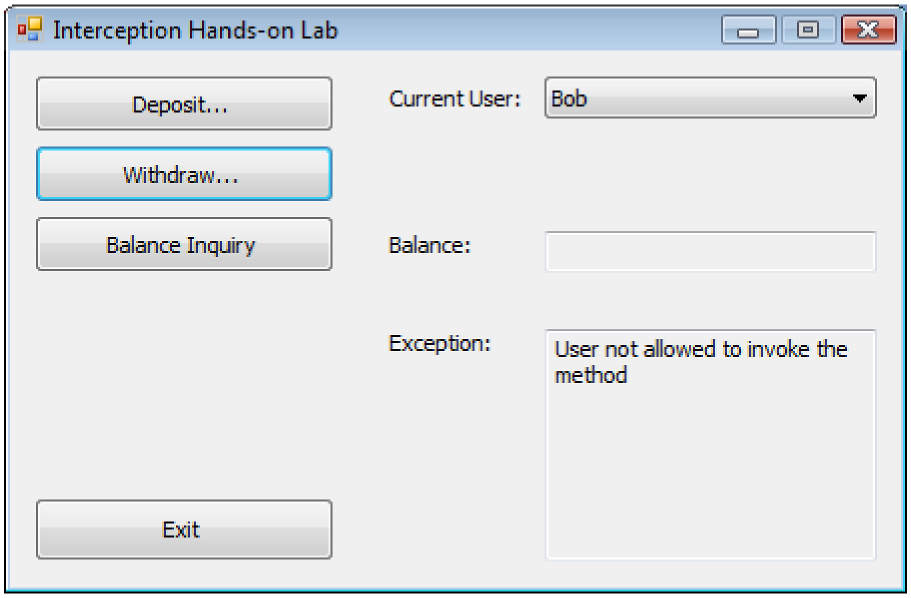
In this section, you will validate that the new call handler is used when the **Withdraw** method is invoked, and that an exception is thrown when a user who does not belong to the allowed Teller role invokes attempts to perform a withdrawal.

The default configuration for the lab assigns the **Teller** role to the user Alice and the **Assistant** role to the user Bob; you can use the drop-down list box in the upper-right corner of the lab's main window to switch the current user while the application runs, as illustrated in the following figure.



To validate that the interception behavior is invoked

1. Build and run the InterceptionHOL project.
2. Perform some actions as the default current user Alice, including a withdrawal for an amount higher than the current balance, which should result in an exception.
3. Switch the current user to Bob.
4. Attempt to perform a withdrawal for an amount lower than the current balance; you should see that an access denied exception is thrown. Deposits should succeed. The following figure illustrates an **UnauthorizedAccessException** is thrown.



User not allowed to invoke the method

Inspecting the interception-updates.log file should show entries for the successful updates and the exceptions caused by withdrawal attempts for an amount higher than the balance. The log should not show attempted withdrawals that failed because of authorization failures. This is a consequence of the position of the new call handler in the call handlers' pipeline. The new call handler appears first in the pipeline, because of the position of the policy defining the handler in the configuration file. When the access check fails and the call is short-circuited with an exception return, the tracing call handler is not invoked. Although entries in the configuration file could be rearranged to position the tracing call handler earlier in the call handlers' pipeline, this is not always desirable, particularly if policies define several handlers that should be intertwined.

Task 3: Specify the Order of Call Handlers in the Configuration File

Still working in the configuration file, set **1** as the value for the **Order** property of the **updates-handler1** handler from the **policy-updates** policy, as shown in the following bold and highlighted line of code. Setting an explicit order for the trace call handler would move it before other call handlers in the pipeline that do not have an explicit order set.

```
<policy name="policy-updates">
  <matchingRule name="updates-rule1" type="TypeMatchingRule">
    ...
  </matchingRule>
  <matchingRule name="updates-rule2" type="MemberNameMatchingRule">
    ...
  </matchingRule>
  <callHandler name="updates-handler1"
               type="InterceptionHOL.TraceCallHandler, InterceptionHOL">
    <lifetime type="singleton" />
    <constructor>
      <param name="source" dependencyName="interception-updates"/>
    </constructor>
    <property name="Order" value="1"/>
  </callHandler>
</policy>
```

Verification

In this section, you will validate that the new call handler is used when the **Withdraw** method is invoked, that an exception is thrown when a user who does not belong to the allowed **Teller** role invokes attempts to perform a withdrawal and it is displayed in the **Exception** text area, and that exceptions caused by failed access checks are properly logged.

To validate that the interception behavior is invoked

1. Build and run the InterceptionHOL project.
2. Perform some actions as the default current user Alice, including a withdrawal for an amount higher than the current balance which should result in an exception.
3. Switch the current user to Bob.

4. Attempt to perform a withdrawal for an amount lower than the current balance; you should see that an **UnauthorizedAccessException** is thrown.

Inspecting the interception-updates.log file should show entries for all updates attempted, including the exceptions caused by access-check failures.

To verify you have completed the lab correctly, you can use the solution provided in the Lab09\After folder.

Lab 10: Use the Policy Injection Application Block to Perform Interception

Estimated time to complete this lab: **20 minutes**

Purpose

In this lab, you will use the Policy Injection Application Block to perform interception.

The Policy Injection Application Block provides a simplified API for performing interception. It uses Unity's interception and policy injection support for its implementation. Beside its simplicity, one of the reasons for using the Policy Injection Application Block instead of Unity's interception support is that the Policy Injection Application Block can be configured by using the Enterprise Library configuration tool. The drawback is that the Policy Injection Application Block can only perform simple object creation (invoking a constructor with the supplied parameters), only uses transparent proxy interception and does not allow for interception behaviors other than policy injection.

While Unity does not ship with call handlers, the Policy Injection Application Block does and the **PolicyInjection** façade is not required in order to use these call handlers. They can also be used with the stand-alone Interception API or with a properly configured Unity container.

For more information about the Policy Injection Application Block, see The Policy Injection Application Block in the Enterprise Library 6 documentation.

For more information about Interception and Policy Injection see Using Interception and Policy Injection in the Unity 3 documentation.

Preparation

Open the solution file Lab10\Before\InterceptionHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Add references to the necessary assemblies.
- Task 2: Use the **PolicyInjection** façade to create a **BankAccount** instance.
- Task 3: Configure policy injection using the **Enterprise Library Configuration Tool.**

Task 1: Add References to the Necessary Assemblies

Add the NuGet package for **Enterprise Library – Policy Injection Application Block.** The Logging Application Block NuGet package has already been included in this project. While the Logging Block will not be used directly, the Logging call handler that will be used in this lab is defined in the Logging block's assembly.

Task 2: Use the PolicyInjection Façade to Create a BankAccount Instance

The **PolicyInjection** façade can be used to create or wrap instances. For the purposes of this lab, a new instance of the **BankAccount** class will be created. For information about creating instances with the **PolicyInjection** façade, see Creating an Instance of an Interceptable Target Class on MSDN.

To use the PolicyInjection façade to create the BankAccount instance

1. Add a **using** directive to the MainForm.cs file in the InterceptionHOL project to make the required types available without full name qualification as shown in the following line of code.

```
using Microsoft.Practices.EnterpriseLibrary.PolicyInjection;
using Microsoft.EnterpriseLibrary.Logging.PolicyInjection;
using Microsoft.Practices.Unity;
using Microsoft.Practices.Unity.InterceptionExtension;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration.Unity;
```

2. Use the **PolicyInjection.Create** method to create the **BankAccount** instance in the **MainForm** class's constructor. Add the following bold and highlighted code.

```
public MainForm()
{
  InitializeComponent();
  PopulateUserList();
  ConfigureLogging();
  IUnityContainer container = new UnityContainer();
  container.AddNewExtension<Interception>();

container.AddNewExtension<TransientPolicyBuildUpExtension>();
      PolicyInjection.SetPolicyInjector(new PolicyInjector(container),
      false);
  bankAccount = PolicyInjection.Create<BusinessLogic.BankAccount>();
}
```

Task 3: Configure Policy Injection Programmatically

To configure policy injection using the configuration tool

1. Configure the **Interception** extension on your **Unity Container**. Add a **Policy** called "Policy" to it.

```
public MainForm()
{
    InitializeComponent();
    PopulateUserList();
    ConfigureLogging();
    IUnityContainer container = new UnityContainer();
    container.AddNewExtension<Interception>();

    container.AddNewExtension<TransientPolicyBuildUpExtension>();
    container.Configure<Interception>()
       .AddPolicy("Policy");
    PolicyInjection.SetPolicyInjector(new PolicyInjector(container),
       false);
    bankAccount = PolicyInjection.Create<BusinessLogic.BankAccount>();
}
```

2. Add a Matching Rule to your configuration of type **TypeMatchingRule** for the **BankAccount** class.

```
public MainForm()
{
     InitializeComponent();
     PopulateUserList();
     ConfigureLogging();
     IUnityContainer container = new UnityContainer();
     container.AddNewExtension<Interception>();
     container.AddNewExtension<TransientPolicyBuildUpExtension>();
     container.Configure<Interception>()
         .AddPolicy("Policy")
         .AddMatchingRule<TypeMatchingRule>("Type Matching Rule",
              new InjectionConstructor(
                  new InjectionParameter(
                      typeof(BusinessLogic.BankAccount)))
          );
     PolicyInjection.SetPolicyInjector(new PolicyInjector(container),
          false);
     bankAccount = PolicyInjection.Create<BusinessLogic.BankAccount>();
}
```

3. Add a **Log Call Handler** to your configuration as specified below.

```
public MainForm()
{
     InitializeComponent();
     PopulateUserList();
     ConfigureLogging();
     IUnityContainer container = new UnityContainer();
     container.AddNewExtension<Interception>();

     container.AddNewExtension<TransientPolicyBuildUpExtension>();
          container.Configure<Interception>()
          .AddPolicy("Policy")
          .AddMatchingRule<TypeMatchingRule>("Type Matching Rule",
                new InjectionConstructor(
                     new InjectionParameter(
                         typeof(BusinessLogic.BankAccount)))
            )
          .AddCallHandler<LogCallHandler>(
                new ContainerControlledLifetimeManager(),
                new InjectionConstructor(0, true, true,
                    "Before Invoking","After Invoking",
                    false,true,true,-1,1));
     PolicyInjection.SetPolicyInjector(new PolicyInjector(container),
          false);
     bankAccount = PolicyInjection.Create<BusinessLogic.BankAccount>();
}
```

This creates a **Log Call Handler** with an **EventId** of 0 that logs both before and after method calls with a before message of "Before Invoking" and an after message of "After Invoking". It is set to include call time and call stack with a priority of -1.

Verification

In this section, you will validate that the standard **LoggingCallHandler** is invoked when the operations in the **BankAccount** class are executed.

To validate that the interception behavior is invoked

1. Build and run the **InterceptionHOL** project.
2. Perform some operations with the application.
3. Open the trace.log file in the application's folder; it should contain a log of all the methods invoked on the **BankAccount** object used by the application.

To verify you have completed the lab correctly, you can use the solution provided in the Lab10\After folder.

More Information

For more information about interception, see the documentation in the Unity 3 Reference documentation and the Developer's Guide to Dependency Injection Using Unity including:

- Chapter 4, Interception
- Chapter 5, Interception using Unity

patterns & practices

proven practices for predictable results