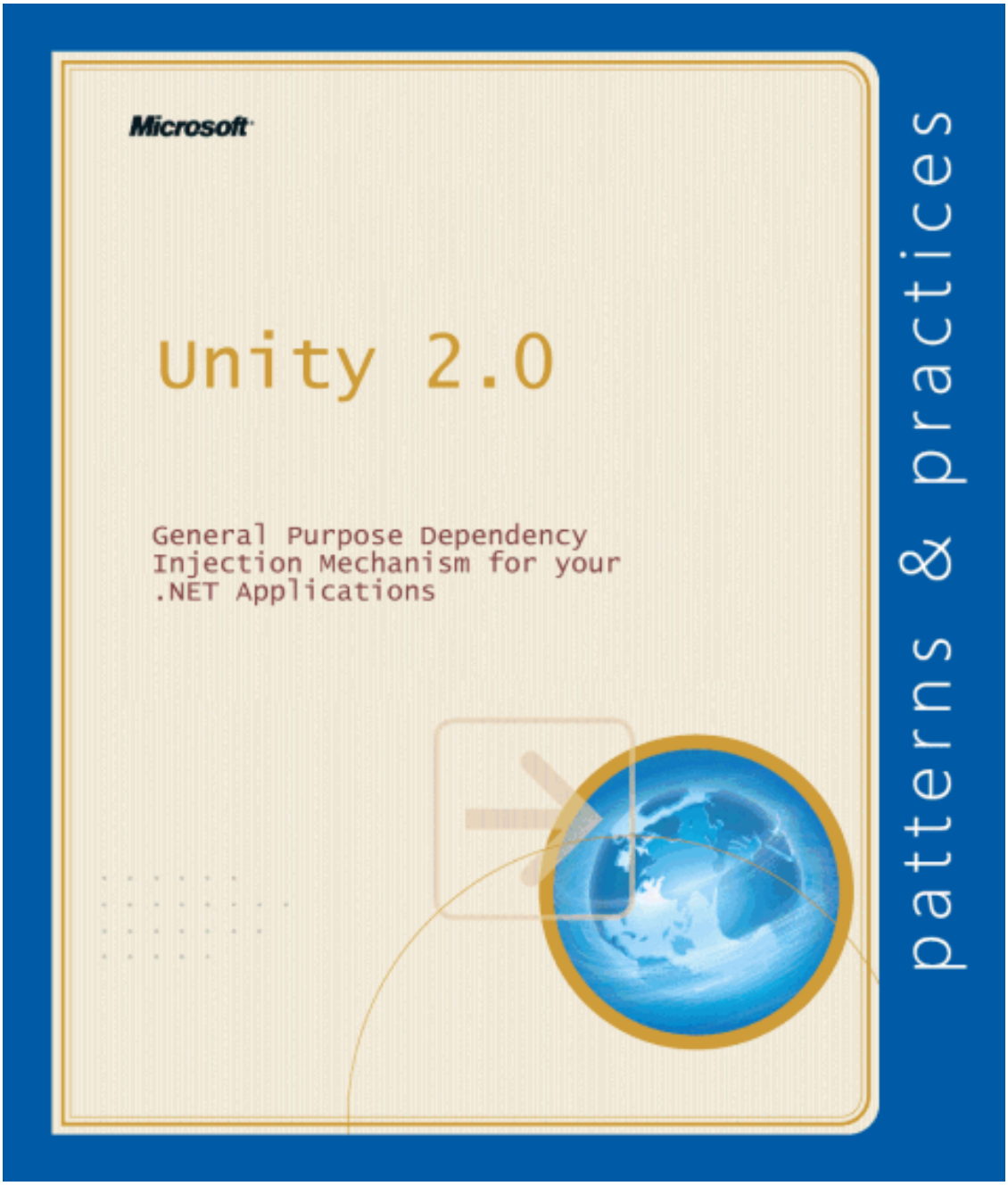

Microsoft
Unity 2.0
General Purpose Dependency Injection Mechanism for your .NET Applications
patterns & practices

April 2010

# COMMUNITY PREVIEW LICENSE

# Unity Dependency Injection

**Welcome to Unity.** The following sections of this guidance describe the ways that you can use Unity dependency injection and Unity interception in your applications. The sections are:

- What Is Unity?
- What Does Unity Do?
- When Should I Use Unity?
- About This Release of Unity
- Configuring Unity
- Using Unity in Applications
- Design of Unity
- Extending and Modifying Unity
- Deployment and Operations
- Unity QuickStarts. This topic walks through the QuickStart applications that demonstrate how to execute common operations in your applications.

# What Is Unity?

Unity is a lightweight, extensible dependency injection container that supports interception, constructor injection, property injection, and method call injection. You can use Unity in a variety of different ways to help decouple the components of your applications, to maximize coherence in components, and to simplify design, implementation, testing, and administration of these applications.

Unity is a general-purpose container for use in any type of Microsoft® .NET Framework-based application. It provides all of the features commonly found in dependency injection mechanisms, including methods to register type mappings and object instances, resolve objects, manage object lifetimes, and inject dependent objects into the parameters of constructors and methods and as the value of properties of objects it resolves.

In addition, Unity is extensible. You can write container extensions that change the behavior of the container, or add new capabilities. For example, the interception feature provided by Unity, which you can use to add policies to objects, is implemented as a container extension.

The following sections of this guidance describe what Unity can do, when you should choose Unity, and the ways that you can use it in your applications:

- What Does Unity Do? This topic provides a brief overview that will help you to understand what Unity can do, and explains some of the concepts and features it incorporates. It also provides a simple example of the way that you can write code to use Unity.
- When Should I Use Unity? This topic will help you to decide if Unity is suitable for your requirements. It explains the benefits of using Unity, and any alternative techniques you may consider. It also provides details of any limitations of Unity that may affect your decision to use it.
- About This Release of Unity. This topic contains information about the changes in this release, the target audience and system requirements, migration and side-by-side execution, and links to other Microsoft patterns & practices resources.
- Configuring Unity. This topic describes how you can populate a Unity container with the type registrations, mappings, extensions, and other information required by your application.
- Using Unity in Applications. This topic explains how to use Unity in your own applications. It explains how to add Unity to your application, how to resolve objects, and how to take advantage of the many other capabilities of Unity.
- Design of Unity. This topic explains the decisions that went into designing Unity and the rationale behind those decisions.
- Extending and Modifying Unity. This topic explains how to extend Unity and how to modify the source code.
- Deployment and Operations. This topic explains how to deploy and update the Unity assemblies and use the instrumentation exposed by Unity.

# What Does Unity Do?

By using dependency injection frameworks and inversion of control mechanisms, you can generate and assemble instances of custom classes and objects that can contain dependent object instances and settings. The following sections explain the ways that you can use Unity, and the features it provides:

- The Types of Objects Unity Can Create
- Registering Existing Types and Object Instances
- Managing the Lifetime of Objects
- Specifying Values for Injection
- Populating and Injecting Arrays, Including Generic Arrays

- Intercepting Calls to Objects

## The Types of Objects Unity Can Create

You can use the Unity container to generate instances of any object that has a public constructor (in other words, objects that you can create using the **new** operator), without registering a mapping for that type with the container. When you call the **Resolve** method and specify the default instance of a type that is not registered, the container simply calls the constructor for that type and returns the result.

Unity exposes a method named **RegisterType** that you can use to register types and mappings with the container. It also provides a **Resolve** method that causes the container to build an instance of the type you specify. The lifetime of the object it builds corresponds to the lifetime you specify in the parameters of the method. If you do not specify a value for the lifetime, the container creates a new instance on each call to **Resolve**. For more information see Registering Types and Type Mappings.

## Registering Existing Types and Object Instances

Unity exposes a method named **RegisterInstance** that you can use to register existing instances with the container. When you call the **Resolve** method, the container returns the existing instance during that lifetime. If you do not specify a value for the lifetime, the instance has a container-controlled lifetime, which means that it effectively becomes a singleton instance. For more information see Creating Instance Registrations.

## Managing the Lifetime of Objects

Unity allows you to choose the lifetime of objects that it creates. By default, Unity creates a new instance of a type each time you resolve that type. However, you can use a lifetime manager to specify a different lifetime for resolved instances. For example, you can specify that Unity should maintain only a single instance (effectively, a singleton). It will create a new instance only if there is no existing instance. If there is an existing instance, it will return a reference to this instead. There are also other lifetime managers you can use. For example, you can use a lifetime manager that holds only a weak reference to objects so that the creating process can dispose them, or a lifetime manager that maintains a separate single instance of an object on each separate thread that resolves it.

You can specify the lifetime manager to use when you register a type, a type mapping, or an existing object using design-time configuration, as shown in Specifying Types in the Configuration File or, alternatively, you can use run-time code to add a registration to the container that specifies the lifetime manager you want to use, as shown in Registering Types and Type Mappings and Creating Instance Registrations.

## Configuring Types for Injection into Constructors, Methods, and Properties

Unity enables you to use techniques such as constructor injection, property injection, and method call injection to generate and assemble instances of objects complete with all dependent objects and settings. For more information see Specifying Values for Injection and Registering Injected Parameter and Property Values.

## Example Application Code for Constructor Injection

As an example of constructor injection, if a class that you instantiate using the **Resolve** method of the Unity container has a constructor that defines one or more dependencies on other classes, the Unity container automatically creates the dependent object instance specified in the parameters of the constructor. For example, the following code shows a class named **CustomerService** that has a dependency on a class named **LoggingService**.

**C#**

```
public class CustomerService
{
  public CustomerService(LoggingService myServiceInstance)
  {
    // work with the dependent instance
    myServiceInstance.WriteToLog("SomeValue");
  }
}
```

**Visual Basic**

```
Public Class CustomerService
  Public Sub New(myServiceInstance As LoggingService)
    ' work with the dependent instance
    myServiceInstance.WriteToLog("SomeValue")
  End Sub
End Class
```

At run time, developers create an instance of the **CustomerService** class using the **Resolve** method of the container, which causes it to inject an instance of the concrete class **LoggingService** within the scope of the **CustomerService** class.

**C#**

```
IUnityContainer uContainer = new UnityContainer();
CustomerService myInstance = uContainer.Resolve<CustomerService>();
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer()
Dim myInstance As CustomerService = uContainer.Resolve(Of CustomerService)()
```

## Example Application Code for Property (Setter) Injection

In addition to constructor injection, described earlier, Unity supports property and method call injection. The following code demonstrates property injection. A class named **ProductService** exposes as a property a reference to an instance of another class named **SupplierData** (not defined in the following code). Unity does not automatically set properties as part of creating an instance; you must explicitly configure the container to do so. One way to do this is to apply the **Dependency** attribute to the property declaration, as shown in the following code.

**C#**

```
public class ProductService
{
  private SupplierData supplier;

  [Dependency]
  public SupplierData SupplierDetails
  {
    get { return supplier; }
    set { supplier = value; }
  }
}
```

**Visual Basic**

```
Public Class ProductService
  Private supplier As SupplierData

  <Dependency()> _
  Public Property SupplierDetails() As SupplierData
    Get
      Return supplier
    End Get
    Set (ByVal value As SupplierData)
      supplier = value
    End Set
  End Property

End Class
```

Now, creating an instance of the **ProductService** class using Unity automatically generates an instance of the **SupplierData** class and sets it as the value of the **SupplierDetails** property of the **ProductService** class.

## Populating and Injecting Arrays, Including Generic Arrays

You can define arrays, including arrays of generic types, and Unity will inject the array into your classes at run time. You can specify the members of an array, or have Unity populate the array automatically by resolving all of the matching types defined in your configuration. You can then use the populated arrays as types to resolve directly, or to set the values of constructor and method parameters and properties. Arrays can be defined in configuration files or by adding the definitions to the container at run time using code.

For details and examples, see the sections on arrays in the following topics:

- Configuring Injected Parameter and Property Values
- Registering Injected Parameter and Property Values
- Registering Generic Parameters and Types

## Intercepting Calls to Objects

The best types in object oriented systems are ones that have a single responsibility. But as systems grow, other concerns tend to creep in. System monitoring, such as logging, event counters, parameter validation, and exception handling are just some examples of areas where this is common. These cross-cutting concerns often require large amounts of repetive code throughout the application and if your design choice changes, for example you change your logging framework, then the logging calls must be changed in many places throughout the code set.

Interception is a technique that enables you to add code that will run before and after a method call on a target object. The call is intercepted and additional processing can happen. When you perform this coding process manually, you are following what is commonly known as the decorator pattern. Writing decorators requires that the types in question be designed for it in the beginning, and requires a different decorator type for each type you are decorating.

The Unity interception system creates the decorators automatically. This allows you to easily reuse code that implements these cross-cutting concerns with minimal, if any, attention to what types the cross-cutting concerns will be applied to.

## Example Application Code

The following example configures the container at run time for interception of the type **TypeToIntercept** by using a **VirtualMethodInterceptor** interceptor with an interception behavior, **ABehavior**, and an additional interface to be implemented.

**C#**

```
IUnityContainer container = new UnityContainer();
container.AddNewExtension<Interception>();
container.RegisterType<TypeToIntercept>(
        new Interceptor<VirtualMethodInterceptor>(),
        new InterceptionBehavior<ABehavior>(),
        new AdditionalInterface<IOtherInterface>());
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer()
container.AddNewExtension(Of Interception)()
container.RegisterType(Of TypeToIntercept)(New Interceptor(Of
VirtualMethodInterceptor)(), New InterceptionBehavior(Of ABehavior)(), New
AdditionalInterface(Of IOtherInterface)())
```

You could then resolve an instance by using the following call:

**C#**

```
TypeToIntercept instance = container.Resolve<TypeToIntercept>();
```

**Visual Basic**

```
Dim instance As TypeToIntercept = container.Resolve(Of TypeToIntercept)()
```

Now when calling methods on an instance, the code in the **ABehavior** class will be executed before and after any call to the instance.

# When Should I Use Unity?

Dependency injection provides opportunities to simplify code, abstract dependencies between objects, and automatically generate dependent object instances. In general, you should use Unity when:

- You wish to build your application according to sound object oriented principles (following the five principles of class design, or SOLID), but doing so would result in large amounts of difficult-to-maintain code to connect objects together.
- Your objects and classes may have dependencies on other objects or classes.
- Your dependencies are complex or require abstraction.
- You want to take advantage of constructor, method, or property call injection features.
- You want to manage the lifetime of object instances.
- You want to be able to configure and change dependencies at run time.
- You want to intercept calls to methods or properties to generate a policy chain or pipeline containing handlers that implement crosscutting tasks.
- You want to be able to cache or persist the dependencies across post backs in a Web application.

---

The following sections provide more information to help you decide whether Unity is suitable for your requirements:

- Scenarios for Unity
- Benefits of Unity
- Limitations of Unity

---

Enterprise Library, also from the Microsoft patterns & practices group, uses Unity as its primary mechanism for generating instances of Enterprise Library objects. For information about this and

other features of Enterprise Library, see the Enterprise Library Community site on CodePlex, or the Microsoft Enterprise Library pages on MSDN®.

## Scenarios for Unity

Unity addresses the issues faced by developers engaged in component-based software engineering. Modern business applications consist of custom business objects and components that perform specific tasks or generic tasks within the application, in addition to components that individually address crosscutting concerns such as logging, authentication, authorization, caching, and exception handling.

The key to successfully building these types of applications is to achieve a decoupled or very loosely coupled design. Loosely coupled applications are more flexible and easier to maintain. They are also easier to test during development. You can mock up shims (lightweight mock implementations) of objects that have strong concrete dependencies, such as database connections, network connections, enterprise resource planning (ERP) connections, and rich user interface components.

Dependency injection is a prime technique for building loosely coupled applications. It provides ways to handle the dependencies between objects. For example, an object that processes customer information may depend on other objects that access the data store, validate the information, and check that the user is authorized to perform updates. Dependency injection techniques can ensure that the customer class correctly instantiates and populates all these objects, especially where the dependencies may be abstract.

The following design patterns define architectural and development approaches that simplify the process:

- **Inversion of Control (IoC) pattern**. This generic pattern describes techniques for supporting a plug-in architecture where objects can look up instances of other objects they require.
- **Dependency Injection (DI) pattern**. This is a special case of the IoC pattern and is an interface programming technique based on altering class behavior without the changing the class internals. Developers code against an interface for the class and use a container that injects dependent object instances into the class based on the interface or object type. The techniques for injecting object instances are interface injection, constructor injection, property (setter) injection, and method call injection.
- **Interception pattern**. This pattern introduces another level of indirection. This technique places a proxy object between the client and the real object. The client behavior is the same as when interacting directly to the real object, but the proxy intercepts the calls and manages their execution by collaborating with the real object and other objects as required.

## Benefits of Unity

Unity provides developers with the following advantages:

- It provides simplified object creation, especially for hierarchical object structures and dependencies, which simplifies application code. It contains a mechanism for building (or assembling) instances of objects, which may contain other dependent object instances.
- It supports abstraction of requirements; this allows developers to specify dependencies at run time or in configuration and simplify the management of crosscutting concerns.
- It increases flexibility by deferring component configuration to the container. It also supports a hierarchy for containers.
- It has a service location capability, which is useful in many scenarios where an application makes repeated use of components that decouple and centralize functionality.
- It allows clients to store or cache the container. This is especially useful in ASP.NET Web applications where developers can persist the container in the ASP.NET session or application.
- It has an interception capability, which allows developers to add functionality to existing components by creating and using handlers that are executed before a method or property call reaches the target component, and again as the calls returns.
- It can read configuration information from standard configuration systems, such as XML files, and use it to configure the container.
- It makes no demands on the object class definition. There is no requirement to apply attributes to classes (except when using property or method call injection), and there are no limitations on the class declaration.
- It supports custom container extensions that you can implement; for example, you can implement methods to allow additional object construction and container features, such as caching.
- It allows architects and developers to more easily implement common design patterns often found in modern applications.

---

### Limitations of Unity

Dependency injection through Unity can have a minor impact on performance, and it can increase complexity where only simple dependencies exist. You should *not* use Unity in the following situations:

- When your objects and classes have no dependencies on other objects or classes.
- When your dependencies are very simple and do not require abstraction.

---

# About This Release of Unity

This section contains the following topics that will help you to understand this release of Unity and help you understand how to use it alongside earlier versions or migrate your applications to this version. This section includes the following topics:

- Changes in This Release
- Target Audience and System Requirements
- Migration and Side-by-Side Execution
- Related patterns & practices Links
- Copyright and Terms of Use

## How to Use This Guidance

The following table shows where you can look to find more information about specific topics covered by this guidance.

| If you want to find out about... | Read this section... |
|---|---|
| What's new in this release of Unity | Changes in This Release |
| What you can do with Unity | What Does Unity Do? |
| When you should choose to use Unity | When Should I Use Unity? |
| How to configure Unity | Configuring Unity |
| How to use Unity in your applications | Using Unity in Applications |
| How to resolve objects in Unity | Resolving Objects |
| How to use injection for methods and properties | Using Injection Attributes |
| How to use interception | Using Interception in Applications |
| How to use interception and create policies | Using Interception and Policy Injection |
| How to use container hierarchies | Using Container Hierarchies |
| Issues with circular references when using Unity | Circular References with Dependency Injection |
| How to deploy Unity | Deployment and Operations |
| The design of Unity | Design of Unity |
| How to extend and modify Unity | Extending and Modifying Unity |
| Additional guidance and information | The Community Web Site on CodePlex |

## More Information

For related information, see the following guidance:

- The Unity Community Web Site on CodePlex

- Loosen Up - Tame Your Software Dependencies for More Flexible Apps by James Kovacs in *MSDN Magazine*
- Design Patterns: Dependency Injection by Griffin Caprio in *MSDN Magazine*
- Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse by Dharma Shukla, Simon Fell, and Chris Sells in *MSDN Magazine*
- Aspect-Oriented Programming by Matthew Deiters on MSDN
- Create a Custom Marshaling Implementation Using .NET Remoting and COM Interop by Jim Sievert in *MSDN Magazine*
- The EFx Architectural-Guidance Software Factory by Jezz Santos on MSDN

# Changes in This Release

Unity 2.0 – April 2010 is a new release of the Microsoft patterns & practices Unity dependency injection and interception system. This release also includes additions in functionality, and has been adapted to work with Microsoft Visual Studio® 2010; and with the Microsoft .NET Framework versions 3.5 SP1 and 4.0.

Go to CodePlex for information on Known Issues.

The following sections discuss the changes to Unity:

Breaking Changes to Unity

Changes in Unity

## Breaking Changes to Unity

This section lists the breaking changes in Unity 2.0.

- Unity 2.0 uses a new streamlined configuration schema for Configuring Unity. Partial backward compatibility is provided for previous configuration schemas. The new schema breaks previous container extension configurations that use <extensionConfig>.
- Unity 2.0 configuration API change:

  The previous API used to load container configuration has been deprecated. Previous versions used the following approach:

  **C#**

```
IUnityContainer myContainer = new UnityContainer();
UnityConfigurationSection section
    = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");
section.Containers["containerName"].Configure(myContainer);
```

In Unity 2.0, you have the two choices. You can use **ConfigurationManager** and **Configure** to load a given container from a specific named configuration section, as shown in the following example.

**C#**

```
IUnityContainer myContainer = new UnityContainer();
UnityConfigurationSection section
    = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");
section.Configure(myContainer, "containerName");
```

Otherwise, you can use **LoadConfiguration** to load a given named container from the default configuration section, as shown in the following example.

**C#**

```
IUnityContainer myContainer = new UnityContainer();
myContainer.LoadConfiguration("containerName");
```

There are several overloads of the new **LoadConfiguration** extension method. See Using Design-Time Configuration for more information.

- The arrangement of call handlers and matching rules has been changed. For more information see Configuring Policy Injection Policies and Enterprise Library Call Handlers.
- The Unity for Silverlight® assembly has changed from **Microsoft.Practices.Unity.DLL** to **Microsoft.Practices.Unity.Silverlight.dll**.

## Changes in Unity

The following changes are implemented in this release:

- The **ObjectBuilder** generic dependency injection mechanism, which was previously distributed as a separate assembly, is now integrated with Unity. You no longer need to reference the ObjectBuilder assembly in your projects.
- The **StaticFactory.StaticFactoryExtension** to Unity, which allows you to specify a factory function that the container will use to create an object, was shipped as a separate project in Unity 1.2. In this release, it is contained within the main Unity assembly but is deprecated. Use the new **InjectionFactory** class.
- A new **InjectionFactory** class has been added. It enables you specify a factory method that the container will use to create the object. It is an easier way to accomplish the same task as the old **StaticFactory.StaticFactoryExtension**.
- Unity provides a new interception process and provides interceptors that enable you to intercept calls to objects and attach behaviors to the intercepted methods. A behaviors pipeline is used instead of the previous call handlers pipeline. Interception is supported with or without a dependency injection container. See Interception with Unity.

- A **HierarchicalLifetimeManager** is provided to register an object in the parent container such that each child container resolves its own instance of the object instead of sharing one with the parent. Understanding Lifetime Managers
- Dependencies can be made optional by using the **<optional>** element in configuration files, the **[OptionalDependency]** attribute, or **OptionalParameter** in your code. See Annotating Objects for Property (Setter) Injection.
- Unity now provides the ability to specify parameter values for constructors, or specific values for properties, at **Resolve** time. These values override what the container would have otherwise supplied. **ParameterOverride**, **PropertyOverride,** and **DependencyOverride** are **ResolverOverride** implementations that provide support for overriding the registration information for resolving instances of types. Resolving Objects by Using Overrides
- A **PerResolveLifetimeManager** registers an object such that the behavior is like a **TransientLifetimeManager**, but also provides a signal to the default build plan, marking the type so that instances are reused across the build up object graph. Understanding Lifetime Managers
- The **Registrations** property and **IsRegistered** method have been added. These let you query the container and retrieve information about what is currently registered. See Retrieving Container Registration Information.
- You can use the .NET standard type **Func<T>** (C#) or **Func(Of T)** (Visual Basic®) with the **Resolve** method if you wish to defer the resolution of an object. For more information see Deferring the Resolution of Objects.
- Specifying a resolved array without providing any members will result in an empty (zero length) array.
    - In a configuration file, the **<array />** element with no child elements will generate a zero-length array for both generic and non-generic types. If you want to resolve all configured types to populate the array (effectively invoking the **ResolveAll** behavior), you must use a **<dependency />** element instead.
    - When performing run-time configuration, the **ResolvedArrayParameter** and **GenericResolvedArrayParameter** classes will generate a zero-length array if you do not specify the members for the array. If you want to resolve all configured types, use the **GenericParameter** class and specify the array type followed by the appropriate brackets to indicate an array type in the language you are using. For example, **GenericParameter("T[]")** in C# or **GenericParameter("T()")** in Visual Basic.

# Target Audience and System Requirements

This guidance is intended for software architects and software developers. To get the greatest benefit from this guidance, you should have an understanding of the following technologies:

- Microsoft Visual C#® or Microsoft Visual Basic .NET
- Microsoft .NET Framework

## System Requirements and Prerequisites

The following are the system requirements for using Unity:

- Operating system: Microsoft Windows® 7 Professional, Enterprise, or Ultimate; Windows Server® 2003 R2; Windows Server 2008 with Service Pack 2; Windows Server 2008 R2; Windows Vista® with Service Pack 2; Windows XP with Service Pack 3.
- Microsoft .NET Framework 3.5 with Service Pack 1 or Microsoft .NET Framework 4.0.

For a rich development environment and use of the integrated configuration tool, the following are recommended:

- Microsoft Visual Studio® 2008 Development System with Service Pack 1 (any edition) or Microsoft Visual Studio 2010 Development System (any edition)

To run the unit tests, the following are also required:

- Microsoft Visual Studio 2008 Professional, Visual Studio 2008 Team Edition, Visual Studio 2010 Premium, Visual Studio 2010 Professional, or Visual Studio 2010 Ultimate edition
    - Moq v3.1 assemblies.

## System Requirements for Unity for Silverlight

The following are the system requirements for using Unity for SIlverlight:

- Operating system: Microsoft Windows® 7 Professional, Enterprise or Ultimate; Windows Server 2003 R2; Windows Server 2008 with Service Pack 2; Windows Server 2008 R2; or Windows Vista with Service Pack 2.
- Microsoft® Silverlight™ 3 or Microsoft Silverlight 4.

For a *rich development environment*, the following are recommended:

- Microsoft Visual Studio® 2008 Development System with Service Pack 1 (any edition) or Microsoft Visual Studio 2010 Development System (any edition).
- Microsoft Silverlight3 Tools for Visual Studio 2008 SP1 or Visual Studio 2010, or Microsoft Silverlight 4 Tools for Visual Studio 2010

To run the *unit tests*, the following are also required:

- Microsoft Visual Studio 2008 Professional, Visual Studio 2008 Team Edition, Visual Studio 2010 Premium, Visual Studio 2010 Professional, or Visual Studio 2010 Ultimate edition.

# Migration and Side-by-Side Execution

In general, applications that use previous versions of Unity will function with this release without the need for any code changes. It will be necessary, however, to update the references to refer to the new assemblies and to update the configuration files to reference the correct version of the assemblies.

This version of Unity can also be installed side by side with earlier versions. You can deploy new applications written for this version of Unity along with applications written for earlier versions. In addition, you can also choose to migrate existing applications to the new version.

If you decide to use side-by-side execution, you must deploy the different Unity versions in different directories. In any specific directory, you cannot mix and match assemblies from different versions.

The shipped project files use data in the AssemblyInfo.cs file to build assemblies that have different version information. This allows you to use strong names and to add different versions to the global assembly cache (GAC) for side-by-side execution.

## Partial Migration

Each assembly in an application can refer to only one version of Unity, but an application that has multiple assemblies can refer to more than one version. For example, you may have an application with two assemblies, both using the Unity 1.1. One assembly can be migrated to use Unity 2.0, and the other assembly can remain unchanged. This means you can gradually migrate your application, one assembly at a time. However, although partial migration is supported, it is somewhat complicated to implement; therefore, it is not recommended.

For details on migrating applications and configuration files see the "Enterprise Library 5.0 and Unity 2.0 Migration Guide" at http://www.codeplex.com/entlib/ .

For details on migrating extensions see Reusing Configuration Files Based on a Previous Schema.

# Reusing Configuration Files Based on a Previous Schema

Although this documentation is based on the Unity 2.0 configuration schema and all examples use Unity 2.0, partial backward compatibility is provided for the Unity 1.2 configuration schema.

However, you cannot simply use a Unity 1.2 configuration file. In order to use the contents of a Unity 1.2 configuration file you must:

1. Create a new configuration file.
2. Edit the **<configSections>** to point to the correct assembly.
3. Add a **<sectionExtension>** section if you are using container extensions for Unity.
4. Cut and paste the portions of the Unity 1.2 configuration file you wish to reuse.

> Check the results as you still may get errors depending upon the specific portions you cut and paste.

For information on using the Unity 1.2 configuration schema see Unity Configuration Schematic on MSDN®.

## Migrating Custom Extensions

Unity 1.2 extension configurations cannot just be copied into a Unity 2.0 configuration file. There is no <**extensionConfig>** section in Unity 2.0. However you can often copy the contents of an old <**extensionConfig>** section to Unity 2.0 <**register>** elements.

For example, the following is an excerpt from a Unity 1.2 configuration file.

**XML**

```
<extensionConfig>
  <add name="interception"

type="Microsoft.Practices.Unity.InterceptionExtension.Configuration.InterceptionC
onfigurationElement, Microsoft.Practices.Unity.Interception.Configuration">
    <interceptors>
      <interceptor type="transparentProxy">
        <default type="wrappable"/>
        <key type="wrappableWithProperty"/>
      </interceptor>
      <interceptor type="virtualMethod">
        <key type="wrappableVirtual" name="name"/>
      </interceptor>
    </interceptors>
  </add>
</extensionConfig>
```

In Unity 2.0 you would do the following:

1. Use <**sectionExtension>** to add the extension elements to the schema and aliases to the configuration section.

   **XML**

   ```
   <sectionExtension
   type="Microsoft.Practices.Unity.InterceptionExtension.Configuration.Interce
   ```

```
ptionConfigurationExtension,
Microsoft.Practices.Unity.Interception.Configuration" />
```

2. You would then use the **<extension>** element to add the **interception** extension to the configuration.

**XML**

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

3. Then you could copy the interceptor element content and change TransparentProxy to TransparentProxyInterceptor.

**XML**

```
<container name="configuringDefaultInterceptor">
    <extension type="Interception" />
    <interceptors>
      <interceptor type="TransparentProxyInterceptor">
        <default type="wrappable"/>
        <key type="wrappableWithProperty"/>
      </interceptor>
    </interceptors>
</container>
```

You can use the container element extension or injection members to add extensions. For more information on Unity 2.0 extensions see Configuration Files for Interception.

# Related patterns & practices Links

For information related to Unity and other tools, and guidance for designing and building applications, see the patterns & practices website and guides:

- Microsoft patterns & practices Developer Center
- Microsoft Application Architecture Guide, 2nd Edition
- Solution Development Fundamentals
- Security Guidance for Applications Index
- .NET Data Access Architecture Guide
- Improving .NET Application Performance and Scalability
- Monitoring in .NET Distributed Application Design
- Deploying .NET Framework-based Applications

# Copyright and Terms of Use



# Configuring Unity

One of the key tasks you need to perform when using Unity is to configure the container with the required aliases, type registrations, mappings, and other information that it requires in order to resolve objects at run time and inject the appropriate objects and values into dependent objects. This section covers all of the configuration options for Unity, and describes how you can configure the container using a configuration file, or at run time using code.

Unity can read configuration information from an XML configuration file. By default, this is the App.config or Web.config file for your application. However, you can load configuration information from any other XML format file or from other sources if required.

In addition, Unity exposes a series of methods that you use in your application code to configure the container. This approach is useful when the registrations and mappings depend on the environment or run-time information, and you can change the configuration at run time using these methods. Run-time configuration is also a good choice if you want to be able to manipulate container hierarchies at run time to change the overall behavior of type resolution, injection, and interception.

You can, of course, use a combination of design-time (configuration files) and run-time configuration to achieve exactly the configuration you require at any point during application execution.

To help you understand how to configure Unity, this section divides the information into two separate subsections—one for design-time configuration and one for run-time configuration. Each section contains basically the same set of topics that describe specific configuration scenarios.

> The configuration files are not encrypted by default. A configuration file may contain sensitive information about connection strings, user IDs, passwords, database servers, and catalogs. You should protect this information against unauthorized read/write operations by using encryption techniques.

If you wish to restrict access to the configuration file, it must be encrypted or protected using Access Control Lists. It is recommended that the configuration store is in the same trust boundary and that decrypting the configuration is done in the same trust boundary after the configuration is read.

The complete set of topics in this section is as follows:

- Design-Time Configuration
  - Using the Configuration Tool
  - Using Design-Time Configuration
  - Specifying Types in the Configuration File
  - Specifying Values for Injection
  - Extending the Unity Configuration Schema
  - Configuration Files for Interception
  - Default Aliases and Assemblies
- Run-Time Configuration
  - Using Run Time Configuration
  - Registering Types and Type Mappings
  - Creating Instance Registrations
  - Registering Injected Parameter and Property Values
  - Registering Generic Parameters and Types
  - Registering Container Extensions
  - Registering Interception

---

For information about using container hierarchies, see Using Container Hierarchies. For information about using Unity, see Using Unity in Applications.

# Design-Time Configuration

This topic describes the techniques you can use to configure Unity containers using a configuration file or a configuration you load into the container at run time.

This documentation is based on the Unity 2.0 configuration schema and all examples use Unity 2.0. Partial backward compatibility is provided for the Unity 1.2 configuration schema.

The following topics describe Unity configuration:

- Using the Configuration Tool. Not available for Unity configuration.
- Using the Unity XSD to Enable Visual Studio IntelliSense. This topic explains how to use the Unity XSD to enable IntelliSense® in Visual Studio to assist in manually editing the Unity configuration.
- Reusing Configuration Files Based on a Previous Schema. This topic explains how to create a current configuration file by reusing a configuration file based on a previous Unity schema.
- Using Design-Time Configuration. This topic explains the overall structure of the Unity configuration file, how you load configuration information at run time, and how you can use alternative configuration sources with Unity.
- Specifying Types in the Configuration File. This topic explains how to configure mappings in the container between types. In general, you will create mappings between an interface and a type that implements the interface, or between a base class and a type that inherits that base class. You can also use this section to specify concrete types for which you want Unity to manage the lifetime.
- The Unity Configuration Schema. This topic describes the configuration schema elements for Unity.
- Specifying Values for Injection. This topic explains how to configure registrations for instance types such as string, date and time, or integer values that you can resolve in your application.
- Extending the Unity Configuration Schema. This topic explains how to configure Unity to load and use container extensions that add additional functionality to the container, and how you can specify configuration information for these extensions.
- Configuration Files for Interception. This topic explains how to configure interceptors, behaviors, policies, handlers, and matching rules that Unity will use when creating instances of types to which you want to add interception capabilities in order to change the behavior of that object or type.

---

For information about how to configure Unity using code that executes at run time and calls the registration methods of the Unity container, see Run-Time Configuration. For information about resolving types at run time, see Resolving Objects.

# Using the Configuration Tool

Not available for Unity configuration.

# Using the Unity XSD to Enable Visual Studio IntelliSense

You can enable IntelliSense in Visual Studio to assist the manual editing of Unity configuration files.

> The XSD is not required. Configuration will work at run time without it. It is only required for IntelliSense in Visual Studio.

In order for the XSD to be used by the Visual Studio editor the **<unity>** element must have an XMLNS attribute with the correct namespace. The following are the two ways to get the correct namespace instead of manually entering it:

- You can force the editor to use the schema by clicking **Schemas** on the **XML** menu and selecting the entry for the unity configuration schema. After that, the **<unity>** element will appear as an alternative in the IntelliSense dropdown and by choosing it from IntelliSense the xmlns attribute will be populated. This is a per-user and per-project setting, so every user working on the project would be required to select this setting.
- You can enter **<unity xmlns="** and then IntelliSense will show a list of namespaces which are targeted by a known schema. You can then choose the right namespace, **http://schemas.microsoft.com/practices/2010/unity**, which will show up in the IntelliSense list when you click on the **xmlns** attribute and complete the entry. Visual Studio will then associate the URL with the actual physical file. This is the recommended option, since the setting persists when you pass the configuration file to another user.

Collection elements, such as aliases, containers, extensions, namespaces, and assemblies are not supported by the xsd, but they do work in the configuration file.

There are some pre-defined type names for some type attributes, such as for lifetime managers, but these are just suggestions and any type name is accepted.

The schema for the **register** element imposes a specific order for its children, an order that is not required by the configuration runtime but makes the schema more robust. The order of children is as follows: one optional **lifetime**, one optional **constructor**, and then as many of **method**, **property**, **interceptor**, **interceptionBehavior**, **addInterface** and any custom element as desired, in any order.

# Using Design-Time Configuration

Using Unity typically requires the configuration of a Dependency Injection (DI) container. You can configure a container by using the Unity API, a .NET configuration file, or to a limited degree by using

attributes. This topic describes how to use an XML configuration file to supply the required configuration information..

Dependency injection is a very flexible pattern, and to be used successfully requires the developer to provide information to the container about his applications. The two most common configuration tasks are setting up type mappings and configuring injection of a type. Type mappings enable you to request a type from the container that results in the container returning an instance of a different type (typically a derived class or interface implementation). Configuring injection for a type entails specifying information such as which constructor gets called, which properties get injected, and what their values are. The Unity configuration schema encompasses these types of configuration and is also extensible to allow for additional kinds of configuration such as Unity interception configuration, see The Unity Configuration Schema. The following sections provide more details:

- Format of the Unity Configuration File
- Loading Configuration File Information into a Container
- Loading the Configuration from Alternative Files

## Format of the Unity Configuration File

Unity uses the System.Configuration namespace supplied with the .NET framework. This means that configuration information can be stored in any .NET configuration file, which is typically your App.config or Web.config file, but it could be stored elsewhere. The following example is a simple XML configuration file.

**XML**

```
<configuration>

  <configSections>
    <section name="unity"
type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
Microsoft.Practices.Unity.Configuration"/>
  </configSections>

  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <alias alias="ILogger" type="MyApp.ILogger, MyApp" />
    <namespace name="MyApp.Implementations" />
    <assembly name="MyApp" />

    <container>
      <register type="ILogger" name="special" mapTo="SpecialLogger" />
    </container>

  </unity>

</configuration>
```

A Unity configuration section consists of a set of type lookup modifiers (aliases, namespaces, and assemblies), and one or more **<container>** elements. A **<container>** element has a set of

**<registration>** elements that provide the configuration for the container's types. Other elements can also be used in the container, see The Unity Configuration Schema for a full description of the schema.

The XML namespace specified with the **xmlns** attribute in the example is not required at run time. However, it is useful because Visual Studio uses it to match the section with the Unity XML schema and to provide IntelliSense in the XML editor.

## Loading Configuration File Information into a Container

Adding a Unity configuration section to a configuration file neither create an actual container nor configures it. You must create a Unity container instance, read the configuration file, and load the configuration file information into the container.

> Alternatively, after you create the container, you can configure it programmatically at run time with registrations, type mappings, and any extensions. For more information about run-time configuration, see Run-Time Configuration.

To load the configuration file information into a container, use the **LoadConfiguration** extension method on **IUnityContainer**. This interprets the default configuration file for your application, seeks the Unity configuration section, and configures the container. The **LoadConfiguration** extension method has several overloads.

Unity offers a convention-based approach to configuring your container that can be applied in most cases. To take advantage of this you must use the default Unity configuration section name of **unity** and specify an un-named container in your App.config or Web.config file. The following example uses this approach.

**C#**

```
IUnityContainer container = new UnityContainer()
    .LoadConfiguration();
```

**Visual Basic**

```
Dim container as IUnityContainer = New UnityContainer()
container.LoadConfiguration()
```

The configuration for a named container can be loaded from the default configuration section by providing the name of the container to the **LoadConfiguration** method as shown in the following example:

**C#**

```
IUnityContainer container = new UnityContainer()
    .LoadConfiguration("otherContainerElement");
```

**Visual Basic**

```
Dim container as IUnityContainer = New UnityContainer()
container.LoadConfiguration("otherContainerElement")
```

If your configuration is in a different section (either with a different name or from a different file entirely), you must first load the section object through **ConfigurationManager** and then pass the section to the **LoadConfiguration** method, as shown in the following example:

**C#**

```
IUnityContainer container = new UnityContainer()
    .LoadConfiguration(section)  // Loads unnamed <container> element
    .LoadConfiguration(section, "otherContainerElement"); // named <container>
element
```

**Visual Basic**

```
Dim container as IUnityContainer = New UnityContainer()
container.LoadConfiguration(section) _
    .LoadConfiguration(section, "otherContainerElement")
```

You can also load multiple configurations into the same container. Non-conflicting configurations will simply be added, and if there is a conflict, such as two mappings for the same type, then the last configuration added will be the one that is used. The previous example illustrates the additive feature for configuration.

There is also an API on the **UnityConfigurationSection** object that can be used to configure a container. Once you have obtained the section object of the **ConfigurationManager**, you can call its **Configure** method to apply configuration to a container:

**C#**

```
var section =
(UnityConfigurationSection)ConfigurationManager.GetSection("unity");
IUnityContainer container = new UnityContainer();
section.Configure(container); // Unnamed <container> element
section.Configure(container, "otherContainerElement"); // named container element
```

**Visual Basic**

```
Dim section = DirectCast(Configurationmanager.GetSection("unity"),
UnityConfigurationSection)
Dim container as IUnityContainer = new UnityContainer()
section.Configure(container)
section.Configure(container, "otherContainerElement")
```

In general, the **LoadConfiguration** extension method is preferred as it is easier to read and use.

## Loading the Configuration from Alternative Files

You are not required to store your container configuration in the standard application configuration file: App.config or Web.config. However, if you do store your configuration in a different file you must use the **ConfigurationManager** methods to explicitly load that specific named file.

If you use a custom configuration file, instead of App.config in an executable application, you must ensure that it is available in the runtime folder of your application. If you create a custom configuration file in a Visual Studio project, open the **Properties** window for the file and set the **Copy to Output Directory** property to **Copy always**.

For example, in order to load configuration information from the named configuration file, **unity.config**, you must first load the section, in this case **UnityConfigurationSection**, as shown in following code.

See ConfigurationManager.OpenMappedExeConfiguration on MSDN for more details.

**C#**

```
using System.Configuration;
var fileMap = new ExeConfigurationFileMap { ExeConfigFilename = "unity.config" };
System.Configuration.Configuration configuration =
    ConfigurationManager.OpenMappedExeConfiguration(fileMap,
ConfigurationUserLevel.None);
var unitySection = (UnityConfigurationSection)configuration.GetSection("unity");

var container = new UnityContainer()
    .LoadConfiguration(unitySection);
```

**Visual Basic**

```
Imports System.Configuration

Dim fileMap as new ExeConfigurationFileMap() With { .ExeConfigFilename =
“unity.config” }
Dim configuration as System.Configuration.Configuration = _
    ConfigurationManager.OpenMappedExeConfiguration(fileMap,
ConfigurationUserLevel.None)
Dim unitySection = DirectCast(configuration.GetSection("unity"),
UnityConfigurationSection)

Dim container as IUnityContainer = new UnityContainer()
container.LoadConfiguration(unitySection)
```

A **<container>** element in the configuration file is not an instance of a **UnityContainer** object. It is a named set of configurations that can be applied to a container instance later and a single **<container>** element can be applied to multiple **UnityContainer** instances.

You cannot nest containers in the configuration file. All **<container>** elements reside at the same level within the **<containers>** element. You configure nested containers by creating the containers in the required hierarchy in your code and then populating them from the appropriate **<container>** elements. If required, you can load more than one container from the same **<container>** element, and you can use more than one container element in a configuration file. For information, see Using Container Hierarchies.

# Specifying Types in the Configuration File

This topic explains how to use types in Unity configuration files.  At the core of Unity's functionality are types and how you specify and handle them. You will need to specify types many times in the typical configuration file. The configuration files have their own set of rules for writing type names—rules that differ from those for types written in C# or Visual Basic. These rules apply everywhere you can specify a type in the Unity configuration section.

This topic contains the following section that describe how you can specify types:



## CLR Type Names

You can specify a type name by using the CLR standard type name syntax, as shown in the following example:

<namespace>.<typename>, <assembly>

You can use either partial assembly names or fully qualified assembly names which include the culture, version, and public key token. These names are straightforward for simple types.

In order to specify a name for a type that is in the global assembly cache, you must use the fully qualified assembly name for the type to be correctly loaded. For example, for System.String, a type in mscorlib, you cannot use **System.String, mscorlib**. You must use the fully qualified assembly name, **System.String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089**.

CLR type names can be very verbose, especially when working with generic types. For example, compare the following simple dictionary in C# or Visual Basic with the CLR example:

**C#**

```
Dictionary<string, int>
```

**Visual Basic**

```
Dictionary(Of String, Integer)
```

**CLR**

```
System.Collections.Generic.Dictionary`2[[System.String, mscorlib, 2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089], [System.Int32, mscorlib,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]], mscorlib,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

In order to expedite the process and make type names less error prone, Unity configuration provides two options you can use in the configuration file: aliases and automatic type lookup.

## Type Aliases

An alias is simply a shorthand name that will be replaced with the full type name when configuration is applied to the container. You specify an alias in the configuration file inside the section, but outside any **<container>** elements, as shown in the following example:

**XML**

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <alias alias="MyAlias" type="full type name" />
    ...
</unity>
```

There are the following rules for using aliases:

- You can have an arbitrary number of **<alias>** elements in the configuration file.
- Anywhere you can give a type name you can use an alias instead.
- There are no recursive aliases, which means that you cannot use an alias to define the type for an alias.
- Alias names are case sensitive: **<alias alias="int" />** and **<alias alias="Int" />** are two different aliases and are not interchangeable.

Aliases only exist at configuration time. They are not available at run time.

## Automatic Type Lookup

In many cases, like for **ILogger** in the Format of the Unity Configuration File example, the name of a type is all that is required. But given Unity's dependence on types and the large number of types typically involved in a configuration, the ability to perform automatic type lookups further expedites the process. By incorporating automatic type lookups, Unity also eliminates the need to define an alias for every type in an assembly, which saves effort and serves to reduce the chance for error from repeatedly typing the namespace and assembly name.

The Unity configuration system can search for types. However, it will only look for types if the type name specified is not a full type name and it is not an alias. You can provide the configuration section with the namespaces and assemblies to look through by using the **<namespace>** and **<assembly>** elements, as shown in the following example.

**XML**

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <namespace name="MyApp.Interfaces" />
    <namespace name="System" />
    <assembly name="MyApp" />
    <assembly name="mscorlib, 2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" />

    ...
</unity>
```

With the configuration shown in the previous example, when the configuration system hits a name it does not recognize as a type name or alias, it will then search through the assemblies and namespaces for a match. So, to find **ILogger**, it will try to match the following names in order:

1. MyApp.Interfaces.ILogger, MyApp
2. System.ILogger, MyApp
3. MyApp.Interfaces.ILogger, mscorlib, 2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
4. System.ILogger, mscorlib, 2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

The search will stop at the first matching type.

The system uses simple string concatenation to create the type name it attempts to load. However, you cannot specify a namespace qualified name plus the type, **MyApp.Interfaces.ILogger, MyApp**, if you have any namespace elements in your configuration section, **<namespace name="System" />**.The namespace from the configuration section will be appended to the namespace, resulting in a search on the wrong name, **System.MyApp.Interfaces.ILogger**. You should put namespaces in the **<namespace>** elements instead of on the type names in the configuration file to avoid this possibility.

If you have a large number of assemblies and namespaces, then the system type search could take a significant amount of time to complete. Normally, containers are only configured at application startup, so this time hit will not be significant during the operation of your application. If you find it becomes a significant issue, then you should consider using an explicit alias for the types that take the greatest search times, since aliases are matched first.

When matching a name with a type, the configuration system performs the following steps in order. The first one to succeed stops the process:

1. Attempt to load a type using the name directly (treated as a full type name)
2. Attempt to match a name to an alias
3. Do automatic type search

## Default Aliases and Assemblies

Some types and assemblies are used frequently in Unity configuration files. The Unity configuration system provides a set of predefined default aliases so you do not have to explicitly add aliases for these common types. Any user-defined entries will overwrite the default ones.

> Aliases are case sensitive.

In addition to the default aliases, the Unity configuration system also automatically adds System and mscorlib assemblies to the list of assemblies that are searched for types.

The following table has the complete list of pre-defined type aliases provided by Unity:

| Default Alias | Type |
|---|---|
| sbyte | System.SByte |
| short | System.Int16 |
| int | System.Int32 |
| integer | System.Int32 |
| long | System.Int64 |
| byte | System.Byte |
| ushort | System.UInt16 |
| uint | System.UInt32 |
| ulong | System.UInt64 |
| float | System.Single |
| single | System.Single |
| double | System.Double |
| decimal | System.Decimal |
| char | System.Char |
| bool | System.Boolean |
| object | System.Object |
| string | System.String |
| datetime | System.DateTime |
| DateTime | System.DateTime |
| date | System.DateTime |
| singleton | Microsoft.Practices.Unity.ContainerControlledLifetimeManager |
| ContainerControlledLifetimeManager | Microsoft.Practices.Unity.ContainerControlledLifetimeManager |

| | |
|---|---|
| transient | Microsoft.Practices.Unity.TransientLifetimeManager |
| TransientLifetimeManager | Microsoft.Practices.Unity.TransientLifetimeManager |
| perthread | Microsoft.Practices.Unity.PerThreadLifetimeManager |
| PerThreadLifetimeManager | Microsoft.Practices.Unity.PerThreadLifetimeManager |
| external | Microsoft.Practices.Unity.ExternallyControlledLifetimeManager |
| ExternallyControlledLifetimeManager | Microsoft.Practices.Unity.ExternallyControlledLifetimeManager |
| hierarchical | Microsoft.Practices.Unity.HierarchicalLifetimeManager |
| HierarchicalLifetimeManager | Microsoft.Practices.Unity.HierarchicalLifetimeManager |
| resolve | Microsoft.Practices.Unity.PerResolveLifetimeManager |
| perresolve | Microsoft.Practices.Unity.PerResolveLifetimeManager |
| PerResolveLifetimeManager | Microsoft.Practices.Unity.PerResolveLifetimeManager |

## Generic Types

The CLR type name syntax for generic types is extremely verbose, and it also does not allow for things like aliases. The Unity configuration system allows for a shorthand syntax for generic types that also allows for aliases and type searching.
To specify a closed generic type, you provide the type name followed by the type parameters in a comma-separated list in square brackets.
The Unity shorthand would look like the following example.

**XML**

```
<container>
    <register type="IDictionary[string,int]" </register>
</container>
```

If you wish to use an assembly name-qualified type as a type parameter, rather than an alias or an automatically found type, you must place that entire name in square brackets, as shown in the following example:

**XML**

```
<register type="IDictionary[string, [MyApp.Interfaces.ILogger, MyApp]]"/>
```

To specify an open generic type you simply leave out the type parameters. You have two options:

- Use the CLR notation of `N where N is the number of generic parameters.
- Use the square brackets, with commas, to indicate the number of generic parameters.

| Generic Type | Configuration file XML using CLR notation | Configuration file XML using comma notation |
|---|---|---|
| IList<T> | IList`1 | IList[] |
| IDictionary<K,V> | IDictionary`2 | IDictionary[,] |

# The Unity Configuration Schema

Unity 2.0 uses a new streamlined configuration schema for configuring Unity.
The following sections describe the schema configuration elements, their child elements, and their attributes in more detail:



## The <unity> Configuration Section

The top-level element for the configuration section is defined by the name that you specify in the **<configSections>** tag when adding it to the configuration file. Typically, the element is **<unity>,** as shown in the following XML example. However, you can choose another name for this section.

**XML**

```
  <configSections>
    <section name="unity"
type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
Microsoft.Practices.Unity.Configuration"/>
  </configSections>

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

```
    <alias alias="ILogger" type="MyApp.ILogger, MyApp" />
    <namespace name="MyApp.Implementations" />
    <assembly name="MyApp" />

    <container>
      …
    </container>

  </unity>
```

The following attribute is the only attribute available for the Unity section:

| Attribute | Description |
|---|---|
| **Xmlns** | XML namespace for this section. Not required for the configuration file to work, but if you want XML IntelliSense support in Visual Studio, set this to http://schemas.microsoft.com/practices/2010/unity. This element is optional. |

The following child elements are available for the **<unity>** section:

| Element | Number | Description |
|---|---|---|
| **<alias>** | Many | Creates a type alias. This element is optional. |
| **<namespace>** | Many | Adds a namespace to the namespace search list. This element is optional. |
| **<assembly>** | Many | Adds an assembly to the assembly search list. This element is optional. |
| **<sectionExtension>** | Many | Adds a section extension, which adds new supported elements to the configuration schema. This element is optional. |
| **<container>** | Many | A set of container configurations. This element is optional. |

## The <container> Element

The **<container>** element contains a set of configurations for a Unity container instance. Within that element, there can be child elements describing type mapping, injection configuration, instance creation, container extensions, or other options made available through any added section extensions. The following table lists the attribute for the **<container>** element.

| Attribute | Description |
|---|---|
| **name** | Name of this container configuration. One container element in the section may omit the name attribute; all others must have unique names. The unnamed <container> element is considered the default, and is the one that will be loaded if a container name is omitted when calling container.LoadConfiguration. This attribute is optional. For more information see Loading Configuration File Information. |

The following example shows the usage of the **<container>** element.

**XML**

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <container> <!-- default container -->
  … content here ...
```

```
  </container>
  <container name="otherContainer">
  ... content here ...
  </container>
</unity>
```

The following table lists the child elements for the **<container>** element.

| Element | Number | Description |
|---|---|---|
| **<register>** | Many | Registration information for a type. This element is optional. |
| **<instance>** | Many | Creates an instance directly in the container. This element is optional. |
| **<extension>** | Many | Adds a container extension to the Unity container. This element is optional. |

For more information see the following:

- The <register> Element
- The <instance> Element
- The <extension> Element

## The <register> Element

The <**register**> element is the basic building block for any configuration file. It enables you to specify type mappings and injection configuration for a type. If you specify a name, that name is used for the type mapping. If you do not specify a name, it creates a default mapping for the specified types. You can specify a lifetime manager for each mapping. If no explicit lifetime manager is configured for a type, it uses the transient lifetime manager.

The following table lists the attributes for the <**register**> element.

| Attribute | Description |
|---|---|
| **type** | The type that is being registered. This is the type that will be requested when calling the **Resolve** method. This attribute is required. |
| **name** | The name of the registration; if omitted the default registration for the type will be created. This attribute is optional. |
| **mapTo** | Type which will actually be created and returned when **Resolve** is called. This sets up a type mapping. It can be a user-defined alias or one of the default aliases. This attribute is optional. |

The following example shows the common usage for the <**register**> element.

**XML**

```
<container>
  <register type="MyService"> ... </register> <!-- Default registration for type
MyService -->
  <register type="ILogger" mapTo="EventLogLogger" /> <!-- type mapping -->
  <register type="ILogger" mapTo="PageAdminLogger" name="emergency" /> <!-- named
registration -->
</container>
```

To register a type mapping at run time, use the **RegisterType** method. The following example registers **EventLogLogger** at run time.

**C#**

```
// Register a default (un-named) type mapping with a transient lifetime
// Specify the registered type as an interface or object type
// and the target type you want returned for that type
myContainer.RegisterType<ILogger, EventLogLogger>();
```

**Visual Basic**

```
' Register a default (un-named) type mapping with a transient lifetime
' Specify the registered type as an interface or object type
' and the target type you want returned for that type
myContainer.RegisterType(Of ILogger, EventLogLogger)()
```

For more information on registering a type at run time see Registering Types and Type Mappings.

The following table lists the child elements for the **<register>** element. When using interception configuration extension, **Microsoft.Practices.Unity.InterceptionExtension.Configuration.InterceptionConfigurationExtension**, additional elements are valid children of a **<register>** element. For more information see Interception Configuration Schema Elements.

| Element | Number | Description |
|---|---|---|
| **<lifetime>** | One | The type that manages instances created by the container. This element is optional. |
| **<constructor>** | One | Configures the constructor to be called when creating instances. This element is optional. |
| **<property>** | Many | Configures properties that will be set when creating instances. This element is optional. |
| **<method>** | Many | Configures methods that will be called when creating instances. This element is optional. |

For more information see the following:

- The <lifetime> Element
- The <constructor> Element
- The <property> Element
- The <method> Element

---

## The <lifetime> Element

The **<lifetime>** element specifies which lifetime manager will be used to manage instances created for a type. If omitted, lifetime defaults to the **TransientLifetimeManager**. See Understanding Lifetime Managers for details about lifetime managers. The following example shows the common usage for the **<lifetime>** element

**XML**

```
<register type="ILogger" mapTo="SerialPortLogger">
  <!-- Simple use of lifetime - singleton instance -->
  <lifetime type="singleton" />
</register>
<register type="TypeWithCustomLifetime">
<!-- Use a Lifetime manager instance created by a type converter.
     Type converter gets passed extra information provided in
     in value attribute -->
  <lifetime type="SessionLifetimeManager"
            value="Session#1" typeConverter="SessionLifetimeConverter" />
</register>
```

The following table lists the attributes for the **<lifetime>** element.

| Attribute | Description |
|---|---|
| **type** | The type of the lifetime manager to create. Type must derive from LifetimeManager. Can be a user-defined alias or one of the default aliases. This attribute is required. |
| **typeConverter** | The type of type converter to use when creating the lifetime manager. If given, the type converter's ConvertFrom method will be called to create the lifetime manager, and pass the contents of the value attribute. If not specified, the default converter for the specified type is used. Aliases are allowed. This attribute is optional. |
| **value** | Any value required to initialize the lifetime manager. This attribute is optional. |

The **<lifetime>** element has no valid child elements.

## The <constructor> Element

The **<constructor>**element configuration for the constructor for this type and contains details of the constructor injection requirements for this object type. The **constructor** element has no attributes. You can include only one **<constructor>** element in each **register** section. The example shows the common usage for the **<constructor>** element.

**XML**

```
<register type="ILogger" mapTo="SerialPortLogger">
    <!-- Call the constructor with one parameter named "port" -->
    <constructor>
        <param name="port" value="COM1:" />
    </constructor>
</register>
```

The following table lists the child element for the **<constructor>** element.

| Element | Number | Description |
|---|---|---|
| **<param>** | Many | Specifies the parameters of the constructor to call. If no **<param>** child elements are present, it indicates that the zero-argument constructor should be called. This element is optional. |

For more information see The <param> Element.

You can specify a constructor for the injection of a specific named instance. In the following example, **InjectionConstructor** indicates which constructor to use based on its arguments, the string "UI," and causes the **TraceSourceLogger** constructor with a single string parameter to be invoked.

**C#**

```
container.RegisterType<ILogger, TraceSourceLogger>(
            "UI",
            new InjectionConstructor("UI"));
container.RegisterType<DriveController>(
          new InjectionProperty("MyProperty"));
container.RegisterType<DriveController>(
           new InjectionMethod("InitializeMe", 42.0,
           new ResolvedParameter(typeof(ILogger), "SpecialLogger")));
```

**Visual Basic**

```
container.RegisterType(Of ILogger, TraceSourceLogger) _
    ("UI", New InjectionConstructor("UI"))
container.RegisterType(Of DriveController) _
    (New InjectionProperty("MyProperty"))
container.RegisterType(Of DriveController) _
    (New InjectionMethod ( _
        "InitializeMe", _
        42, _
        New ResolvedParameter(GetType(ILogger), "SpecialLogger")))
```

For more information on using injection at run time see Registering Injected Parameter and Property Values.

## The <property> Element

The **<property>**element configures a property for injection. It contains details of the property injection requirements for this object type. You can include one or more **<property>**elements in each **<register>** element. See Specifying Values for Injection for details on how to specify what value to inject for the property. No explicit value means to inject the default value for the type of the property.

**XML**

```
<register type="ILogger" mapTo="SerialPortLogger">
    <!-- Inject the property "Settings" -->
    <property name="Settings" />
</register>
```

The following table lists the attributes for the **<property>** element.

| Attribute | Description |
|---|---|
| **name** | The name of the property. This attribute is required. |
| **dependencyName** | If present, resolve the value of the property using this name rather than the default. This attribute is optional. |

| | |
|---|---|
| **dependencyType** | If present, indicates the type to resolve for the value of the property. If not given, the type of the property is the one resolved. This attribute is optional. |
| **value** | Value to store in the property. This string is converted to the type of the property using the default TypeConverter for that type. This attribute is optional. |

The **<property>** element can take a single child element, which can be one of the following:

| Element | Number | Description |
|---|---|---|
| **<dependency>** | One | Specifies how to resolve the value to store in the property. This element is optional. |
| **<value>** | One | Specifies a literal value to be stored in the property. This element is optional. |
| **<optional>** | One | Specifies that an optional value should be resolved for the property. This element is optional. |
| **<array>** | One | Configures injection of an array value for properties of array types. This element is optional. |

For information about the child elements of the **<property>** element, see the following:

- The <dependency> Element
- The <value> Element
- The <optional> Element
- The <array> Element

## The <method> Element

The **<method>** element configures a method that will be called as part of creating an instance. It contains details of the method injection requirements for this object type. You can include one or more **<method>** elements for each <**register>** element.

**XML**

```
<register type="MyLogger">
    <method name="Initialize">
        <param name="loggerSettings" />
    </method>
</register>
```

The following table lists the attributes for the **<method>** element.

| Attribute | Description |
|---|---|
| **name** | The name of the method to call. This attribute is required. |

The **<method>** element can take the following child elements. These correspond to the parameters of the method you want to specify.

| Element | Number | Description |
|---|---|---|
| **<param>** | Many | Specifies the parameters of the method to call. If no **<param>** child elements are present, it means that the method has no arguments This element is optional. |

For more information see the <param> element.

## The <param> Element

The **<param>** element specifies a parameter for constructor or method injection. It is used to determine which constructor or method overload is called, based on the names (and optionally types) of the parameters for the constructor or method.

**XML**

```
<constructor>
    <param name="param1" />
    <param name="param2" value="Hello World!" />
</constructor>
```

See Specifying Values for Injection for more information about how the values for the parameters are provided.

The following table lists the attributes for the **<param>** element.

| Attribute | Description |
|---|---|
| **name** | The name of the parameter. This attribute is required. |
| **type** | The type of parameter. This attribute is only required if there are two different overloads with the same parameter name and you need to differentiate between them based on type. Normally, the parameter name alone is sufficient. This attribute is optional. |
| **dependencyName** | Name to use to resolve dependencies. This attribute is optional. |
| **dependencyType** | The type of dependency to resolve. This attribute is optional. |
| **value** | Value to inject for this parameter. This attribute is optional. |

The **<param>** element can take one of the following child elements. Only one child element is allowed.

| Element | Number | Description |
|---|---|---|
| **<dependency>** | One | Resolve value for parameter through the container. This element is optional. |
| **<optional>** | One | Resolve optional value through the container. This element is optional. |
| **<value>** | One | Gives explicit value to use for parameter value. This element is optional. |
| **<array>** | One | If parameter is of an array type, specifies what values to put into the resolved array. This element is optional. |

For information about the child elements of the **<param>** element, see the following:

- The <value> Element
- The <dependency> Element
- The <optional> Element
- The <array> Element

Value elements are a schema extension point, so other child elements may be allowed depending on which schema extensions are available. See Extending the Unity Configuration Schema.

## The <dependency> Element

The **<dependency>** element is used when a value is to be resolved through the container. By default, with no other configuration, this element results in a callback into the container for the default value of the type of the containing parameter or property. Attributes may be used to customize what is resolved. If the value cannot be resolved, then an exception is thrown at resolve time. The following example shows the common usage for the **<dependency>** element.

**XML**

```
<constructor>
    <param name="param1">
        <dependency name="otherRegistration" />
    </param>
</constructor>
```

The following table lists the attributes for the **<dependency>** element.

| Attribute | Description |
|---|---|
| **name** | The name of the named type or mapping registered in the container to use to resolve this dependency. If omitted, the default registration will be resolved. This attribute is optional. |
| **type** | Type to resolve from the container. It must be a type compatible with the type of the parameter or property. If omitted, Unity resolves the type based on the type of the parent parameter or property. This attribute is optional. |

The **<dependency>** element has no valid child elements.

## The <value> Element

The **<value>** element is used to specify a specific value for a parameter or property. This element lets you specify a string that is then passed through a **TypeConverter** to create the actual object to be injected. If no specific type converter type is given, then the default type converter for the type of parameter or property is used.

The string is converted to an object using the invariant culture; if you wish to use a locale-specific culture you must provide a custom **TypeConverter** to do so.

The following table lists the attributes for the **<value>** element.

| Attribute | Description |
|---|---|

| | |
|---|---|
| **value** | The string value to be converted to an object. This attribute is required. |
| **typeConverter** | Type of the **TypeConverter** to use to convert the value to an object. If omitted, the default type converter for the containing parameter or property type will be used. This attribute is optional. |

The following example shows the common usage of the **<value>** element.

**XML**

```
<constructor>
    <param name="param1">
        <value value="42" />
    </param>
    <param name="param2">
        <value value="aieou" typeConverter="VowelTypeConverter" />
    </param>
</constructor>
```

The **<value>** element has no valid child elements.

## The <optional> Element

The **<optional>** element is used when a value should be resolved through the container but its presence is optional. The container will try to resolve this parameter or property using the types and mappings registered in the container, but if the resolution fails, the container returns **null** instead of throwing an exception.

The following table lists the attributes for the **<optional>** element.

| Attribute | Description |
|---|---|
| **name** | The name of the named type or mapping registered in the container to use to resolve this optional dependency. If omitted the default registration will be resolved. This attribute is optional. |
| **type** | The type to use to resolve an optional dependency mapping or registration. Must be a type compatible with the type of the parameter or property. If you do not specify a type, Unity will resolve the type of the parameter or property. This attribute is optional. |

**XML**

```
<constructor>
    <param name="param1">
        <optional name="otherRegistration" />
    </param>
</constructor>
```

The **<optional>** element has no valid child elements.

## The <array> Element

The <**array>** element can be used to customize which elements are returned in the array if a parameter or property is of an array type. See Specifying Values for Injection for more details.

The <**array>** element takes no attributes.

The <**array>** element has a collection of zero or more child elements. If there are no child elements, an empty, zero-length array for both generic and non-generic types is injected. If there is more than one child, any value element such as **<dependency>**, **<optional>**, or **<value>** may be used, and each one corresponds to an element in the returned array.

If you want to have the default container behavior for arrays, do not specify the <**array>** element; use the **<dependency>** element instead.

The following example shows the common usage of the <**array>** element.

**XML**

```
<register type="ILogger" mapTo="NetworkLogger">

    <!-- Empty array -->
    <property name="noSettings">
       <array />
    </property>

    <!-- Type NetworkSettings[], get default array injection -->
    <property name="allSettings" />

    <!--Type NetworkSettings[], get only the specified three values -->
    <property name="someSettings">
        <array>
            <dependency name="fastSettings" />
            <optional name="turboSettings" />
             <value value="port=1234;protocol=tcp;timeout=60”
                typeConverter="ConnectionSettingsTypeConverter" />
        </array>
    </property>
</register>
```

## The <extension> Element

The **<extension>** element is used to add a Unity container extension to a container and contains the list of extensions to register with the Unity container.

The following example shows the common usage for the **<extension>** element.

**XML**

```
<container>
    <extension
type="Microsoft.Practices.Unity.Interception.InterceptionExtension,
Microsoft.Practices.Unity.Interception" />
</container>
```

The following example adds an extension at run time. This code creates a new instance of an extension class named **MyCustomExtension** that you have previously created, and which resides in this or a referenced project, and adds it to the container.

**C#**

```
IUnityContainer container = new UnityContainer();
container.AddNewExtension<MyCustomExtension>();
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer()
container.AddNewExtension(Of MyCustomExtension)()
```

For more information on adding extensions at run time see Registering Container Extensions.

The following table describes the single attribute for the **<extension>** element.

| Attribute | Description |
|---|---|
| **type** | The type of extension to add to the container. Can be a user-defined alias or one of the default aliases. This attribute is required. |

There are no child elements for the **<extension>** element.

## The <instance> Element

The **<instance>** element is used to specify a value to be placed in the container and to indicate that you do not want that value to be created by the container but that it will be created by using a **TypeConverter** and then placed in the container by using the **RegisterInstance** method.

Instances added through configuration tend to be simple because it is hard to represent them with a string value, but they could be arbitrarily complex if there is a type converter that can handle the conversion from a string.

You cannot use an existing instance with the **<instance>** element. The **<instance>** section results in the creation of a new instance, much like something registered with **RegisterType** would. This is in contrast to the use of **RegisterInstance**, which requires the user to create the instance manually and then register it.

The following table lists the attributes for the **<instance>** element.

| Attribute | Description |
|---|---|
| **name** | The name to use when registering this instance. This attribute is optional. |
| **type** | The type of the value. Can be a user-defined alias or one of the default aliases. If omitted, the assumed type is **System.String**. This attribute is optional. |
| **value** | The value to pass to the type converter to create the object. If omitted, **null** is passed to the type converter. This attribute is optional. |

| | |
|---|---|
| **typeConverter** | The type converter to use to construct the value. If not specified, the default converter for the type of the value is used. This attribute is optional. |

The following example shows how the **<instance>** element is used in a configuration file.

**XML**

```
<container>
    <!-- A named string value -->
    <instance name="NorthwindDb" value="SqlConnectionStringHere" />

    <!-- A typed value -->
    <instance type="ConnectionSettings" value="port=1234"
typeConverter="ConnectionSettingsTypeConverter" />
</container>
```

The following example shows how you can register instances at run time. The following example creates a default (unnamed) registration for an object of type **EmailService** that implements the **IMyService** interface.

**C#**

```
EmailService myEmailService = new EmailService();
myContainer.RegisterInstance<IMyService>(myEmailService);
```

**Visual Basic**

```
Dim myEmailService As New EmailService()
myContainer.RegisterInstance(Of IMyService)(myEmailService)
```

The **<instance>** element has no child elements.

## The <namespace> Element

The **<namespace>** element is used to declare namespaces that will be automatically searched for types, as explained in the section Specifying Types in the Configuration File.

The following table lists the attribute for the **<namespace>** element.

| Attribute | Description |
|---|---|
| **name** | The name of the namespace that is to be searched for the types. This attribute is required. |

The following example shows how the **<namespace>** element is used.

**XML**

```
<namespace name="ObjectsInstancesExamples.MyTypes" />
```

See Specifying Types in the Configuration File for more examples.

The **<namespace>** element has no allowed child elements.

## The <assembly> Element

The **<assembly>** element is used to declare assemblies that Unity will automatically search for types, as described in the section Specifying Types in the Configuration File.

The following table lists the attribute for the **<assembly>** element.

| Attribute | Description |
|---|---|
| **name** | The name of the assembly to search. The assembly name must be sufficiently qualified for the common language runtime (CLR) to load it. Assemblies in the GAC, for instance, must have a fully-qualified assembly name. This attribute is required. |

The following example shows how the **<assembly>** element is used. See Specifying Types in the Configuration File for more examples.

**XML**

```
<assembly name="ObjectsInstancesExamples" />
```

See the Automatic Type Lookup section for the search rules for matches.

The **<assembly>** element has no allowed child elements.

## The <alias> Element

The **<alias>** element is used to define a type alias, as explained in the section Specifying Types in the Configuration File.

The following table lists the attributes for the **<alias>** element.

| Attribute | Description |
|---|---|
| **alias** | The alias used to refer to the specified type. This attribute is required. |
| **type** | The type that the alias refers to. This name must be a CLR type name. Aliases or automatic type lookups do not apply with this attribute. This attribute is required. |

The following example shows how the **<alias>** element is used.

**XML**

```
<unity>
    <alias alias="MyLogger" type="MyNamespace.MyLogger, MyProject" />
    <alias alias="AGenericType" type="MyNamespace.MyGenericType`1, MyProject" />
</unity>
```

The **<alias>** element has no valid child elements.

## The <sectionExtension> Element

The **<sectionExtension>** element is used to load a schema extension. Unlike the **<extension>** element, which is used to load a container extension object to modify the run-time behavior of a Unity container instance, this element loads a schema extension, which modifies the allowed elements in the configuration file itself. This is a section-level element; it must occur outside any **<container>** elements.

See Extending the Unity Configuration Schema for more details.

The following table describes the attributes for the **<sectionExtension>** element.

| Attribute | Description |
|---|---|
| **type** | The type that implements the section extension. This attribute is required. |
| **prefix** | Specifies a prefix that will be appended to all the extension elements. This allows you to avoid collisions if two different extensions add an element with the same name. If left out, no prefix will be added. This attribute is optional. |

**XML**

```
<unity>
    <sectionExtension type="MyProject.MySectionExtension, MyProject" />

    <sectionExtension prefix="ext" type="MyOtherSectionExtension" />
    ...
</unity>
```

The **<sectionExtension>** element has no allowed child elements.

# Specifying Values for Injection

This topic explains how to configure the information required so that Unity will automatically populate constructor and method parameters and property values when it resolves instances of types. The **<param>** and **<property>** elements both let you specify a value to be supplied for the parameter or property, respectively. There are many different kinds of values that can be specified, each with a separate element. In addition, the Unity configuration schema supports a shorthand notation for the most common cases.

For more information about the format of the Unity configuration file, see Using Design-Time Configuration.

This topic contains the following sections describing values for injection:

Resolving Values from the Container

Giving Values in Configuration

Configuring Array Values

## Resolving Values from the Container

Probably the most common use is the case when you want the value for a parameter or property to be resolved from the container. You can use the <dependency> element to accomplish this. The type of the dependency to resolve is, by default, the type of the parameter or property being injected. If you want to use a different type, then you must specify the type attribute on the **<dependency>** element. Also, the container will resolve the default value for that type unless you use the name attribute to specify a named registration to resolve.

There are several shorthand attributes to expedite the configuration process. If you want to resolve the default registration for the type of the parameter or property, you can simply leave out the **<dependency>** element, as shown in the following example.

**XML**

```
<register type="ILogger" mapTo="SerialPortLogger">
    <property name="Settings" />
</register>
```

Similarly, since specifying a dependency is so common, instead of using the child **<dependency>** element, you can use the **dependencyType** and **dependencyName** attributes on the containing elements. The following example uses the **<dependency>** element.

**XML**

```
<!-- Explicit dependency element -->
<register type="ILogger" mapTo="SerialPortLogger">
    <property name="Settings">
        <dependency type="SpecialPortSettings" name="highSpeedSettings" />
    </property>
</register>
```

The following example uses attributes instead of the **<dependency>** element to accomplish the same configuration.

**XML**

```
<!--Use attributes instead of the dependency element -->
<register type="ILogger" mapTo="SerialPortLogger">
    <property name="Settings" dependencyType="SpecialPortSettings”
        dependencyName="highSpeedSettings" />
</register>
```

The previous examples use the **<property>** element, but the attributes can also be used on the **<param>** element in exactly the same way.

Optional values, specified by using the **<optional>** element, are resolved from the container. If the container cannot resolve an optional value, for example when attempting to resolve an interface that is not registered, then instead of getting an exception, as you would for a normal dependency, you get **null** injected. There are no shorthand attributes for optional dependencies; you must use the explicit child element.

## Giving Values in Configuration

The other most common scenario is to specify a particular value to be injected, such as a string or integer. The explicit way to do is by using the <value> element. This lets you specify a string, which will be passed to a **TypeConverter** to create the underlying object. The type of the value is the type of the enclosing property or parameter.

Like the **<dependency>** element, the **<value>** element has shorthand attributes. If you only need the default type converter to turn the string into a value, which is commonly the case for most value types like strings and numbers, you can simply use the **value** attribute on your enclosing parameter or property, as shown in the following example.

**XML**

```
<register type="ILogger" mapTo="NetworkLogger">
    <constructor>

        <!-- Explicit value child element -->
        <param name="logHost">
            <value value="loghost.example.org" />
        </param>

        <!-- using shorthand attribute -->
        <param name="header" value="Log Entry:" />

        <!-- custom typeconverter, no shorthand available -->
        <param name="connectionSettings">
            <value value="port=123;protocol=tcp;timeout=30"
                typeConverter="ConnectionSettingsTypeConverter" />
        </param>
    </constructor>
</register>
```

## Configuring Array Values

The container handles parameter or properties of array types uniquely. There are several options for how to configure array values; the choice depends on what you want to accomplish. See Registering Injected Parameter and Property Values.

The simplest approach for arrays is the default behavior of the container for arrays. In that case simply use the **<dependency>** child element or, for the equivalent shorthand attribute, just leave the attribute out.

If you wish to have only explicit values injected into the array, then you can use the <array> element with child elements. This element lets you specify explicitly what values you want to have injected.

If you want an empty array, use **<array>** with no child elements. If you want an array with values, include as children a set of other value elements, such as **<dependency>**, **<optional>**, or **<value>** to give the values for the various array elements. For example:

**XML**

```
<register type="ILogger" mapTo="NetworkLogger">
```

```
    <!-- Empty array -->
    <property name="noSettings">
       <array />
    </property>

    <!-- Type NetworkSettings[], get default array injection -->
    <property name="allSettings" />

    <!--Type NetworkSettings[], get only the specified three values -->
    <property name="someSettings">
        <array>
            <dependency name="fastSettings" />
            <optional name="turboSettings" />
             <value value="port=1234;protocol=tcp;timeout=60"
                typeConverter="ConnectionSettingsTypeConverter" />
        </array>
    </property>
</register>
```

# Extending the Unity Configuration Schema

The Unity container is highly customizable. No one fixed configuration format can cover everything that you might want to do with the container. As a result, the Unity configuration system itself is extensible, allowing you to add new valid elements to your configuration file. The **<sectionExtension>** element allows you to load the code that adds these new options to the configuration file. This lets you specify an implementation of the **SectionExtension** type.
Section extensions can do the following to the configuration:

- Add new predefined aliases
- Add new container-level elements
- Add new registration-level elements
- Add new value-level elements

You can create your own custom extensions, or use extensions created by third parties, with Unity. Unity also uses default extensions to implement its own functionality. For information about using extensions, see Creating and Using Container Extensions.

One example of a section extension is the **InterceptionConfigurationExtension** section extension, which ships with the Unity package. This section extension adds the following elements and aliases to the schema:

- Aliases are defined for each of the types (like **VirtualMethodInterceptor**, **TransparentProxyInterceptor**, and various matching rules) that are used by the interception configuration.
- The **<interception>** element is added as a valid element child element for the **<container>** element.
- The **<interceptor>**, **<interceptionBehavior>**, **<addInterface>**, and **<policyInjection>** elements are added as valid child elements for the **<register>** element.

This extension mechanism allows for almost unlimited extensibility of the configuration file on an opt-in basis. Though the schema extension will modify the schema allowed at run time, it does not modify the XSD file used by Visual Studio IntelliSense. As a result, you will still get warnings in the Visual Studio XML editor even though the file will work fine at run time. In order to resolve this problem, the section extension author must provide an updated XSD document for use with their extension.

> The **InterceptionConfigurationExtension** is supported by the schema shipped with Unity.

The **<sectionExtension>** element also accepts a user-provided prefix attribute. This is useful in the case where two section extensions both provide extension elements with the same name. In the case of a collision, you can specify a prefix for one or both section extensions, and then use that prefix to disambiguate them.
Consider two schema extensions, both of which add a **<containerCustomization>** element. Using the prefix attribute, a configuration file that uses both would look like the following example.

**XML**

```
<unity>
    <sectionExtension prefix="ext1" type="MyFirstExtension, MyStuff" />
    <sectionExtension prefix="ext2" type="MySecondExtension, MyOtherStuff" />
    <container>
        <ext1.containerCustomization />
        <ext2.containerCustomization />
    </container>
</unity>
```

# Configuration Files for Interception

Unity 2.0 treats interception like any extension you add to Unity. As with any extension in Unity 2.0, the Unity interception mechanism can be configured through either the API or through a Unity configuration section.

> Unity provides partial backward compatibility for implementing interception through a container. Earlier versions used a container extension named **InterceptionExtension**, which resides in the assembly named **Microsoft.Practices.Unity.Interception.dll**. To configure interception, you specify

this extension in the <**extensions>** element of your application configuration, and then define the behavior of interception in the <**extensionConfig**> section.

Using the **extension** and **register** elements in Unity 2.0 is comparable to **Interceptor** element use in the **extensionConfig** section in earlier versions.

For more information on backward compatibility, see Reusing Configuration Files Based on a Previous Schema.

For more information about Unity 1.2 interception, see Using Interception with Unity on MSDN.

This topic contains the following sections to describe the interception configuration file:



## Using the Configuration File to Enable Interception

Interception is not part of the default Unity configuration schema. Before you can configure interception you must add the correct **<sectionExtension>** element to your configuration section in the configuration file. The interception configuration extension is the type **Microsoft.Practices.Unity.InterceptionExtension.Configuration.InterceptionConfigurationExtension**, in the Microsoft.Practices.Unity.Interception.Configuration assembly.

The following extract from a configuration file adds the interception extension, **InterceptionConfigurationExtension**, by using the <**sectionExtension>** element.

**XML**

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <sectionExtension
type="Microsoft.Practices.Unity.InterceptionExtension.Configuration.InterceptionC
onfigurationExtension, Microsoft.Practices.Unity.Interception.Configuration" />

</unity>
```

Loading the interception section extension supplies a set of Standard Interception Aliases and additional configuration elements.

Loading the section extension only enables the interception configuration to be given in the configuration file. Interception itself will not work unless you also load the interception container

extension in your Unity container instance. This can also be done in the configuration file, as shown in the following example.

**XML**

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <sectionExtension
type="Microsoft.Practices.Unity.InterceptionExtension.Configuration.InterceptionC
onfigurationExtension, Microsoft.Practices.Unity.Interception.Configuration" />

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

</unity>
```

You do not need to explicitly alias the interception container extension's type or add a **<namespace>** or **<assembly>** element. When you add the **InterceptionConfigurationExtension** to the **<sectionExtension>** element, the alias "Interception" is automatically added, along with the other default aliases for common types used in interception configuration.

## Standard Interception Aliases

The following table contains the list of predefined type aliases provided by the Unity interception extension. All of these types are in the Microsoft.Practices.Unity.Interception.dll assembly and the Microsoft.Practices.Unity.InterceptionExtension namespace.

| Alias | Description |
|---|---|
| Interception | The interception container extension. |
| IInterceptionBehavior | Interface for interception behaviors. |
| PolicyInjectionBehavior | Behavior implementing policy injection. |
| Policy Injection Types | Policy injection types. |
| IMatchingRule | The matching rule interface. |
| ICallHandler | The policy injection call handler interface. |
| Policy Injection Matching Rules | Policy injection matching rules. |
| AssemblyMatchingRule | Match based on being in a particular assembly. |
| CustomAttributeMatchingRule | Match based on having a given attribute. |
| MemberNameMatchingRule | Match based on member name. |
| ParameterTypeMatchingRule | Match based on parameter types. |
| PropertyMatchingRule | Match based on a property name. |
| TagAttributeMatchingRule | Match based on having the Tag attribute. |
| TypeMatchingRule | Match based on type. |
| Interceptor Types | Interceptor type. |

| | |
|---|---|
| VirtualMethodInterceptor | Virtual method type interceptor. |
| InterfaceInterceptor | Interface interceptor. |
| TransparentProxyInterceptor | Transparent proxy interceptor. |

## Enabling Interception on a Type

To turn on interception for a type in the container, you specify the interceptor and behaviors using the following child elements of the **<register>** element.

- <interceptor>
- <interceptionBehavior>
- <policyInjection>
- <addInterface>

The following example turns on interception using the transparent proxy interceptor and performs policy injection.

**XML**

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <sectionExtension
type="Microsoft.Practices.Unity.InterceptionExtension.Configuration.InterceptionC
onfigurationExtension, Microsoft.Practices.Unity.Interception.Configuration" />

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

    <register type="MyType">
      <!-- Other children, like constructor or property -->
      <interceptor type="TransparentProxyInterceptor" />
      <policyInjection />
    </register>
  </container>
</unity>
```

After loading this configuration, any objects of type **MyType** will be intercepted by using the transparent proxy interceptor and they will have the policy injection behavior applied to them.

## Configuring Policy Injection Policies

Policy injection polices can also be configured through the Unity configuration section. This section describes how to perform the configuration along with several options you can use when defining policies.

Policies are defined on a per-container basis inside an **<interception>** element. The following example defines two policies, with matching rules and a call handler applied.

**XML**

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <sectionExtension
type="Microsoft.Practices.Unity.InterceptionExtension.Configuration.InterceptionC
onfigurationExtension, Microsoft.Practices.Unity.Interception.Configuration" />

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

    <interception>
      <policy name="addDataAccessTypes">
        <matchingRule name="DataLayerMatch" type="NamespaceMatchingRule">
          <constructor>
            <param name="namespaceName" value="MyApp.DataAccess" />
          </constructor>
        </matchingRule>
        <callHandler name="LogHandler" type="LoggingCallHandler" />
        <callHandler name="SecurityHandler"
            type="DatabaseSecurityCheckHandler" />
      </policy>

      <policy name="webMethods">
        <matchingRule name="MatchWebRequestMethods" />
        <callHandler name="LogWebMethodHandler" type="LoggingCallHandler" />
      </policy>
    </interception>

    <register type="IMatchingRule" name="MatchWebRequestMethods"
        mapTo="MemberNameMatchingRule">
      <constructor>
        <param name="nameToMatch" value="Begin*Request" />
      </constructor>
    </register>
  </container>
</unity>
```

This example demonstrates the two most common approaches for defining matching rules and call handlers in a configuration file. The first approach, the inline style, declares the type and injection members directly within the policy definition. This is what was used in the first matching rule declaration, as shown in the following extract from the previous example.

**XML**

```
...
         <matchingRule name="DataLayerMatch" type="NamespaceMatchingRule">
          <constructor>
            <param name="namespaceName" value="MyApp.DataAccess" />
          </constructor>
        </matchingRule>
...
```

This approach is designed to work much like a **<register>** element. The only difference is that you cannot provide a type mapping. The underlying type given to the container is always **IMatchingRule**.

Any valid child element for the **<register>** element can be used with the **<matchingRule>** element (including **<lifetime>**, **<constructor>**, and **<property>** elements). In this example, the actual matching rule type is **NamespaceMatchingRule**, and when the matching rule object must be created, it is created by calling the constructor that takes a single string parameter named **namespaceName**. The value for that will be **MyApp.DataAccess**. The intent of this matching rule configuration is to match all methods on the types in that namespace.

The same design applies to the **<callHandler>** element, except that you are defining injection for a call handler (a type that implements **ICallHandler**) instead of a matching rule.

The second most common approach is to use a named reference, as in the following example with the **MatchWebRequestMethods** named reference.

**XML**

```
...
        <matchingRule name="MatchWebRequestMethods" />
...
    <register type="IMatchingRule" name="MatchWebRequestMethods"
        mapTo="MemberNameMatchingRule">
      <constructor>
        <param name="nameToMatch" value="Begin*Request" />
      </constructor>
    </register>
...
```

In this approach, a type of matching rule is not specified. Instead, a name is provided. At configuration time, the container will attempt to resolve an **IMatchingRule** with the name **MatchWebRequestMethods** that you specified. The result is that the container will look for a registration for the type **ICallHandler** with the name **MatchWebRequestMethods**. That registration is defined a few lines lower in the previous example file, which works because the order of definition in the file does not matter. The same design applies to the **<callHandler>** element.

The name is always required by the configuration system. However, when using an inline style registration, the name is ignored in the container.

The approach you choose to use is a matter of personal preference. The following are factors to consider when choosing an approach to use for policies, matching rules, and call handlers in your configuration file:

- If your policies are fairly small and initialization of the matching rules and call handlers is straightforward, the inline style will be the simplest approach. Everything related to your policies is then in one place.
- If you want to reuse a call handler definition or matching rule across policies, a named reference will facilitate this approach. Using a named reference lets you define the configuration for the call handler or matching rule in a single place and simply reference it from the policies. Choosing the name to describe what the matching rule or call handler's purpose is makes it easier to track them; for example, MatchWebRequestMethods is clearly for matching methods that do Web requests.

- If you have a large number of policies or many matching rules and call handlers, using named references will facilitate your work. Using the inline style can quickly bloat your policy definitions, making it hard to tell what the policies are and which ones are grouped together.

For more information on registering interception at run time, see Registering Interception.

For more information on registering policies, matchingRules, and callHandlers at run time see Registering Policy Injection Components.

## Legacy Interception Configuration

In earlier versions of Unity, enabling interception on a type was done using a separate element entirely. Unity 2.0 supports this approach as well, in order to allow some reuse of other configuration files. The syntax is not identical, so you will still need to copy and update your configuration file; you cannot use an older Unity configuration file directly. For new development, configuring interception through the **<register>** element as explained above is the recommended approach.

The **<interceptors>** element can appear within a **<container>** element, and you can then specify a series of interceptors and the types they should intercept.

**XML**

```
<container>
  <extension type="Interception" />
  <interceptors>
    <interceptor type="VirtualMethodInterceptor">
      <default type="wrappableVirtual"/>
    </interceptor>
    <interceptor type="TransparentProxyInterceptor">
      <key type="wrappable"/>
    </interceptor>
    <interceptor type="TransparentProxyInterceptor">
      <key type="wrappable" name="name"/>
    </interceptor>
  </interceptors>
</container>
```

The **<interceptor>** element lets you specify a single interceptor type. You can use the child elements of **<interceptor>** to specify the type that this interceptor should be applied to. See <the <default> Element and the <key> Element for more information.

When defining interception through the **<interceptors>** element, policy injection is automatically enabled for the types applied, and you cannot add any additional interception behaviors.

For more information on backward compatibility, see Reusing Configuration Files Based on a Previous Schema.

For more information about Unity 1.2 interception, see Using Interception with Unity on MSDN.

# Interception Configuration Schema Elements

When configuring files for interception, the following elements may appear as children of a **<register>** element. For more information see The register Element. These elements are used to describe the <**interceptors>** and <**policies>** elements, their child elements, and their attributes in more detail:

- The <interceptor> Element
- The <interceptionBehavior> Element
- The <addInterface> Element
- The <interception> Element
- The <policy> Element
- The <matchingRule> Element
- The <callHandler> Element
- The <interceptors> Element
- The interceptors <interceptor> Element
- The <default> Element
- The <key> Element

For more information about interception, and selecting the objects and their members to add a handler pipeline, see Using Interception and Policy Injection.

## The <interceptor> Element

The **interceptor** element specifies a type that should be intercepted and provides the type of interceptor to use.

The **<interceptor>** element is a valid child of the <register> element.

The following table lists the attributes for the **interceptor** element.

| Attribute | Description |
|---|---|
| **name** | The name to use when registering this interceptor. This attribute is required only when there are two or more entries for the same interceptor type. |
| **type** | The type of the interceptor. This attribute is required. |
| **isDefaultForType** | Default is false. If true, then this interceptor will be used for all registrations for this type, ignoring the name that the registration is under. This flag is useful if you are registering many implementations of the same interface with different names, and want all of them |

| | |
|---|---|
| | intercepted. Setting this to "true" lets you define interception once instead of in every registration. This attribute is optional. |

There are no valid child elements for the **<interceptor>** element.

## The <interceptionBehavior> Element

This **<interceptionBehavior>** element specifies which interception behaviors will be executed when the intercepted object is called. This element may appear multiple times, with each instance adding a separate behavior interface.

The **<interceptionBehavior>** element is a valid child of the <register> Element.

The following table lists the attributes for the **interceptionBehavior** element.

| Attribute | Description |
|---|---|
| **name** | The name, if provided, used to resolve the interceptor type. Behaviors are resolved out of the container. This attribute is optional. |
| **type** | The Type of behavior to create. This attribute is required. |
| **isDefaultForType** | Default is false. If true, then this behavior will be applied for all registrations for this type, ignoring the name that the registration is under. This flag is useful if you are registering many implementations of the same interface with different names, and want all of them intercepted. Setting this to **true** lets you define the behavior once instead of in every registration. This attribute is optional. |

There are no valid child elements for the **<interceptionBehavior>** element.

## The <addInterface> Element

This **<addInterface>** element lets you implement additional interfaces. It is possible for the interception system to add extra interfaces to the proxy classes that are created. These are interfaces above and beyond the interfaces that are implemented by the intercepted types. These interfaces must be implemented by an appropriate interception behavior. In most cases, the behavior itself will indicate when it adds an interface. There are occasions, such as for a mock object framework, when the behavior itself could implement any interface or a large set of interfaces. If you are using such a behavior, you need to explicitly indicate which interfaces to add. This element enables you to do that. This element may appear multiple times with each appearance adding an individual interface.

The **<addInterface>** element is a valid child of the <register> Element.

The following table lists the attributes for the **addInterface** element.

| Attribute | Description |
|---|---|
| **type** | The interface to add. This attribute is required. |

There are no valid child elements for the **<addInterface>** element.

**XML**

```
<register type="MyViewModel">
```

```
  <interceptor type="VirtualMethodInterceptor" />
  <interceptionBehavior type="SpecialSupportBehavior" />
 <addInterface type="ISupportInterface1" />
</register>
```

## The <interception> Element

This **<interception>** element is used to group together a set of policy injection policy definitions.

The **<interception>** element is a valid child of the <container> Element.

The **<interception>** element has no valid Attributes.

The **<interception>** element has one valid child element, the <policy> element.

## The <policy> Element

Each **policy** element is used to define a policy injection policy and specifies the complete configuration for a single policy.

The **<policy>** element is a valid child of the <interception> element.

The following table lists the attributes for the **policy** element.

| Attribute | Description |
|---|---|
| **name** | The name to use when registering this policy. All policies must have unique names. This attribute is required. |

The **policy** element has two valid child elements, the <matchingRule> Element and the <callHandler> Element, that specify details of the policy.

## The <matchingRule> Element

Each **matchingRule** element defines a matching rule object that will be used when determining if its containing policy will be applied in policy injection. All of the matching rules within a **matchingRules** element must evaluate to **True** for a policy to apply to a specific type or member of a type.

The **<matchingRule>** element can be used in multiple ways to define a matching rule object directly in the policy or to reference one that has been defined elsewhere in the container. See Configuring Policy Injection Policies for more information.

The **<matchingRule>** element is a valid child of the <policy> element.

The following table describes the attributes of the **matchingRule** element.

| Attribute | Description |
|---|---|
| **name** | The name by which code in Unity and the configuration tools will refer to this matching rule. This attribute is required. |

| | |
|---|---|
| **type** | The type of object that will be created and registered with the container. If omitted, the container will be used to resolve the matching rule. The container will search for a mapping of **IMatchingRule** with the name supplied in the **name** attribute<br>This attribute is not required if the types are registered elsewhere within the configuration, and the **name** attribute matches one of these named registrations. |

Matching rules are only used to determine which policies apply for each interceptable method. They are not used at run time.

The **matchingRule** element has the valid child element, the <**lifetime**> Element, and any other elements derived from **InjectionMemberElement**.

## The <callHandler> Element

Each **callHandler** element defines a call handler object that will be used when determining if its containing policy will be applied in policy injection. It provides the details of a single call handler. The **<callHandler >** element can be used in multiple ways to define a call handler object directly in the policy or to reference one that has been defined elsewhere in the container. See Configuring Policy Injection Policies for more information.

The **<callHandler>** element is a valid child of the <policy> element.

The following table describes the attributes of the **callHandler** element.

| Attribute | Description |
|---|---|
| **name** | The name of the call handler and the name by which code in Unity and the configuration tools will refer to this call handler. The name must be unique within the policy. This attribute is required. |
| **type** | The type of the call handler. If provided, this is the type of object that will be created and registered with the container. If omitted, the container will be used to resolve the matching rule, looking for a mapping of **ICallHandler** with the name supplied in the **name** attribute.<br>This attribute is not required if the types are registered elsewhere within the configuration, and the **name** attribute matches one of these named registrations. |

The **matchingRule** element has the valid child element, the <lifetime> element, and any other elements derived from **InjectionMemberElement**.

## The <interceptors> Element

The **<interceptors>** element is part of the legacy support for Unity 1.x configuration files. It is recommended that for new development you define interception directly in the <register> Element by using the <interception> element and the rest of the interception elements. See Legacy Interception Configuration for more information.

The **<interceptors>** element is a valid child of the <container> element.

There are no the attributes of the **interceptors** element.

The **<interceptors>** element has one valid child element, the <interceptor> element>.

## The interceptors <interceptor> Element

This **<interceptor>** element, when placed within the **<interceptors>** element, is used to enable interception on a type that is otherwise registered elsewhere in the configuration file.

> There are two **<interceptor>** elements. They are actually two different elements (different C# classes) that have the same name in the XML. The meaning changes based on context (where they are in the XML file. They can appear in the **<register>** element or in the **<interceptors>** element and are defined accordingly.

It is recommended that for new development you define interception directly in the <register> Element by using the <interceptor> Element and the rest of the interception elements. See Legacy Interception Configuration for more information about the **<interceptors>** element.

The following table lists the attributes for the **interceptor** element.

| Attribute | Description |
|---|---|
| **type** | The type of the interceptor to create. This attribute is required. |
| **value** | If a type converter is used to create the interceptor, this value is passed to the type converter. If left out, the type converter will receive **null**. This attribute is optional. |
| **typeConverter** | The type converter to use to create the interceptor. If omitted, the interceptor is created with **Activator.CreateInstance**. |

There are two valid child elements for the **<interceptor>** element, the <default> element and the <key> element.

## The <default> Element

The **<default>** element specifies which types to apply the containing interceptor to. The interceptor will be applied every time this type is resolved from the container, regardless of which name is used to resolve it.

It is recommended that for new development you define interception directly in the <register> Element by using the <interceptor> element and the rest of the interception elements

The **<default>** element is a valid child of the interceptors <interceptor> element. . See Legacy Interception Configuration for more information about the **<interceptors>** element and its child elements.

The following table lists the attributes for the **key** element.

| Attribute | Description |
|---|---|

| | |
|---|---|
| **type** | Type to apply the interceptor to. This attribute is required. |

There are no valid child elements for the **<default>** element.

### The <key> Element

The **<key>** element is used to specify which types to apply the containing interceptor to. The interceptor will be applied only to the type/name pair specified in the attributes of this element. The **name** and **type** attributes of the **<key>**element specify the type of the object that should be intercepted when resolved.

It is recommended that for new development you define interception directly in the <register> Element by using the <interceptor> element and the rest of the interception elements.

The **<key>** element is a valid child of the interceptors <interceptor> element. . See Legacy Interception Configuration for more information about the **<interceptors>** element and its child elements.

The following table lists the valid attributes for the **<key>**element.

| Attribute | Description |
|---|---|
| **name** | The name of the registration to apply interception to. If omitted, the interceptor is applied to the default registration. The **name** attribute is only required where you configure more than one key for the same type. This attribute is optional. |
| **type** | The type to apply the interceptor to. This attribute is required. |

There are no valid child elements for the **<key>** element.

# Run-Time Configuration

This topic explores the techniques you can use to configure Unity containers using code that executes at run time and calls the registration methods of the Unity container. It contains the following topics:

- Using Run-Time Configuration. This topic describes the fluent interface that Unity exposes, and other issues you should be aware of when configuring the container at run time using code.
- Registering Types and Type Mappings. This topic explains how to register mappings in the container between types. In general, you will create mappings between an interface and a type that implements the interface, or between a base class and a type that inherits that base class.
- Creating Instance Registrations. This topic explains how to register existing objects in the container that you can resolve in your application. This technique is useful if you want Unity to manage the lifetime of the objects you register.

- Registering Injected Parameter and Property Values. This topic explains how to register the information required so that Unity will automatically populate constructor and method parameters and property values when it resolves instances of types.
- Registering Generic Parameters and Types. This topic explains how you can register the information required for injection for generic types, including generic arrays.
- Registering Container Extensions. This topic explains how to register information that instructs Unity to load and use container extensions that add additional functionality to the container, and how you can register configuration information for these extensions.
- Registering Interception. This topic explains how to register behaviors, policies, handlers, and matching rules that Unity will use when creating instances of types to which you want to add interception capabilities to change the behavior of that object or type.

---

For information about how to configure Unity at design time, including the techniques for loading configuration from a file or other source, see Design-Time Configuration. For information about resolving types at run time, see Resolving Objects.

# Using Run Time Configuration

This topic discusses some of the factors you should keep in mind when using run-time configuration, and explains some features of the Unity run-time configuration mechanism. For details on how to specify configuration using a configuration file, see Design-Time Configuration.

## Using the UnityContainer Fluent Interface

The API for the Unity container provides a fluent configuration interface which enables you to chain method calls in one statement. To use the fluent interface, call all the methods you want, one after the other, in a single statement, as shown in the following code.

**C#**

```
IUnityContainer myContainer = new UnityContainer()
   .RegisterType<IMyService, DataService>()
   .RegisterType<IMyService, EmailService>()
   .RegisterType<MyServiceBase, LoggingService>();
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer() _
   .RegisterType(Of IMyService, DataService)() _
   .RegisterType(Of IMyService, EmailService)() _
   .RegisterType(Of MyServiceBase, LoggingService)()
```

Method chaining is not a requirement, but it does expedite coding by relieving you from repeatedly typing "container." You can use this approach to chain any of the methods of the **UnityContainer**

class when you create the container. For example, you can use any combination of the **RegisterType**, **RegisterInstance**, **AddExtension**, and **Configure** methods to prepare the container.

Extensions use the **Configure** method of the container to provide access to a configuration interface for the extension. Many extensions expose a fluent interface that makes it easier to configure the features of that extension. For example, interception is implemented as a container extension and exposes the **AddPolicy** method that you use to define an interception policy at run time. You specify the type of matching rule, **MatchingRuleType**, and type of call handler, **HandlerType**.

**C#**

```
IUnityContainer myContainer = new UnityContainer()
   .AddNewExtension<Interception>()
   .Configure<Interception>()
       .AddPolicy("PolicyName")
           .AddMatchingRule<MatchingRuleType>()
           .AddCallHandler<HandlerType>()
       .Container;
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer() _
   .AddNewExtension(Of Interception)() _
   .Configure(Of Interception)() _
   .AddPolicy("PolicyName") _
       .AddMatchingRule(Of MatchingRuleType)() _
       .AddCallHandler(Of HandlerType)() _
   .Container
```

When using Visual Basic, you should use the full syntax for declaring the container, as shown in the previous code example. The fluent interface methods all return IUnityContainer, not UnityContainer. If you use the short syntax **Dim myContainer As New UnityContainer()**, the fluent interface mechanism will result in a type mismatch and compile errors.

# Registering Types and Type Mappings

This topic explains how to register types in the container. Registering a type lets you configure how the container creates instances of the specified type. In general, you will create mappings between an interface and a type that implements the interface, or between a base class and a type that inherits that base class. However, you can register types in the container without creating a mapping.

The **RegisterType** method registers a type with the container. At the appropriate time, the container will build an instance of the type you specify. This could be in response to dependency injection through class attributes or when you call the **Resolve** method. The lifetime of the object it builds will correspond to the lifetime you specify in the parameters of the method. If you do not specify a value

for the lifetime, the type is registered for a transient lifetime, which means that a new instance will be created on each call to **Resolve**.

This topic contains the following sections that explain use of the **RegisterType** method:



## Registering an Interface or Class Mapping to a Concrete Type

This is the most common scenario for dependency injection using Unity. It involves registering a mapping between types such as an interface or a base class and a corresponding concrete class that implements or inherits from it.

First you must create a new instance of the **UnityContainer** class or obtain a reference to an existing instance so that you can call the **RegisterType** method of the container in which you want to register the mapping or type. The following example creates a new instance of the **UnityContainer** class by using the **new** operator.

**C#**

```
IUnityContainer myContainer = new UnityContainer();
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer()
```

You call the **RegisterType** method to specify the registered type as an interface or object type and the target type you want returned in response to a query for that type. The target type must implement the interface, or inherit from the class, that you specify as the registered type. The following code creates a default (unnamed) mapping using an interface as the dependency key.

**C#**

```
myContainer.RegisterType<IMyService, CustomerService>();
```

**Visual Basic**

```
myContainer.RegisterType(Of IMyService, CustomerService)()
```

You can map a class or object type to a more specific class that inherits from it by using the same syntax but provide the inherited object as the registered type, as shown in the following example.

**C#**

```
myContainer.RegisterType<MyServiceBase, DataService>();
```

**Visual Basic**

```
myContainer.RegisterType(Of MyServiceBase, DataService)()
```

The registration controls how you will retrieve objects from the container. It is the type that you will specify when you call the **Resolve** or **ResolveAll** method to retrieve the concrete object instance.

You can create more than one registration or mapping for the same type, by creating a named (non-default) mapping by specifying a name as a parameter, as shown in the following example.

**C#**

```
myContainer.RegisterType<IMyService, CustomerService>("Customers");
```

**Visual Basic**

```
myContainer.RegisterType(Of IMyService, CustomerService)("Customers")
```

If the target class or object specifies any dependencies of its own, the instance returned will have these dependent objects injected automatically. For information about using constructor, property, or method call injection techniques, see Using Injection Attributes.

## Registering a Named Type

You can create simple type registrations at run time using the Unity API. To register a type with a name, you simply specify the name as a parameter of the **RegisterType** call. The following example simply registers a named type in the container **myContainer**.

**C#**

```
IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType(typeof(MyEmailService), "MyBestEmail");
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(GetType(MyEmailService), "MyBestEmail")
```

## Registering Type Mappings with the Container

Mapping types is useful for retrieving instances of different objects that implement the same specified interface or that inherit from the same specified base class. The target type for the mapping must inherit from or implement the base type or interface of the source. You can generate both default and named mappings for a type registration by using the generic overloads of the container methods.

**C#**

```
// Register a default (un-named) type mapping
myContainer.RegisterType<IMyObject, MyRealObject>();
```

```
// Following code will return a new instance of MyRealObject
myContainer.Resolve<IMyObject>();

// Register a named type mapping
myContainer.RegisterType<IMyObject, MyRealObject>("MyMapping");
// Following code will return a new instance of MyRealObject
myContainer.Resolve<IMyObject>("MyMapping");
```

**Visual Basic**

```
' Register a default (un-named) type mapping
myContainer.RegisterType(Of IMyObject, MyRealObject)()
' Following code will return a new instance of MyRealObject
myContainer.Resolve(Of IMyObject)()

' Register a named type mapping
myContainer.RegisterType(Of IMyObject, MyRealObject)("MyMapping")
' Following code will return a new instance of MyRealObject
myContainer.Resolve(Of IMyObject)("MyMapping")
```

The following code registers a type mapping between an example interface named **IRepository** and a concrete type named **SqlRepository** that implements this interface. The code then registers a mapping for the **SqlRepository** type as an open generic type and shows how you can resolve a specific instance using the **IRepository** interface as the dependency identifier. For more information on generics see Registering Generic Parameters and Types.

**C#**

```
public interface IRepository<TEntity>
{
  TEntity GetById(int id);
}

public class SqlRepository<TEntity> : IRepository<TEntity>
{
  public TEntity GetById(int id)
  {
    ...
  }
}

IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType(typeof(IRepository<>), typeof(SqlRepository<>));
IRepository<Customer> result = myContainer.Resolve<IRepository<Customer>>();
```

**Visual Basic**

```
Public Interface IRepository(Of TEntity)
  Function GetById(ByVal id As Integer) As TEntity
End Interface

Public Class SqlRepository(Of TEntity)
  Implements IRepository(Of TEntity)
  Public Function GetById(ByVal id As Integer) As TEntity _
                  Implements IRepository(Of TEntity).GetById
```

```
    ...
  End Function
End Class

Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(GetType(IRepository(Of )), GetType(SqlRepository(Of )))
Dim result As IRepository(Of Customer) = myContainer.Resolve(Of IRepository(Of
Customer)())()
```

## Using a Lifetime Manager with the RegisterType Method

You can also provide a lifetime manager when calling **RegisterType**. The LifetimeManager you specify when you register a type controls when object instances are created and disposed. If you do not specify a lifetime manager for your type registration, the object instances returned by the container have a transient lifetime. The container does not store a reference to the object, and creates a  new instance of the type each time you call the **Resolve** method. If you want a different object lifetime, specify a lifetime manager when you call the **RegisterType** method.

For more information on lifetime managers see Understanding Lifetime Managers.

The most common scenario is to create an instance that behaves like a singleton, so that the container creates the object the first time you call **Resolve** and then returns the same instance for all subsequent calls to **Resolve** for as long as the container is in scope. Register a singleton mapping by including an instance of the **ContainerControlledLifetimeManager** class in the parameters to the **RegisterType** method.

First you must create a new instance of the **UnityContainer** class or obtain a reference to an existing instance so that you can call the **RegisterType** method of the container in which you want to register the mapping or type.

**C#**

```
IUnityContainer myContainer = new UnityContainer();
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer()
```

The following example creates a default (unnamed) singleton mapping by adding **new ContainerControlledLifetimeManager()** to the **RegisterType** method parameters.

**C#**

```
myContainer.RegisterType<IMyService, CustomerService>(new
ContainerControlledLifetimeManager());
```

**Visual Basic**

```
myContainer.RegisterType(Of IMyService, CustomerService)(New
ContainerControlledLifetimeManager())
```

You can also just register a specific type as a singleton by specifying as the registration type the concrete type you want returned in response to a query for that type. Include an instance of the **ContainerControlledLifetimeManager** class in the parameters to the **RegisterType** method. The following example creates a default (unnamed) singleton registration for the type **CustomerService**.

**C#**

```
myContainer.RegisterType<CustomerService>(new
ContainerControlledLifetimeManager());
```

**Visual Basic**

```
myContainer.RegisterType(Of CustomerService)(New
ContainerControlledLifetimeManager())
```

You can create more than one registration using the same registered type, **CustomerService**, by creating a named (non-default) registration by specifying a name as a parameter. Each named registration uses a separate lifetime manager. In this example, calling **container.Resolve<CustomerService>()** and **container.Resolve<CustomerService>("Customers")** will result in two different instances. But calling **container.Resolve<CustomerService>("Customers")** multiple times will give the same instance each time.

**C#**

```
myContainer.RegisterType<CustomerService>("Customers",
                         new ContainerControlledLifetimeManager());
```

**Visual Basic**

```
myContainer.RegisterType(Of CustomerService)("Customers", _
                         New ContainerControlledLifetimeManager())
```

You can register more than one mapping for an object type, **IMyService**, **CustomerService** that will return a singleton using the registered type. You can do this by creating a named (non-default) registration by specifying a name as a parameter, as shown in the following example.

**C#**

```
myContainer.RegisterType<IMyService, CustomerService>("Customers",
                         new ContainerControlledLifetimeManager());
```

**Visual Basic**

```
myContainer.RegisterType(Of IMyService, CustomerService)("Customers", _
                         New ContainerControlledLifetimeManager())
```

The following example registers a default (unnamed) type mapping for the type **MyRealObject** with a per-thread lifetime by specifying the **PerThreadLifetimeManager()** class in the parameter list.

**C#**

```
// Specify a default type mapping with an per thread lifetime
myContainer.RegisterType<IMyObject, MyRealObject>(new
PerThreadLifetimeManager());
// Following code will return a reference to the object
myContainer.Resolve<IMyObject>();

// Specify a default type mapping with an externally-controlled lifetime
myContainer.RegisterType<IMyObject, MyRealObject>(new
ExternallyControlledLifetimeManager());
// Following code will return a singleton instance of MyRealObject
// Container will hold only a weak reference to the object
myContainer.Resolve<IMyObject>();
```

**Visual Basic**

```
' Specify a default type mapping with an per thread lifetime
myContainer.RegisterType(Of IMyObject, MyRealObject)(New
PerThreadLifetimeManager())
' Following code will return a reference to the object
myContainer.Resolve(Of IMyObject)()

' Specify a default type mapping with an externally-controlled lifetime
myContainer.RegisterType(Of IMyObject, MyRealObject)(New
ExternallyControlledLifetimeManager())
' Following code will return a singleton instance of MyRealObject
' Container will hold only a weak reference to the object
myContainer.Resolve(Of IMyObject)()
```

The following code shows additional uses of the **RegisterType** method with a lifetime manager to specify the container behavior. It shows how you can generate both default and named mappings for a type registration.

**C#**

```
IUnityContainer myContainer = new UnityContainer();

// Register a default (un-named) type mapping with a singleton lifetime
myContainer.RegisterType<IMyObject, MySingletonObject>(new
ContainerControlledLifetimeManager());
// Following code will return a singleton instance of MySingletonObject
// Container will take over lifetime management of the object
myContainer.Resolve<IMyObject>();

// Register a named type mapping with a singleton lifetime
myContainer.RegisterType<IMyObject, MySingletonObject>("MyMapping", new
ContainerControlledLifetimeManager());
// Following code will return a singleton instance of MySingletonObject
// Container will take over lifetime management of the object
myContainer.Resolve<IMyObject>();
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer()

' Register a default (un-named) type mapping with a singleton lifetime
myContainer.RegisterType(Of IMyObject, MySingletonObject)(new
ContainerControlledLifetimeManager())
' Following code will return a singleton instance of MySingletonObject
' Container will take over lifetime management of the object
myContainer.Resolve(Of IMyObject)()

' Register a named type mapping with a singleton lifetime
myContainer.RegisterType(Of IMyObject, MySingletonObject)("MyMapping", new
ContainerControlledLifetimeManager())
' Following code will return a singleton instance of MySingletonObject
' Container will take over lifetime management of the object
myContainer.Resolve(Of IMyObject)()
```

## Summary of the RegisterType Method Overloads

The following table summarizes the method overloads you can use to register type mappings with the container at run time.

| Method | Description |
|---|---|
| RegisterType<*TFrom*, *TTo*>(params InjectionMember[] *injectionMembers*) | Registers a default type mapping with the container so that it returns an instance of the type specified as *TTo* when a **Resolve** method requests an instance of the type *TFrom*. The *injectionMembers* parameter configures specific objects to be injected. |
| RegisterType<*TFrom*, *TTo*>(LifetimeManager *lifetime,* params InjectionMember[] *injectionMembers*) | Registers a default type mapping with the container so that it returns an instance of the type specified as *TTo* when a **Resolve** method requests an instance of the type *TFrom*. Also registers the specified **LifetimeManager** instance with the container to manage the lifetime of the returned object. The *injectionMembers* parameter configures specific objects to be injected. |
| RegisterType<*TFrom*, *TTo*>(String *name,* params InjectionMember[] *injectionMembers*) | Registers a named type mapping with the container so that it returns an instance of the type specified as *TTo* when a **Resolve** method requests an instance of the type *TFrom* with the specified *name*. If *name* is null, generates a default type mapping. Names are case sensitive. The *injectionMembers* parameter configures specific objects to be injected. |
| RegisterType<*TFrom*, *TTo*>(String *name*, LifetimeManager *lifetime, params InjectionMember[] injectionMembers*) | Registers a named type mapping with the container so that it returns an instance of the type specified as *TTo* when a **Resolve** method requests an instance of the type *TFrom* with the specified *name*. If *name* is null, generates a default type mapping. Names are case sensitive. Also registers the specified **LifetimeManager** instance with the container to manage the lifetime of the returned object. The *injectionMembers* parameter configures specific objects to be injected. |
| RegisterType<*T*>(LifetimeManager *lifetime,* params InjectionMember[] *injectionMembers*) | Creates a default type registration with the container so that it returns an instance of that type in response to a call from the **Resolve** method. Allows you to register a **LifetimeManager** with the container to manage the lifetime of the registered type. The *injectionMembers* parameter configures specific objects to be injected. |
| RegisterType<*T*>(String *name*, LifetimeManager *lifetime,* params InjectionMember[] *injectionMembers*) | Creates a named type registration with the container so that it returns an instance of that type in response to a call from the **Resolve** method with the specified *name*. Allows you to register a **LifetimeManager** with the container to manage the lifetime of the registered type. The *injectionMembers* parameter configures specific objects to be injected. |
| RegisterType(Type *from*, Type *to,* params InjectionMember[] *injectionMembers*) | Registers a default type mapping with the container so that it returns an instance of the type specified as *to* when a **Resolve** method requests an instance of the type *from*. The *injectionMembers* parameter configures specific objects to be injected. |
| RegisterType(Type *from*, Type *to*, String *name*, params InjectionMember[] *injectionMembers*) | Registers a named type mapping with the container so that it returns an instance of the type specified as *to* when a **Resolve** method requests an instance of the type *from* with the specified *name*. If *name* is null, generates a default type mapping. Names are case sensitive. The *injectionMembers* parameter configures specific objects to be |

| | |
|---|---|
| | injected. |
| RegisterType(Type *from*, Type *to*, LifetimeManager *lifetime*, params InjectionMember[] *injectionMembers*) | Registers a default type mapping with the container so that it returns an instance of the type specified as *to* when a **Resolve** method requests an instance of the type *from*. Also registers the specified **LifetimeManager** instance with the container to manage the lifetime of the returned object. The *injectionMembers* parameter configures specific objects to be injected. |
| RegisterType(Type *from*, Type *to*, String *name*, LifetimeManager *lifetime,* params InjectionMember[] *injectionMembers*) | Registers a named type mapping with the container so that it returns an instance of the type specified as *to* when a **Resolve** method requests an instance of the type *from* with the specified *name*. If *name* is null, generates a default type mapping. Names are case sensitive. Also registers the specified **LifetimeManager** instance with the container to manage the lifetime of the returned object. The *injectionMembers* parameter configures specific objects to be injected. |
| RegisterType(Type *t*, LifetimeManager *lifetime,* params InjectionMember[] *injectionMembers*) | Creates a default type registration with the container so that it returns an instance of that type in response to a call from the **Resolve** method. Allows you to register a **LifetimeManager** with the container to manage the lifetime of the registered type. The *injectionMembers* parameter configures specific objects to be injected. |
| RegisterType(Type *t*, String *name*, LifetimeManager *lifetime* params InjectionMember[] *injectionMembers)* | Creates a named type registration with the container so that it returns an instance of that type in response to a call from the **Resolve** method with the specified *name*. Allows you to register a **LifetimeManager** with the container to manage the lifetime of the registered type. The *injectionMembers* parameter configures specific objects to be injected. |

The following example uses **RegisterType** to configure class methods for injection by the container.

**C#**

```
IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType<MyObject>(
    new InjectionConstructor(12, "Hello Unity!"),
    new InjectionProperty(“MyProperty”));
```

**Visual Basic**

```
Dim myContainer as IUnityContainer = New UnityContainer()
myContainer.RegisterType(Of MyObject)( _
   new InjectionConstructor(12, "Hello Unity!"), _
   new InjectionProperty(“MyProperty”))
```

## More Information

For more information about the techniques discussed in this topic, see the following topics:

- Creating Instance Registrations
- Registering Injected Parameter and Property Values
- Resolving Objects by Using Overrides
- Understanding Lifetime Managers
- Using Container Hierarchies

# Creating Instance Registrations

This topic explains how to register existing objects in the container that you can resolve in your application. This technique is useful if you already have an instance of an object that you have previously created and want Unity to manage its lifetime, or if you want Unity to inject that object into other objects that it is resolving.

The **RegisterInstance** method registers an existing instance with the container. You specify the instance type and optional lifetime in the parameter list. The container will return the specified existing instance for the duration of the specified lifetime. The **RegisterInstance** method overloads are very similar to the **RegisterType** method overloads, but they accept an additional parameter, the object instance to register. The use of the registration type and an optional name are identical for the two methods.

When resolving types with dependencies, instances of objects added to the container with the **RegisterInstance** method act just like those registered through **RegisterType**. The **RegisterType** method with a **ContainerControlledLifetimeManager** automatically generates this single instance the first time your code calls it, while the **RegisterInstance** method accepts an existing instance for which it will return references. If you do not specify a lifetime manager, the container will use a **ContainerControlledLifetimeManager** and it will return a reference to the original object on each call to **Resolve**.

This topic contains the following sections, which explain the use of the **RegisterInstance** method:



## Registering an Existing Object Instance of an Interface or Type to a Container

Instance registration is similar to type registration, except that instead of the container creating the instance you first create the instance directly and then use the **RegisterInstance** method to add that instance to the container. Therefore, the container does not need to create the instance on the first **Resolve** request. The **LifetimeManager** still controls the lifetime of the object, including when object instances are created and disposed, and the container returns the existing instance for the duration of the specified lifetime. For example, you first create the instance but the lifetime manager determines whether that instance is a singleton or has a per-thread lifetime.

By default, the **RegisterInstance** method registers existing object instances with a container-controlled lifetime, **ContainerControlledLifetimeManager**, and holds onto a strong reference to the object for the life of the container. You can change this behavior by using a different lifetime manager to control the lifetime and disposal of the object. For more information, see Understanding Lifetime Managers. For information about how to create custom lifetime managers, see Creating Lifetime Managers.

The Unity container exposes overloads of the **RegisterInstance** method that allow you to register dependency injection mappings that return references to a single existing instance of an object. Each of these methods accepts a type parameter that identifies the object interface or type and an existing instance of the object. Optionally, you can provide a name to disambiguate instances when there is more than one registration for the same type. In all cases you must first create or obtain a reference to a container.

Once you have a reference to the proper container, call the **RegisterInstance** method of that container to register the existing object. Specify as the registration type an interface that the target object implements, an object type from which the target object inherits, or the concrete type of the target object.

The following example creates a default (unnamed) registration for an object of type **EmailService** that implements the **IMyService** interface.

**C#**

```
EmailService myEmailService = new EmailService();
myContainer.RegisterInstance<IMyService>(myEmailService);
```

**Visual Basic**

```
Dim myEmailService As New EmailService()
myContainer.RegisterInstance(Of IMyService)(myEmailService)
```

The registration type controls how you will retrieve objects from the container. It is the type that you will specify when you call the **Resolve** or **ResolveAll** method to retrieve the concrete object instance. In this case, **IMyService theDataService = myContainer.Resolve<IMyService>().**

If you do not want to map the existing object type to an interface or base class type, you can specify the actual concrete type, **myEmailService**, for the registered type, **EmailService**, as shown in the following example.

**C#**

```
myContainer.RegisterInstance<EmailService>(myEmailService);
```

**Visual Basic**

```
myContainer.RegisterInstance(Of EmailService)(myEmailService)
```

If you want to create more than one existing object registration using the same registered type, you can create named (non-default) mappings by specifying a name as a parameter, such as "Email" as shown in the following example.

**C#**

```
myContainer.RegisterInstance<EmailService>("Email", myEmailService);
```

**Visual Basic**

```
myContainer.RegisterInstance(Of EmailService)("Email", myEmailService)
```

> In general, you should never need to specify the same registered type in calls to the **RegisterType** and **RegisterInstance** methods. In other words, if you register a mapping for **IMyObject** using the **RegisterType** method, you should avoid using **IMyObject** as the registration type parameter of the **RegisterInstance** method. You are likely to get unexpected behavior because type mapping occurs before instances are resolved.

If you want to manage the instance lifetime, provide a reference to the existing object and an instance of the lifetime manager. The following example creates a default (unnamed) externally managed registration using an interface, **IMyService**, as the registered type and specifies the lifetime manager by providing the **ExternallyControlledLifetimeManager** class as a parameter of the **RegisterInstance** method.

**C#**

```
EmailService myEmailService = new EmailService();
myContainer.RegisterInstance<IMyService>(myEmailService,
                            new ExternallyControlledLifetimeManager());
```

**Visual Basic**

```
Dim myEmailService As New EmailService()
myContainer.RegisterInstance(Of IMyService)(myEmailService, _
                            New ExternallyControlledLifetimeManager())
```

> When you use the **RegisterInstance** method to register an existing object, dependency injection will **not** take place on that object because it has already been created outside of the influence of the Unity container. You can force property and method call injection to occur by passing the object through the **BuildUp** method, but you cannot force constructor injection to occur because the class constructor will never execute for an existing object. For information about using constructor, property, or method call injection techniques, see Registering Injected Parameter and Property Values .

The following example uses the **RegisterInstance** method to register existing instances of objects that implement the **IMyService** interface. This example uses both the generic and the non-generic overloads of the container methods and shows the **Resolve** method call for each instance.

**C#**

```
DataService myDataService = new DataService();
EmailService myEmailService = new EmailService();
LoggingService myLoggingService = new LoggingService();
```

```
// Create container and register existing object instance
IUnityContainer myContainer = new UnityContainer();

myContainer.RegisterInstance<IMyService>(myDataService);
myContainer.RegisterInstance<IMyService>("Email", myEmailService);
myContainer.RegisterInstance(typeof(IMyService), "Logging", myLoggingService);

// Retrieve an instance of each type
IMyService theDataService = myContainer.Resolve<IMyService>();
IMyService theEmailService = myContainer.Resolve<IMyService>("Email");
IMyService theLoggingService
  = (IMyService)myContainer.Resolve(typeof(IMyService), "Logging");
```

**Visual Basic**

```
Dim myDataService As New DataService()
Dim myEmailService As New EmailService()
Dim myLoggingService As New LoggingService()

' Create container and register existing object instance
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterInstance(Of IMyService)(myDataService)
myContainer.RegisterInstance(Of IMyService)("Email", myEmailService)
myContainer.RegisterInstance(GetType(IMyService), "Logging", myLoggingService)

' Retrieve an instance of each type
Dim theDataService As IMyService = myContainer.Resolve(Of IMyService)()
Dim theEmailService As IMyService = myContainer.Resolve(Of IMyService)("Email")
Dim theLoggingService As IMyService = myContainer.Resolve(GetType(IMyService),
"Logging")
```

You can use the **RegisterInstance** method to register simple types and more complex or custom types. For example, you can register **String** values such as database connection strings with the container and have them injected into your code as required.

If you register a type more than once using the **RegisterInstance** method (in other words, if you register more than one instance of the same type), only the last one you register remains in the container and is returned when you execute the **Resolve** or **ResolveAll** method.

## Using a Lifetime Manager with the RegisterInstance Method

The lifetime manager you specify when you register an existing object enables you to manage object lifetimes. If you do not specify a lifetime manager for your instance registration, by default, the object instances returned by the container when you use the **RegisterInstance** method have a container-controlled lifetime. If you want to specify the lifetime of the object, specify a lifetime manager when you call the **RegisterInstance** method.

Some objects, such as unmanaged components and other applications, may hold onto references to an object that you pass to the **RegisterInstance** method. In this case, you should use the **ExternallyControlledLifetimeManager**. However, you should always design your code so that disposable objects always have only one "owner."

The most common scenario is to create an instance that behaves like a singleton, so that the container creates the object the first time you call **Resolve** and then returns the same instance for all subsequent calls to **Resolve** for as long as the container is in scope. You can specify one of the following Unity Built-In Lifetime Managers types or your custom type when you call the **RegisterInstance** method:

- ContainerControlledLifetimeManager
- ExternallyControlledLifetimeManager
- HierarchicalLifetimeManager

It is not appropriate to use either **PerResolveLifetimeManager** or **TransientLifetimeManager** with **RegisterInstance** since they both create a new instance on every call to resolve.

The following example summarizes how you can use the **RegisterInstance** method with a lifetime manager to specify the container behavior. It shows how you can generate both default and named mappings when you register an existing object. This example uses the generic overloads of the container methods.

**C#**

```
IUnityContainer myContainer = new UnityContainer();

// Register an existing object as a default (un-named) registration with
// the default container-controlled lifetime.
myContainer.RegisterInstance<IMyObject>(MyRealObject);
// Following code will return a singleton instance of MyRealObject
// Container will take over lifetime management of the object
myContainer.Resolve<IMyObject>();

// Register an existing object as a named registration with the default
// container-controlled lifetime.
myContainer.RegisterInstance<IMyObject>("MySingleton", MyRealObject);
// Following code will return a singleton instance of MyRealObject
// Container will take over lifetime management of the object
myContainer.Resolve<IMyObject>("MySingleton");

// Same as above, but specify the default lifetime manager
myContainer.RegisterInstance<IMyObject>("MySingleton", MyRealObject, new
ContainerControlledLifetimeManager());
// Following code will return a singleton instance of MyRealObject
```

```
// Container will take over lifetime management of the object
myContainer.Resolve<IMyObject>("MySingleton");

// Register an existing object as a default (un-named) registration
// with an externally controlled lifetime.
myContainer.RegisterInstance<IMyObject>(MyRealObject, new
ExternallyControlledLifetimeManager());
// Following code will return a singleton instance of MyRealObject
// Container will hold only a weak reference to the object
myContainer.Resolve<IMyObject>();

// Register an existing object as a named registration
// with an externally controlled lifetime.
myContainer.RegisterInstance<IMyObject>("MySingleton", MyRealObject, new
ExternallyControlledLifetimeManager());
// Following code will return a singleton instance of MyRealObject
// Container will hold only a weak reference to the object
myContainer.Resolve<IMyObject>("MySingleton");

// Register an existing object as a named registration
// with a per thread lifetime.
myContainer.RegisterInstance<IMyObject>("MySingleton", MyRealObject, new
PerThreadLifetimeManager());
// Following code will return a singleton instance of MyRealObject
// Container will hold only a weak reference to the object
myContainer.Resolve<IMyObject>("MySingleton");
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer()

' Register an existing object as a default (un-named) registration with
' the default container-controlled lifetime.
myContainer.RegisterInstance(Of IMyObject)(MyRealObject)
' Following code will return a singleton instance of MyRealObject
' Container will take over lifetime management of the object
myContainer.Resolve(Of IMyObject)()

' Register an existing object as a named registration with the default
' container-controlled lifetime.
myContainer.RegisterInstance(Of IMyObject)("MySingleton", MyRealObject)
' Following code will return a singleton instance of MyRealObject
' Container will take over lifetime management of the object
myContainer.Resolve(Of IMyObject)("MySingleton")

' Same as above, but specify the default lifetime manager
myContainer.RegisterInstance(Of IMyObject)("MySingleton", MyRealObject, new
ContainerControlledLifetimeManager())
' Following code will return a singleton instance of MyRealObject
' Container will take over lifetime management of the object
myContainer.Resolve(Of IMyObject)("MySingleton")

' Register an existing object as a default (un-named) registration
```

```
' with an externally controlled lifetime.
myContainer.RegisterInstance(Of IMyObject)(MyRealObject, new
ExternallyControlledLifetimeManager())
' Following code will return a singleton instance of MyRealObject
' Container will hold only a weak reference to the object
myContainer.Resolve(Of IMyObject)()

' Register an existing object as a named registration
' with an externally controlled lifetime.
myContainer.RegisterInstance(Of IMyObject)("MySingleton", MyRealObject, new
ExternallyControlledLifetimeManager())
' Following code will return a singleton instance of MyRealObject
' Container will hold only a weak reference to the object
myContainer.Resolve(Of IMyObject)("MySingleton")
' Register an existing object as a named registration
' with a per thread lifetime.
myContainer.RegisterInstance(Of IMyObject)("MySingleton", MyRealObject, new
PerThreadLifetimeManager())
' Following code will return a singleton instance of MyRealObject
' Container will hold only a weak reference to the object
myContainer.Resolve(Of IMyObject)("MySingleton")
```

For more information about lifetime managers see Understanding Lifetime Managers.

## Summary of the RegisterInstance Overloads

The following table summarizes the method overloads you can use to register existing object instances with the container at run time.

| Method | Description |
|---|---|
| RegisterInstance<*TInterface*>(TInterface *instance*) | Registers a default instance mapping with the container so that it returns the specified *instance* when a **Resolve** method requests an instance of the type *TInterface* (which can be an implemented interface instead of the actual type). The container takes over the lifetime of the instance. |
| RegisterInstance<*TInterface*>(TInterface *instance*, LifetimeManager *lifetime*) | Registers a default instance mapping with the container so that it returns the specified *instance* when a **Resolve** method requests an instance of the type *TInterface* (which can be an implemented interface instead of the actual type). Also registers the specified **LifetimeManager** instance with the container to manage the lifetime of the returned object. |
| RegisterInstance<*TInterface*>(String *name*, TInterface *instance*) | Registers a named instance mapping with the container so that it returns the specified *instance* when a **Resolve** method requests an instance of the type *TInterface* (which can be an implemented interface instead of the actual type) and the specified *name*. The container takes over the lifetime of the instance. Names are case sensitive. |
| RegisterInstance<*TInterface*>(String *name*, TInterface *instance*, LifetimeManager *lifetime*) | Registers a named instance mapping with the container so that it returns the specified *instance* when a **Resolve** method requests an instance of the type *TInterface* (which can be an implemented interface instead of the actual type) and the specified *name*. Names are case sensitive. Also registers the specified **LifetimeManager** instance with the container to |

| | |
|---|---|
| | manage the lifetime of the returned object. |
| RegisterInstance(Type *t*, Object *instance*) | Registers a default instance mapping with the container so that it returns the specified *instance* when a **Resolve** method requests an instance of the type *t* (which can be an implemented interface instead of the actual type). The container takes over the lifetime of the instance. |
| RegisterInstance(Type *t*, Object *instance*, LifetimeManager *lifetime*) | Registers a default instance mapping with the container so that it returns the specified *instance* when a **Resolve** method requests an instance of the type *t* (which can be an implemented interface instead of the actual type). Also registers the specified **LifetimeManager** instance with the container to manage the lifetime of the returned object. |
| RegisterInstance(Type *t*, String *name*, Object *instance*) | Registers a named instance mapping with the container so that it returns the specified *instance* when a **Resolve** method requests an instance of the type *t* (which can be an implemented interface instead of the actual type) and the specified *name*. Names are case sensitive. Also registers the specified **LifetimeManager** instance with the container to manage the lifetime of the returned object. |
| RegisterInstance(Type *t*, String *name*, Object *instance*, LifetimeManager *lifetime*) | Registers a named instance mapping with the container so that it returns the specified *instance* when a **Resolve** method requests an instance of the type *t* (which can be an implemented interface instead of the actual type) and the specified *name*. Names are case sensitive. Also registers the specified **LifetimeManager** instance with the container to manage the lifetime of the returned object. |

## More Information

For more information about the  techniques discussed in this topic, see the following topics:

- Resolving Objects by Using Overrides
- Understanding Lifetime Managers.

# Registering Injected Parameter and Property Values

This topic explains how to configure a container to perform dependency injection at run time by using the **RegisterType** method overloads with the **InjectionMembers** parameter and avoid relying on annotating the classes to resolve with attributes. This topic includes information on configuring Unity to automatically populate constructor and method parameters and property values when it resolves instances of types.

This topic contains the following sections to explain the use of the **InjectionMembers** methods:

- Registering Injection for Parameters Properties and Methods Using InjectionMembers

- Injecting Arrays at Run Time
- Summary of the InjectionMember Methods and Overloads
- For More Information on InjectionMembers

---

## Registering Injection for Parameters, Properties, and Methods using InjectionMembers

The **RegisterType** overloads allow for configuring injection by accepting **InjectionMembers**. Include the **InjectionConstructor**, **InjectionProperty**, and **InjectionMethod** classes as a RegisterType parameter to provide dependency injection configuration in a container for InjectionMember objects.

The following example shows the general syntax for using an **InjectionMember** subclass, **InjectionConstructor,** with the **RegisterType** method. In this example the default constructor is called.

**C#**

```
IUnityContainer container = new UnityContainer()
    .RegisterType<AType>(new InjectionConstructor());

AType aType = container.Resolve<AType>();
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer()_
    .RegisterType(Of AType)(New InjectionConstructor())

Dim aType As AType = container.Resolve(Of AType)()
```

You can also use attributes applied to target class members to instruct Unity to inject dependent objects. For more information, see Using Injection Attributes.

You can use the **RegisterType** overloads to do the following:

- Register Constructors and Parameters
- Specify a Property for Injection
- Specify a Method for Injection

---

## Register Constructors and Parameters

You can specify a constructor for the injection of a specific named instance. When you include a call to the **InjectionConstructor** method with a specific name, this provides the necessary information on which constructor to use and you do not need to call **RegisterInstance**. If you use the annotated constructor, you require a call to **RegisterInstance**.

In the following example, **InjectionConstructor** indicates which constructor to use based on its arguments, the string "UI," and causes the **TraceSourceLogger** constructor with a single string parameter to be invoked.

**C#**

```
container.RegisterType<ILogger, TraceSourceLogger>(
            "UI",
            new InjectionConstructor("UI"));
```

**Visual Basic**

```
container.RegisterType(Of ILogger, TraceSourceLogger) _
         ("UI", New InjectionConstructor("UI"))
```

**C#**

```
[InjectionConstructor]
public TraceSourceLogger(TraceSource traceSource)
{
    this.traceSource = traceSource;
}
```

**Visual Basic**

```
<InjectionConstructor> _
Public Sub New(traceSource As TraceSource)
      Me.traceSource = traceSource
End Sub
```

**InjectionConstructor()** takes precedence over the constructor annotated with the **InjectionConstructor** attribute.

You could also use the **InjectionConstructor** method to configure the container to call the constructor with the provided values and build the instance instead of just registering an instance created elsewhere. The following example passes values to the constructor.

**C#**

```
IUnityContainer container = new UnityContainer()
            .RegisterType<MyObject>(
                  new InjectionConstructor(someInt, someString, someDouble));
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer()
           .RegisterType(Of MyObject)
                  (New InjectionConstructor(someInt, someString, someDouble))
```

You could use constructor injection to specify any of a series of constructors or method overloads; however, you could inadvertently cause infinite recursion. To avoid the infinite recursion, specify which constructor to call in the **RegisterType** call. For more information see Circular References with Dependency Injection.

## Specify a Property for Injection

You can configure the container to inject a property with or without a value for the property.

In the following example, property injection is configured for the property named **MyProperty** to use the default configuration of the container. The container resolves the value for this property using registrations and mappings within the container.

**C#**

```
container.RegisterType<DriveController>(
          new InjectionProperty("MyProperty"));
```

**Visual Basic**

```
container.RegisterType(Of DriveController) _
         (New InjectionProperty("MyProperty"))
```

The following example configures property injection for the property named IntProperty to use a specified value. The container sets the property to the value "8."

**C#**

```
int expectedInt = 8;
container.RegisterType<DriveController>(
new InjectionProperty("IntProperty", expectedInt));
```

**Visual Basic**

```
Dim expectedInt As Integer = 8
container.RegisterType(Of DriveController)(New InjectionProperty("IntProperty",
expectedInt))
```

In the following example, the **InjectionProperty** object is configured, indicating that the property with the name **Logger** will be injected. Because there is no further configuration for this property, the unnamed instance for the property's type, **ILogger**, will be resolved when obtaining the value to inject to the property.

**C#**

```
using (IUnityContainer container = new UnityContainer())
{
  container
    .RegisterType<IStocksTickerView, StocksTickerForm>()
    // Configure property injection.
    .RegisterType<IStockQuoteService, MoneyCentralStockQuoteService>(
                                      new InjectionProperty("Logger"))
    .RegisterType<ILogger, ConsoleLogger>()
    .RegisterType<ILogger, TraceSourceLogger>("UI")
    .RegisterInstance(new TraceSource("UI", SourceLevels.All));
```

```
  StocksTickerPresenter presenter = container.Resolve<StocksTickerPresenter>();

  Application.Run((Form)presenter.View);
}
```

**Visual Basic**

```
Using container As IUnityContainer = New UnityContainer()
  container _
    .RegisterType(Of IStocksTickerView, StocksTickerForm)() _
    .RegisterType(Of IStockQuoteService, MoneyCentralStockQuoteService) _
                                         (New InjectionProperty("Logger")) _
    .RegisterType(Of ILogger, ConsoleLogger)() _
    .RegisterType(Of ILogger, TraceSourceLogger)("UI") _
    .RegisterInstance(New TraceSource("UI", SourceLevels.All))

  Dim presenter As StocksTickerPresenter = container.Resolve(Of
StocksTickerPresenter)()

  Application.Run(DirectCast(presenter.View, Form))

End Using
```

This is similar to the use of the **Dependency** attribute without further configuration to annotate the property. Add the **Dependency** attribute to the **Logger** property, to indicate that the property should be injected with the result that the property's type, **ILogger**, is resolved in the container. The following example shows the comparable use of the **Dependency** attribute.

**C#**

```
private ILogger logger;
[Dependency]
public ILogger Logger
{
  get { return logger; }
  set { logger = value; }
}
```

**Visual Basic**

```
Private logger As ILogger
<Dependency> _
Public Property Logger() As ILogger
  Get
    Return logger
  End Get
  Set
    logger = value
  End Set
End Property
```

## Specify a Method for Injection

You can configure a container to call methods during injection. The following example configures the method named **InitializeMe** for injection. It is configured with two parameters, a **Double** value and an instance of a class that implements the **ILogger** interface. It passes the value 42.0 to the first parameter and resolves the **ILogger** type through the container by looking for a mapping for that type with the name **SpecialLogger** and passes the result to the second parameter of the method.

**C#**

```
container.RegisterType<DriveController>(
           new InjectionMethod("InitializeMe", 42.0,
           new ResolvedParameter(typeof(ILogger), "SpecialLogger")));
```

**Visual Basic**

```
container.RegisterType(Of DriveController)(_
        New InjectionMethod("InitializeMe", 42, _
        New ResolvedParameter(GetType(ILogger), "SpecialLogger")))
```

## Injecting Arrays at Run Time

Although an array can be injected as a value by supplying it when configuring injection just as you would for any other CLR object, Unity enables you to configure array injection by using the result of resolving other keys as elements.

Unity provides API registration support for arrays by providing a resolver object that resolves all the named instances of the type registered in a container and an **InjectionParameterValue** that is used to configure parameters for constructor or method injection. The implementation for the **ResolvedArray** API support relies on the **ResolveAll** method in **IUnityContainer**. The **ResolveAll** method enables you to specify the type and returns all instances of all registered types requested even after you register multiple typenames of the same type.

The injection configuration APIs are based on the subclasses of **InjectionParameterValue**. You can also provide other types of objects when setting up injection. The objects are translated to a **InjectionParameterValue** according to the following rules:

1. If the object is an instance of a subclass of the **InjectionParameterValue** class, the injection system uses the object.
2. If the object is an instance of the **Type** class, the injection system creates a **ResolvedParameter** that describes how the container should perform injection to resolve an instance of that type.

> In order to specify an instance of the **Type** class as a literal argument and ensure that the override is not resolved in the container as an instance of the specified type per this rule, you must inject an **InjectionParameter** with the **Type** object.

**C#**

```
container.Resolve<Cars>(new DependencyOverride(typeof(Type), new
InjectionParameter(typeof(int))));
```

**Visual Basic**

```
container.Resolve(Of Cars)(New DependencyOverride(GetType(Type), _
        New InjectionParameter(GetType(Integer))))
```

3. In all other cases, the configuration API creates an **InjectionParameter** instance the container uses to get the value to be injected into a property.
4. You cannot supply **null** as the value to inject. You must explicitly provide an **InjectionParameter**, as with types.

---

The injection rules are located in the **ToParameter** method of the **InjectionParameterValue** class. You can examine this class to see the implementation. The **InjectionConstructor**, **InjectionProperty**, and **InjectionMethod** subclasses of the **InjectionMember** base class are used to describe the injection at the time the BuildPlan for a key is created. The BuildPlan is created the first time a key is resolved. After that, injection does not use these three subclasses. You can create your own subclasses based on this class if you want to extend or change the behavior to suit your own specific requirements. You can also create your own subclasses based on the **InjectionParameterValue** class if you want to customize the handling of individual parameters.

There are two kinds of array injection in Unity:

- Injecting an array that contains all the instances of the array's element type registered in the container (in the order they were registered).

  This is comparable to using the **<dependency>** or **<optional>** element in the configuration. Attribute support for injecting all registered members of an array is provided by using the **[Dependency]** attribute. Using the **[Dependency]** attribute produces the equivalent results for injection. You can also use the **[OptionalDependency]** attribute just like you use the **Dependency** attribute only it is for optional dependencies.

- Injecting an array containing the result of resolving a specific key by using **ResolvedArrayParameter**. There is no attribute support in this case. Use **InjectionParameterValues** to specify how each element in the array should be resolved.

  Using **ResolvedArrayParameter** is comparable to using or the **<array>** element in the configuration file where you would use the **<array>** element to specify how each element should be resolved.

---

Injection of properties can be specified simultaneously in the configuration file and through code. The injection specification using the API overrides any previous specification from the configuration file, but non-overlapping definitions will be observed regardless of their origin because the underlying mechanism is the same. Injection of properties indicated through attributes in the resolved types is observed only when there are no other injection specifications for any property in the same type.

## Injecting Specific Array Instances

The **ResolvedArrayParameter** works like any other **InjectionParameterValue** used to specify how to supply a constructor argument or a property value. In the following example the **ILogger[]** parameter is injected with a two-element array based on the two arguments for **ResolvedArrayParameter**. The first element is the result of resolving the **ILogger** interface without a name per the **typeof(ILogger)** argument.

> **typeof(ILogger)** is equivalent to **new ResolvedParameter<ILogger>()**.

The second element is the result of resolving the **ILogger** interface with the **UI** name per the **new ResolvedParameter<ILogger>("UI")** argument expression. When using this kind of array injection, literal values can be specified as members of the array and as unnamed instances.

**C#**

```
// Map the ILogger interface to the CompositeLogger class to inject
// an array of specific instances through the constructor.
container.RegisterType<ILogger, CompositeLogger>(
  "composite",
  new InjectionConstructor(
    new ResolvedArrayParameter<ILogger>(
        typeof(ILogger),
        new ResolvedParameter<ILogger>("UI"))));
```

**Visual Basic**

```
' Map the ILogger interface to the CompositeLogger class to inject
' an array of specific instances through the constructor.
container.RegisterType(Of ILogger, CompositeLogger)( _
  "composite", _
  New InjectionConstructor( _
    New ResolvedArrayParameter(Of ILogger)( _
        GetType(ILogger), _
        New ResolvedParameter(Of ILogger)("UI"))))
```

The following example shows how to inject a specific instance of an array. Start with the class **MyClass** with a member, **myLogger,** which is an **ILogger** array.

**C#**

```
class MyClass{
  public MyClass(ILogger[] myLogger)
  { }
}
```

**Visual Basic**

```
Class MyClass
  Public  Sub New(ByVal myLogger() As ILogger)
  End Sub
End Class
```

The following example code injects a specific instance, **myLogger**, of an array, **ILogger**, into **MyClass**.

**C#**

```
container.RegisterType<MyClass>(
  new InjectionConstructor(new ResolvedArrayParameter<ILogger>(myLogger)));
```

**Visual Basic**

```
container.RegisterType(MyClass)( _
  New InjectionConstructor(New ResolvedArrayParameter(Of ILogger)(myLogger)))
```

By using **ResolvedArrayParameter** as a constructor parameter you will get the registered instances of the specified array, **myLogger**. At resolve time, the container calls **ResolveAll<ILogger>** and injects the resulting array into the constructor.

The following example configures the container to inject specific array values, **log1** and **logger2**. **ResolvedArrayParameter** supplies the **InjectionConstructor** constructor argument. **typeof(ILogger)** is shorthand for **newResolvedParameter<ILogger>()**.

**C#**

```
ILogger logger2 = new SpecialLogger();
IUnityContainer container = new UnityContainer();
container.RegisterType<MyTypeArray>(new InjectionConstructor(
        new ResolvedArrayParameter(
              typeof(ILogger),
              new ResolvedParameter<ILogger>("log1"),
              typeof(ILogger), logger2)));
```

**Visual Basic**

```
Dim logger2 As ILogger = New SpecialLogger()
Dim container As IUnityContainer = New UnityContainer()
container.RegisterType(Of MyTypeArray) (New InjectionConstructor( _
        New ResolvedArrayParameter( _
            GetType(ILogger), _
            New ResolvedParameter(Of ILogger)("log1"), _
            GetType(ILogger), logger2)))
```

## Injecting All Array Named Instances

The following example configures the container to call the constructor with an array parameter. The entire array will be injected.

**C#**

```
ILogger o1 = new MockLogger();
ILogger o2 = new SpecialLogger();
container.RegisterType<MyArrayType>(
                      new InjectionConstructor(typeof(ILogger[])));
```

**Visual Basic**

```
Dim o1 As ILogger =  New MockLogger()
Dim o2 As ILogger =  New SpecialLogger()
container.RegisterType(Of MyArrayType)( _
```

```
New InjectionConstructor(GetType(ILogger()))
```

## Summary of the InjectionMember Methods and Overloads

The **RegisterType** method overloads take an optional **InjectionMember** parameter. For the table of RegisterType method overloads see the "Summary of the RegisterType Method Overloads" table in the Registering Types and Type Mappings topic.

The following table summarizes the **InjectionMembers** methods used in **RegisterType** method overloads that allow for configuring injection of parameters, properties, and methods**.** You can use these to register the information required for injecting types and values into constructor and method parameters and properties at run time.

| Method | Description |
|---|---|
| RegisterType(Type from, Type to, string name, LifetimeManager lifetimeManager,<br>params InjectionMember[] injectionMembers); | General form for using the **RegisterType** method. Registers a type mapping with the container so that it returns an instance of the type specified as *to* when a **Resolve** method requests an instance of the type *from* with the specified *name*, null if a default registration,<br>The *lifetimeManager* controls the lifetime of the returned instance. Objects configured to get injected by the container are specified by *injectionMembers*. |
| RegisterType<AType>(new InjectionConstructor()) | Creates a new instance of **InjectionConstructor** that uses the default constructor. |
| RegisterType<AType>(new InjectionConstructor(params object[] parameterValues)) | Creates a new instance of **InjectionConstructor** that looks for a constructor with the given set of parameters.<br>*parameterValues*: The values for the parameters. |
| RegisterType<AType>(new InjectionProperty(string propertyName)) | Configures the container to inject the given property name, resolving the value via the container. The *propertyName* parameter is the name of the property to inject. |
| RegisterType<AType>(new InjectionProperty((string propertyName, object propertyValue)) | Configures the container to inject the given property name, using the value supplied. The *propertyName* parameter is the name of the property to inject. The *propertyValue* parameter is the value for the property. |
| RegisterType<AType>(new InjectionParameter(object parameterValue)) | Creates an instance of **InjectionParameter** that stores the given value, using the runtime type of that value as the type of the parameter. |
| RegisterType<AType>(new InjectionParameter(System.Type parameterType, object parameterValue)) | Creates an instance of InjectionParameter that stores the given value, associated with the given type.<br>*parameterType*: Type of the parameter.<br>*parameterValue*: Value of the parameter |
| RegisterType<AType>(new InjectionParameter<TParameter>) | A generic version of **Microsoft.Practices.Unity.InjectionParameter**. Creates a new **InjectionParameter<TParameter>**. *TParameter* = value for the parameter. |
| RegisterType<AType>(new InjectionMethod(string methodName, params object[] methodParameters) | Configures the container to call the given methods with the given parameters. |

| | |
|---|---|
| | *methodName*: Name of the method to call.<br>*methodParameters*: Parameter values for the method. |

### More Information

For more information about the  techniques discussed in this topic, see the following topics:

- Registering Types and Type Mappings
- Resolving Objects by Using Overrides
- Using Injection Attributes

# Registering Generic Parameters and Types

This topic explains how you can register the information required for injection for generic types, including generic arrays. You can specify a generic type when you register a type in the Unity container in almost exactly the same way as you register non-generic types. Unity provides two classes specifically for registering generics, **GenericParameter** for specifying that an instance of a generic type parameter should be resolved, and **GenericResolvedArrayParameter** for specifying that an array containing the registered instances of a generic type parameter should be resolved.

See the "Specifying Types in the Configuration File" section in the Specifying Types in the Configuration File topic for more details on generics, including a discussion of unbounded, closed, and open generic types.

This topic contains the following sections that explain registering generics:

- Registering Generic Interfaces and Classes
- Registering Type Mappings For Generics
- Registering Generic Arrays
- Support for Generic Decorator Chains
- Methods for Registering Generic Parameters and Types
- More Information

## Registering Generic Interfaces and Classes

Registering closed generic types works exactly like it does for non-generic types. For more information see Registering Types and Type Mappings, Creating Instance Registrations and Registering Injected Parameter and Property Values.

You can use the **RegisterInstance** method to register open generic types, types with all their generic type parameters left unspecified, with non-ambiguous constructors.

Unity generic parameter injection run time API configuration support for **parameters** and **properties** is provided by the **GenericParameter** class. Use the **RegisterType** overloads with **GenericParameter** to register generic types. The following example registers the open generic interface, **IGenericClass**, in the container as the registered type, and registers **GenericClass** as the target type to be returned in response to a query for **IGenericClass**. Due to the syntax limitations in C# and Visual Basic .NET, you must use the overloads of **RegisterType** that take explicit **Type** objects instead of the generic version of **RegisterType**.

**C#**

```
container.RegisterType(typeof(IGenericClass<>), typeof(GenericClass<>));
```

**Visual Basic**

```
container.RegisterType(GetType(IGenericClass(Of )), GetType(GenericClass(Of )))
```

The following example class taking a generic parameter is used in the generic type registration examples.

**C#**

```
public class MyClass1<T>
{
    public T InjectedValue;
    public MyClass1 (string s, object o)
    {
    }

    public MyClass1 (T injectedValue)
    {
         InjectedValue = injectedValue;
    }
}
```

**Visual Basic**

```
Public Class MyClass1(Of T)
    Public InjectedValue As T
    Public Sub New(s As String, o As Object)
    End Sub

    Public Sub New(injectedValue As T)
        InjectedValue = injectedValue
    End Sub
End Class
```

The following examples show the registration for open generic types and type mappings using **RegisterType** with a generic parameter.

Here is how to call a non-generic constructor on an open generic type.

**C#**

```
// Where MyClass1 has at least one generic parameter
container.RegisterType(typeof (MyClass1<>),
                       new InjectionConstructor("Name",
                       new InjectionParameter<object>("objectName")));
```

**Visual Basic**

```
' Where MyClass1 has at least one generic parameter
container.RegisterType(GetType(MyClass1(Of )), _
                       New InjectionConstructor("Name", _
                       New InjectionParameter(Of Object)("objectName")))
```

Here is how to call a constructor that takes a generic parameter. Here we designate the generic parameter by using **GenericParameter**.

**C#**

```
// Where MyClass1 has at least one generic parameter
// Inject constructor with argument GenericParameter.
container.RegisterType(typeof (MyClass1<>),
                       new InjectionConstructor(new GenericParameter("T")));
Account a = new Account();
container.RegisterInstance<Account>(a);
```

**Visual Basic**

```
' Where MyClass1 has at least one generic parameter
' Inject constructor with argument GenericParameter.
container.RegisterType(GetType(MyClass1(Of )), _
                       New InjectionConstructor(New GenericParameter("T")))
Dim a As New Account()
container.RegisterInstance(Of Account)(a)
```

Here we configure a named resolution of **GenericParameter**() and designate the generic parameter by using **GenericParameter**.

**C#**

```
// Where MyClass1 has at least one generic parameter.
// Inject constructor with argument.
container.RegisterType(typeof(MyClass1<>),
          new InjectionConstructor(new GenericParameter("T", "named")));
//Create Account instances and register using registerInstance
Account a = new Account();
container.RegisterInstance<Account>(a);
Account named = new Account();
container.RegisterInstance<Account>("named", named);
```

**Visual Basic**

```
' Where MyClass1 has at least one generic parameter.
' Inject constructor with argument.
```

```
container.RegisterType(GetType(MyClass1(Of )), _
          New InjectionConstructor(New GenericParameter("T", "named")))
'Create Account instances and register using registerInstance
Dim a As New Account()
container.RegisterInstance(Of Account)(a)
Dim named As New Account()
container.RegisterInstance(Of Account)("named", named)
```

If you need to specify a mapping—for example, if an instance is registered to supply the implementation for an interface, the generic type argument must be provided with the **RegisterInstance** method.

## Registering Type Mappings for Generics

You can use the **RegisterType** method to register mappings in the container that include generics. The following example maps an open generic interface, **IOpenGenericInterface<,>**, to an open class, **MyOpenClass<,>,** with the **validating** name.

**C#**

```
// Map the open generic interface IOpenGenericInterface to the
// open class MyOpenClass.
public MyOpenClass(IOpenGenericInterface<T> repository,
                   IValidator<T> validator)
{
  ...
}
public class MyOpenClass<T> : IOpenGenericInterface<T>
{
    …
}

// Map the open generic interface IOpenGenericInterface
// to the open class MyOpenClass with the name "validating".
// Use the typeof keyword to specify Type instances.
container.RegisterType(typeof(IOpenGenericInterface<>),
                       typeof(MyOpenClass<>),
                       "validating");
```

**Visual Basic**

```
' Map the open generic interface IOpenGenericInterface to the
' open class MyOpenClass.
Public Class MyOpenClass(repository As IOpenGenericInterface(Of T), _
                         validator As IValidator(Of T))
  ...
End Class
Public Class MyOpenClass(Of T)
    Implements IOpenGenericInterface(Of T)
    ...
End Class
```

```
' Map the open generic interface IOpenGenericInterface
' to the open class MyOpenClass with the name "validating".
' Use the GetType keyword to specify Type instances.
container.RegisterType(GetType(IOpenGenericInterface(Of )), _
                       GetType(MyOpenClass(Of )), _
                       "validating")
```

> Open generic types cannot be used as generic type arguments. The version of the **RegisterType** method taking **Type** instances as parameters is used instead of the version with generic type parameters. The version with generic type parameters benefits from compile-time type checks.

The following example maps a closed generic interface to a non-generic class by using the **RegisterType** method to map the closed **IValidator<StockQuote>** interface to the non-generic **RandomStockQuoteValidator**. The **RandomStockQuoteValidator** class is a non-generic class that implements the closed generic interface, **IValidator<StockQuote>**.

**C#**

```
container.RegisterType<IValidator<StockQuote>, RandomStockQuoteValidator>();
```

**Visual Basic**

```
container.RegisterType(Of IValidator(Of StockQuote), _
                       RandomStockQuoteValidator)()
```

## Registering Generic Arrays

Unity provides the **GenericResolvedArrayParameter** to enable you to specify that an array containing the registered instances of a generic type parameter should be resolved. You can specify the entire array or specific named instances.

The following examples use the **MyClass2** class. **MyClass2** has a constructor with a generic array parameter.

**C#**

```
public class MyClass2<T>
{
    public T[] injectedValue;
    public readonly bool DefaultConstructorCalled;

    public MyClass2()
    {
        DefaultConstructorCalled = true;
    }

    public MyClass2(T[] injectedValue)
    {
        DefaultConstructorCalled = false;

        this.injectedValue = injectedValue;
```

```
    }

    public T[] InjectedValue
    {
        get { return this.injectedValue; }
        set { this.injectedValue = value; }
    }
}
```

**Visual Basic**

```
Public Class MyClass2(Of T)
    Public injectedValue As T()
    Public ReadOnly DefaultConstructorCalled As Boolean

    Public Sub New()
        DefaultConstructorCalled = True
    End Sub

    Public Sub New(injectedValue As T())
        DefaultConstructorCalled = False

        Me.injectedValue = injectedValue
    End Sub

    Public Property InjectedValue() As T()
        Get
            Return Me.injectedValue
        End Get
        Set
            Me.injectedValue = value
        End Set
    End Property
End Class
```

The following example registers a generic array by using the **RegisterType** method with a **GenericResolvedArrayParameter** parameter.

Here we call a constructor by using constructor injection, **InjectionConstructor**, that takes a generic array, as a parameter, **new GenericResolvedArrayParameter("T")**.

**C#**

```
IUnityContainer container = new UnityContainer()
    .RegisterType(
         typeof(MyClass2<>),
         new InjectionConstructor(new GenericResolvedArrayParameter("T")));
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer() _
    .RegisterType( _
         GetType(MyClass2(Of )), _
         New InjectionConstructor(New GenericResolvedArrayParameter("T")))
```

Then register three named instances, a0, a1 and a3, for the type **Account**:

**C#**

```
Account a0 = new Account();
container.RegisterInstance<Account>("a0", a0);
Account a1 = new Account();
container.RegisterInstance<Account>("a1", a1);
Account a2 = new Account();
container.RegisterInstance<Account>(a2);
```

**Visual Basic**

```
Dim a0 As New Account()
container.RegisterInstance(Of Account)("a0", a0)
Dim a1 As New Account()
container.RegisterInstance(Of Account)("a1", a1)
Dim a2 As New Account()
container.RegisterInstance(Of Account)(a2)
```

You resolve the array as follows:

**C#**

```
MyClass2<Account> result = container.Resolve<MyClass2<Account>>();
```

**Visual Basic**

```
Dim result As MyClass2(Of Account) = container.Resolve(Of MyClass2(Of Account))()
```

Then specify the named instances to resolve. The **InjectionConstructor** argument is again supplied by **new GenericResolvedArrayParameter** but has its arguments supplied by **new GenericParameter("T", "a2")** that returns a named instance.

**C#**

```
IUnityContainer container = new UnityContainer()
      .RegisterType(
                  typeof(MyClass2<>),
                  new InjectionConstructor(
                      new GenericResolvedArrayParameter(
                            "T",
                            new GenericParameter("T", "a2"),
                            new GenericParameter("T", "a1"))));
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer() _
      .RegisterType(GetType(MyClass2(Of )), _
           New InjectionConstructor( _
               New GenericResolvedArrayParameter( _
                   "T", _
                   New GenericParameter("T", "a2"), _
                   New GenericParameter("T", "a1"))))
```

Now, set a property with a generic parameter array type. Property injection for **MyClass2** is specified by **InjectionProperty**, which injects the property named **InjectedValue** with the generic array returned by **new GenericResolvedArrayParameter("T")**.

**C#**

```
IUnityContainer container = new UnityContainer()
```

```
    .RegisterType(typeof(MyClass2<>),
        new InjectionConstructor(),
        new InjectionProperty("InjectedValue",
                              new GenericResolvedArrayParameter("T")));
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer() _
    .RegisterType(GetType(MyClass2(Of )), _
        New InjectionConstructor(), _
        New InjectionProperty("InjectedValue", _
                              New GenericResolvedArrayParameter("T")))
```

## Support for Generic Decorator Chains

Support for generic decorator chains is provided by the generic parameter. The following example uses a generic decorator with a generic class and a generic array parameter.

**C#**

```
public class SupportClass
{ }

public class ClassGeneric<T>
{
    private T[] arrayProperty;
    public T[] ArrayProperty
    {
        get { return arrayProperty; }
        set { arrayProperty = value; }
    }

    private T[] arrayCtor;
    public T[] ArrayCtor
    {
        get { return arrayCtor; }
        set { arrayCtor = value; }
    }

    public ClassGeneric()
    { }

    // A generic class with an generic array property
    public ClassGeneric(T[] arrayCtor)
    {
        ArrayCtor = arrayCtor;
    }
}
```

**Visual Basic**

```
Public Class SupportClass
End Class

Public Class ClassGeneric(Of T)
```

```
    Private m_arrayProperty As T()
    Public Property ArrayProperty() As T()
        Get
            Return m_arrayProperty
        End Get
        Set(ByVal value As T())
            m_arrayProperty = value
        End Set
    End Property

    Private m_arrayCtor As T()
    Public Property ArrayCtor() As T()
        Get
            Return m_arrayCtor
        End Get
        Set(ByVal value As T())
            m_arrayCtor = value
        End Set
    End Property

    Public Sub New()
    End Sub

    ' A generic class with an generic array property
    Public Sub New(ByVal arrayCtor__1 As T())
        ArrayCtor = arrayCtor__1
    End Sub

End Class
```

The following example programmatically configures property injection. The **RegisterType** method registers a property with the name **ArrayProperty**, with **ClassGeneric**. **InjectionProperty** provides the property name and value. The name of the property is specified by the first parameter for **InjectionProperty** and the value for the property is specified by the second parameter, **GenericResolvedArrayParameter("T")**.

**C#**

```
public void ConfigureGenericInjection()
{
  IUnityContainer container = new UnityContainer()
    .RegisterType(typeof(ClassGeneric<>),
                  new InjectionProperty("ArrayProperty",
                  new GenericResolvedArrayParameter("T")));
}
```

**Visual Basic**

```
Public  Sub ConfigureGenericInjection()
  Dim container As IUnityContainer = New UnityContainer() _
    .RegisterType(GetType(ClassGeneric(Of )), _
                  New InjectionProperty("ArrayProperty", _
                  New GenericResolvedArrayParameter("T")))
```

```
End Sub
```

## Methods for Registering Generic Parameters and Types

The following table summarizes the methods you can use to register generic parameters and types with the container at run time.

| Method | Description |
|---|---|
| GenericResolvedArrayParameter(string genericParameterName, params object[] elementValues) | Creates a new **GenericResolvedArrayParameter** instance that specifies that the given named generic parameter should be resolved where *genericParameterName* is the generic parameter name to resolve and *elementValues* represents the values for the elements that will be converted to *InjectionParameterValue* objects. |
| GenericParameter(string genericParameterName) | Creates a new **GenericParameter** instance that specifies that the given named generic parameter should be resolved where *genericParameterName* is the generic parameter name to resolve. |
| GenericParameter(string genericParameterName, string resolutionKey) | Creates a new **GenericParameter** instance that specifies that the given named generic parameter should be resolved where *genericParameterName* is the generic parameter name to resolve and *resolutionKey* is the name to use when looking up the value in the container. |

## More Information

For more information about the techniques discussed in this topic, see the following topics:

- Specifying Types in the Configuration File
- Registering Types and Type Mappings

---

# Registering Container Extensions

This topic explains how to register information that instructs Unity to load and use container extensions that add additional functionality to the container, and how you can register configuration information for these extensions.

This topic contains the following sections that explain container extensions:

- Adding and Removing Extensions
- Accessing Configuration Information for Extensions
- Methods for Registering and Configuring Container Extensions

- More Information

## Adding and Removing Extensions

You can add a custom or third-party extension to the Unity container at run time using the **AddExtension** and **AddNewExtension** methods, and you can remove extensions using the **RemoveAllExtensions** method.

To add an extension, use the **AddNewExtension** or **AddExtension** method of the container. For example, this code creates a new instance of an extension class named **MyCustomExtension** that you have previously created, and which resides in this or a referenced project, and adds it to the container.

**C#**

```
IUnityContainer container = new UnityContainer();
container.AddNewExtension<MyCustomExtension>();
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer()
container.AddNewExtension(Of MyCustomExtension)()
```

If you already have an existing extension instance, you add it to the container using the **AddExtension** method. For example, this code adds an extension instance referenced by the variable named **myExistingExtension** to the container.

**C#**

```
IUnityContainer container = new UnityContainer();
container.AddExtension(myExistingExtension);
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer()
container.AddExtension(myExistingExtension)
```

The **AddNewExtension** method creates the actual extension object by resolving it from the container. The previous example  using **AddNewExtension** is equivalent to the following example.

**C#**

```
IUnityContainer container = new UnityContainer();
container.AddExtension(container.Resolve<MyCustomExtension>());
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer()
container.AddExtension(container.Resolve(Of MyCustomExtension)())
```

You cannot remove individual extensions from the container. However, you can remove all currently registered extensions and then add back any you want to use. To remove all existing extensions, use the **RemoveAllExtensions** method of the container. For example, this code removes all extensions from an existing container referenced in the variable named **container**.

**C#**

```
container.RemoveAllExtensions();
```

**Visual Basic**

```
container.RemoveAllExtensions()
```

Important parts of the container's core functionality are implemented using the extension mechanism. You will be unable to successfully resolve objects after you call **RemoveAllExtensions** unless you explicitly add those core parts back to the container.

## Accessing Configuration Information for Extensions

Container extensions will usually add strategies and policies to the container. However, extensions can expose configuration interfaces that allow them to read and expose configuration information. The extension must provide its own features to read and manage configuration information.

To access the configuration information for an extension, use the **Configure** method. This method of the **UnityContainer** class walks the list of extensions added to the container and returns the first one that implements the type you specify in the parameter of the **Configure** method. The method returns the required configuration interface, or it returns **null** (**Nothing** in Visual Basic) if the configuration type is not found.

There are two overrides of the **Configure** method. You can specify the extension type in the generic overload as a configurator (a class that implements the **IUnityContainerExtensionConfigurator** interface), or you can use the non-generic overload that accepts and returns an **Object** type. The following code shows both overrides of the **Configure** method used to retrieve the configuration as a type named **MyConfigInterface**.

**C#**

```
// using a "configurator" where MyConfigInterface implements
// the IUnityContainerExtensionConfigurator interface
MyConfigInterface configurator = container.Configure<MyConfigInterface>();

// using an Object type
Object config = container.Configure(typeof(MyConfigInterface));
```

**Visual Basic**

```
' using a "configurator" where MyConfigInterface implements
' the IUnityContainerExtensionConfigurator interface
Dim configurator As MyConfigInterface = container.Configure(Of
MyConfigInterface)()

' using the configuration interface type
Dim config As Object = container.Configure(GetType(MyConfigInterface))
```

For more information about configuration for container extensions, see Creating and Using Container Extensions.

## Methods for Registering and Configuring Container Extensions

The following table summarizes the method overloads you can use to register and configure container extensions at run time.

| Method | Description |
|---|---|
| AddExtension(UnityContainerExtension *extension*) | Adds the specified extension object, which must be of type **UnityContainerExtension**, to the container. Returns a reference to the container, equivalent to **this** in C# or **Me** in Visual Basic. |
| AddNewExtension<*TExtension*>( ) | Creates a new extension object of type *TExtension* and adds it to the container. Returns a reference to the container, equivalent to **this** in C# or **Me** in Visual Basic. |
| Configure<*TConfigurator*>( ) | Returns the configuration of the specified extension interface as an object of type *TConfigurator* or null if the specified extension interface is not found. Extensions can expose configuration interfaces in addition to adding strategies and policies to the container. This method walks the list of extensions and returns the first one that implements the specified type. |
| Configure(Type *configurationInterface*) | Returns the configuration of the specified extension interface as an object of type *configurationInterface* or null if the specified extension interface is not found. Extensions can expose configuration interfaces in addition to adding strategies and policies to the container. This method walks the list of extensions and returns the first one that implements the specified type. |
| RemoveAllExtensions( ) | Removes all installed extensions from the container, including all extensions that implement the default behavior, but it does not remove registered instances and singletons already set up in the container. To use the container again after executing this method, you must add either the default extensions or your own custom extensions. Returns a reference to the container, equivalent to **this** in C# or **Me** in Visual Basic. |

## More Information

For more information about the techniques discussed in this topic, see the following topics:

- Extending the Unity Configuration Schema
- Configuration Files for Interception
- Creating and Using Container Extensions
- Design of Unity
- Registering Interception

# Registering Interception

This topic explains run-time registration of the various interception elements, including interceptors, behaviors, policies, handlers, and matching rules that Unity uses to configure a container for interception. The configuration information is used when creating instances of types for which you want to add interception capabilities to change the behavior of that object or type. In order to provide backward compatibility, Unity 2.0 supports calling the older API **SetInterceptorFor** and **SetDefaultInterceptorFor** methods on the **Interception** container extension in addition to supporting the Unity 2.0 approach using the **RegisterType** API to explicitly configure interceptors, behaviors, and additional interfaces.



For information on using a configuration file to configure a container for interception, see Configuration Files for Interception.

For information on the design of Unity interception see Interception with Unity.

For information on using interception without a dependency injection (DI) container, see the "Stand Alone Unity Interception" section in Using Interception in Applications.

## Registering Interceptors and Interceptor Behaviors Explicitly Using RegisterType

Unity 2.0 enables interception like any other container extension by using **container.AddNewExtension**. Then you can configure a type for interception using an interceptor of your choosing, with behaviors of your choosing. In Unity 2.0 you explicitly configure which object is to be intercepted by which interception mechanism and specify the behavior by using **InterceptionBehavior**; Unity 1.2 implicitly set up policy injection when you configured an interceptor. The following example shows how to configure interception for a type and turn on a custom behavior. This example first adds the **Interception** extension by calling **AddNewExtension**, and then uses **RegisterType** to register a **VirtualMethodInterceptor** and an interception behavior. The behavior must be defined elsewhere.

**C#**

```
IUnityContainer container = new UnityContainer();
container.AddNewExtension<Interception>();
container.RegisterType<TypeToIntercept>(
          new Interceptor<VirtualMethodInterceptor>(),
          new InterceptionBehavior<CustomBehavior>());
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer()
container.AddNewExtension(Of Interception)()
container.RegisterType(Of TypeToIntercept)( _
          New Interceptor(Of VirtualMethodInterceptor)(), _
```

```
        New InterceptionBehavior(Of CustomBehavior)())
```

Using this overload of the **Interceptor** constructor actually tells the container to resolve the interceptor through the container. You can pass an optional string, which becomes the name to resolve with. In most applications you would simply leave this blank, but if you have implemented custom interceptors, you might want to provide additional configuration. There is another overload of the **Interceptor** constructor you can use to specify the interceptors and behaviors by creating instances and passing the actual instances into the container, as done in the following example:

**C#**

```
// Add the interception extension to the container
IUnityContainer container = new UnityContainer();
container.AddNewExtension<Interception>();
// Configure interception
container.RegisterType<IInterface, BaseClass>(
    "myInterceptor",
    new Interceptor(new InterfaceInterceptor()),
    new InterceptionBehavior(new CustomBehavior()),
    new InterceptionBehavior(new SomeOtherBehavior()));
```

**Visual Basic**

```
' Add the interception extension to the container
Dim container As IUnityContainer = New UnityContainer()
container.AddNewExtension(Of Interception)()
' Configure interception
container.RegisterType(Of IInterface, BaseClass) _
        ("myInterceptor", _
        New Interceptor(New InterfaceInterceptor()), _
        New InterceptionBehavior(New CustomBehavior()), _
        New InterceptionBehavior(New SomeOtherBehavior()))
```

## Default Interceptor for a Type

When using multiple named registrations for the same type, you often want all implementations of that type to be intercepted. Rather than requiring you to specify the interceptor for all registrations, you can instead specify a default interceptor on any one of the registrations. This interceptor will be applied any time that type is resolved, regardless of the name used to resolve it. Additionally, you can also provide default interception behaviors, which apply the behaviors to all instances of that type regardless of the instance name.

**C#**

```
// Add the interception extension to the container
IUnityContainer container = new UnityContainer();
container.AddNewExtension<Interception>();
// Configure default interception for BaseClass
container.RegisterType<IInterface>(
    new DefaultInterceptor<InterfaceInterceptor>(),
    new DefaultInterceptionBehavior(new CustomBehavior()),
    new DefaultInterceptionBehavior(new SomeOtherBehavior());
```

```
// Configure type mappings
container.RegisterType<IInterface, BaseClass>("myInterceptor")
    .RegisterType<IInterface, OtherClass>("interceptedToo");
```

**Visual Basic**

```
' Add the interception extension to the container
Dim container As IUnityContainer = New UnityContainer()
container.AddNewExtension(Of Interception)()
container.RegisterType(Of IInterface)( _
    new DefaultInterceptor(Of InterfaceInterceptor)(), _
    new DefaultInterceptionBehavior(new CustomBehavior()), _
    new DefaultInterceptionBehavior(new SomeOtherBehavior())

' Configure type mappings
container.RegisterType(Of IInterface, BaseClass)("myInterceptor") _
    .RegisterType(Of IInterface, OtherClass)("interceptedToo")
```

Visit MSDN for information on how previous versions of Unity provided Configuration Support for Interception.

## Registering Additional Interfaces

The interception mechanism can add entirely new interfaces to objects. The interfaces are implemented by the interception behaviors. In most cases, the behaviors themselves provide the information about which additional interfaces they implement. In rare cases such as for a mock object behavior, the behavior can implement any interface, so extra information is required about exactly which extra interfaces to add to the object.

The **InterceptionExtension.AdditionalInterface** class enables you to implement additional interfaces on target objects. It contains information for additional interfaces that will be added to the intercepted object and configures a container accordingly.

In the following example, **IOtherInterface** is an **AdditionalInterface** interface that configures the interceptor to implement a specified interface in addition to the original interfaces implemented by the target object, and **MyInterceptionBehavior** is an implementation of the **IInterceptionBehavior** interface which defines an interception behavior.

**C#**

```
IUnityContainer container = new UnityContainer();
container.RegisterType<IInterface, BaseClass>(
    "sample",
    new Interceptor<VirtualMethodInterceptor>(),
    new AdditionalInterface<IOtherInterface>(),
    new InterceptionBehavior<MyInterceptionBehavior>());
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer()
container.RegisterType(Of IInterface, BaseClass)( _
        "sample", _
```

```
        New Interceptor(Of VirtualMethodInterceptor)(), _
        New AdditionalInterface(Of IOtherInterface)(), _
        New InterceptionBehavior(Of MyInterceptionBehavior)())
```

For information on using a configuration file to configure interception for additional interfaces see Configuration Files for Interception.

# Registering Policy Injection Components

This topic explains how to register the various elements, including interceptors, behaviors, policies, call handlers, and matching rules that Unity uses to configure a container for interception and for a policy injection behavior.

When you configure policy injection you must specify which objects will be intercepted with the policy injection behavior and which policies in the container are to be used. Then when building the object, the policy injection behavior is set up using the policies already defined in the container.

This topic contains the following sections:

- Policy Injection Run-Time Configuration
- Defining Policies by Using the API

## Policy Injection Run-Time Configuration

There are two steps to configuring a type for policy injection. First, you must register the type in the container. In that registration, you must configure an interceptor and enable the **PolicyInjectionBehavior**. Second, you must configure the policy injection policies that determine which call handlers execute on which methods.

**C#**

```
int intercepted = 0;
var container = new UnityContainer();
container
    .AddNewExtension<Interception>()
    .RegisterType<ActionCallHandler>()
    // Register the type to be intercepted
    .RegisterType<InterceptedType>(
            new Interceptor<TransparentProxyInterceptor>(),
            new InterceptionBehavior<PolicyInjectionBehavior>())
    // Configure policies
    .Configure<Interception>()
        .AddPolicy("policy")
            .AddCallHandler(new ActionCallHandler(() => intercepted++))
            .AddMatchingRule(new MemberNameMatchingRule("MethodX"));
```

**Visual Basic**

```
Dim intercepted As Integer = 0
Dim container = New UnityContainer()
container _
    .AddNewExtension(Of Interception)() _
    .RegisterType(Of ActionCallHandler)() _
    ' Register the type to be intercepted
    .RegisterType(Of InterceptedType)( _
           New Interceptor(Of TransparentProxyInterceptor)(), _
           New InterceptionBehavior(Of PolicyInjectionBehavior)()) _
    ' Configure policies
    .Configure(Of Interception)() _
        .AddPolicy("policy") _
            .AddCallHandler (New ActionCallHandler(Function() _
             System.Math.Max(System.Threading.Interlocked.Increment _
                 (intercepted),intercepted - 1)))
```

## Defining Policies by Using the API

The streamlined **InterceptionExtension.PolicyDefinition** APIs provide a simplified way to wire up **RuleDriven** policies and their **IMatchingRules** and **ICallHandlers**. The general-purpose APIs require repeated calls to the **RegisterType** method. The streamlined extension APIs reduce the overhead required to manage the various strings and cross links, thus making the process more obvious and convenient.

Everything you can do with the **InterceptionExtension.PolicyDefinition** API can be done with the general-purpose APIs.

For more information about interception and selecting the objects and their members to add a handler pipeline, see Using Interception and Policy Injection.

The following are streamlined configuration **InterceptionExtension.PolicyDefinition** methods:

- **AddPolicy**. These methods are a set of methods on the interception type.
- **AddMatchingRule**. These methods are on the **PolicyDefinition** class you get when you call **AddPolicy**.
- **AddCallHandler**. These methods are on the **PolicyDefinition** class you get when you call **AddPolicy**.

---

These methods are only used for configuring rule-driven policies, which are also the only policies configurable with the standard installation and setup. For user-defined policies, you must use the general-purpose APIs.

The streamlined **InterceptionExtension.PolicyDefinition** API is similar to the expanded **RegisterType** API in that you can provide a lifetime manager (just like **RegisterType**), the mapping, and the injection configuration. It also differs in the following ways:

- The entry point for the API is the **AddPolicy** method in the interception extension.

- The result of this method is a transient **PolicyDefinition** object, which can be used to add matching rules and add call handlers. These methods add rules and handlers to the policy, but they also configure the container as necessary.
- The signatures for these methods are similar to those of **RegisterType**, but the names imply the interface being registered instead of relying on generic type parameters, as with the general-purpose **RegisterTypeMethods**. The following is an example:

**C#**

```
// Instead of:
RegisterType<ICallHandler, MyCallHandler>(...)
// you use:
AddCallHandler<MyCallHandler>(...).
```

**Visual Basic**

```
' Instead of:
RegisterType(Of ICallHandler, MyCallHandler)(...)
' you use:
AddCallHandler(Of MyCallHandler)(...).
```

There are three approaches for using the streamlined interception methods:

- **Supply a string parameter**. Use a string parameter to indicate that you want to use an object that was configured elsewhere. If you used the general-purpose API to configure a handler in the container, you would just link to it with this approach.

  > You can supply a string and configure the corresponding rule or handler at a later time.

  The following is an example of a policy with externally configured rules and handlers.

**C#**

```
public void PolicyUseExample()
{
    IUnityContainer container = new UnityContainer();
    container.AddNewExtension<Interception>();
    container
        .Configure<Interception>()
            .AddPolicy("MyPolicy")
            .AddMatchingRule("rule1")
            .AddCallHandler("handler1")
            .AddCallHandler("handler2")
    .Interception.Container
            .RegisterType<IMatchingRule, AlwaysMatchingRule>("rule1")
        .RegisterType<ICallHandler,LogCallHandler>(
            "handler1",
            new InjectionConstructor("handler1"))
        .RegisterType<ICallHandler,LogCallHandler>(
            "handler2",
            new InjectionConstructor("handler2"),
            new InjectionProperty("Order", 10));
        .RegisterType<TypeToIntercept>("wrappable",
            new Interceptor<TransparentProxyInterceptor>(),
```

```
            new InterceptionBehavior<PolicyInjectionBehavior>());

    LogCallHandler.Calls.Clear();
    var wrappable1 = container.Resolve<TypeToIntercept>("wrappable");
    wrappable1.Method2();
}
```

**Visual Basic**

```
Public Sub PolicyUseExample()
    Dim container As IUnityContainer = New UnityContainer()
    container.AddNewExtension(Of Interception)()
    container _
        .Configure(Of Interception)() _
            .AddPolicy("MyPolicy") _
            .AddMatchingRule("rule1") _
            .AddCallHandler("handler1") _
            .AddCallHandler("handler2") _
    .Interception.Container _
                .RegisterType(Of IMatchingRule, AlwaysMatchingRule) _
                          ("rule1") _
            .RegisterType(Of ICallHandler, LogCallHandler) _
                     ("handler1", New InjectionConstructor("handler1")) _
            .RegisterType(Of ICallHandler, LogCallHandler) _
                 ("handler2",  _
                 New InjectionConstructor("handler2"),  _
                 New InjectionProperty("Order", 10)) _
        .RegisterType(Of TypeToIntercept)("wrappable", _
            new Interceptor(Of TransparentProxyInterceptor)(), _
            new InterceptionBehavior(Of PolicyInjectionBehavior)())

    LogCallHandler.Calls.Clear()
    container _
           .Configure(Of Interception)() _
             .SetInterceptorFor(Of TypeToIntercept) _
               ("wrappable", New TransparentProxyInterceptor())
    Dim wrappable1 As TypeToIntercept = container.Resolve(Of
TypeToIntercept)("wrappable")
    wrappable1.Method2()
End Sub
```

- **Supply an instance**. Use this case when you already have the object and want only the new policy to use it.

**C#**

```
public void APolicyGivenRulesAndHandlers()
{
    IUnityContainer container = new UnityContainer();
    container.AddNewExtension<Interception>();
    IMatchingRule rule1 = new AlwaysMatchingRule();
    ICallHandler handler1 = new CallCountHandler();
    container
        .Configure<Interception>()
            .AddPolicy("MyPolicy")
```

```
            .AddMatchingRule(rule1)
            .AddCallHandler(handler1);
        .RegisterType<TypeToIntercept>("wrappable",
            new Interceptor<TransparentProxyInterceptor>(),
            new InterceptionBehavior<PolicyInjectionBehavior>());
    var wrappable1 = container.Resolve<TypeToIntercept>("wrappable");
    wrappable1.Method2();
}
```

**Visual Basic**

```
Public Sub APolicyGivenRulesAndHandlers()
    Dim container As IUnityContainer = New UnityContainer()
    container.AddNewExtension(Of Interception)()
    Dim rule1 As IMatchingRule = New AlwaysMatchingRule()
    Dim handler1 As ICallHandler = New CallCountHandler()
    container _
        .Configure(Of Interception)() _
            .AddPolicy("MyPolicy") _
            .AddMatchingRule(rule1) _
            .AddCallHandler(handler1)
    container.RegisterType(Of TypeToIntercept)("wrappable",
        new Interceptor(Of TransparentProxyInterceptor)(), _
        new InterceptionBehavior(Of PolicyInjectionBehavior)())
    Dim wrappable1 = container.Resolve(Of TypeToIntercept)("wrappable")
    wrappable1.Method2()

End Sub
```

- **Supply a type**. Supply a type either as a generic type parameter or a normal parameter, and, optionally, a name, a lifetime container, and injection configuration in any combination. Use this if the matching rule or call handler object is unique to the policy it is defined in, as it allows you to centralize the configuration of the policy in one spot. Normally, you do not need to specify a name for the matching rules or call handlers. If you do, that name can be used in other policies to reuse the named matching rule or call handler using the configuration approach, as shown in supply a string parameter. In this approach, you describe how to resolve for injection.

**C#**

```
public void APolicyGivenRulesAndHandlersTypes()
{
    IUnityContainer container = new UnityContainer();
    container.AddNewExtension<Interception>();
    container
       .Configure<Interception>()
            .AddPolicy("MyPolicy")
                .AddMatchingRule(typeof(AlwaysMatchingRule)),
                    new InjectionConstructor("rule1"))
                .AddCallHandler(typeof(LogCallHandler)),
                    new InjectionConstructor("handler1"),
                    new InjectionProperty("Order", 10))
    container.RegisterType<TypeToIntercept>("wrappable",
        new Interceptor<TransparentProxyInterceptor>(),
```

```
        new InterceptionBehavior<PolicyInjectionBehavior>());
    LogCallHandler.Calls.Clear();
    TypeToIntercept wrappable1 =
container.Resolve<TypeToIntercept>("wrappable");
    wrappable1.Method2();
}
```

**Visual Basic**

```
Public Sub APolicyGivenRulesAndHandlersTypes()
    Dim container As IUnityContainer = New UnityContainer()
    container.AddNewExtension(Of Interception)()
    container.Configure(Of Interception)() _
        .AddPolicy("MyPolicy") _
        .AddMatchingRule(GetType(AlwaysMatchingRule)),
            new InjectionConstructor("rule1")) _
        .AddCallHandler(GetType(LogCallHandler)),
            new InjectionConstructor("handler1"), _
            new InjectionProperty("Order", 10))
    container.RegisterType(Of TypeToIntercept)("wrappable", _
        new Interceptor(Of TransparentProxyInterceptor)(), _
        new InterceptionBehavior(Of PolicyInjectionBehavior)())
    LogCallHandler.Calls.Clear()
    Dim wrappable1 As TypeToIntercept = _
        container.Resolve(Of TypeToIntercept)("wrappable")
    wrappable1.Method2()
End Sub
```

One significant difference between the **RegisterType** methods and this streamlined API is that when you do not specify a name, a name is generated for you, so all anonymous definitions without names will not be overwritten by other definitions without names. If you do specify a name, it is used and may override previous existing policies with the same name.

You can use generic parameter support when you configure for injection, just like you do with **RegisterType**.

The generic versions of the **AddMatchingRule** and **AddCallHandler** methods contain the types you can supply, unlike **RegisterType**. **RegisterType** is a general-purpose method and places no constraints on the types you can supply. For **AddCallHandler<TCallHandler>()**, you can only provide a type value for **TCallHandler** that implements **ICallHandler**. The non-generic version of the method is limited to performing run-time checks. Similarly, **AddMatchingRule<TRule>()** is constrained to types implementing **IMatchingRule**.

Once you have defined a policy, you must also set up an interceptor and turn on policy injection for each type you want intercepted. Setting the interceptor just says how to do interception on a type if it has members that match any rules. Without the policy injection behavior, rules will not be checked. And you must have at least one matching rule defined or nothing will happen.

In general, you do not want to use **AlwaysMatchingRule**, since that matches absolutely everything, which is rather indiscriminate (and is only part of the test suite, not the main .dll file). It is useful as a shortcut for testing purposes; but, using the **Type** or **Namespace** matching rules is better for more generalized matches.

# Using Unity in Applications

This topic describes how to develop applications using Unity, and how to create and build instances of objects. It assumes that you understand how to configure the Unity container. This section includes the following topics:

- Application Design Concepts with Unity. This topic explains how Unity can help you to implement common design patterns and achieve decoupling and coherence in your designs.
- Adding Unity to Your Application. This topic describes how to add Unity to your project, and how to reference the appropriate assemblies in your code.
- Resolving Objects. This topic contains a series of sections that describe how you can resolve objects through the Unity container so that it creates the appropriate type and optionally populates any dependencies specified for these types.
- Understanding Lifetime Managers. This topic describes the way that Unity manages the lifetime of objects it creates, and how you can use the lifetime managers included with Unity.
- Using Container Hierarchies. This topic explains how you can use a hierarchy of nested Unity containers to achieve finely grained control over the configuration of Unity and manage this configuration at run time.

---

For information on how to configure Unity, see Configuring Unity.

Unity ships as both source code and signed binary assemblies. You can use the signed assemblies directly. If you intend to compile the source code, see Target Audience and System Requirements.

# Application Design Concepts with Unity

Features such as inversion of control, dependency injection, interception, factory, and lifetime (some of which are described in the "Scenarios for Unity" section of the topic When Should I Use Unity?) provide several major advantages when building applications that consist of many individual classes and components. Designing applications that conform to these patterns can provide the following:

- The capability to substitute one component for another using a pluggable architecture.
- The capability to centralize and abstract common features and to manage crosscutting concerns such as logging, authentication, caching, and validation.
- Increased configuration flexibility.
- The capability to locate and instantiate services and components, including singleton instances of these services and components.

- Simplified testability for individual components and sections of the application.
- Simplified overall design, with faster and less error-prone development.
- Ease of reuse for common components within other applications.

---

Of course, implementing these patterns can initially make the design and development process more complex, but the advantages easily justify this extra complexity. In addition, the use of a comprehensive dependency injection mechanism can actually make the task of designing and developing applications much easier.

Fundamentally, there are two approaches to using a dependency injection mechanism:

- You can arrange to have dependent objects automatically injected, using techniques such as constructor injection, property (setter) injection, and method call injection that inject dependent objects immediately when you instantiate the parent object. This approach is generally most appropriate for applications that require a pluggable architecture or where you want to manage crosscutting concerns.
- You can have objects injected only on demand, by calling the **Resolve** method of the container only when you need to retrieve a reference to a specific object.
- This approach is known as service locator. It is more intrusive into your application, but can be simpler if your architecture does not lend itself to having a central container.

---

In addition to dependency injection, developers may wish to implement patterns such as Interception, Decorator, Chain Of Responsibility, and Intercepting Filter, where a call from a client or process passes through a graph of objects, with each one able to access and act upon details of the call, such as the method or property name, the parameter types and values, the returned type and value, and other information. Unity achieves this through interception of method calls, providing opportunities to apply policies to objects using a technique often referred to as policy injection.

Unity provides a comprehensive dependency injection and interception mechanism, and is easy to incorporate into your applications. However, it does change the way that you design these applications. The following sections of this topic describe areas where dependency injection is useful:

- Pluggable Architectures
- Managing Crosscutting Concerns
- Service and Component Location
- Policy Injection through Interception

---

## Pluggable Architectures

By designing applications to use a pluggable architecture, developers and users can add and modify the functionality of an application without changing the core code or processes. ASP.NET is an example of a pluggable architecture, where you can add and remove providers for features such as

authentication, caching, and session management. Unity also facilitates a pluggable architecture that allows you to substitute the container for one of your own design and add extensions through the container extensions architecture.

By using dependency injection mechanisms, such as Unity container, developers can map different components that implement a specific interface (such as a provider interface) or that inherit from a specific base class to the appropriate concrete implementations. Developers can then obtain a reference to the appropriate provider component at run time, without having to specify exactly which implementation of the component is required. The following sections of this topic, Managing Crosscutting Concerns and Service and Component Location, show how you can use dependency injection to create instances of a specific class based on a request for a more general class (such as an interface or base class definition).

In addition, the providers may require instances of other components. For example, a caching provider may need to use the services of a cryptography component or service. When components require the services of other components, you can use dependency injection to automatically instantiate and inject instances of these components into a component, based on either configuration settings in the container or on code within the application.

For more information about how you can use dependency injection in this way, see Using Injection Attributes.

## Managing Crosscutting Concerns

The features and tasks implemented in applications are often referred to as concerns. The tasks specific to the application are core concerns. The tasks that are common across many parts of the application, and even across different applications, are crosscutting concerns. Most large applications require services such as logging, caching, validation, and authorization, and these are crosscutting concerns.

The simplest way to centralize these features is to build separate components that implement each of the required features, and then use these components wherever the management of that concern is required—in one or more applications. By using dependency injection mechanisms, such as a Unity container, developers can register components that implement crosscutting concerns and then obtain a reference to the component at run time, without having to specify exactly which implementation of the component is required.

For example, if you have a component named **FileLogger** that performs logging tasks, your application can instantiate this component using the **new** operator, as shown in the following code.

**C#**

```
FileLogger myLogger = new FileLogger();
```

**Visual Basic**

```
Dim myLogger As New FileLogger()
```

If you then want to change the application to use the new component, **FastFileLogger**, you must change the application code. However, if you have an interface **ILogger** that all implementations of the logging component uses, you might instead write the code like the following.

**C#**

```
ILogger myLogger = new FileLogger();
```

**Visual Basic**

```
Dim myLogger As ILogger = New FileLogger()
```

However, this does not solve the problem because you still must find everywhere in the code where you used **new()** to create an instance of the specific class. Instead, you can use a dependency injection mechanism to map the **ILogger** interface to a specific concrete instance of the logging component, or even to multiple implementations, so that the application can specify which one it requires. The following code uses Unity to register a mapping for both a default type of logging component and a named type with the **ILogger** interface. Alternatively, you can specify these mappings using the Unity configuration file.

**C#**

```
// Create container and register types
IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType<ILogger, FileLogger>();       // default instance
myContainer.RegisterType<ILogger, FastFileLogger>("FastLogger");
```

**Visual Basic**

```
' Create container and register types
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(Of ILogger, FileLogger)()   ' default instance
myContainer.RegisterType(Of ILogger, FastFileLogger)("FastLogger")
```

The application can then obtain a reference to the default logging component using the following code.

**C#**

```
ILogger myLogger = myContainer.Resolve<ILogger>();
```

**Visual Basic**

```
Dim myLogger As ILogger = myContainer.Resolve(Of ILogger)()
```

In addition, the application can use a variable (perhaps set in configuration or at run time) to specify a different implementation of the logging component interface as required, as shown in the following code.

**C#**

```
// Retrieve logger type name from configuration
String loggerName = ConfigurationManager.AppSettings["LoggerName"].ToString();
ILogger myLogger = myContainer.Resolve<ILogger>(loggerName);
```

**Visual Basic**

```
' Retrieve logger type name from configuration
Dim loggerName As String =
ConfigurationManager.AppSettings("LoggerName").ToString()
Dim myLogger As ILogger = myContainer.Resolve(Of ILogger)(loggerName)
```

For more information about registering types, type mappings, and resolving instances, see Resolving Objects.

## Service and Component Location

Frequently, applications require the use of services or components that are specific to the application; examples are business logic components, data access components, interface components, and process controllers. In some cases, these services may be instance-based, so that each section of the application or each task requires a separate individual instance of the service. However, it is also common for services to be singleton-based, so that every user of the service references the same single instance of the service.

A service location facility makes it easy for an application to obtain a reference to a service or component, without having to specify where to look for the specific service or whether it is a singleton-based or an instance-based service. By using dependency injection mechanisms, such as the Unity container, developers can register services in the appropriate way and then obtain a reference to the service at run time, without having to specify exactly which implementation of the service is required or what type of instance it actually is.

For example, if you have a singleton service class named **CustomerData** that you interact with to read and update information for any customer, your application obtains a reference to this service usually by calling a static **GetInstance** method of the service (which ensures that it is a singleton and that only one instance can exist), as shown in the following code.

**C#**

```
CustomerData cData = CustomerData.GetInstance();
```

**Visual Basic**

```
Dim cData As CustomerData = CustomerData.GetInstance()
```

Instead, you can use the Unity container to set the **CustomerData** class type with a specific lifetime that ensures it behaves as a singleton so that every request for the service returns the same instance, as shown in the following code. Alternatively, you could specify these mappings using the Unity configuration file.

**C#**

```
// Create container and register type as a singleton instance
IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType<CustomerData>(new ContainerControlledLifetimeManager());
```

**Visual Basic**

```
' Create container and register type as a singleton instance
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(Of CustomerData)(New
ContainerControlledLifetimeManager())
```

The application can then obtain a reference to the single instance of the **CustomerData** service using the following code. If the instance does not yet exist, the container creates it.

**C#**

```
CustomerData cData = myContainer.Resolve<CustomerData>();
```

**Visual Basic**

```
Dim cData As CustomerData = myContainer.Resolve(Of CustomerData)()
```

In addition, perhaps a new **CustomerFile** component you decide to use in your application inherits the same base class named **CustomerAccessBase** as the **CustomerData** service, but it is not a singleton—instead, it requires that your application instantiate an instance for each customer. In this case, you can specify mapping names when you register one component as a singleton type and one component as an instance type (with the default transient lifetime), and then use the same application code to retrieve the required instance.

For example, the following code shows how you can register two named mappings for objects that inherit from the same base class, then—at run time—collect a string value from elsewhere in the application configuration that specifies which of the mappings to use. In this case, the value comes from the **AppSettings** section of the configuration file. If the value with the key **CustomerService** contains **CustomerDataService**, the code returns an instance of the **CustomerData** class. If it contains the value **CustomerFileService**, the code returns an instance of the **CustomerFile** class.

**C#**

```
IUnityContainer myContainer = new UnityContainer();
// Register CustomerData type as a singleton instance
myContainer.RegisterType<CustomerAccessBase, CustomerData>("CustomerDataService",
                        new ContainerControlledLifetimeManager());
// Register CustomerFile type with the default transient lifetime
myContainer.RegisterType<CustomerAccessBase,
CustomerFile>("CustomerFileService");
...
String serviceName =
ConfigurationManager.AppSettings["CustomerService"].ToString();
CustomerAccessBase cData
  = (CustomerAccessBase)myContainer.Resolve<CustomerAccessBase>(serviceName);
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer()
' Register CustomerData type as a singleton instance
myContainer.RegisterType(Of CustomerAccessBase,
CustomerData)("CustomerDataService", _
                         New ContainerControlledLifetimeManager())
' Register CustomerFile type with the default transient lifetime
myContainer.RegisterType(Of CustomerAccessBase,
CustomerFile)("CustomerFileService")
...
Dim serviceName As String =
ConfigurationManager.AppSettings("CustomerService").ToString()
Dim cData As CustomerAccessBase = myContainer.Resolve(Of
CustomerAccessBase)(serviceName)
```

For more information about registering types, type mappings, and resolving instances, see Resolving Objects. For more information about using lifetime managers to control the creation, lifetime, and disposal of objects, see Understanding Lifetime Managers.

## Policy Injection through Interception

Unity interception with its built-in policy injection module enables you to effectively capture calls to objects you resolve through the Unity DI container, and apply a policy that adds additional functionality to the target object. Typically, you will use this technique to change the behavior of existing objects, or to implement the management of crosscutting concerns through reusable handlers. You can specify how to match the target object using a wide range of matching rules, and construct a policy pipeline that contains one or more call handlers.

Calls to the intercepted methods or properties of the target object then pass through the call handlers in the order you add them to the pipeline, and return through them in the reverse order. Your call handlers can access the values in the call, change these values, and control execution of the call. For example, the call handlers might authorize users, validate parameter values, cache the return value, and shortcut execution so that the target method does not actually execute where this is appropriate.

You can configure Unity for policy injection by using a configuration file at design time, see Configuring Policy Injection Policies, or by using the API at run time, see Registering Policy Injection Components.

The following example uses the Unity API to demonstrate how you can configure Unity to perform interception on a target object, using a policy that contains a logging handler and a validation handler. Notice that the logging handler is added first, so that it will log calls even if validation fails and the validation handler shortcuts the pipeline instead of calling the method of the target object. You can use the streamlined policy definition API provided by the Unity interception container extension to configure the container at run time or you can specify the same behavior at design time by using a configuration file. For an example using the API, see Registering Policy Injection Components. For a design time example, see Configuration Files for Interception.

**C#**

```
// Create a container and add the interception extension.
IUnityContainer myContainer = new UnityContainer();
myContainer.AddNewExtension<Interception>();

// Configure the container with a policy named MyPolicy
// that uses a TypeMatchingRule to match a custom class
// and adds a logging handler and a validation handler
// to the handler pipeline. You must specify at least one
// matching rule or the policy will not be applied.
myContainer.Configure<Interception>()
    .AddPolicy("MyPolicy")
         .AddMatchingRule<TypeMatchingRule>(
                  new InjectionConstructor("MyCustomType"))
         .AddCallHandler(typeof(MyLoggingCallHandler))
         .AddCallHandler(typeof(MyValidationCallHandler));

// Configure the container to intercept calls to the
// custom class using a TransparentProxyInterceptor.
myContainer.RegisterType<MyCustomType>("myType",
    new Interceptor<TransparentProxyInterceptor>(),
```

```
    new InterceptionBehavior<PolicyInjectionBehavior>());


// Resolve the custom type through the container when
// you are ready to use it. When you call a method or
// set a property on it, the call will pass through
// the logging handler and the validation handler.
MyCustomType myNewInstance = myContainer.Resolve<MyCustomType>("myType");
```

**Visual Basic**

```
' Create a container and add the interception extension.
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.AddNewExtension(Of Interception)()

' Configure the container with a policy named MyPolicy
' that uses a TypeMatchingRule to match a custom class
' and adds a logging handler and a validation handler
' to the handler pipeline. You must specify at least one
' matching rule or the policy will not be applied.
myContainer.Configure(Of Interception)() _
    .AddPolicy("MyPolicy") _
        .AddMatchingRule(Of TypeMatchingRule)( _
            new InjectionConstructor ("MyCustomType")) _
               .AddCallHandler(GetType(MyLoggingCallHandler)) _
               .AddCallHandler(typeof(MyValidationCallHandler))

' Configure the container to intercept calls to the
' custom class using a TransparentProxyInterceptor.
myContainer.RegisterType(Of MyCustomType)( _
    New Interceptor(Of TransparentProxyInterceptor)(), _
    New InterceptionBehavior(Of PolicyInjectionBehavior)())

' Resolve the custom type through the container when
' you are ready to use it. When you call a method or
' set a property on it, the call will pass through
' the logging handler and the validation handler.
Dim myNewInstance As MyCustomType = myContainer.Resolve(Of
MyCustomType)("myType");
```

For more information about interception and policy injection, see Using Interception and Policy Injection.

# Adding Unity to Your Application

Unity is designed to support a range of common scenarios for resolving instances of objects that, themselves, depend on other objects or services. However, you must first prepare your application to use Unity. The following procedure describes how to include the necessary assemblies and elements in your code.

**To prepare your application**

1. Add a reference to the Unity assembly. In Visual Studio, right-click your project node in Solution Explorer, and then click **Add Reference**. Click the **Browse** tab and find the location of the **Microsoft.Practices.Unity.dll** assembly. Select the assembly, and then click **OK** to add the reference.
2. (Optional) If you intend to use the configuration types when you create extensions for Unity, use the same procedure to set a reference to the Unity configuration assembly, named **Microsoft.Practices.Unity.Configuration.dll**.
3. (Optional) If you intend to use the interception and policy injection features of Unity, use the same procedure to set a reference to the Unity interception assembly, named **Microsoft.Practices.Unity.Interception.dll**.
4. (Optional) If you intend to use the configuration types for the interception and policy injection features of Unity, use the same procedure to set a reference to the Unity interception configuration assembly, named **Microsoft.Practices.Unity.Interception.Configuration.dll**.
5. (Optional) To use elements from Unity without fully qualifying the element reference, add the following **using** statements (C#) or **Imports** statements (Visual Basic) to the top of your source code file as required.

   **C#**

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

   **Visual Basic**

   ```
   Imports Microsoft.Practices.Unity
   Imports Microsoft.Practices.Unity.Configuration
   Imports Microsoft.Practices.Unity.InterceptionExtension
   ```

6. (Optional) If you are using the **IServiceLocator** interface, add a reference to the service location binary **Microsoft.Practices.ServiceLocation.dll.** Visual Studio may automatically copy this file to your bin directory when it compiles, but you do not need to include it unless you are explicitly using the **UnityServiceLocatorAdapter** class.
7. Add your application code. For more information about how you can use Unity in your own applications, see What Does Unity Do?

---

For Visual Basic projects, you can also use the **References** page of the Project Designer to manage references and imported namespaces. To access the **References** page, select a project node in Solution Explorer, and then click **Properties** on the **Project** menu. When the Project Designer appears, click the **References** tab.

There are limitations when using Unity in a partial trust environment. For more information, see Using Unity in Partial Trust Environments.

# Dependency Injection with Unity

If you are using dependency injection (DI) through a DI container approach to your application development, you can use any available DI container including the container provided by Unity. Using the Unity dependency injection container provides opportunities for you to more easily decouple components, business objects, and services you use in applications, and can simplify how you organize and architect these applications.

You can create instances of objects using the DI container provided by Unity. Unity is available as a stand-alone dependency injection mechanism.

The following sections of this topic will help you to understand the overall process, and use Unity dependency injection in your applications:

- Using BuildUp to Wire Up Objects Not Created by the Container. This topic explains how to use BuildUp to pass existing object instances through the container in order to apply dependency injection to that object. This is an alternative to resolving the object using any of the other techniques available with Unity.
- Using Injection Attributes. This topic contains a series of sections that describe how you can use attributes applied to members of target classes to instruct Unity to inject dependent objects for constructor and method parameters, and as the values of properties.
- Circular References with Dependency Injection. This topic describes how you should be aware of the possibility of circular references arising when using dependency injection techniques.

# Resolving Objects

You can use the Unity container to generate instances of any object that has a public constructor (in other words, objects that you can create using the **new** operator), without registering a mapping for that type with the container. When you call the **Resolve** method and specify the default instance of a type that is not registered, the container simply generates and returns an instance of that type. However, the only time that this is realistically practical is when the object you are generating contains dependency attributes that the container will use to inject dependent objects into the requested object.

The Unity container identifies type registrations and type mappings in the container using a type and, optionally, a name. The type is an interface or a class (usually an interface or base class) that the desired concrete object type implements or inherits. This identifies the mapping so that the container can retrieve the correct object type in response to a call to the **Resolve** or **ResolveAll** method. Where there is more than one mapping for the same type, the optional name differentiates these mappings and allows code to specify which of the mappings for that type to use.

The provision of both generic and non-generic overloads of many of the Unity container methods ensures that Unity can be used in languages that do not support generics. You can use either approach (the generic or the non-generic overloads) in your code and mix them as required. For example, you can register mappings using the generic overloads and then retrieve object instances using the non-generic overloads, and vice versa.

When you attempt to resolve an abstract base class or interface where there is no matching type mapping in the container, Unity will attempt to create a new instance of the class you specified. As it cannot construct and populate an instance of an abstract class or an interface, Unity will raise an exception.

When you attempt to resolve a non-mapped concrete class that does not have a matching registration in the container, Unity will create an instance of that class and populate any dependencies.

The following topics describe how you can resolve objects using the **Resolve** or **ResolveAll** methods:

- Resolving an Object by Type.
- Resolving an Object by Type and Registration Name.
- Resolving All Objects of a Particular Type.
- Resolving Objects by Using Overrides
- Retrieving Container Registration Information

For more information about how you can configure Unity with type registrations and mappings, see Configuring Unity.

For more information about how you can perform dependency injection on existing object instances, see Using BuildUp to Wire Up Objects Not Created by the Container.

# Resolving an Object by Type

Unity provides a method named **Resolve** that you can use to resolve an object by type, and optionally by providing a registration name. Registrations that do not specify a name are referred to as default registrations. This topic describes how to use the **Resolve** method to resolve types and mappings registered as default registrations. For information about resolving named registrations, see Resolving an Object by Type and Registration Name.

## The Resolve Method Overloads for Default Registrations

The following table describes the overloads of the **Resolve** method that return instances of objects based on the default registrations and mappings with the container. The API for the Unity container contains both generic and non-generic overloads of this method so that you can use it with languages that do not support the generics syntax.

| Method | Description |
|---|---|
| Resolve<*T*>( ) | Returns an instance of the default type registered with the container as the type *T*. |
| Resolve(Type *t*) | Returns an instance of the default type registered with the container as the type *t*. |

If you call the **Resolve** method and specify the default instance of a type that is not registered, the container simply generates and returns an instance of that type. However, the only time that this is useful is when the object you are generating contains dependency attributes that the container will use to inject dependent objects into the requested object, or you want to intercept calls to the object to inject a policy.

If you register a default type or a default type mapping more than once using the **RegisterType** method (in other words, if you register more than one type or type mapping that specifies the same types and does not specify a registration name) only the last registration remains in the container and is applied when you execute the **Resolve** method.

The default registration is the same as a registration where the name is null or an empty string.

## Using the Resolve Method with Default Registrations

The following examples show how you can use the **Resolve** method to create or obtain a reference to an object defined in the container configuration. Typically you will register a type mapping between an interface and a concrete type that implements it, or between a base class and a concrete type that inherits it. The examples use the run-time methods of the container to register the types it will resolve. For more information about how you can configure Unity with type registrations and mappings, see Configuring Unity.

## Resolving Types Registered as Interfaces

The following code registers a mapping for an interface named **IMyService** and specifies that the container should return an instance of the **CustomerService** class (which implements the **IMyService** interface). In this case, the type **IMyService** identifies the registration type. Code that requests an instance of the type **IMyService** receives an instance of the **CustomerService** class. The following example uses the generic overloads of the container methods.

**C#**

```
IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType<IMyService, CustomerService>();
IMyService myServiceInstance = myContainer.Resolve<IMyService>();
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(Of IMyObject, CustomerService)()
Dim myServiceInstance As IMyService = myContainer.Resolve(Of IMyService)()
```

Alternatively, you can use the non-generic overloads of the methods. The following code achieves the same result.

**C#**

```
IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType(typeof(IMyService), typeof(CustomerService));
IMyService myServiceInstance =
(IMyService)myContainer.Resolve(typeof(IMyService));
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(GetType(IMyService), GetType(CustomerService))
Dim myServiceInstance As IMyService = myContainer.Resolve(GetType(IMyService))
```

## Resolving Types Registered as Base Classes

When you need to register a mapping for a base class or other object type (instead of an interface), you use the overloads of the **RegisterType** and **Resolve** methods that accept object type names. The following examples show the use of the overloads of the **RegisterType** and **Resolve** methods that accept object type names as the registration identifier.

The following code registers a mapping for an object named **MyBaseService** and specifies that the container should return an instance of the **CustomerService** class (which inherits from the **MyBaseService** class). In this case, the type **MyBaseService** identifies the registration. Code that requests an instance of the type **MyBaseService** receives an instance of the **CustomerService** class.

**C#**

```
IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType<MyBaseService, CustomerService>();
MyBaseService myServiceInstance = myContainer.Resolve<MyBaseService>();
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(Of MyBaseService, CustomerService)()
Dim myServiceInstance As MyBaseService = myContainer.Resolve(Of MyBaseService)()
```

Alternatively, you can use the non-generic overloads of the methods. The following code achieves the same result.

**C#**

```
IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType(typeof(MyBaseService), typeof(CustomerService));
```

```
MyBaseService myServiceInstance =
(MyBaseService)myContainer.Resolve(typeof(MyBaseService));
```

**Visual Basic**

```
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(GetType(MyBaseService), GetType(CustomerService))
Dim myServiceInstance As MyBaseService =
myContainer.Resolve(GetType(MyBaseService))
```

If the target class or object specifies any dependencies of its own, using constructor, property, or method call injection attributes, the instance returned will have these dependent objects injected automatically.

By default, the **RegisterType** method registers a type with a transient lifetime, which means that the container will not hold onto a reference to the objects it creates when you call the **Resolve** method. Each time you call one of these methods, the container generates a new instance of the specified or mapped type. However, you can use lifetime managers to control the creation, lifetime, and disposal of objects if required.

# Resolving an Object by Type and Registration Name

Unity provides a method named **Resolve** that you can use to resolve an object by type, and optionally by providing a registration name. Registrations that specify a name are referred to as named registrations. This topic describes how to use the **Resolve** method to resolve types and mappings registered as named registrations. For information about resolving default registrations, see Resolving an Object by Type.

## The Resolve Method Overloads for Named Registrations

The following table describes the overloads of the **Resolve** method that return instances of objects based on named registrations and mappings with the container. The API for the Unity container contains both generic and non-generic overloads of this method so that you can use it with languages that do not support the generics syntax.

| Method | Description |
|---|---|
| Resolve<*T*>(string *name*, *params ResolverOverride[] resolverOverrides*) | Returns an instance of the type registered with the container as the type ***T*** with the specified name and where **params** provide any constructor overrides for the **Resolve** calls. Names are case sensitive. |

| | |
|---|---|
| Resolve(Type *t*, string *name*, *params ResolverOverride[] resolverOverrides*) | Returns an instance of the default type registered with the container as the type ***t*** with the specified name and where **params** provide any constructor overrides for the **Resolve** calls. Names are case sensitive. |

If you call the **Resolve** method and specify a name as well as the registration type, and there is no mapping registered for that type and name, the container will attempt to create an instance of the type you resolved. If that type is a base class or interface, which cannot be constructed, Unity will raise an exception.

Registration (mapping) names are just strings, so they can contain spaces if required. However, they are case sensitive. For example, the names **Mymapping** and **MyMapping** will refer to two different registration mappings.

You can use the same name for different registrations if they register different object types. The key used to retrieve a registration is effectively a combination of the registered type and the name.

If you register a named type or a named type mapping more than once using the same type and name, only the last registration remains in the container and is applied when you execute the **Resolve** method.

## Using the Resolve Method with Named Registrations

If you need to register multiple mappings for the same type, you can specify a name to differentiate each mapping. Then, to retrieve an object of the appropriate type, you specify the name and the registered type. Typically you will register a type mapping between an interface and a concrete type that implements it, or between a base class and a concrete type that inherits it.

The following examples show how you can retrieve concrete instances of registered objects from a Unity container using the **Resolve** method when you have registered one or more mappings for a type and differentiated them by name. The examples use the run-me methods of the container to register the types it will resolve. For more information about how you can configure Unity with type registrations and mappings, see Configuring Unity.

## Resolving Types Registered as Interfaces

The following code demonstrates how you can register two mappings for the same interface class and have the container return the appropriate object type depending on the type and name you specify in the call to the **Resolve** method. The following example uses the generic overloads of the container methods (though you can also use the non-generic method overloads).

**C#**

```
// Create container and register types
IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType<IMyService, DataService>("Data");
myContainer.RegisterType<IMyService, LoggingService>("Logging");

// Retrieve an instance of each type
IMyService myDataService = myContainer.Resolve<IMyService>("Data");
IMyService myLoggingService = myContainer.Resolve<IMyService>("Logging");
```

**Visual Basic**

```
' Create container and register types
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(Of IMyService, DataService)("Data")
myContainer.RegisterType(Of IMyService, LoggingService)("Logging")

' Retrieve an instance of each type
Dim myDataService As IMyService = myContainer.Resolve(Of IMyService)("Data")
Dim myLoggingService As IMyService = myContainer.Resolve(Of
IMyService)("Logging")
```

## Resolving Types Registered as Base Classes

The following code demonstrates how you can register two mappings for the same base or other class using the overloads of the **RegisterType** and **Resolve** methods that accept type names, and then have the container return the appropriate object type depending on the type and name you specify in the call to the **Resolve** method. This example uses the non-generic overloads of the container methods (though you can also use generic method overloads).

**C#**

```
// Create container and register types
IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType(typeof(MyServiceBase), typeof(DataService), "Data");
myContainer.RegisterType(typeof(MyServiceBase), typeof(LoggingService),
"Logging");

// Retrieve an instance of each type
MyServiceBase myDataService =
(MyServiceBase)myContainer.Resolve(typeof(MyServiceBase), "Data");
MyServiceBase myLoggingService =
(MyServiceBase)myContainer.Resolve(typeof(MyServiceBase), "Logging");
```

**Visual Basic**

```
' Create container and register types
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(GetType(MyServiceBase), GetType(DataService), "Data")
myContainer.RegisterType(GetType(MyServiceBase), GetType(LoggingService),
"Logging")

' Retrieve an instance of each type
Dim myDataService As MyServiceBase = myContainer.Resolve(GetType(MyServiceBase),
"Data")
```

```
Dim myLoggingService As MyServiceBase =
myContainer.Resolve(GetType(MyServiceBase), "Logging")
```

If the target class or object specifies any dependencies of its own, using constructor, property, or method call injection attributes, the instance returned will have these dependent objects injected automatically.

By default, the **RegisterType** method registers a type with a transient lifetime, which means that the container will not hold onto a reference to the objects it creates when you call the **Resolve** method. Each time you call one of these methods, the container generates a new instance of the specified or mapped type. However, you can use lifetime managers to control the creation, lifetime, and disposal of objects, if required.

## Resolving Generic Types by Name

You can also resolve generic types by name, just as you do with non-generic types.

You cannot resolve unbound generic types. When an instance is created by Unity, concrete type arguments replace all of the generic type parameters.

For more information see the "Resolving Generic Types" section in Resolving an Object by Type.

You can specify the parameter that defines the base class or interface type you to resolve as the actual type by using the **Resolve** method generic overload. The following example assumes that you have registered a named mapping between the **IMyService** type and a concrete type named **MyDataService**. The **Resolve** method will return the named instance, Personal, of the **MyDataService** class.

**C#**

```
var myInstance = myContainer.Resolve<IMyService>("Personal");
```

**Visual Basic**

```
Dim myInstance = myContainer.Resolve(Of IMyService)("Personal")
```

## More Information

For more information about the techniques discussed in this topic, see the following:

- Configuring Unity
- Using Injection Attributes
- Understanding Lifetime Managers

## Resolving Generic Types

You resolve generic types in much the same way as you resolve non-generic types. The primary difference is with unbound types. The specification of the type arguments depend on the definition of the mapped type or the type you are resolving:

- If the mapped type or the type you are resolving is a **bound** type, you can only resolve an instance of the type using the defined type arguments. For example, if the mapped type has type arguments of type **string** and **DateTime**, you must specify these in the call to the **Resolve** method.
- If the mapped type or the type you are resolving is an **unbound** type, you can resolve an instance of the type using any types for the type arguments. The target class must be able to process arguments of the type you specify. For example, if one of the type arguments you specify is the type **Boolean**, the class must be able to handle **Boolean** values for that argument and not attempt to parse the value into a **DateTime** instance.

> You cannot resolve unbound generic types. You can register an unbound generic type but you must specify a mapping to a concrete type that Unity can resolve.

The examples in this section use the following aliases and registrations:

**XML**

```
<!-- Generic type aliases -->
<alias alias="IMyClosedInterface" type=
   "TypeMappingsExamples.MyTypes.IMyInterface`2[
    System.String,System.DateTime], TypeMappingsExamples" />
<alias alias="IMyUnboundInterface" type="TypeMappingsExamples.
    MyTypes.IMyInterface`2, TypeMappingsExamples" />
<alias alias="MyClosedClass" type="TypeMappingsExamples.
    MyTypes.MyClass`2[System.String,System.DateTime], TypeMappingsExamples" />
<alias alias="MyUnboundClass" type="TypeMappingsExamples.
    MyTypes.MyClass`2, TypeMappingsExamples" />

<container name="container1">

    <!-- Generic type mappings -->
    <register type="IMyClosedInterface" mapTo="MyClosedClass"
name="Closed(String,DateTime)ToClosed(String,DateTime)" />
    <register type="IMyUnboundInterface" mapTo="MyUnboundClass"
name="UnboundToUnbound" />
    <register type="IMyUnboundInterface" mapTo="MyClosedClass"
name="UnboundToClosed(String,DateTime)" />

</container>
```

The following example resolves an instance using closed generic types.

**C#**

```
// Resolve closed IMyInterface type using type parameters String and DateTime.
var result1 = myContainer.Resolve<IMyInterface<string, DateTime>>(
```

```
                            "Closed(String,DateTime)ToClosed(String,DateTime)");
```

**Visual Basic**

```
' Resolve the closed IMyInterface type using type parameters String and DateTime.
Dim result1 = myContainer.Resolve(Of IMyInterface(Of String, DateTime))( _
                            "Closed(String,DateTime)ToClosed(String,DateTime)")
```

The result is resolved through the mapping between the closed types, and is valid because the combination of type arguments supplied in the call to the **Resolve** method matches those for the mapping of the closed types. Unity returns the following result:

```
'MyClass`2[System.String,System.DateTime]'
```

The following example resolves instances for unbound generic types.

**C#**

```
// Resolve unbound IMyInterface type using type parameters Integer and Decimal.
var result2 = myContainer.Resolve<IMyInterface<int, decimal>>(
                                    "UnboundToUnbound");

// Resolve unbound IMyInterface type using type parameters String and DateTime.
var result3 = myContainer.Resolve<IMyInterface<string, DateTime>>(
                                    "UnboundToClosed(String,DateTime)");
```

**Visual Basic**

```
' Resolve the unbound IMyInterface type using type parameters Integer and
Decimal.
Dim result2 = myContainer.Resolve(Of IMyInterface(Of Integer, Decimal))( _
                                    "UnboundToUnbound")

' Resolve the unbound IMyInterface type using type parameters String and
DateTime.
Dim result3 = myContainer.Resolve(Of IMyInterface(Of String, DateTime))( _
                                    "UnboundToClosed(String,DateTime)")
```

The examples return the following results:

```
Resolving with the mapping 'UnboundToUnbound'
- Unity returns 'MyClass`2[System.Int32,System.Decimal]'

Resolving IMyInterface using the mapping 'UnboundToClosed(String,DateTime)'
- Unity returns 'MyClass`2[System.String,System.DateTime]'
```

The UnboundToUnbound example returns a closed instance. The call to the **Resolve** method specifies the use of the mapping between the unbound types. However, as the call to this method includes the definition of the type arguments, Unity will resolve an instance of the closed type. The types used for the type arguments in the call to the **Resolve** method can be any type that the target class can process.

The UnboundToClosed example call is resolved through the mappings between the unbound interface and a closed type. The call to the **Resolve** method specifies the appropriate types for the closed types that will be returned.

## More Information

For more information about the techniques discussed in this topic, see the following:

- Configuring Unity
- Using Injection Attributes
- Understanding Lifetime Managers

# Resolving All Objects of a Particular Type

When you want to obtain a list of all the registered objects of a specific type, you can use the **ResolveAll** method. The two overloads of this method accept either an interface or a type name, and they return an instance of **IEnumerable** that contains references to all registered objects of that type that are **not** default mappings. The list returned by the **ResolveAll** method contains only named instance registrations. The **ResolveAll** method is useful if you have registered multiple object or interface types using the same type but different names. You can also use the **params** to provide constructor overrides for the **ResolveAll** calls.

## The ResolveAll Method Overloads

The following table shows the overloads of the **ResolveAll** method. You can only request a list of objects using the object type. The API for the Unity container contains both generic and non-generic overloads of this method so that you can use it with languages that do not support the generics syntax.

| Method | Description |
| --- | --- |
| ResolveAll<*T*>( params ResolverOverride[] resolverOverrides) | Returns a list of **IEnumerable<T>**, where **T** is the type registered as a non-default mapping with the container as the type *T and where* ***params*** *provide any constructor overrides for the* **ResolveAll** *calls*. |
| ResolveAll(Type *t, params ResolverOverride[] resolverOverrides*) | Returns a list of **IEnumerable<object>**, where **object** is the type registered as a non-default mapping with the container as the type *t and where* ***params*** *provide any constructor overrides for the* **ResolveAll** *calls*. |

It is important to remember that you must register type mappings using a name in addition to the registration type (which identifies the registration) if you want to be able to retrieve a list of mapped types using the **ResolveAll** method. In other words, you must use overloads of the **RegisterType** and **RegisterInstance** methods that take a name (as a **String**) and the dependency type, or specify the name in your configuration of type mappings. This behavior is by design. The expectation is that you will either use only named mappings or use only default mappings.

If you register a named type or a named type mapping more than once using the same type and name, only the last registration remains in the container and is applied when you execute the **ResolveAll** method. If the container does not contain any named (non-default) mappings for the specified type, it will return **null** (in C#) or **Nothing** (in Visual Basic).

## Using the ResolveAll Method

The following examples show how you can retrieve a list of all registered types for a specific registration type. They use the run-time methods of the container to register the types it will resolve. For more information about how you can configure Unity with type registrations and mappings, see Configuring Unity.

**C#**

```
// Create container and register types using a name for each one
IUnityContainer myContainer = new UnityContainer();
myContainer.RegisterType<IMyService, DefaultService>();
myContainer.RegisterType<IMyService, DataService>("Data");
myContainer.RegisterType<IMyService, LoggingService>("Logging");

// Retrieve a list of non-default types registered for IMyService
// List will only contain the types DataService and LoggingService
IEnumerable<IMyService> serviceList = myContainer.ResolveAll<IMyService>();
```

**Visual Basic**

```
' Create container and register types using a name for each one
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(Of IMyService, DefaultService)()
myContainer.RegisterType(Of IMyService, DataService)("Data")
myContainer.RegisterType(Of IMyService, LoggingService)("Logging")

' Retrieve a list of non-default types registered for IMyService
' List will only contain the types DataService and LoggingService
Dim serviceList As IEnumerable(Of IMyService) = myContainer.ResolveAll(Of
IMyService)
```

Alternatively, you can use the non-generic methods of the container to achieve the same result, except that the return value this time is an **IEnumerable** list of type **Object**.

**C#**

```
// Create container and register types using a name for each one
IUnityContainer myContainer = new UnityContainer();
```

```
myContainer.RegisterType(typeof(MyServiceBase), typeof(DefaultService));
myContainer.RegisterType(typeof(MyServiceBase), typeof(DataService), "Data");
myContainer.RegisterType(typeof(MyServiceBase), typeof(LoggingService),
"Logging");

// Retrieve a list of non-default types registered for MyServiceBase
// List will only contain the types DataService and LoggingService
foreach(Object mapping in myContainer.ResolveAll(typeof(MyServiceBase)))
{
  MyServiceBase myService = (MyServiceBase)mapping;
}
```

**Visual Basic**

```
' Create container and register types using a name for each one
Dim myContainer As IUnityContainer = New UnityContainer()
myContainer.RegisterType(GetType(MyServiceBase), GetType(DefaultService))
myContainer.RegisterType(GetType(MyServiceBase), GetType(DataService), "Data")
myContainer.RegisterType(GetType(MyServiceBase), GetType(LoggingService),
"Logging")

' Retrieve a list of non-default types registered for MyServiceBase
' List will only contain the types DataService and LoggingService
For Each mapping As Object In myContainer.ResolveAll(GetType(MyServiceBase))
  Dim myService As MyServiceBase = CType(mapping, MyServiceBase)
Next
```

## Resolving All Generic Types by Name

You can use the **ResolveAll** method with generic types, just as you do with non-generic types, to retrieve an enumerable list of resolved instances for all of the named mappings for a generic type. However, all of the registrations or mappings must resolve to a type that has the same set of type arguments, and you must specify these type arguments in the call to the **ResolveAll** method.

> Where you resolve a closed type, and there are both unbound and closed registrations in the container that potentially match the type you are resolving, Unity will preferentially use the registration or type mapping defined for the *closed* generic type rather than the *unbound* one.

For more information see Resolving All Objects of a Particular Type.

# Resolving Objects by Using Overrides

The parameter and dependency overrides, **ParameterOverride** and **DependencyOverride,** are **ResolverOverride** implementations that provide support for overriding the registration information for resolving instances of types. When you call the **Resolve** method, these classes enable you to

override values specified when the type was registered, such as by a **RegisterType** or **RegisterInstance** statement. In effect, **RegisterType** supplied values are overridden by **Resolve** supplied values.

Use **ParameterOverride** to override the specified constructor parameter or parameters. The override applies everywhere the parameter appears unless you use **OnType** to constrain the override to a specified type. Since the purpose of overrides is to affect the resolution of dependencies for all relevant created objects, not just the object requested in the call to **Resolve**, unconstrained overrides can produce errors if there are unconstrained **ParameterOverride** parameters that match parameters with the same name but different types on the selected constructors for objects created in a given resolve operation.

Use **PropertyOverride** to override the value of the specified property or properties. The override applies everywhere the property appears unless you use **OnType** to constrain the override to a specified type.

Use **DependencyOverride** to override the value injected whenever there is a dependency of the given type. **DependencyOverride** overrides all instances where the type matches. Both parameter overrides and dependency overrides support generic types and multiple overrides.

> If the overridden object was previously created and is a Singleton, the override is ignored. The lifetime manager takes precedence and Singletons always return the same instance.
>
> The container does not store a reference for the overridden object.

Overrides work with the constructor that is selected for the type, by attribute or configuration. If the constructor to be used is not identified with an attribute or explicit container configuration, then the default behavior is that the constructor with the most parameters will be used.

A parameter and property override never affects what element gets selected. They only control the value of the specified parameter or property. You do not change which constructor is called with an override, and you do not change which properties get set with an override.

> If the property is not set as a dependency through attribute, container API, or configuration file, then the override does nothing.

This topic contains the following sections to explain overrides in more detail:

## Using Parameter Overrides

**ParameterOverride** enables you to pass in values for constructor parameters to override a parameter passed to a given named constructor. Only the parameter value is overridden, not the constructor.

**ParameterOverride** can be used only for constructors.

The following code constrains the override to the **NewCar** instance.

**C#**

```
var anInstance = container.Resolve<ClassCar>(new ParameterOverride("car", "new
car").OnType<NewCar>());
```

**Visual Basic**

```
Dim anInstance = container.Resolve(Of ClassCar)(New ParameterOverride("car", "new
car").OnType(Of NewCar)())
```

If the parameter does not exist, then the override is ignored.

When the parameters you set to override do not match the constructor signature, then the default parameter values set during registration are used.

You can override the constructor parameters but not the constructor.

The two parameters for **ParameterOverride** (**parameterName,** and **parameterValue)**, specify the name of the parameter to override and the value to use for the parameter, respectively. In the following example the value **ExpectedValue** overrides the constructor's value for the constructor's parameter **x** for the type **MyObject**. The **OnType** method enables you to optionally specify a type to constrain the override to.

**C#**

```
// Arbitrary values
const int ConfiguredValue = 15;
const int ExpectedValue = 42;

// Create a container and register type "MyObject"
var container = new UnityContainer()
.RegisterType<MyObject>(new InjectionConstructor(ConfiguredValue));

// Override the constructor parameter "x" value
// Use "ExpectedValue" instead of "ConfiguredValue"
var result = container.Resolve<MyObject>(
    new ParameterOverride("x", ExpectedValue)
        .OnType<MyOtherObject>());
```

**Visual Basic**

```
' Arbitrary values
Const  ConfiguredValue As Integer = 15
Const  ExpectedValue As Integer = 42

' Create a container and register type "MyObject"
Dim container = New UnityContainer().RegisterType(Of MyObject)(New
InjectionConstructor(ConfiguredValue))

' Override the constructor parameter "x" value
' Use "ExpectedValue" instead of "ConfiguredValue"
Dim result = container.Resolve(Of MyObject)( _
    New ParameterOverride("x", ExpectedValue) _
        .OnType(Of MyOtherObject)())
```

Unity also supports overriding multiple parameters and the use of generics.

**ParameterOverrides** is a convenience form of **ParameterOverride** that lets you specify multiple parameter overrides in one statement rather than having to construct multiple objects, one for each override. The following example overrides the two parameters, **y** and **x,** with **ExpectedValue**.

**C#**

```
const int ExpectedValue = 42;
var container = new UnityContainer();
var result = container.Resolve<MyObject>(
    new ParameterOverrides
    {
          { "y", ExpectedValue * 2 },
          { "x", ExpectedValue }
    }
    .OnType<MyOtherObject>());
```

For Visual Basic .NET  you must either use discrete **ParameterOverride** objects instead of the grouped one, or do something like the following:

**Visual Basic**

```
Const  ExpectedValue As Integer = 42
Dim container = New UnityContainer()
Dim overrides as new ParameterOverrides
With overrides
    .Add("y", ExpectedValue * 2)
    .Add("x", ExpectedValue)
End With

Dim result = container.Resolve(Of MyObject)(overrides.OnType(Of MyOtherObject)())
```

This could be useful in cases where your object contains both entities and services. You might wish to change an entity, but the service is fine as is. For example, you could process a series of customers and their information from your database and want to log the process. You have the following class where **IErrorLogger logger** and **IDataAccess db** are services.

**C#**

```
class Customer
{
```

```
    public customer(System.Int32 id, System.String name, IErrorLogger logger,
IDataAccess db)
    {
    }
}
```

**Visual Basic**

```
Class Customer
   Private Function customer(ByVal id As System.Int32, ByVal name As
System.String, _
                             ByVal logger As IErrorLogger, ByVal db As
IDataAccess)
   End Function
End Class
```

If you register the type and specify the **id** and **name** for a customer, say **Sam**, then **container.Resolve<Customer>** will resolve with **Sam** every time. What you really want as you process customers in your database is to resolve a different customer every time you use **container.Resolve<Customer>**.

Using the plural form, **ParameterOverrides,** you can override just the relevant parameters for the constructor. Your object will have the services and you will have a new customer to work with.

**C#**

```
container.Resolve<Customer>(
          new ParameterOverrides
          {
              {"id", userform.id},
              { "name", "Bob"}
          }.OnType<Customer>()
);
```

Visual Basic .NET lacks the dictionary initialization syntax to accomplish the same thing. You must either use discrete **ParameterOverride** objects instead of the grouped one, or do something like the following:

**Visual Basic**

```
Dim overrides as new ParameterOverrides
With overrides
    .Add("id", userform.id)
    .Add("name", "Bob")
End With

Dim result = container.Resolve(Of customer)(overrides.OnType(Of Customer)))
```

## Using Property Overrides

**PropertyOverride** enables you to override the value for a specified property. Only the property value is overridden, not the properties selected. Its behavior is the same as **ParameterOverride**. The use of the **OnType** method enables you to specify a type to constrain the override to.

The following code specifies a value, **overrideValue**, to use to override the injected property value, **defaultObject**.

**C#**

```
container.RegisterType<TargetTypeForInjection>(
    new InjectionProperty("InjectedObject", defaultObject));
var result1 = container.Resolve<TargetTypeForInjection>(
    new PropertyOverride("InjectedObject", overrideValue));
```

**Visual Basic**

```
container.RegisterType(Of TargetTypeForInjection)( _
    New InjectionProperty("InjectedObject", defaultObject))
Dim result1 = container.Resolve(Of TargetTypeForInjection)( _
    New PropertyOverride("InjectedObject", overrideValue))
```

The following code constrains the override to the **TargetType2** instance by invoking **OnType**.

**C#**

```
var result = container.Resolve<TargetTypeForInjection>(
    new PropertyOverride("InjectedObject", overrideObject)
    .OnType<TargetType2>());
```

**Visual Basic**

```
Dim result = container.Resolve(Of TargetTypeForInjection)( _
    New PropertyOverride("InjectedObject", overrideObject) _
    .OnType(Of TargetType2)())
```

> If the property is not set as a dependency through an attribute, the container API, or by a configuration file, then the override has no effect.

**PropertyOverrides** is a convenience form of **PropertyOverride** that lets you specify multiple property overrides in one statement rather than having to construct multiple objects, one for each override. Its behavior is the same as **ParameterOverrides**. The following example overrides the values for three properties of **MyObject** and constrains the overrides to **MyOtherObject** by using **OnType**.

**C#**

```
return container.Resolve<MyObject>(
    new PropertyOverrides
        {
             {"Property1", Value1},
             {"Property2", Value2},
             {"Property3", Value3}
```

```
        }
        .OnType<MyOtherObject>()
);
```

For Visual Basic .NET you must either use discrete **PropertyOverride** objects instead of the grouped one, or do something like the following:

**Visual Basic**

```
Dim overrides as new PropertyOverrides
With overrides
    .Add("Property1", Value1)
    .Add("Property2", Value2)
    .Add("Property3", Value3)
End With

Dim result = container.Resolve(Of MyObject)(overrides.OnType(Of MyOtherObject)())
```

## Using Dependency Overrides

**DependencyOverride** enables you to specify an override of the registered value injected for the specified dependency type, and enables you to pass in a different object that will be provided by type. The use of the **OnType** method enables you to constrain the override to a specified type with the dependency instead of being propagated throughout all types with the dependency.

> **DependencyOverride** will be in effect everywhere the specified dependency type shows up and there is an exact type match.

The two parameters for **DependencyOverride**, **typeToConstruct** and **dependencyValue**, specify the **Type** of the dependency to construct and the value to use for the parameter respectively. Overriding named dependencies is not supported. You should not try to override a named dependency if there is a default registration for that named dependency.

The values for dependency overrides are interpreted using the same rules used for values in constructor, property, and method injection. These rules are described in Registering Injected Parameter and Property Values. So for dependency overrides if you wish to supply an instance of the **Type** class to be used as a literal argument you must inject an **InjectionParameter** with the **Type** object, otherwise the override type is resolved in the container as an instance of the supplied type (see rule #2 in Registering Injection for Parameters, Properties, and Methods using InjectionMembers for details).

**C#**

```
class Cars
{
  public Cars(Type typeObject)
  { ... }
}
```

**Visual Basic**

```
Class Cars
   Public Sub New(ByVal typeObject As Type)
```

```
    ...
   End Sub
End Class
```

In the following example the override will apply to parameters of type **Type**, replacing the container's configuration with a resolved instance of type **int**.

**C#**

```
container.Resolve<Cars>(new DependencyOverride(typeof(Type), typeof(int)));
```

**Visual Basic**

```
container.Resolve(Of Cars)(New DependencyOverride(GetType(Type),
GetType(Integer)))
```

In the following example, the supplied instance of the **Type** class will be used as a literal argument and your intent is explicit.

**C#**

```
container.Resolve<Cars>(new DependencyOverride(typeof(Type),
                        new InjectionParameter(typeof(int))));
```

**Visual Basic**

```
container.Resolve(Of Cars)(New DependencyOverride(GetType(Type), _
                           New InjectionParameter(GetType(Integer))))
```

You cannot use the **GenericParameters** as values for overrides since they inherit from **InjectionParameterValue** and do not make sense when resolving. They only make sense when configuring an unbound generic type with **RegisterType.**

The following example creates an instance of **DependencyOverride** to override the given type with the given value. **FirstObject** is the type to override and **overrideValue** is the value used for the override.

**C#**

```
var container = new UnityContainer()
    .RegisterType<ObjectDependentOnFirstObject>(
        new InjectionProperty("OtherObject"))
    .RegisterType<FirstObject>(new InjectionConstructor());

// Use an arbitrary value for the overrideValue object
var overrideValue = new FirstObject(15);

var result = container.Resolve<ObjectDependentOnFirstObject>(
    new DependencyOverride<FirstObject>(overrideValue));
```

**Visual Basic**

```
Dim container = New UnityContainer() _
    .RegisterType(Of ObjectDependentOnFirstObject)( _
        New InjectionProperty("OtherObject")) _
    .RegisterType(Of FirstObject)(New InjectionConstructor())
```

```
' Use an arbitrary value for the overrideValue object
Dim overrideValue = New FirstObject(15)

Dim result = container.Resolve(Of ObjectDependentOnFirstObject)( _
    New DependencyOverride(Of FirstObject)(overrideValue))
```

Where **FirstObject** and **ObjectDependentOnFirstObject** are defined as follows:

**C#**

```
public class FirstObject
{
    public FirstObject()
    { }

    public FirstObject(int x)
    {
        X = x;
    }

    public int X { get; private set; }
}

public class ObjectDependentOnFirstObject
{
    public FirstObject TestObject { get; set; }

    public ObjectDependentOnFirstObject(FirstObject testObject)
    {
        TestObject = testObject;
    }

    public FirstObject OtherObject { get; set; }
}
```

**Visual Basic**

```
Public Class FirstObject

    Public Sub New()
    End Sub

    Public Sub New(ByVal x__1 As Integer)
        X = x__1
    End Sub

    Private _X As Integer
    Public Property X() As Integer
        Get
            Return _X
        End Get
        Private Set(ByVal value As Integer)
            _X = value
        End Set
```

```
    End Property

End Class

Public Class ObjectDependentOnFirstObject

    Private _TestObject As FirstObject
    Public Property TestObject() As FirstObject
        Get
            Return _TestObject
        End Get
        Set(ByVal value As FirstObject)
            _TestObject = value
        End Set
    End Property

    Public Sub New(ByVal testObject__1 As FirstObject)
        TestObject = testObject__1
    End Sub

    Private _OtherTestObject As FirstObject
    Public Property OtherObject() As FirstObject
        Get
            Return _OtherObject
        End Get
        Set(ByVal value As FirstObject)
            _OtherObject = value
        End Set
    End Property

End Class
```

- **DependencyOverrides** is a convenience form of **DependencyOverride** that lets you specify multiple dependency overrides in one shot rather than having to construct multiple objects. You must specify each type to be overridden and the value to use for the override. The following example will override **FirstObject** and **SecondObject** with **overrideValue1** and **overrideValue2**, respectively.

**C#**

```
var container = new UnityContainer()
    .RegisterType<ObjectDependentOnFirstObject>(
        new InjectionProperty("OtherObject"))
    .RegisterType<FirstObject>(new InjectionConstructor());

// Use an arbitrary value for the overrideValue objects
var overrideValue1 = new FirstObject(15);
var overrideValue2 = new FirstObject(25);

var result = container.Resolve<ObjectDependentOnFirstObject>(
    new DependencyOverrides
    {
        {typeof(FirstObject),overrideValue1},
```

```
        {typeof(SecondObject),overrideValue2},
    }
);
```

For Visual Basic .NET you must either use discrete **DependencyOverride** objects instead of the grouped one, or do something like the following:

**Visual Basic**

```
Dim container = New UnityContainer() _
    .RegisterType(Of ObjectDependentOnFirstObject)( _
        New InjectionProperty("OtherObject")) _
    .RegisterType(Of FirstObject)(New InjectionConstructor())

' Use an arbitrary value for the overrideValue objects
Dim overrideValue1 = New FirstObject(15)
Dim overrideValue2 = New FirstObject(25)
Dim overrides as new DependencyOverrides
With overrides
    .Add("FirstObject ", overrideValue1)
    .Add("SecondObject ", overrideValue2)
End With

Dim result = container.Resolve(Of ObjectDependentOnFirstObject)(overrides)
```

- **DependencyOverride<T>** is a convenience overload of **DependencyOverride** that lets you specify the dependency type using generic syntax. The following example uses a generic version of **DependencyOverride** that specifies the type the constructor is for as a generic parameter, type **T** (where **T** is a generic for the **Type**).

**C#**

```
var result = container.Resolve<ObjectDependentOnT>(
    new DependencyOverride<T>(overrideValue));
```

**Visual Basic**

```
Dim result = container.Resolve(Of ObjectDependentOnT)( _
    New DependencyOverride(Of T)(overrideValue))
```

## More information

For more information about the techniques discussed in this topic, see the following:

- Resolving Objects
- Configuring Unity
- Using Injection Attributes

# Deferring the Resolution of Objects

Unity provides a technique to facilitate holding a reference to an object you need, but do not want to construct right away. You wish to defer resolution of the object. Instead of creating a factory for the type and injecting the factory into your class, then using it to create the type you want you can use the .NET standard type **Func<T>** (C#) or **Func(Of T)** (Visual Basic) with the **Resolve** method. This returns a delegate that, when invoked, calls into the container and returns an instance of the specified type (in this case, **T**).

You can even create a delegate in this way without creating a registration or mapping for the specified type in the container if you wish. Because the resolve action only takes place when you invoke the delegate, subsequent registrations added to the container are available when the target object is resolved. This means that you can manipulate the registrations and mappings in the container at any point before you resolve the target object (although you can obviously register the type before you create the delegate if you prefer).

For example, you can create a delegate for a component named MyClass, and then register a mapping for it and perform deferred resolution when required using the following code.

**C#**

```
// Create a Unity container
IUnityContainer myContainer = new UnityContainer();

// Create a delegate for the IMyClass interface type
var resolver = myContainer.Resolve<Func<IMyClass>>();

// ... other code here...

// Register a mapping for the IMyClass interface to the MyClass type
myContainer.RegisterType<IMyClass, MyClass>();

// Resolve the mapped target object
IMyClass myClassInstance = resolver();
```

**Visual Basic**

```
' Create a Unity container
Dim myContainer As IUnityContainer = New UnityContainer()

' Create a delegate for the IMyClass interface type
Dim resolver = myContainer.Resolve(Of Func(Of IMyClass))()

' ... other code here...

' Register a mapping for the IMyClass interface to the MyClass type
myContainer.RegisterType(Of IMyClass, MyClass)()

' Resolve the mapped target object
Dim myClassInstance As IMyClass = resolver()
```

You can use this approach when you resolve the type using the **Resolve** method, or you can specify the delegate when you configure constructor, property setter, or method call injection. You can also use named (non-default) registrations by including the registration name in the call to the **Resolve** method and the **RegisterType** method, just as you would when using these methods for non-deferred resolution.

In addition, you can use this feature to perform deferred resolution of multiple named registrations, as an alternative to using the ResolveAll method. For example, if you have multiple named registrations for the **IMyClass** interface to suitable concrete types, you can obtain a collection of the resolved types. The following code illustrates this.

**C#**

```
// Create a Unity container
IUnityContainer myContainer = new UnityContainer();

// Create an IEnumerable resolver for the IMyClass interface type
var resolver = myContainer.Resolve<Func<IEnumerable<IMyClass>>>();

// ... other code here...

// Register mappings for the IMyClass interface to appropriate concrete types
myContainer.RegisterType<IMyClass, FirstClass>("First");
myContainer.RegisterType<IMyClass, SecondClass>("Second");
myContainer.RegisterType<IMyClass, ThidClass>("Third");

// Resolve a collection of the mapped target objects
IEnumerable<IMyClass> myClassInstances = resolver();
```

**Visual Basic**

```
' Create a Unity container
Dim myContainer As IUnityContainer = New UnityContainer()

' Create an IEnumerable resolver for the IMyClass interface type
Dim resolver = myContainer.Resolve(Of Func(Of IEnumerable(Of IMyClass)))()

' ... other code here...

' Register mappings for the IMyClass interface to appropriate concrete types
myContainer.RegisterType(Of IMyClass, FirstClass)("First")
myContainer.RegisterType(Of IMyClass, SecondClass)("Second")
myContainer.RegisterType(Of IMyClass, ThidClass)("Third")

' Resolve a collection of the mapped target objects
Dim myClassInstances As IEnumerable(Of IMyClass) = resolver()
```

You can also use the deferred resolver to resolve instance registrations. For example, the following code shows how you can resolve an IEnumerable collection of string values.

**C#**

```
// Create a Unity container
IUnityContainer myContainer = new UnityContainer();

// Create an IEnumerable resolver for string instance registrations
var resolver = myContainer.Resolve<Func<IEnumerable<string>>>();

// ... other code here...

// Register mappings for the IMyClass interface to appropriate concrete types
```

```
myContainer.RegisterInstance("one", "FirstString");
myContainer.RegisterInstance("two", "SecondString");
myContainer.RegisterInstance("three", "ThirdString");

// Resolve a collection of the strings
IEnumerable<string> myStringInstances = resolver();
```

**Visual Basic**

```
' Create a Unity container
Dim myContainer As IUnityContainer = New UnityContainer()

' Create an IEnumerable resolver for string instance registrations
Dim resolver = myContainer.Resolve(Of Func(Of IEnumerable(Of String)))()

' ... other code here...

' Register mappings for the IMyClass interface to appropriate concrete types
myContainer.RegisterInstance("one", "FirstString")
myContainer.RegisterInstance("two", "SecondString")
myContainer.RegisterInstance("three", "ThirdString")

' Resolve a collection of the strings
Dim myStringInstances As IEnumerable(Of String) = resolver()
```

# Retrieving Container Registration Information

You can retrieve a list of registrations from a container, and check if a specific registration is in the container.

This topic contains the following sections:



## Viewing the Container Registrations and Mappings

The Unity container exposes the **Registrations** property which returns an **IEnumerable** list of the registrations within that container. Each registration is an instance of the **ContainerRegistration** class, which exposes information such as the registered type, the registration name (if any), the mapped type (if any), and the lifetime manager that the registration uses.

The following example uses this feature to display the contents of a container. It queries the **Count()** extension method for **IEnumerable<T>**, and then iterates through it displaying a list of the registrations and mappings.

**C#**

```
void DisplayContainerRegistrations(IUnityContainer theContainer)
{
    string regName, regType, mapTo, lifetime;
    Console.WriteLine("Container has {0} Registrations:",
            theContainer.Registrations.Count());
    foreach (ContainerRegistration item in theContainer.Registrations)
    {
    regType = item.RegisteredType.Name;
    mapTo = item.MappedToType.Name;
    regName = item.Name ?? "[default]";
    lifetime = item.LifetimeManagerType.Name;
    if (mapTo != regType)
    {
        mapTo = " -> " + mapTo;
    }
    else
    {
        mapTo = string.Empty;
    }
    lifetime = lifetime.Substring(0, lifetime.Length - "LifetimeManager".Length);
    Console.WriteLine("+ {0}{1}  '{2}'  {3}", regType, mapTo, regName, lifetime);
    }
}
```

**Visual Basic**

```
Sub DisplayContainerRegistrations(ByVal theContainer As IUnityContainer)
    Dim regName As String, regType As String, _
            mapTo As String, lifetime As String
    Console.WriteLine("Container has {0} Registrations:", _
            theContainer.Registrations.Count())
    For Each item As ContainerRegistration In theContainer.Registrations
        regType = item.RegisteredType.Name
        mapTo = item.MappedToType.Name
        regName = If(item.Name, "[default]")
        lifetime = item.LifetimeManagerType.Name
        If mapTo <> regType Then
            mapTo = " -> " & mapTo
        Else
            mapTo = String.Empty
        End If
        lifetime = lifetime.Substring( _
                0, lifetime.Length - "LifetimeManager".Length)
        Console.WriteLine("+ {0}{1} '{2}' {3}", _
                regType, mapTo, regName, lifetime)
    Next
End Sub
```

The **Name** property of a **ContainerRegistration** instance returns **null** (C#) or **Nothing** (Visual Basic) for default (unnamed) registrations. The **MappedToType** property has the same value as the **RegisteredType** when this is a non-mapped concrete type registration (rather than a base class or interface mapped to a concrete type).

## Checking for the Existence of a Specific Registration

You can also use the methods of the container to check if a specific registration or mapping exists. The **IsRegistered** method takes a container, a type, and optionally a registration name, and returns **True** or **False**. For example, to check if there is a registration in the container for the type **MyEmailService**, you would use the following code.

**C#**

```
IUnityContainer container = new UnityContainer();
bool result = container.IsRegistered<MyEmailService>();
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer()
Dim result As Boolean = container.IsRegistered(Of MyEmailService)()
```

This will return **True** if there is a default (unnamed) registration for the **MyEmailService** type. If you want to check for the presence of a named registration, you simply specify the name in the call to the **IsRegistered** method, as shown in the following example.

**C#**

```
bool result = IsRegistered<MyEmailService>("EMailHandler");
```

**Visual Basic**

```
Dim result As Boolean = IsRegistered(Of MyEmailService)("EMailHandler")
```

# Understanding Lifetime Managers

The Unity container manages the creation and resolution of objects based on a lifetime you specify when you register the type of an existing object, and uses the default lifetime if you do not specify a lifetime manager for your type registration.

When you register a type in configuration, or by using the **RegisterType** method, the default behavior is for the container to use a transient lifetime manager. It creates a new instance of the registered, mapped, or requested type each time you call the **Resolve** or **ResolveAll** method or when the dependency mechanism injects instances into other classes. The container does not store a reference to the object. However, when you want nontransient behavior (such as a singleton) for objects Unity creates, the container must store a reference to these objects. It must also take over management of the lifetime of these objects.

Unity uses specific types that inherit from the **LifetimeManager** base class (collectively referred to as lifetime managers) to control how it stores references to object instances and how the container disposes of these instances.

When you register an existing object using the **RegisterInstance** method, the default behavior is for the container to take over management of the lifetime of the object you pass to this method using the **ContainerControlledLifetimeManager**. This means that at the end of the container lifetime, the existing object is disposed. You can also use this lifetime manager when defining registrations in

configuration, or when using the **RegisterType** method, to specify that Unity should manage the object as a singleton instance.

Using a non-default lifetime manager with **RegisterInstance** will result in different behaviors, depending on the context of the requests.

- **Resolve** requests in the same context where the **RegisterInstance** call was made, such as the same thread if using a per-thread manager, or the same parent container when using the hierarchical one, will return the registered instances.
- **Resolve** requests in other contexts, such as a different thread if using a per-thread manager, or a child container when using the hierarchical lifetime manager, will result in a new instance being created by the container and it will be made the singleton for that context. The creation of an instance under these circumstances could fail if the container cannot resolve the instance, for example if you registered an instance for an interface with no mappings to a matching class.

For information about using lifetime managers with the **RegisterType** and **RegisterInstance** methods, see Registering Types and Type Mappings and Creating Instance Registrations in the Run-Time Configuration section of this documentation. For information about specifying the lifetime of objects at design time, see Specifying Types in the Configuration File, and The <instance> Element in the Design-Time Configuration section of this documentation.

## Unity Built-In Lifetime Managers

Unity includes six lifetime managers that you can use directly in your code, but you can create your own lifetime managers to implement specific lifetime scenarios. Unity includes the following lifetime managers:

- **TransientLifetimeManager**. For this lifetime manager Unity creates and returns a new instance of the requested type for each call to the **Resolve** or **ResolveAll** method. This lifetime manager is used by default for all types registered using the **RegisterType**, method unless you specify a different lifetime manager.
- **ContainerControlledLifetimeManager** which registers an existing object as a singleton instance. For this lifetime manager Unity returns the same instance of the registered type or object each time you call the **Resolve** or **ResolveAll** method or when the dependency mechanism injects instances into other classes. This lifetime manager effectively implements a singleton behavior for objects. Unity uses this lifetime manager by default for the **RegisterInstance** method if you do not specify a different lifetime manager. If you want singleton behavior for an object that Unity will create when you specify a type mapping in configuration or when you use the **RegisterType** method, you must explicitly specify this lifetime manager.

  If you registered a type mapping using configuration or using the **RegisterType** method, Unity creates a new instance of the registered type during the first call to the **Resolve** or **ResolveAll** method or when the dependency mechanism injects instances into other classes. Subsequent requests return the same instance.

If you registered an existing instance of an object using the **RegisterInstance** method, the container returns the same instance for all calls to **Resolve** or **ResolveAll** or when the dependency mechanism injects instances into other classes, provided that one of the following is true:

- You have specified a container-controlled lifetime manager
- You have used the default lifetime manager
- You are resolving in the same context in which you registered the instance when using a different lifetime manager.

Once you have a reference to the proper container, call the **RegisterInstance** method of that container to register the existing object. Specify as the registration type an interface that the object implements, an object type from which the target object inherits, or the concrete type of the object.

The following example creates a named (non-default) mapping by specifying the name, Email and uses the default lifetime.

**C#**

```
myContainer.RegisterInstance<EmailService>("Email", myEmailService);
```

**Visual Basic**

```
myContainer.RegisterInstance(Of EmailService)("Email", myEmailService)
```

When the container is disposed, it calls the **Dispose** method of the object and allows it to be garbage collected. Therefore, you must ensure that your code does not maintain a reference to the object.

- **HierarchicalLifetimeManager.** For this lifetime manager, as for the **ContainerControlledLifetimeManager,** Unity returns the same instance of the registered type or object each time you call the **Resolve** or **ResolveAll** method or when the dependency mechanism injects instances into other classes. The distinction is that when there are child containers, each child resolves its own instance of the object and does not share one with the parent. When resolving in the parent, the behavior is like a container controlled lifetime; when resolving the parent and the child you have different instances with each acting as a container-controlled lifetime. If you have multiple children, each will resolve its own instance.

**C#**

```
IUnityContainer Parent = new UnityContainer();
IUnityContainer child = Parent.CreateChildContainer();

MyType newInstance = new MyType();
Parent.RegisterInstance<MyType>(newInstance, new
HierarchicalLifetimeManager());
```

**Visual Basic**

```
Dim Parent As IUnityContainer = New UnityContainer()
Dim child As IUnityContainer = Parent.CreateChildContainer()
```

```
Dim newInstance As New MyType()
Parent.RegisterInstance(Of MyType)(newInstance, New
HierarchicalLifetimeManager())
```

- **PerResolveLifetimeManager**. For this lifetime manager the behavior is like a **TransientLifetimeManager**, but also provides a signal to the default build plan, marking the type so that instances are reused across the build-up object graph. In the case of recursion, the singleton behavior applies where the object has been registered with the **PerResolveLifetimeManager**. The following example uses the **PerResolveLifetimeManager**.

**C#**

```
public void ViewIsReusedAcrossGraph()
{
    var container = new UnityContainer()
        .RegisterType<IPresenter, MockPresenter>()
        .RegisterType<IView, View>(new PerResolveLifetimeManager());
    var view = container.Resolve<IView>();
    var realPresenter = (MockPresenter) view.Presenter;
}
```

**Visual Basic**

```
Public Sub ViewIsReusedAcrossGraph()
    Dim container = New UnityContainer() _
        .RegisterType(Of IPresenter, MockPresenter)() _
        .RegisterType(Of IView, View)(New PerResolveLifetimeManager())
    Dim view = container.Resolve(Of IView)()
    Dim realPresenter = DirectCast(view.Presenter, MockPresenter)
End Sub
```

The following small object graph illustrates the per-build behavior for the example. **MockPresenter** inherits from **IPresenter** and contains a **View** and **MockPresenter** object. **IView** contains a **Presenter** object and **View** class contains a **Presenter** dependency. **View** is reused across the graph per resolve call because **View** is registered with a **PerResolveLifetimeManager**.

**C#**

```
public interface IPresenter
{ }

public class MockPresenter : IPresenter
{
    public IView View { get; set; }

    public MockPresenter(IView view)
    {
        View = view;
    }
}

public interface IView
{
    IPresenter Presenter { get; set; }
```

```
}

public class View : IView
{
    [Dependency]
    public IPresenter Presenter { get; set; }
}
```

**Visual Basic**

```
Public Interface IPresenter
End Interface

Public Class MockPresenter
    Implements IPresenter

    Private _View As IView
    Public Property View() As IView
        Get
            Return _View
        End Get
        Set(ByVal value As IView)
            _View = value
        End Set
    End Property

    Public Sub New(ByVal view__1 As IView)
        View = view__1
    End Sub
End Class

Public Interface IView
    Property Presenter() As IPresenter
End Interface

Public Class View
    Implements IView

    Private _Presenter As IPresenter
    <Dependency()> _
    Public Property Presenter() As IPresenter
        Get
            Return _Presenter
        End Get
        Set(ByVal value As IPresenter)
            _Presenter = value
        End Set
    End Property

End Class
```

- **PerThreadLifetimeManager**. For this lifetime manager Unity returns, on a per-thread basis, the same instance of the registered type or object each time you call the **Resolve** or **ResolveAll** method or when the dependency mechanism injects instances into other classes.

This lifetime manager effectively implements a singleton behavior for objects on a per-thread basis. **PerThreadLifetimeManager** returns different objects from the container for each thread.

If you registered a type mapping using configuration or using the **RegisterType** method, Unity creates a new instance of the registered type the first time the type is resolved in a specified thread, either to answer a call to the **Resolve** or **ResolveAll** method for the registered type or to fulfill a dependency while resolving a different type. Subsequent resolutions on the same thread return the same instance.

**PerThreadLifetimeManager** returns the object desired or permits the container to create a new instance if no such object is currently stored for the current thread. A new instance is also created if called on a thread other than the one that set the value.

> When using the **PerThreadLifetimeManager**, it is recommended that you use **RegisterType** and do not use **RegisterInstance** to register an object.
>
> When you register an instance with the **PerThreadLifetimeManager**, the instance is registered only for the executing thread. Calls to **Resolve** on other threads result in per-thread singletons for container-built instances. If you request a type registered with the **PerThreadLifetimeManager** in any thread other than the thread it was registered for, the lifetime container for that object finds that there is no registered instance for that thread and, therefore, permits the container to build a new instance for that thread.
>
> The result is that for threads other than the one registering the instance, the behavior is the same as if you registered the container lifetime with **RegisterType**.

This lifetime manager does not dispose the instances it holds. The thread object is reclaimed by garbage collection when the thread is disposed, but note that the object is not disposed in the sense that the **Dispose** method is not invoked.

- **ExternallyControlledLifetimeManager**. The **ExternallyControlledLifetimeManager** class provides generic support for externally managed lifetimes. This lifetime manager allows you to register type mappings and existing objects with the container so that it maintains only a weak reference to the objects it creates when you call the **Resolve** or **ResolveAll** method or when the dependency mechanism injects instances into other classes based on attributes or constructor parameters within that class. This allows other code to maintain the object in memory or dispose it and enables you to maintain control of the lifetime of existing objects or allow some other mechanism to control the lifetime. Using the **ExternallyControlledLifetimeManager** enables you to create your own custom lifetime managers for specific scenarios. Unity returns the same instance of the registered type or object each time you call the **Resolve** or **ResolveAll** method or when the dependency mechanism injects instances into other classes. However, since the container does not hold onto a strong reference to the object after it creates it, the garbage collector can dispose of the object if no other code is holding a strong reference to it.

---

> Using the **RegisterInstance** method to register an existing object results in the same behavior as if you just registered the lifetime container with **RegisterType**. Therefore, it is recommended that

you do not use the **RegisterInstance** method to register an existing object when using the non-default lifetime managers except for the thread in which the **RegisterInstance** was invoked.

### More Information

- Creating Lifetime Managers.
- "Using a Lifetime Manager with the RegisterInstance Method" in Creating Instance Registrations
- "Using a Lifetime Manager with the RegisterType Method" in Registering Types and Type Mappings

# Using BuildUp to Wire Up Objects Not Created by the Container

Unity exposes a method named **BuildUp** that you can use to pass existing object instances through the container in order to apply dependency injection to that object. This is an alternative to resolving the object using any of the other techniques available with Unity. However, remember that the **BuildUp** method cannot inject dependent objects into constructor parameters, because the object has already been created; it is not created by Unity.

The **BuildUp** method is useful when you do not have control of the construction of an instance, but you still want property or method call injection performed. For example, ASP.NET pages, Windows Communication Foundation (WCF) applications, and XAML code often create instances of objects and pass a reference to your code. The **BuildUp** method will usually return the original object after passing it through the container, although container extensions may add other features that cause the method to return a different object that is type-compatible with the existing object. For example, an injection strategy may create and return a proxy for an object or a derived object instead of the actual object.

If you have created or added extensions to the Unity container, these extensions can access and use a name that you specify when you execute the **BuildUp** method. This allows the extensions to change their behavior, depending on the value you specify. For example, they may use the name to control how dependencies are resolved or to control features such as event wiring or interception. The actual behavior depends on the individual extension.

### The BuildUp Method Overloads

The following table describes the overloads of the **BuildUp** method. The API for the Unity container contains both generic and non-generic overloads of these methods so that you can use them with languages that do not support the generics syntax.

| Method | Description BuildUp |
|---|---|
| BuildUp<*T*>(T *existing*) | Passes the existing object of type *T* through the container and performs all configured injection upon it. |
| BuildUp<*T*>(T *existing*, string *name*) | Passes the existing object of type *T* with the specified *name* through the container and performs all configured injection upon it. |
| BuildUp(Type *t*, object *existing*) | Passes the existing object of type *t* through the container and performs all configured injection upon it. |
| BuildUp(Type *t*, object *existing*, string *name*) | Passes the existing object of type *t* with the specified *name* through the container and performs all configured injection upon it. |

The **UnityContainer** class also exposes the **Teardown** method, which theoretically would reverse the build-up process. However, the base implementation of this method does nothing. The implementation exists so that container extensions can execute their own custom overrides of this method if required.

## Using the BuildUp Method

The following example shows how you can use the **BuildUp** method to apply dependency injection to existing object instances named **myDataService** and **myLoggingService**, which implement the interface **IMyService**. The examples use both the generic and the non-generic overloads of the container methods.

**C#**

```
IMyService myDataService = new DataService();
IMyService myLoggingService = new LoggingService();
IMyService builtupDataService = myContainer.BuildUp<IMyService>( myDataService);
IMyService builtupLoggingService
  = (IMyService)myContainer.BuildUp(typeof(IMyService), myLoggingService);
```

**Visual Basic**

```
Dim myDataService As IMyService = New DataService()
Dim myLoggingService As IMyService = New LoggingService()
Dim builtupDataService As IMyService = myContainer.BuildUp(Of
IMyService)(myDataService)
Dim builtupLoggingService As IMyService _
  = myContainer.BuildUp(GetType(IMyService), myLoggingService)
```

If you have created or added extensions to the Unity container, these extensions can access and use a name that you specify when you execute the **BuildUp** method. This allows the extensions to change their behavior depending on the value you specify. For example, they may use the name to control how dependencies are resolved or to control features such as event wiring or interception. The actual behavior depends on the individual extension.

The following code shows how you can execute the **BuildUp** method and provide a name. It uses both the generic and the non-generic overloads of the container methods.

**C#**

```
IMyService myDataService = new DataService();
IMyService myLoggingService = new LoggingService();
IMyService builtupDataService
  = myContainer.BuildUp<IMyService>(myDataService, "Data");
IMyService builtupLoggingService
  = (IMyService)myContainer.BuildUp(typeof(IMyService), myLoggingService,
"Logging");
```

**Visual Basic**

```
Dim myDataService As IMyService = New DataService()
Dim myLoggingService As IMyService = New LoggingService()
Dim builtupDataService As IMyService _
  = myContainer.BuildUp(Of IMyService)(myDataService, "Data")
Dim builtupLoggingService As IMyService _
  = myContainer.BuildUp(GetType(IMyService), myLoggingService, "Logging")
```

### More Information

For more information about the techniques discussed in this scenario, see the following topics:

- Configuring Unity
- Annotating Objects for Constructor Injection
- Annotating Objects for Property (Setter) Injection

# Using Injection Attributes

One of the most useful and powerful techniques when using Unity is to take advantage of dependency injection for the parameters of class constructors and methods, and for the values of properties. This approach allows you to resolve and populate the entire hierarchy of objects used in your application based on type registrations and mappings defined in the container, with the subsequent advantages this offers.

You can specify constructor, property, and method call injection information in configuration or by adding registrations to the container at run time. You can also apply attributes to members of your classes. When you resolve these classes through the container, Unity will generate instances of the dependent objects and wire up the target class with these instances.

Unity performs constructor injection automatically on resolved classes, choosing the most complex constructor and populating any parameters for which you do not provide values when it constructs the object. You can also specify which constructor Unity should use to construct the object. For more information, see Annotating Objects for Constructor Injection.

Property and method call injection do not occur automatically unless you have registered injection types in the container at design time or run time. If you have not registered injection types in the container, you can add attributes to the members of your resolved class to force injection of

dependent objects when the target class is resolved. For more information, see Annotating Objects for Property (Setter) Injection and Annotating Objects for Method Call Injection.

For information about registering injection types in the configuration at design time or run time, see Configuring Unity.

# Annotating Objects for Constructor Injection

Unity supports automatic dependency injection for class constructors. You can use the Unity container to generate instances of dependent objects and wire up the target class with these instances. This topic explains how to use both the automatic constructor injection mechanism and an attribute applied to the constructor of a class to define the dependency injection requirements of that class. The attribute can also specify parameters that the constructor will pass to the dependent object that the container generates.

To perform injection of dependent classes into objects you create through the Unity container, you can use the following techniques:

- Single Constructor Automatic Injection. With this technique, you allow the Unity container to satisfy any constructor dependencies defined in parameters of the constructor automatically. You use this technique when there is a single constructor in the target class.
- Specifying Named Type Mappings. With this technique, you specify named mappings for dependencies in the parameters of a class constructor. Named mappings allow you to specify more than one mapping for an interface or base class, or for a type registration.
- Multiple Constructor Injection Using an Attribute. With this technique, you apply attributes to the class constructor(s) that specify the dependencies. You use this technique when there is more than one constructor in the target class.

Constructor injection is a form of mandatory injection of dependent objects, as long as developers use the Unity container to generate the target object. The dependent object instance is generated when the Unity container creates an instance of the target class using the constructor. For more information, see Notes on Using Constructor Injection.

## Single Constructor Automatic Injection

For automatic constructor injection, you simply specify as parameters of the constructor the dependent object types. You can specify the concrete type, or specify an interface or base class for which the Unity container contains a registered mapping.

**To use automatic single-constructor injection to create dependent objects**

1. Define a constructor in the target class that takes as a parameter the concrete type of the dependent class. For example, the following code shows a target class named **MyObject** containing a constructor that has a dependency on a class named **MyDependentClass**.

**C#**

```
public class MyObject
{
  public MyObject(MyDependentClass myInstance)
  {
    // work with the dependent instance
    myInstance.SomeProperty = "SomeValue";
    // or assign it to a class-level variable
  }
}
```

**Visual Basic**

```
Public Class MyObject
  Public Sub New(myInstance As MyDependentClass)
    ' work with the dependent instance
    myInstance.SomeProperty = "SomeValue"
    ' or assign it to a class-level variable
  End Sub
End Class
```

2. In your run-time code, use the **Resolve** method of the container to create an instance of the target class. The Unity container will instantiate the dependent concrete class and inject it into the target class. For example, the following code shows how you can instantiate the example target class named **MyObject** containing a constructor that has a dependency on a class named **MyDependentClass**.

**C#**

```
IUnityContainer uContainer = new UnityContainer();
MyObject myInstance = uContainer.Resolve<MyObject>();
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer()
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()
```

3. Alternatively, you can define a target class that contains more than one dependency defined in constructor parameters. The Unity container will instantiate and inject an instance of each one. For example, the following code shows a target class named **MyObject** containing a constructor that has dependencies on two classes named **DependentClassA** and **DependentClassB**.

**C#**

```
public class MyObject
{
  public MyObject(DependentClassA depA, DependentClassB depB)
  {
    // work with the dependent instances
    depA.SomeClassAProperty = "SomeValue";
    depB.SomeClassBProperty = "AnotherValue";
```

```
    // or assign them to class-level variables
  }
}
```

**Visual Basic**

```
Public Class MyObject
  Public Sub New(depA As DependentClassA, depB As DependentClassB)
    ' work with the dependent instance
    depA.SomeClassAProperty = "SomeValue"
    depB.SomeClassBProperty = "AnotherValue"
    ' or assign them to class-level variables
  End Sub
End Class
```

4. In your run-time code, use the **Resolve** method of the container to create an instance of the target class. The Unity container will create an instance of each of the dependent concrete classes and inject them into the target class. For example, the following code shows how you can instantiate the example target class named **MyObject** containing a constructor that has constructor dependencies.

**C#**

```
IUnityContainer uContainer = new UnityContainer();
MyObject myInstance = uContainer.Resolve<MyObject>();
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer()
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()
```

5. In addition to using concrete types as parameters of the target object constructor, you can use interfaces or base class types and then register mappings in the Unity container to translate these types into the correct concrete types. Define a constructor in the target class that takes as parameters the interface or base types of the dependent class. For example, the following code shows a target class named **MyObject** containing a constructor that has a dependency on a class that implements the interface named **IMyInterface** and a class that inherits from **MyBaseClass**.

**C#**

```
public class MyObject
{
  public MyObject(IMyInterface interfaceObj, MyBaseClass baseObj)
  {
    // work with the concrete dependent instances
    // or assign them to class-level variables
  }
}
```

**Visual Basic**

```
Public Class MyObject
  Public Sub New(interfaceObj As IMyInterface, baseObj As MyBaseClass)
    ' work with the dependent instance
    ' or assign them to class-level variables
  End Sub
```

```
End Class
```

6. In your run-time code, register the mappings you require for the interface and base class types, and then use the **Resolve** method of the container to create an instance of the target class. The Unity container will instantiate an instance of each of the mapped concrete types for the dependent classes and inject them into the target class. For example, the following code shows how you can instantiate the example target class named **MyObject** containing a constructor that has a dependency on the two objects of type **IMyInterface** and **MyBaseClass**.

**C#**

```
IUnityContainer uContainer = new UnityContainer()
   .RegisterType<IMyInterface, FirstObject>()
   .RegisterType<MyBaseClass, SecondObject>();
MyObject myInstance = uContainer.Resolve<MyObject>();
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer() _
   .RegisterType(Of IMyInterface, FirstObject)() _
   .RegisterType(Of MyBaseClass, SecondObject)()
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()
```

## Specifying Named Type Mappings

The preceding example shows how you can resolve types for constructor parameters using the default (unnamed) mappings in the container. If you register more than one mapping for a type, you must differentiate them by using a name. In this case, you can specify which named mapping the container will use to resolve each constructor parameter type.

**To use attributed constructor injection with named container type mappings**

1. Define a constructor in the target class that takes as a parameter the concrete type of the dependent class, and apply a **Dependency** attribute to the parameter that specifies the name of the registered mapping to use. For example, the following code shows a target class named **MyObject** containing a constructor that has a dependency on a service registered with the name **myDataService**, and which implements the **IMyService** interface. It assumes that the container contains a mapping defined with the name **DataService** between the **IMyService** interface and a concrete implementation of this interface.

**C#**

```
public class MyObject
{
  public MyObject([Dependency("DataService")] IMyService myDataService)
  {
    // work with the service here
  }
}
```

**Visual Basic**

```
Public Class MyObject
  Public Sub New(<Dependency("DataService")> myDataService As IMyService)
    ' work with the service here
  End Sub
End Class
```

2. In your run-time code, use the **Resolve** method of the container to create an instance of the target class. The Unity container will instantiate the dependent concrete class defined in the mapping named **DataService** and inject it into the target class. For example, the following code shows how you can instantiate the example target class shown above.

**C#**

```
IUnityContainer uContainer = new UnityContainer();
MyObject myInstance = uContainer.Resolve<MyObject>();
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer()
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()
```

You can use the **Dependency** attribute on more than one constructor parameter. You can also use it when the constructor defines more than one parameter of the same type to differentiate the mappings and ensure that the appropriate concrete type is returned for each parameter.

If you specify a named mapping and there is no mapping registered for that type and name, the container will raise an exception.

## Multiple Constructor Injection Using an Attribute

When a target class contains more than one constructor with the same number of parameters, you must apply the **InjectionConstructor** attribute to the constructor that the Unity container will use to indicate which constructor the container should use. As with automatic constructor injection, you can specify the constructor parameters as a concrete type, or you can specify an interface or base class for which the Unity container contains a registered mapping.

**To use attributed constructor injection when there is more than one constructor**

1. Apply the **InjectionConstructor** attribute to the constructor in the target class that you want the container to use. In the simplest case, the target constructor takes as a parameter the concrete type of the dependent class. For example, the following code shows a target class named **MyObject** containing two constructors, one of which has a dependency on a class named **MyDependentClass** and has the **InjectionConstructor** attribute applied.

**C#**

```
public class MyObject
{

  public MyObject(SomeOtherClass myObjA)
  {
```

```
    ...
  }

  [InjectionConstructor]
  public MyObject(MyDependentClass myObjB)
  {
    ...
  }

}
```

**Visual Basic**

```
Public Class MyObject

  Public Sub New(myObjA As SomeOtherClass)
    ...
  End Sub

  <InjectionConstructor()> _
  Public Sub New(myObjB As MyDependentClass)
    ...
  End Sub

End Class
```

2. In your run-time code, use the **Resolve** method of the container to create an instance of the target class. The Unity container will instantiate the dependent concrete class defined in the attributed constructor and inject it into the target class. For example, the following code shows how you can instantiate the example target class named **MyObject** containing an attributed constructor that has a dependency on a class named **MyDependentClass**.

**C#**

```
IUnityContainer uContainer = new UnityContainer();
MyObject myInstance = uContainer.Resolve<MyObject>();
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer()
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()
```

3. Alternatively, you can define a multiple-constructor target class that contains more than one dependency defined in the target constructor parameters. The Unity container will instantiate and inject an instance of each one. For example, the following code shows a target class named **MyObject** containing an attributed constructor that has dependencies on two classes, **DependentClassA** and **DependentClassB**.

**C#**

```
public class MyObject
{

  public MyObject(SomeClassA objA, SomeClassB objB)
  {
    ...
```

```
  }

  [InjectionConstructor]
  public MyObject(DependentClassA depA, DependentClassB depB)
  {
    ...
  }

}
```

**Visual Basic**

```
Public Class MyObject

  Public Sub New(objA As SomeClassA, objB As SomeClassB)
    ...
  End Sub

  <InjectionConstructor()> _
  Public Sub New(depA As DependentClassA, depB As DependentClassB)
    ...
  End Sub

End Class
```

4. In your run-time code, use the **Resolve** method of the container to create an instance of the target class. The Unity container will create an instance of each of the dependent concrete classes defined in the attributed constructor and inject them into the target class. For example, the following code shows how you can instantiate the example target class named **MyObject** containing a constructor that has constructor dependencies

**C#**

```
IUnityContainer uContainer = new UnityContainer();
MyObject myInstance = uContainer.Resolve<MyObject>();
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer()
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()
```

5. In addition to using concrete types as parameters of the target object constructor, you can use interfaces or base class types, and then register mappings in the Unity container to translate these types into the correct concrete types. For details, see steps 5 and 6 of the procedure Single Constructor Automatic Injection.

## Notes on Using Constructor Injection

The following notes will help you to get the greatest benefit from using constructor injection with Unity.

## How Unity Resolves Target Constructors and Parameters

When a target class contains more than one constructor, Unity will use the one that has the **InjectionConstructor** attribute applied. If there is more than one constructor, and none carries the **InjectionConstructor** attribute, Unity will use the constructor with the most parameters. If there is more than one constructor that is the "longest" with the same number of parameters, Unity will raise an exception.

## Constructor Injection with Existing Objects

If you use configuration or the **RegisterInstance** method to register an existing object, constructor injection does not take place on that object because it has already been created outside of the influence of the Unity container. Even if you call the **BuildUp** method of the container and pass it the existing object, constructor injection will never take place because the constructor will not execute.

Instead, you can use property (setter) injection by applying a **Dependency** attribute to a public property. Alternatively, create an **Initialize** method for the class that takes the type to resolve as a parameter and apply the **InjectionMethod** attribute to this method. Unity will resolve the parameter and call the method. Inside the method, you can store a reference to the resolved object for use in your code. For more details, see Annotating Objects for Method Call Injection.

## Avoiding Circular References

Dependency injection mechanisms can cause application errors if there are circular references between objects that the container will create. For more details, see Circular References with Dependency Injection.

## When to Use Constructor Injection

You should consider using constructor injection in the following situations:

- When you want to perform dependency injection on new instances of objects or classes created through the Unity container. Constructor injection cannot be used with existing object instances.
- When you want to instantiate dependent objects automatically when you instantiate the parent object.
- When you want a simple approach that makes it easy to see in the code what the dependencies are for each class.
- When the parent object does not require a large number of constructors that forward to each other.
- When the parent object constructors do not require a large number of parameters.

- When you want to be able to hide field values from view in the application code by not exposing them as properties or methods.
- When you want to control which objects are injected by editing the code of the dependent object instead of the parent object or application.

---

If you are not sure which type of injection to use, the recommendation is that you use constructor injection unless you are working with an existing instance of an object or class.

> You can also configure constructor injection at design time or run time. For more information, see Configuring Unity.

# Annotating Objects for Property (Setter) Injection

Unity supports dependency injection to set the values or properties through attributes applied to members of the target class. You can use the Unity container to generate instances of dependent objects and wire up the target class properties with these instances. This topic explains how to use an attribute that is applied to one or more property declarations of a class to define the dependency injection requirements of that class. The attribute can specify parameters for the attribute to control its behavior, such as the name of a registered mapping.

To perform property injection of dependent classes into objects you create through the Unity container, you apply the **Dependency** attribute to the property declarations of a class. The Unity container will create an instance of the dependent class within the scope of the target object (the object you specify in a **Resolve** method call) and assign this dependent object to the attributed property of the target object.

Property injection is a form of optional injection of dependent objects, as long as developers use the Unity container to generate the target object. The dependent object instance is generated before the container returns the target object. In addition, unlike constructor injection, you must apply the appropriate attribute in the target class to initiate property injection. You can also perform property injection of optional dependent classes by applying the **OptionalDependency** attribute. This simply marks a dependency as optional, which means that the container will try to resolve it, and return **null** if the resolution fails rather than throw an exception. For more information, see Notes on Using Property (Setter) Injection.

**To use property (setter) injection to create dependent objects for a class**

1. Define a property in the target class and apply the **Dependency** attribute to it to indicate that the type defined and exposed by the property is a dependency of the class. The following code demonstrates property injection for a class named **MyObject** that exposes as

a property a reference to an instance of another class named **SomeOtherObject** (not defined in this code).

**C#**

```
public class MyObject
{
  private SomeOtherObject _dependentObject;

  [Dependency]
  public SomeOtherObject DependentObject
  {
    get { return _dependentObject; }
    set { _dependentObject = value; }
  }
}
```

**Visual Basic**

```
Public Class MyObject
  Private _dependentObject As SomeOtherObject

  <Dependency()> _
  Public Property DependentObject() As SomeOtherObject
    Get
      Return _dependentObject
    End Get
    Set(ByVal value As SomeOtherObject)
      _dependentObject = value
    End Set
  End Property

End Class
```

2. In your run-time code, use the **Resolve** method of the container to create an instance of the target class, and then reference the property containing the dependent object. The Unity container will instantiate the dependent concrete class defined in the attributed property and inject it into the target class. For example, the following code shows how you can instantiate the example target class named **MyObject** containing an attributed property that has a dependency on a class named **SomeOtherObject** and then retrieve the dependent object from the **DependentObject** property.

**C#**

```
IUnityContainer uContainer = new UnityContainer();
MyObject myInstance = uContainer.Resolve<MyObject>();

// now access the property containing the dependency
SomeOtherObject depObj = myInstance.DependentObject;
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer()
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()

' now access the property containing the dependency
```

```
Dim depObj As SomeOtherObject = myInstance.DependentObject
```

3. In addition to using concrete types for the dependencies in target object properties, you can use interfaces or base class types, and then register mappings in the Unity container to translate these types into the correct concrete types. Define a property in the target class as an interface or base type. For example, the following code shows a target class named **MyObject** containing properties named **InterfaceObject** and **BaseObject** that have dependencies on a class that implements the interface named **IMyInterface** and on a class that inherits from **MyBaseClass**.

**C#**

```
public class MyObject
{

  private IMyInterface _interfaceObj;
  private MyBaseClass _baseObj;

  [Dependency]
  public IMyInterface InterfaceObject
  {
    get { return _interfaceObj; }
    set { _interfaceObj = value; }
  }

  [Dependency]
  public MyBaseClass BaseObject
  {
    get { return _baseObj; }
    set { _baseObj = value; }
  }

}
```

**Visual Basic**

```
Public Class MyObject

  Private _interfaceObj As IMyInterface
  Private _baseObj As MyBaseClass

  <Dependency()> _
  Public Property InterfaceObject() As IMyInterface
    Get
      Return _interfaceObj
    End Get
    Set(ByVal value As IMyInterface)
      _interfaceObj = value
    End Set
  End Property

  <Dependency()> _
  Public Property BaseObject() As MyBaseClass
    Get
```

```
      Return _baseObj
    End Get
    Set(ByVal value As MyBaseClass)
      _baseObj = value
    End Set
  End Property

End Class
```

4. In your run-time code, register the mappings you require for the interface and base class types, and then use the **Resolve** method of the container to create an instance of the target class. The Unity container will create an instance of each of the mapped concrete types for the dependent classes and inject them into the target class. For example, the following code shows how you can instantiate the example target class named **MyObject** containing two properties that have dependencies on the two classes named **FirstObject** and **SecondObject**.

**C#**

```
IUnityContainer uContainer = new UnityContainer()
   .RegisterType<IMyInterface, FirstObject>()
   .RegisterType<MyBaseClass, SecondObject>();
MyObject myInstance = uContainer.Resolve<MyObject>();

// now access the properties containing the dependencies
IMyInterface depObjA = myInstance.InterfaceObject;
MyBaseClass depObjB = myInstance.BaseObject;
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer() _
   .RegisterType(Of IMyInterface, FirstObject)() _
   .RegisterType(Of MyBaseClass, SecondObject)()
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()

' now access the properties containing the dependencies
Dim depObjA As IMyInterface = myInstance.InterfaceObject
Dim depObjB As MyBaseClass = myInstance.BaseObject
```

5. You can register multiple named mappings with the container for each dependency type, if required, and then use a parameter of the **Dependency** attribute to specify the mapping you want to use to resolve the dependent object type. For example, the following code specifies the mapping names for the **Key** property of the **Dependency** attribute for two properties of the same type (in this case, an interface) in the class **MyObject**.

**C#**

```
public class MyObject
{

  private IMyInterface _objA, _objB;

  [Dependency("MapTypeA")]
  public IMyInterface ObjectA
  {
```

```
    get { return _objA; }
    set { _objA = value; }
  }

  [Dependency("MapTypeB")]
  public IMyInterface ObjectB
  {
    get { return _objB; }
    set { _objB = value; }
  }

}
```

**Visual Basic**

```
Public Class MyObject

  Private _objA, _objB As IMyInterface

  <Dependency("MapTypeA")> _
  Public Property ObjectA() As IMyInterface
    Get
      Return _objA
    End Get
    Set(ByVal value As IMyInterface)
      _objA = value
    End Set
  End Property

  <Dependency("MapTypeB")> _
  Public Property ObjectB() As IMyInterface
    Get
      Return _objB
    End Get
    Set(ByVal value As IMyInterface)
      _objB = value
    End Set
  End Property

End Class
```

6. In your run-time code, register the named (non-default) mappings you require for the two concrete types that the properties will depend on, and then use the **Resolve** method of the container to create an instance of the target class. The Unity container will instantiate an instance of each of the mapped concrete types for the dependent classes and inject them into the target class. For example, the following code shows how you can instantiate the example target class named **MyObject** containing two properties that have dependencies on the two classes named **FirstObject** and **SecondObject**.

**C#**

```
IUnityContainer uContainer = new UnityContainer()
   .RegisterType<IMyInterface, FirstObject>("MapTypeA")
   .RegisterType<IMyInterface, SecondObject>("MapTypeB");
```

```
MyObject myInstance = uContainer.Resolve<MyObject>();

// now access the properties containing the dependencies
IMyInterface depObjA = myInstance.ObjectA;
IMyInterface depObjB = myInstance.ObjectB;
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer() _
   .RegisterType(Of IMyInterface, FirstObject)("MapTypeA") _
   .RegisterType(Of IMyInterface, SecondObject)("MapTypeB")
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()

' now access the properties containing the dependencies
Dim depObjA As IMyInterface = myInstance.ObjectA
Dim depObjB As IMyInterface = myInstance.ObjectB
```

## Using Optional Dependencies

An optional dependency works like a regular dependency with the difference that if the optional dependency cannot be resolved, **null** is returned by the container.

**C#**

```
public class ObjectWithOptionalProperty
{
  [OptionalDependency]
  public ICarInterface Car
  {
….  get;
….  set;
  }
}
```

**Visual Basic**

```
Public Class ObjectWithOptionalProperty
..<OptionalDependency> _
..Public Property Car() As ICarInterface
      Get
      End Get
      Set
      End Set
..End Property
End Class
```

## Notes on Using Property (Setter) Injection

Remember that property injection is optional and that you must apply the **Dependency** attribute to target class properties if you want property injection of dependent types to occur.

## Using the Dependency Attribute with Constructor and Method Parameters

You can also apply the **Dependency** attribute to the parameters of a constructor or method to specify the named mapping you want the container to use. If you do not specify a named mapping, Unity uses the default (unnamed) mapping. For more details, see Annotating Objects for Constructor Injection and Annotating Objects for Method Call Injection.

## Property Injection with Existing Objects

If you use configuration or the **RegisterInstance** method to register an existing object, property (setter) injection does not take place on that object because it has already been created outside of the influence of the Unity container. However, you can call the **BuildUp** method of the container and pass it the existing object to force property injection to take place on that object.

## Avoiding the Use of Public Properties

To use property injection, you must expose a public property from your class to which you can apply the **Dependency** attribute. If the only reason you are exposing this property is to implement injection, and you would prefer not to expose it as a public property, you can use method call injection instead. Create an **Initialize** method for the class that takes the type to resolve as a parameter and apply the **InjectionMethod** attribute to this method. Unity will resolve the parameter and call the method. Inside the method, you can store a reference to the resolved object for use in your code. For more details, see Annotating Objects for Method Call Injection.

## Avoiding Circular References

Dependency injection mechanisms can cause application errors if there are circular references between objects that the container will create. For more details, see Circular References with Dependency Injection.

## When to Use Property (Setter) Injection

You should consider using property injection in the following situations:

- When you want to instantiate dependent objects automatically when you instantiate the parent object.
- When you want a simple approach that makes it easy to see in the code what the dependencies are for each class.
- When the parent object requires a large number of constructors that forward to each other, making debugging and maintenance difficult.
- When the parent object constructors require a large number of parameters, especially if they are of similar types and the only way to identify them is by position.

- When you want to make it easier for users to see what settings and objects are available, which is not possible using constructor injection.
- When you want to control which objects are injected by editing the code of the dependent object instead of the parent object or application.

---

If you are not sure which type of injection to use, the recommendation is that you use **constructor** injection. This is likely to satisfy almost all general requirements unless you need to perform injection on existing object instances.

> You can also configure property injection at design time or run time. For more information, see Configuring Unity.

# Annotating Objects for Method Call Injection

Unity supports dependency injection to set the values of parameters of methods specified through attributes applied to members of the target class. You can use the Unity container to generate instances of dependent objects and wire up the target class method parameters with these instances. This topic explains how to use an attribute that is applied to one or more method declarations of a class to define the dependency injection requirements of that class.

To perform injection of dependent classes into objects you create through the Unity container, you apply the **InjectionMethod** attribute to the method declarations of a class. The Unity container will force the target object (the object you specify in a **Resolve** method call) to create an instance of the dependent class and then call the target method. If required, your code in the method can save this instance by assigning it to a class-level variable.

Method call injection is a form of optional injection of dependent objects that you can use if you use the Unity container to generate the target object. Unity instantiates dependent objects defined in parameters of methods that carry the **InjectionMethod** attribute within the scope of the target object. Then it calls the attributed method of the target object before returning the object to the caller. You must apply the **InjectionMethod** attribute in the target class to initiate method call injection. For more information, see Notes on Using Method Call Injection.

**To use method call injection to create dependent objects for a class**

1. Define a method in the target class and apply the **InjectionMethod** attribute to it to indicate that any types defined in parameters of the method are dependencies of the class. The following code demonstrates the most common scenario—saving the dependent object instance in a class-level variable—for a class named **MyObject** that exposes a method named **Initialize** that takes as a parameter a reference to an instance of another class named **SomeOtherObject** (not defined in this code).

**C#**

```
public class MyObject
{

  private SomeOtherObject dependentObject;

  [InjectionMethod]
  public void Initialize(SomeOtherObject dep)
  {
    // assign the dependent object to a class-level variable
    dependentObject = dep;
  }
}
```

**Visual Basic**

```
Public Class MyObject

  Private dependentObject As SomeOtherObject

  <InjectionMethod()> _
  Public Sub Initialize(dep As SomeOtherObject)
    ' assign the dependent object to a class-level variable
    dependentObject = dep
  End Sub

End Class
```

2. In your run-time code, use the **Resolve** method of the container to create an instance of the target class. The Unity container will instantiate the dependent concrete class defined in the attributed method, inject it into the target class, and execute the method. For example, the following code shows how you can instantiate the example target class named **MyObject** containing an attributed method that has a dependency on a class named **SomeOtherObject**.

**C#**

```
IUnityContainer uContainer = new UnityContainer();
MyObject myInstance = uContainer.Resolve<MyObject>();
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer()
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()
```

3. In addition to using concrete types for the dependencies in target object methods, you can use interfaces or base class types and then register mappings in the Unity container to translate these types into the appropriate concrete types. Define a method in the target class that takes as parameters interfaces or base types. For example, the following code shows a target class named **MyObject** containing a method named **Initialize** that takes as parameters an object named **interfaceObj** that implements the interface named **IMyInterface** and an object named **baseObj** that inherits from the class **MyBaseClass**.

**C#**

```
public class MyObject
{

  private IMyInterface depObjectA;
  private MyBaseClass depObjectB;

  [InjectionMethod]
  public void Initialize(IMyInterface interfaceObj, MyBaseClass baseObj)
  {
    depObjectA = interfaceObj;
    depObjectB = baseObj;
  }
}
```

**Visual Basic**

```
Public Class MyObject

  Private depObjectA As IMyInterface
  Private depObjectB As MyBaseClass

  <InjectionMethod()> _
  Public Sub Initialize(interfaceObj As IMyInterface, baseObj As
MyBaseClass)
    depObjectA = interfaceObj
    depObjectB = baseObj
  End Sub

End Class
```

4. In your run-time code, register the mappings you require for the interface and base class types, and then use the **Resolve** method of the container to create an instance of the target class. The Unity container will instantiate an instance of each of the mapped concrete types for the dependent classes, and inject them into the target class. For example, the following code shows how you can instantiate the example target class named **MyObject** containing an attributed method that has dependencies on the two classes, **FirstObject** and **SecondObject**.

**C#**

```
IUnityContainer uContainer = new UnityContainer()
   .RegisterType<IMyInterface, FirstObject>()
   .RegisterType<MyBaseClass, SecondObject>();
MyObject myInstance = uContainer.Resolve<MyObject>();
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer() _
   .RegisterType(Of IMyInterface, FirstObject)() _
   .RegisterType(Of MyBaseClass, SecondObject)()
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()
```

## Specifying Named Type Mappings

The preceding example shows how you can resolve types for method parameters using the default (unnamed) mappings in the container. If you register more than one mapping for a type, you must differentiate them by using a name. In this case, you can specify which named mapping the container will use to resolve the method parameter types.

**To use attributed method call injection with named container type mappings**

1. Define a method in the target class that takes as a parameter the concrete type of the dependent class, and apply a **Dependency** attribute to the parameter that specifies the name of the registered mapping to use. For example, the following code shows a target class named **MyObject** containing a method named **Initialize** that has a dependency on a service that implements the **IMyService** interface. The code assumes that the container contains a mapping defined with the name **DataService** between the **IMyService** interface and a concrete implementation of that interface.

**C#**

```
public class MyObject
{
  private IMyService myDataService;

  [InjectionMethod]
  public void Initialize([Dependency("DataService")] IMyService theService)
  {
    // assign the dependent object to a class-level variable
    myDataService = theService;
  }
}
```

**Visual Basic**

```
Public Class MyObject

  Private myDataService As IMyService

  <InjectionMethod()> _
  Public Sub Initialize(<Dependency("DataService")> theService As
IMyService)
    ' assign the dependent object to a class-level variable
    myDataService = theService
  End Sub

End Class
```

2. In your run-time code, use the **Resolve** method of the container to create an instance of the target class. The Unity container will instantiate the dependent concrete class defined in the attributed method, inject it into the target class, and execute the method. For example, the following code shows how you can instantiate the example class shown above.

**C#**

```
IUnityContainer uContainer = new UnityContainer();
MyObject myInstance = uContainer.Resolve<MyObject>();
```

**Visual Basic**

```
Dim uContainer As IUnityContainer = New UnityContainer()
Dim myInstance As MyObject = uContainer.Resolve(Of MyObject)()
```

If you specify a named mapping and there is no mapping registered for that type and name, the container will raise an exception.

## Notes on Using Method Call Injection

The following notes will help you to get the greatest benefit from using method call injection with Unity.

## Method Call Injection with Existing Objects

If you use configuration or the **RegisterInstance** method to register an existing object, method call injection does not take place on that object because it has already been created outside of the influence of the Unity container. However, you can call the **BuildUp** method of the container and pass it the existing object to force method call injection to take place on that object.

## Avoiding Circular References

Dependency injection mechanisms can cause application errors if there are circular references between objects that the container will create. For more details, see Circular References with Dependency Injection.

## When to Use Method Call Injection

You should consider using method call injection in the following situations:

- When you want to instantiate dependent objects automatically when you instantiate the parent object.
- When you want a simple approach that makes it easy to see in the code what the dependencies are for each class.
- When you want to avoid exposing a public property just to implement injection, and instead use an **Initialize** method.
- When the parent object requires a large number of constructors that forward to each other, making debugging and maintenance difficult.
- When the parent object constructors require a large number of parameters, especially if they are of similar types and the only way to identify them is by position.

- When you want to hide the dependent objects by not exposing them as properties.
- When you want to control which objects are injected by editing the code of the dependent object instead of the parent object or application.

If you are not sure which type of injection to use, the recommendation is that you use **constructor** injection. This is likely to satisfy almost all general requirements, unless you need to perform injection on existing object instances.

> You can also configure method call injection at design time or run time. For more information, see Configuring Unity.

# Using Container Hierarchies

Unity supports nested containers, allowing you to build container hierarchies. Nesting containers enables you to control the scope and lifetime of singleton objects, and register different mappings for specific types. This topic contains the following sections that describe how you can create container hierarchies and use them in your applications:

- Constructing and Disposing Unity Containers
- Controlling Object Scope and Lifetime
- Registering Different Mappings for Specific Types

## Constructing and Disposing Unity Containers

The following methods enable you to create a new default **UnityContainer**, create a child container that has a specified **UnityContainer** as its parent, and dispose an existing container. For more information about configuring containers, see Configuring Unity.

| Method | Description |
|---|---|
| UnityContainer( ) | Creates a default **UnityContainer**. Returns a reference to the new container. |
| CreateChildContainer( ) | Creates a new nested **UnityContainer** as a child of the current container. The current container first applies its own settings, and then it checks the parent for additional settings. Returns a reference to the new container. |
| Dispose( ) | Disposes this container instance and any child containers. Also disposes any registered object instances whose lifetimes are managed by the container. |

## Controlling Object Scope and Lifetime

When the container creates singleton objects, it manages the lifetime of these singletons. They remain in scope until you (or the garbage collector) dispose the container. At this point, it disposes the registered singleton instances it contains. In addition, if you dispose the parent container in a nested container hierarchy, it automatically disposes all child containers and the registered singletons they contain.

Therefore, if you require two separate sets of such objects that must have different lifetimes, you can use hierarchical containers to store and manage each set. Register instances that you want to be able to dispose separately in one or more child containers that you can dispose without disposing the parent container.

The following code demonstrates the use of a child container to manage the lifetime of specific singleton instances while maintaining the singleton instances in the parent container.

**C#**

```
// Create parent container
IUnityContainer parentCtr = new UnityContainer();

// Register type in parent container
parentCtr.RegisterType<MyParentObject>(new ContainerControlledLifetimeManager());

// Create nested child container in parent container
IUnityContainer childCtr = parentCtr.CreateChildContainer();

// Register type in child container
childCtr.RegisterType<MyChildObject>(new ContainerControlledLifetimeManager());

// Create instance of type stored in parent container
MyParentObject parentObj = parentCtr.Resolve<MyParentObject>();

// Create instance of type stored in child container
MyChildObject childObj = childCtr.Resolve<MyChildObject>();

// ... can use both generated objects here ...

// Dispose child container
childCtr.Dispose();

// ... can use only object in parent container here ...

// Dispose parent container
parentCtr.Dispose();
```

**Visual Basic**

```
' Create parent container
Dim parentCtr As IUnityContainer = New UnityContainer()

' Register type in parent container
parentCtr.RegisterType(Of MyParentObject)(New
ContainerControlledLifetimeManager())
```

```
' Create nested child container in parent container
Dim childCtr As IUnityContainer = parentCtr.CreateChildContainer()

' Register type in child container
childCtr.RegisterType(Of MyChildObject)(New ContainerControlledLifetimeManager())

' Create instance of type stored in parent container
Dim parentObj As MyParentObject = parentCtr.Resolve(Of MyParentObject)()

' Create instance of type stored in child container
Dim childObj As MyChildObject = childCtr.Resolve(Of MyChildObject)()

' ... can use both generated objects here ...

' Dispose child container
childCtr.Dispose()

' ... can use only object in parent container here ...

' Dispose parent container
parentCtr.Dispose()
```

You can also specify the lifetime of objects by using the **LifetimeManager** parameter when you register types or existing objects with the container. For more information about how to use the **LifetimeManager** parameter, see Understanding Lifetime Managers and Configuring Unity. For more information about how to create custom lifetime managers, see Creating Lifetime Managers.

## Registering Different Mappings for Specific Types

You can use nested containers when you have slightly different dependency injection requirements for specific objects but want to provide a fallback facility for objects that implement a specific interface or are of a specific type. For example, you may have a general requirement for objects that implement the **IMyObject** interface to map to the type **MyStandardObject**. However, in specific parts of the application code, you may want the **IMyObject** interface to map to the type **MySpecialObject**.

In this case, you can register the general mapping in the parent container and register the specific case in a child container. Then, when you want to obtain an instance of the object, you call the **Resolve** method on the appropriate container. If you call the method on the child container, it returns an object of type **MySpecialObject**. If you call the method on the parent container, it returns an object of type **MyStandardObject**.

However, the advantage with nested containers is that, if the child container cannot locate a mapping for the requested interface or type, it passes the request to its parent container and onward through the hierarchy until it reaches the root or base container. Therefore, for objects not mapped in the child container, the mapping in the parent container (or in an ancestor container where there are more than two levels in the hierarchy) defines the object type returned.

The following code shows how you can implement the preceding scenario.

**C#**

```
// Create parent container
IUnityContainer parentCtr = new UnityContainer();

// Register two mappings for types in parent container
parentCtr.RegisterType<IMyObject, MyStandardObject>();
parentCtr.RegisterType<IMyOtherObject, MyOtherObject>();

// Create nested child container in parent container
IUnityContainer childCtr = parentCtr.CreateChildContainer();

// Register mapping for specific type in child container
childCtr.RegisterType<IMyObject, MySpecialObject>();

// Now retrieve instances of the mapped objects using the child container.
// Using the interface as the type for the returned objects means that it
// does not matter which container returns the actual object.

// This code returns an object of type MySpecialObject using the mapping
// registered in the child container:
IMyObject specialObject = childCtr.Resolve<IMyObject>();

// This code returns an object of type MyOtherObject using the mapping
// registered in the parent container because there is no mapping in
// the child container for this type:
IMyOtherObject otherObject = childCtr.Resolve<IMyOtherObject>();

// Now retrieve instance of the standard object using the parent container.
// This code returns an object of type MyStandardObject using the mapping
// registered in the parent container:
IMyObject standardObject = parentCtr.Resolve<IMyObject>();

// Dispose parent container and child container
parentCtr.Dispose();
```

**Visual Basic**

```
' Create parent container
Dim parentCtr As IUnityContainer = New UnityContainer()

' Register two mappings for types in parent container
parentCtr.RegisterType(Of IMyObject, MyStandardObject)()
parentCtr.RegisterType(Of IMyOtherObject, MyOtherObject)()

' Create nested child container in parent container
Dim childCtr As IUnityContainer = parentCtr.CreateChildContainer()

' Register mapping for specific type in child container
childCtr.RegisterType(Of IMyObject, MySpecialObject)()

' Now retrieve instances of the mapped objects using the child container.
' Using the interface as the type for the returned objects means that it
```

```
' does not matter which container returns the actual object.

' This code returns an object of type MySpecialObject using the mapping
' registered in the child container:
Dim specialObject As IMyObject = childCtr.Resolve(Of IMyObject)()

' This code returns an object of type MyOtherObject using the mapping
' registered in the parent container because there is no mapping in
' the child container for this type:
Dim otherObject As IMyOtherObject = childCtr.Resolve(Of IMyOtherObject)()

' Now retrieve instance of the standard object using the parent container.
' This code returns an object of type MyStandardObject using the mapping
' registered in the parent container:
Dim standardObject As IMyObject = parentCtr.Resolve(Of IMyObject)()

' Dispose parent container and child container
parentCtr.Dispose()
```

# Circular References with Dependency Injection

Dependency injection mechanisms carry the risk of unintentional circular references, which are not easy to detect or prevent. This topic describes the situations where you may inadvertently cause circular references to occur, resulting in a stack overflow and application error. The most common causes of circular references with dependency injection are the following:

- Objects generated through constructor injection that reference each other in their constructor parameters
- Objects generated through constructor injection where an instance of a class is passed as a parameter to its own constructor
- Objects generated through method call injection that reference each other
- Objects generated through property (setter) injection that reference each other

For example, the following code shows two classes that reference each other in their constructors.

**C#**

```
public class Class1
{
  public Class1(Class2 test2)
  { ... }
}

public class Class2
```

```
{
  public Class2(Class1 test1)
  { ... }
}
```

**Visual Basic**

```
Public Class Class1
  Public Sub New(test2 As Class2)
    ...
  End Sub
End Class

Public Class Class2
  Public Sub New (test 1 As Class1)
    ...
  End Sub
End Class
```

It is the responsibility of the developer to prevent this type of error by ensuring that the members of classes they use with dependency injection do not contain circular references.

> You could use constructor injection to specify any of a series of constructors or method overloads; however, you could inadvertently cause endless recursion. To avoid the endless recursion, specify which constructor to call in the **RegisterType** call.

Unity's default behavior is to resolve the constructor with the most parameters. This would cause endless recursion in the following example.

**C#**

```
container.RegisterType<IServiceProvider, ServiceContainer>();
var sp = container.Resolve<IServiceProvider>();
```

**Visual Basic**

```
container.RegisterType(Of IServiceProvider, ServiceContainer)()
Dim sp = container.Resolve(Of IServiceProvider)()
```

To avoid the endless recursion, specify which constructor to call in the **RegisterType** call, as in the following example:

**C#**

```
container.RegisterType<IServiceProvider, ServiceContainer>(new
InjectionConstructor());
```

**Visual Basic**

```
container.RegisterType(Of IServiceProvider, ServiceContainer) _
          (New InjectionConstructor())
```

In this case, when creating the service container, the zero argument constructor is explicitly requested.

# Design of Unity

This topic describes the design goals, the architecture, and the design highlights of Unity. You do not have to understand the design to use Unity; however, this topic will help you to understand how it works and how it interacts with the underlying ObjectBuilder subsystem.

## Design Goals

Unity was designed to achieve the following goals:

- Promote the principles of modular design through aggressive decoupling.
- Raise awareness of the need to maximize testability when designing applications.
- Provide a fast and lightweight dependency injection container mechanism for creating new object instances and managing existing object instances.
- Expose a compact and intuitive API for developers to work with the container.
- Support a wide range of code languages, with method overrides that accept generic parameters where the language supports these.
- Implement attribute-driven injection for constructors, property setters, and methods of target objects.
- Provide extensibility through custom and third-party container extensions.
- Provide the performance required in enterprise-level line of business (LOB) applications.

## Operation

The public methods of the UnityContainer that developers use fall into two main categories:

- **Methods that register mappings or types**. The methods **RegisterType** and the **RegisterInstance** create the appropriate entries within the current container context.
- **Methods that retrieve objects**. These include overrides of the **Resolve**, **ResolveAll**, and **BuildUp** methods. These methods retrieve the required instances of the objects.

# Extending and Modifying Unity

If required, you can extend and modify Unity to better suit your own requirements. You can extend Unity by doing the following:

- Creating Lifetime Managers that control how and when the container will dispose of instances of objects it resolves.
- Creating and Using Container Extensions that can change the behavior of the container, the instance generation mechanism, and the dependency injection and interception features.
- Creating Policy Injection Matching Rules that provides alternative techniques for selecting classes and class members to which Unity will attach a handler pipeline.
- Creating Interception Policy Injection Call Handlers that perform the task-specific processing you require for method invocations and property accessors.
- Creating Interception Handler Attributes that cause Unity to add built-in or custom call handlers to the handler pipeline. If you create a custom handler, you may also want to create a custom attribute that developers can use to apply your handler by adding the attribute directly to classes or class members within the source code of an application.
- Creating Interception Behaviors that describe what to do when an object is intercepted.

# Creating Lifetime Managers

Unity supports several approaches for registering classes, interfaces, and instances of existing objects. You can use the **RegisterType** and **RegisterInstance** methods to register object instances at run time, or specify the mappings and registrations at design time in configuration. You can also specify a lifetime manager with all of these approaches that will control how Unity resolves instances of the specified types and how it holds references to these instances. The built-in lifetime managers allow you to specify objects as singletons, with weak references, or as per-thread instances.  For more details, see Understanding Lifetime Managers.

You can create custom **LifetimeManager** classes if you require additional functionality not available in the default lifetime managers. Documentation to help you do this is available from the Unity Community Web site on CodePlex.

# Creating and Using Container Extensions

You can create your own custom Unity container extensions, or use container extensions created by third parties with Unity. Unity uses default container extensions to implement its own functionality.

For example, the interception mechanism provided by Unity is implemented as a container extension.

Documentation to help you understand ObjectBuilder, and the steps required to create custom container extensions, is available from the Unity Community Web site on CodePlex.

# Creating Policy Injection Matching Rules

Unity defines an interface named **IMatchingRule**, which all classes that implement matching rules must implement. This interface declares a single method named **Matches** that takes a **MethodBase** instance and returns a **Boolean** value.

**C#**

```
public interface IMatchingRule
{
  bool Matches(MethodBase member);
}
```

**Visual Basic**

```
Public Interface IMatchingRule
  Function Matches(ByVal member As MethodBase) As Boolean
End Interface
```

Inside a concrete implementation of this interface, a custom matching rule class can access details of the current member (method or property) of the target object and determine whether Unity should add a handler pipeline to this member. The **Matches** method should return **True** if the current member matches the requirements of this matching rule; if the current member does not match the requirements of this matching rule, it should return **False**.

> Remember that there may be several matching rules defined for a policy, and every one must return **True** when Unity calls their **Matches** method in order for Unity to add the handler pipeline to the target class member.

## Example: The TagAttributeMatchingRule

As an example of a matching rule, this following extract shows the **TagAttributeMatchingRule** class that implements the built-in tag attribute matching rule. This rule examines the current target object member, passed to the **Matches** method as a parameter, looking for any attributes of type **TagAttribute**, whose value matches the string value passed to the class constructor.

**C#**

```
[ConfigurationElementType(typeof(TagAttributeMatchingRuleData))]
```

```
public class TagAttributeMatchingRule : IMatchingRule
{
  private readonly string tagToMatch;
  private readonly bool ignoreCase;

  // This constructor takes only the tag value.
  public TagAttributeMatchingRule(string tagToMatch)
       : this(tagToMatch, false)
  { }

  // This constructor takes the tag value and the case-sensitivity setting.
  public TagAttributeMatchingRule(string tagToMatch, bool ignoreCase)
  {
    this.tagToMatch = tagToMatch;
    this.ignoreCase = ignoreCase;
  }

  // The returns true only if a matching TagAttribute exists on this member.
  public bool Matches(MethodBase member)
  {
    foreach (TagAttribute tagAttribute in
             ReflectionHelper.GetAllAttributes<TagAttribute>(member, true))
    {
      if (string.Compare(tagAttribute.Tag, tagToMatch, ignoreCase) == 0)
      {
        return true;
      }
    }
    return false;
  }
}
```

**Visual Basic**

```
<ConfigurationElementType(GetType(TagAttributeMatchingRuleData))> _
Public Class TagAttributeMatchingRule : Implements IMatchingRule

  Dim tagToMatch As String
  Dim ignoreCase As Boolean

  ' This constructor takes only the tag value.
  Public Sub New(ByVal tagToMatch As String)
    Me.New(tagToMatch, false)
  End Sub

  ' This constructor takes the tag value and the case-sensitivity setting.
  Public Sub New(ByVal tagToMatch As String, ByVal ignoreCase As Boolean)
    Me.tagToMatch = tagToMatch
    Me.ignoreCase = ignoreCase
  End Sub

  ' The returns true only if a matching TagAttribute exists on this member.
  Public Function Matches(ByVal member As MethodBase) As Boolean Implements
IMatchingRule.Matches
```

```
    For Each tagAttribute As TagAttribute In _
        ReflectionHelper.GetAllAttributes(Of TagAttribute)(member, true)
      If string.Compare(tagAttribute.Tag, tagToMatch, ignoreCase) = 0 Then
        Return True
      End If
    Next
    Return False
  End Sub
End Class
```

When Unity first creates an instance of the target object, it reads a list of matching rules from the configuration. For each rule it finds, it creates an instance of the appropriate class and passes to it as parameters the values of any attributes defined within the configuration for that matching rule. You can see that the constructors for the **TagAttributeMatchingRule** class accept the name of the attribute and a case-sensitivity value.

Internally, the **TagAttributeMatchingRule** class uses a separate helper class named **ReflectionHelper** that gets a list of all the attributes of type **TagAttribute** on the target member so that the **Matches** method can detect any that has the specified value—taking into account the setting of the **ignoreCase** property for this handler instance.

Notice that the **TagAttributeMatchingRule** class has a **ConfigurationElementType** attribute that defines the class (**TagAttributeMatchingRuleData**) that the configuration system uses to expose the data from the configuration to the matching rule. As an alternative approach, you can also use the **CustomMatchingRuleData** class in this attribute. This eliminates the need to create your own configuration data class; however, configuration information will be provided as a dictionary and will not be strongly typed.

# Creating Interception Policy Injection Call Handlers

To create a call handler for policy injection, you must understand the way that Unity passes calls through the policy pipeline. This topic explains how the pipeline executes call handlers, and how it can block or abort execution when an error occurs, or on demand (such as when a validation handler detects a validation error or an authorization handler detects an unauthorized user). This topic contains the following sections:

## The ICallHandler Interface and Pipeline Execution

The Unity interception mechanism defines an interface named **ICallHandler**, which all classes that implement handlers for a call handler pipeline must implement. This interface defines two delegates and a single method named **Invoke**. The first delegate, named **InvokeHandlerDelegate**, is called by the previous handler to execute this handler—and therefore has the same signature as the **Invoke** method. The second, named **GetNextHandlerDelegate**, determines which is the next handler in the chain to invoke (the handler to which this handler should pass control when it completes its own preprocessing stage).

The signature for the **InvokeHandlerDelegate** and the **Invoke** method contains an instance of a class that implements the **IMethodInvocation** interface and a reference to an instance of the **GetNextHandlerDelegate** class. The **IMethodInvocation** implementation instance contains information about the current method invocation or property access, including the parameters to pass to the method or property.

The **Invoke** method returns an instance of a class that implements the **IMethodReturn** interface. This class instance contains any return value from the method or property accessor and any other information to return to the client. Usually, this is a concrete instance of the **ReturnMessage** class. The following extract shows the complete **ICallHandler** interface.

**C#**

```
public interface ICallHandler
{
  IMethodReturn Invoke(IMethodInvocation input, GetNextHandlerDelegate  getNext);
  int Order { get; set; }
}

public delegate IMethodReturn InvokeHandlerDelegate(IMethodInvocation input,
                              GetNextHandlerDelegate getNext);

public delegate InvokeHandlerDelegate GetNextHandlerDelegate();
```

**Visual Basic**

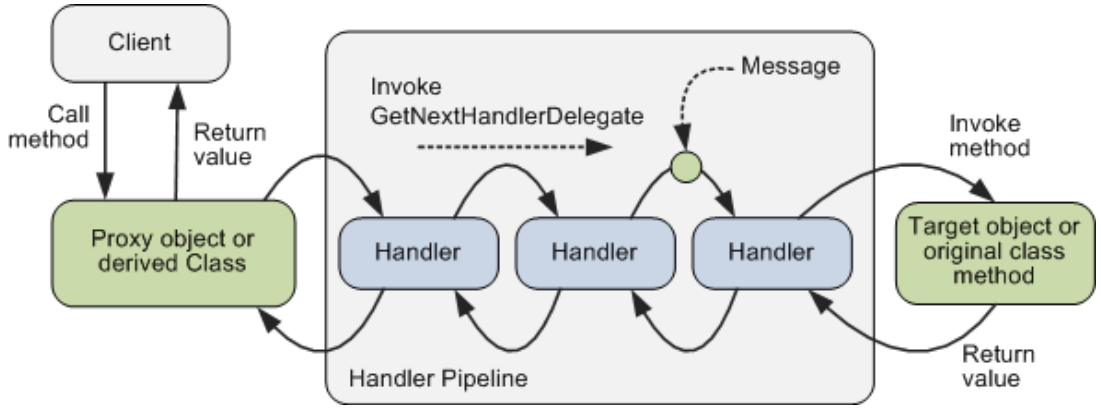
```
Public Interface ICallHandler
  Function Invoke(ByVal input As IMethodInvocation, _
                  ByVal getNext As GetNextHandlerDelegate) As IMethodReturn
  Property Order() As Integer
End Interface

Public Delegate Function InvokeHandlerDelegate(ByVal input As IMethodInvocation,
_
                  ByVal getNext As GetNextHandlerDelegate) As IMethodReturn

Public Delegate Function GetNextHandlerDelegate() As InvokeHandlerDelegate
```

The call handler pipline is part of the policy injection behavior which, if used, is just one behavior in the behaviors pipeline when performing Interception with Unity.

The following schematic shows the process for invocation through the call handler pipeline under normal conditions.




## Outline Implementation of a Call Handler

The following extract shows the basic outline of the **Invoke** method for a custom handler class, indicating where you can add code to perform both the preprocessing and postprocessing tasks you require. The code excludes the definition of the constructors and the **Order** property.

**C#**

```
[ConfigurationElementType(typeof(CustomCallHandlerData))]
public class ExampleHandler : ICallHandler
{
  public IMethodReturn Invoke(IMethodInvocation input,
                              GetNextHandlerDelegate getNext)
  {
    // Declare any variables required for values used in this method here.
    ...
    ...

    // Perform any pre-processing tasks required in the custom handler here.
    // This code executes before control passes to the next handler.
    ...
    ...

    // Use the following line of code in any handler to invoke the next
    // handler that Unity should execute. This code gets
    // the current return message that you must pass back to the caller:
    IMethodReturn msg = getNext()(input, getNext);

    // Perform any post-processing tasks required in the custom handler here.
    // This code executes after the invocation of the target object method or
    // property accessor, and before control passes back to the previous
    // handler as the Invoke call stack unwinds. You can modify the return
    // message if required.
    ...
    ...

    // Return the message to the calling code, which may be the previous
    // handler or, if this is the first handler in the chain, the client.
    return msg;
  }
```

```
  ... other class members as required here
}
```

**Visual Basic**

```
<ConfigurationElementType(GetType(CustomCallHandlerData))> _
Public Class ExampleHandler : Implements ICallHandler

  Public Function Invoke(ByVal input As IMethodInvocation, _
                      ByVal getNext As GetNextHandlerDelegate) As IMethodReturn _
         Implements ICallHandler.Invoke

    ' Declare any variables required for values used in this method here.
    ...
    ...

    ' Perform any pre-processing tasks required in the custom handler here.
    ' This code executes before control passes to the next handler.
    ...
    ...

    ' Use the following line of code in any handler to invoke the next
    ' handler that Unity should execute. This code gets
    ' the current return message that you must pass back to the caller:
    Dim msg As IMethodReturn = getNext()(input, getNext)

    ' Perform any post-processing tasks required in the custom handler here.
    ' This code executes after the invocation of the target object method or
    ' property accessor, and before control passes back to the previous
    ' handler as the Invoke call stack unwinds. You can modify the return
    ' message if required.
    ...
    ...

    ' Return the message to the calling code, which may be the previous
    ' handler or, if this is the first handler in the chain, the client.
    Return msg
  End Function

  ... other class members as required here

End Class
```

The **Invoke** method receives a reference to an instance of a concrete implementation of the **IMethodInvocation** class. Your handler can access the properties of this instance to get information about the current method or property accessor call. For example, you can access the target object instance (**Target**), the name of the target method or property (**MethodBase.Name**), a count of the number of inputs (**Inputs.Count**), and a collection of all the parameter values (**Arguments**).

Your code can change the values of the properties of the **IMethodInvocation** instance. However, be aware that changing the values of some properties may cause unexpected behavior. You should avoid changing any properties that affect the name, type, or signature of the target member.

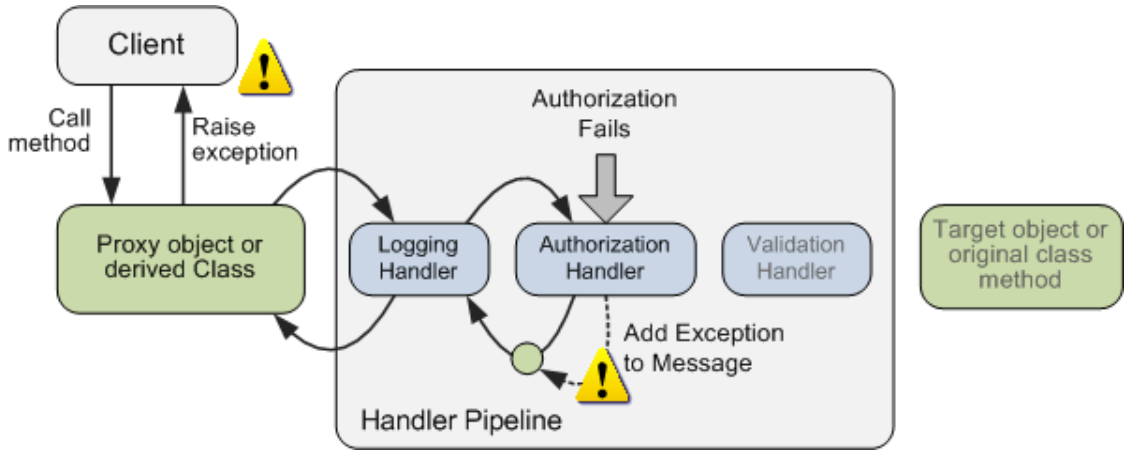
Unity reuses handler instances in different pipelines to minimize the number of objects it must create. Therefore, handlers should not store any per-call state in member variables. In multi-threaded applications, the action of multiple calls passing through the same handler instance is likely to corrupt the internal state.

## Exceptions and Aborted Pipeline Execution

Exceptions may occur in a call handler, or a call handler may wish to abort or short-circuit execution so that the subsequent handlers in the pipeline do not execute and Unity does not invoke the target method or property accessor. In this case, your code should simply avoid calling the next handler and generate a suitable return message. In other words, you omit the code line shown in the previous example that calls the **getNext** method. This causes all the previous handlers to execute their postprocessing tasks as the **Invoke** stack unwinds.

If your handler has to abort processing completely without allowing previous handlers to execute, you can create and raise a suitable exception type—depending on the tasks that your handler carries out. However, in most cases, you should *avoid* this approach and, instead, add to the message any exceptions you want to raise to the client so that previous handlers (which may perform logging or process exceptions) can execute their postprocessing tasks.

For example, the following schematic illustrates a scenario where a handler in the handler pipeline aborts execution. In this scenario, the handler pipeline contains—in execution order—a handler that performs logging, a handler that performs authorization, and a handler that validates the values in the parameters of the method call.




If, as shown in the schematic, the authorization handler detects an authorization failure, it can return an exception to the original caller by adding it to the message that passes along the handler

pipeline. This allows handlers earlier in the pipeline to carry out their post-processing tasks (such as, in this example, creating and writing a log message) as the stack unwinds. If, instead of passing it back through the message, a handler raises an unhandled exception, the previous handlers will not be able to execute their post-processing tasks.

To add an exception to the message, you create a new instance of the **ReturnMessage** class using the constructor that accepts a **System.Exception** instance or a subclass of **System.Exception**. For example, this code shows how you can add an exception to the return message when the current method request for some business-related activity occurs on a Saturday or Sunday.

**C#**

```
public IMethodReturn Invoke(IMethodInvocation input, GetNextHandlerDelegate
getNext)
{
  GregorianCalendar cal = new GregorianCalendar();
  DayOfWeek weekDay = cal.GetDayOfWeek(DateTime.Now);
  if ((weekDay == DayOfWeek.Saturday) || (weekDay == DayOfWeek.Sunday))
  {
    // Create the exception to return and the return message.
    Exception ex = new Exception("Available on weekdays only");
    IMethodReturn msg = input.CreateExceptionMethodReturn(ex);
    return msg;
  }
  else
  {
    // Do nothing except invoke the next handler.
    return getNext()(input, getNext);
  }
}
```

**Visual Basic**

```
Public Function Invoke(ByVal input As IMethodInvocation, _
                  ByVal getNext As GetNextHandlerDelegate) As IMethodReturn _
       Implements ICallHandler.Invoke

  Dim cal As New GregorianCalendar()
  Dim weekDay As DayOfWeek = cal.GetDayOfWeek(DateTime.Now)
  If (weekDay = DayOfWeek.Saturday) Or (weekDay = DayOfWeek.Sunday) Then

    ' Create the exception to return and the return message.
    Dim ex As New Exception("Available on weekdays only")
    Dim msg As IMethodReturn = input.CreateExceptionMethodReturn(ex)
    Return msg

  Else
    ' Do nothing except invoke the next handler.
    return getNext()(input, getNext)
  End If
End Function
```

# Creating Interception Handler Attributes

Handler attributes allow developers to apply handlers to classes and class members directly, without configuring them in the application configuration file. Developers creating custom handlers may want to provide an attribute for their handlers. To build a custom handler attribute, you create a class that derives from the **HandlerAttribute** base class shown here.

**C#**

```
public abstract class HandlerAttribute : Attribute
{
  /// Derived classes implement this method. When called, it creates a
  /// new call handler as specified in the attribute configuration.
  /// The parameter "container" specifies the IUnityContainer
  /// to use when creating handlers, if necessary.
  /// returns a new call handler object.
  public abstract ICallHandler CreateHandler(IUnityContainer container);

  private int executionorder;
  /// <summary>
  /// Gets or sets the order in which the handler will be executed.
  /// </summary>
  public int Order
  {
    get { return this.executionorder; }
    set { this.order = value; }
  }
}
```

**Visual Basic**

```
Public MustInherit Class HandlerAttribute : Inherits Attribute

  ''' Derived classes implement this method. When called, it creates a
  ''' new call handler as specified in the attribute configuration.
  ''' The parameter container specifies the IUnityContainer
  ''' to use when creating handlers, if necessary.
  ''' Returns a new call handler object.
  Public MustOverride Function CreateHandler(container As IUnityContainer) As
ICallHandler

  Private executionorder As Integer
  ''' <summary>
  ''' Gets or sets the order in which the handler will be executed.
  ''' </summary>
  Public Property Order As Integer
    Get
      Return Me. executionorder
    End Get
```

```
    Set
      Me.order = value
    End Set
  End Property

End Class
```

In your custom attribute class, you must implement one or more constructors that accept values from the attribute, and/or implement named properties that the developer can use to set the properties of the class. Then you simply override the **CreateHandler** abstract method declared within the base class to create and return the required handler class as an **ICallHandler** instance.

## Example Call Handler Attribute

As an example, you could create a call handler attribute for a call handler similar to that described in the topic Creating Interception Policy Injection Call Handlers that prevents invocation of business processes on weekend days. In this case, assume that the handler has a property named **SaturdayOK** that allows you to set it to allow calls to occur on a Saturday. The call handler has two constructors: one that takes a parameter that sets the value of the **SaturdayOK** property to the specified value (true or false), and one that takes no parameters and sets the default value (false) for the **SaturdayOK** property. The following code shows an implementation of the **WeekdayOnlyCallHandlerAttribute**.

**C#**

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Property |
AttributeTargets.Method)]
public class WeekdayOnlyCallHandlerAttribute : HandlerAttribute
{
  private bool allowSaturday;

  public WeekdayOnlyCallHandlerAttribute()
  {
    allowSaturday = false;
  }

  public WeekdayOnlyCallHandlerAttribute(bool SaturdayOK)
  {
    allowSaturday = SaturdayOK;
  }

  public override ICallHandler CreateHandler(IUnityContainer ignored)
  {
    return new WeekdayOnlyCallHandler(allowSaturday, Order);
  }
}
```

**Visual Basic**

```
<AttributeUsage(AttributeTargets.Class Or AttributeTargets.Property Or
AttributeTargets.Method)> _
Public Class WeekdayOnlyCallHandlerAttribute : Inherits HandlerAttribute
```

```
    Private allowSaturday As Boolean

    Public Sub New()
        allowSaturday = False
    End Sub

    Public Sub New(SaturdayOK As Boolean)
        allowSaturday = SaturdayOK
    End Sub

    Public Overrides Function CreateHandler(ignored As IUnityContainer) As
ICallHandler
        Return New WeekdayOnlyCallHandler(allowSaturday, Order)
    End Function

End Class
```

Notice the **AttributeUsage** attribute that specifies where developers can apply the new custom attribute (on a class, a property, or a method), and—in this case—the provision of two constructors. The first (default) constructor uses the default value (false), while the second accepts a value for the **SaturdayOK** property. The **CreateHandler** method override instantiates the **WeekdayOnlyCallHandler** class with the appropriate values and returns this as an **ICallHandler** reference.

# Creating Interception Behaviors

Unity uses the **Interceptor** class to specify how interception happens, and the **InterceptionBehavior** class to describe what to do when an object is intercepted. Unity interception utilizes a behavior pipeline to for the behaviors. The Interception Behavior Pipeline maintains a list of interception behaviors and manages them, calling them in the proper order with the correct inputs.

For information on the details about interception behaviors see Interception with Unity and Behaviors for Interception.

# Deployment and Operations

When you use Unity in your applications, you must deploy the required assemblies with your application or install the assemblies on the target computer in the global assembly cache (GAC). You must include the assembly named **Microsoft.Practices.Unity.dll**. If you are using interception, you will also require the assembly named **Microsoft.Practices.Unity.Interception.dll**.

You can deploy an application that uses Unity in one of two configurations:

- As private assemblies in the application folder hierarchy
- As shared assemblies in any file system location or in the global assembly cache

For advice on using Unity with applications that run in partial trust modes, see Using Unity in Partial Trust Environments.

For advice on updating existing versions of Unity assemblies, see Updating the Unity Assemblies.

When you compile the installed version of Unity source code, the assemblies produced will not be strong named. As a result, they cannot be installed in the global assembly cache, nor will they have the other benefits associated with strong-named assemblies. To learn how to strong name Unity assemblies, see Strong Naming the Unity Assemblies.

Unity throws (and handles internally) **LockSynchronization** exceptions. **LockSynchronization** exceptions may be observed in the debugger output but they are handled internally and no action is required.

## Using XCopy

You are not required to use strong names or to install Unity assemblies in the global assembly cache. You can compile and deploy Unity—without modifying it—in the directory structure of any application that uses Unity. This simplifies deployment because you can use the **xcopy** command to install the entire application, including the Unity assemblies, on the target computer. However, if multiple applications on the same computer use Unity, you must install a copy of the assemblies in each application's folder hierarchy.

## Using the Global Assembly Cache

Alternatively, you can sign Unity assemblies with a strong name key. This will ensure that their names are globally unique and will also provide versioning. If you take this additional step, you can deploy Unity assemblies in a shared location and multiple applications can use them. For example, you could deploy Unity assemblies in the global assembly cache. If you do this, all applications on the computer can use Unity. To learn how to strong name Unity assemblies, see Strong Naming the Unity Assemblies.

## Installing an Assembly in the Global Assembly Cache

You can use one of the following tools to install an assembly in the global assembly cache:

- An installer program, such as the Microsoft Windows Installer version 2.0
- The global assembly cache tool command line utility (Gacutil.exe)
- The .NET Framework configuration tool (Mscorcfg.msc)

For more information about deploying .NET applications, see Deploying.NET Framework-based Applications on MSDN.

### Versioning

All the assemblies provided with Unity are strong named, which provides versioning and naming protection. This means that the name of the binary code is guaranteed to be unique and that the versions being loaded are the ones intended to be used with the application. Because the assemblies are strong named, you can install them in the global assembly cache, where they can be shared by multiple applications on the computer.

The Unity installation package includes compiled assemblies for Unity. It also includes the source code and scripts that you can run to create the complied assemblies. The assemblies you create this way are not strong named. To sign an assembly with a strong name, you must have a public/private key pair. For more information, see Strong-Named Assemblies and Versioning Tutorial.

It is not recommended that you use Unity namespaces because it makes it difficult to identify the source of the code. Instead, you should choose a root namespace that meets your own coding and versioning standards.

# Using Unity in Partial Trust Environments

Unity uses dynamically generated methods to perform injection, and the .NET Framework security model imposes some security limitations that you should be aware of if you want to use Unity in applications that will run in less than full trust environments. The limitation when using Unity in a partial trust environment is that you cannot register and use mappings using the **RegisterType** methods where the target class is **internal** (C#), **Friend** (Visual Basic .NET), **private** (C#), or **Private** (Visual Basic .NET).

For more information about security issues when using dynamically generated Microsoft intermediate language (MSIL) code, see Security Issues in Reflection Emit for .NET 3.5 and Security Issues in Reflection Emit for .NET 2.0.

# Updating the Unity Assemblies

If an upgraded version of Unity becomes available, you can install the new version and have all applications use the updated assembly. However, if the new version introduces compatibility problems for certain applications, you can install the new version in the global assembly cache and

configure some applications to use the updated version, while others continue to use the earlier version.

### Updating Private Assemblies

If a Unity assembly has been deployed as a private assembly, you can deploy the upgrade by just replacing the earlier version of the DLL in the application folder hierarchy with the new one.

> You should keep a copy of the earlier version so that if you experience any compatibility issues with the new assembly, you can revert to the earlier version.

### Updating Shared Assemblies

The easiest way to upgrade a Unity assembly in a shared configuration is to install the updated DLL in the global assembly cache. By default, the common language runtime tries to load the assembly that has the latest build and revision numbers, but the same major and minor version numbers, as the assembly the application was built with. Therefore, if the major and minor version numbers have not changed, adding the later version to the global assembly cache automatically updates all applications that refer to the assembly.

If the major or minor version numbers are incremented, or if the new version causes compatibility problems with existing applications, you can override the default version policy. To specify that a particular version of an assembly is used, edit the application's configuration file (for individual applications), or the machine policy file. Alternatively, you can distribute the new version of the assembly with a publisher policy file to redirect assembly requests to the new version.

# Strong Naming the Unity Assemblies

If you build Unity from the source code, you may decide to apply strong naming to the assemblies. A strong name consists of the assembly's identity—the simple text name, version number, and culture information (if provided)—plus a public key and a digital signature. The strong name is generated from an assembly file (the file that contains the assembly manifest, which in turn contains the names and hashes of all the files that make up the assembly), using the corresponding private key. Signing an assembly with a strong name ensures that its name is globally unique. Assemblies with the same strong name are expected to be identical.

For example, if you intend to share Unity assemblies among several applications, you can install them into the global assembly cache. Each assembly in the global assembly cache must have a globally unique name. You can use a strong name to ensure this. Even if you only use the assemblies within a single application, you can strong name the assemblies to ensure that your application uses the correct version of the assemblies.

Strong names satisfy the following requirements:

- Strong names guarantee name uniqueness by relying on unique key pairs. No one can generate the same assembly name that you can because an assembly generated with one private key has a different name than an assembly generated with another private key.
- Strong names protect the version lineage of an assembly. A strong name can ensure that no one can produce a subsequent version of your assembly. Users can be sure that a version of the assembly they are loading comes from the same publisher that created the version originally provided with the application.
- Strong names provide a strong integrity check. Passing the .NET Framework security checks guarantees that the contents of the assembly have not been changed since it was built. However, note that strong names themselves do not imply a level of trust such as the level provided by, for example, a digital signature and supporting certificate.

---

For information about deploying assemblies into the global assembly cache, see Working with Assemblies and the Global Assembly Cache.

## Using Visual Studio to Strong Name the Unity Assemblies

You  can strong name Unity assemblies with Visual Studio. To sign an assembly with a strong name, you must have a public/private key pair. This public and private cryptographic key pair is used during compilation to create a strong-named assembly. If many developers are using Unity, they should all use the same strong-named assembly. This means that everyone should use a single key pair to sign the assemblies. This key pair should be stored securely.

The first procedure describes how to create a key pair. (You can also use an existing key pair. If you have an existing key pair, you can skip this procedure.) The second procedure describes how to assign the public key to an assembly.

**To create a key pair**

1. At the Visual Studio command prompt, go to the directory that will hold the key pair.
2. To create a key pair, type **sn –k keyfile.snk**.

---

You can also use the **Create Strong Name** dialog box in Visual Studio to create a key pair. To access this dialog box, select a project node in Solution Explorer. On the **Project** menu, click **Properties**. When the Project Designer appears, click the **Signing** tab. On the **Signing** page, select **Sign the assembly**, and then click **New** in the **Choose a strong name key file** drop-down list box.

The next procedure describes how to assign the key to the Unity assembly. Open the Unity.sln solution file and add any of the optional projects you require, such as the Security.Azman project. Follow these steps for each project, including the design projects. You may consider using a batch editor, such as that in Visual Studio, to update each project as they are defined in XML files.

**To assign the public key to a project assembly**

1. In Visual Studio, select the project node in Solution Explorer. On the **Project** menu, click **Properties** (or right-click the project node in Solution Explorer, and then click **Properties**).
2. In the Project Designer, click the **Signing** tab.
3. Select the **Sign the assembly** check box.
4. In the **Choose a strong name key file** drop-down list box, click **Browse**.
5. In the **Select File** dialog box, navigate to the key file you created or enter its path in the **File name** box. Click **Open** to select it.
6. Close the Properties window and save the changes.

# Unity QuickStarts

The instructions in this QuickStart topic are directed at the Silverlight solution. Though there are many similarities, some instructions in this topic, such as references to Program.cs or Program.vb, do not apply to the Silverlight project. The following QuickStart applications demonstrate some of the key features of Unity:

- Walkthrough: The Unity StopLight QuickStart. This QuickStart demonstrates dependency injection techniques. This is the only QuickStart project in the Silverlight solution.
- Walkthrough: The Unity Event Broker Extension QuickStart. This QuickStart provides an example extension for the Unity container. The Silverlight solution does no include the Event Broker QuickStart.

Unity QuickStarts are only available if you install the standalone Unity MSI. The Unity MSI is available at patterns & practices - Unity on CodePlex.

## Building the QuickStarts

The QuickStarts ship as source code, which means that you must compile them before you can run them. You can use Visual Studio 2005 to build the QuickStarts. If you open the QuickStarts in Visual Studio 2008, you will be prompted to upgrade the projects to the Visual Studio 2008 format. If you decide to upgrade them, you may want to keep a copy of the original Visual Studio 2005 projects so that you can refer to them and use them if required.

**To build the Unity QuickStarts**

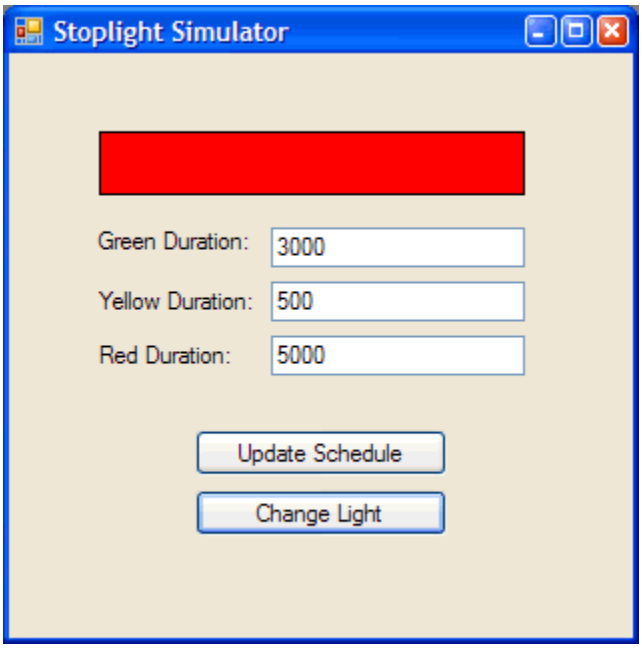
1. Ensure Unity Source Code is installed.
2. Open the source code folder in Windows Explorer or from the Start menu.

3. Open the QuickStarts folder, open the CS folder (for C#) or VB folder (for Visual Basic .NET), and then open the StopLight or EventBroker folder.
4. Double-click the Visual Studio solution file for the QuickStart.
5. Visual Studio opens, displaying the solution file. Click the **Run** button on the toolbar, or press F5, to start the application.

---

# Walkthrough: The Unity StopLight QuickStart

The StopLight QuickStart demonstrates the ways that you can use Unity and the Unity container in your applications. The user interface is a simple Windows Forms application that displays the three colors of a stop light or traffic light—it shows red, yellow, and green, in turn, for specified periods. You can configure the display periods for each color. The following illustration shows the user interface.

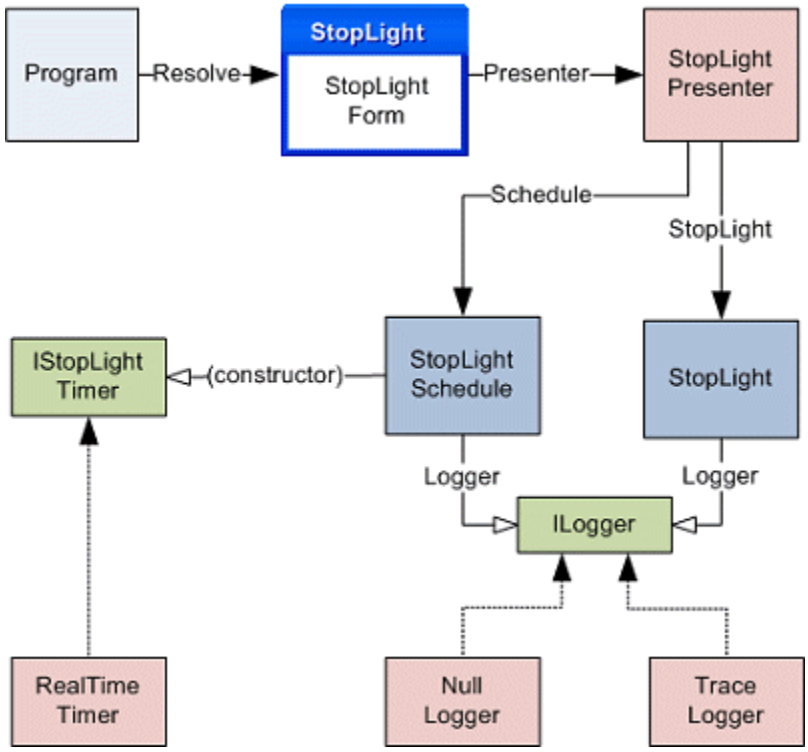



The StopLight QuickStart demonstrates the following features of Unity:

- Registering mappings for types with the container
- Implementing the Model View Presenter pattern by injecting a presenter into the user interface
- Injecting a business component into objects using property (setter) injection
- Implementing a configurable pluggable architecture

---

The following diagram shows the classes and architecture of the StopLight QuickStart application.




## Registering Mappings for Types with the Container

The StopLight QuickStart uses two services that the Unity container maps from interfaces to concrete service implementations. These two services are injected into classes that require them automatically when unity creates instances of the class.

The two interfaces are **ILogger**, which Unity maps to the concrete service class named **TraceLogger**, and **IStoplightTimer**, which Unity maps to the concrete service class named **RealTimeTimer**. The mapping registration occurs in the **Program** file that executes when the application starts. It uses the following code.

**C#**

```
IUnityContainer container = new UnityContainer()
                            .RegisterType<ILogger, TraceLogger>()
                            .RegisterType<IStoplightTimer, RealTimeTimer>();
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer() _
                            .RegisterType(Of ILogger, TraceLogger)() _
                            .RegisterType(Of IStoplightTimer, RealTimeTimer)()
```

This sets up two default (unnamed) mappings so that requests to the container for the **ILogger** interface will return a new instance of the **TraceLogger** component and requests to the container for the **IStoplightTimer** interface will return a new instance of the **RealTimeTimer** component.

The **TraceLogger** component (in the **ServiceImplementations** folder) exposes a single method named **Write** that writes a specified message to the application's current **System.Diagnostics.Trace** instance.

The **RealTimeTimer** component is a single-shot timer implemented in the file **RealTimeTimer** (in the **ServiceImplementations** folder). It exposes a property named **Duration** (in milliseconds), a **Start** method, and it raises the **OnExpired** event when the timer reaches the specified duration.

After registering the required mappings, the code must force dependency injection to be applied to the child objects the application uses. The parent object for the application is the Windows Forms user interface class named **StoplightForm**. To ensure that all dependent objects are injected, the code in the **Program** class instantiates the **StoplightForm** using the **Resolve** method of the Unity container, as shown here.

**C#**

```
Application.Run(container.Resolve<StoplightForm>());
```

**Visual Basic**

```
Application.Run(container.Resolve(Of StoplightForm)())
```

The **Run** method of the **Application** class accepts as a parameter an object that is the class to execute. The **Resolve** method of the Unity container generates an instance of the **StoplightForm** class, applying dependency injection to all child classes to create or reference the objects these child classes require.

## Implementing the Model-View-Presenter Pattern

When the program starts, it loads and displays the user interface—the Windows Form named **StoplightForm**. This form implements the view for the Model View Presenter (MVP) pattern, so it must expose a property that is a reference to its associated presenter. To obtain a reference to its presenter, a class named **StoplightPresenter**, the form uses property (setter) injection by applying the **Dependency** attribute to the property, as show in this code.

**C#**

```
public partial class StoplightForm : Form, IStoplightView
{
  private StoplightPresenter presenter;

  [Dependency]
  public StoplightPresenter Presenter
  {
    get { return presenter; }
    set
    {
      presenter = value;
      presenter.SetView(this);
```

```
    }
  }
  ...
```

**Visual Basic**

```
Public Partial Class StoplightForm
    Inherits Form
    Implements IStoplightView

  Private _presenter As StoplightPresenter

  <Dependency()> _
  Public Property Presenter() As StoplightPresenter
    Get
      Return _presenter
    End Get
    Set(ByVal value As StoplightPresenter)
      _presenter = value
      _presenter.SetView(Me)
    End Set
  End Property
  ...
```

This ensures that Unity will create the presenter before it creates and displays the view (the **StoplightForm** form). The dependent class type for dependency injection is the concrete class **StoplightPresenter**, so the Unity container can create this object without requiring a mapping in the container.

The view exposes event handlers that the presenter can use to manipulate the controls on the form and access the values.

## Injecting a Business Component using Property (Setter) Injection

The presenter for the user interface, the class named **StoplightPresenter**, requires an instance of the class named **Stoplight** that represents the set of colors and methods of a real stop light. The **Stoplight** class uses an enumeration of the three colors (red, yellow, and green), exposes the **Next** method that the application can use to change the colors in a predefined sequence, and raises the **StoplightChangedHandler** event when a color change occurs. The argument for this event is an instance of the class named **LightChangedEventArgs** that exposes the current color.

To obtain an instance of this class, the **StoplightPresenter** exposes it as a property and applies the **Dependency** attribute to the property, as shown in the following code.

**C#**

```
private Stoplight stoplight;

[Dependency]
public Stoplight Stoplight
{
  get { return stoplight; }
```

```
  set { stoplight = value; }
}
```

**Visual Basic**

```
Private _stoplight As StopLight.Logic.Stoplight

<Dependency()> _
Public Property Stoplight() As StopLight.Logic.Stoplight
  Get
    Return _stoplight
  End Get
  Set(ByVal value As StopLight.Logic.Stoplight)
    _stoplight = value
  End Set
End Property
```

The dependent class type for dependency injection is the concrete class, **Stoplight**, so the Unity container can create this object without requiring a mapping in the container.

The presenter also has a dependency on the **StoplightSchedule** class, which maintains references to the **StoplightTimer** and **ILogger**, exposes methods to update the durations for the colors and force a change to the next color, and responds to the **OnTimerExpired** event raised by the **RealTimeTimer** class.

To get a reference to a new instance of the **StoplightSchedule** class, the **StoplightPresenter** exposes it as a property named **Schedule** and applies the **Dependency** attribute to the property, as shown in the following code.

**C#**

```
private StoplightSchedule schedule;

[Dependency]
public StoplightSchedule Schedule
{
  get { return schedule; }
  set { schedule = value; }
}
```

**Visual Basic**

```
Private _schedule As StoplightSchedule

<Dependency()> _
Public Property Schedule() As StoplightSchedule
  Get
    Return _schedule
  End Get
  Set(ByVal value As Stoplight)
    _schedule = value
  End Set
End Property
```

Again, the dependent type is a concrete class, in this case the **StoplightSchedule** class, and so the Unity container can create this object without requiring a mapping in the container.

## Implementing a Configurable Pluggable Architecture

The **Stoplight** and **StoplightSchedule** classes have dependencies that are not concrete classes. Both of these classes have a dependency on one of the concrete implementations of the **ILogger** interface—either the **NullLogger** or the **TraceLogger**. At run time, the code can choose which concrete class to instantiate. It can use any class that implements the **ILogger** interface, which defines just the single method **Write** that takes a message and writes it to the appropriate output.

This is an example of a pluggable architecture. The actual class chosen at run time, and injected into the application, depends on the mapping in the Unity container. The QuickStart maps the **ILogger** interface to the concrete type **TraceLogger** using the method **RegisterType<ILogger, TraceLogger>()** in the main **Program** class. Therefore, to use a different concrete implementation, the developer just has to change the mapping. For example, to use a new class **ReallyFastLogger**, the developer would just change the mapping to **RegisterType<ILogger, ReallyFastLogger>()**.

In addition, by using a configuration file to populate the Unity container or reading the configuration information from another source such as a database, the developer allows the user to change the actual concrete logging class without requiring recompilation of the application code.

The **Stoplight** and **StoplightSchedule** classes force injection of the currently mapped logger class by exposing it as a property named **Logger** (of type **ILogger**) that has the **Dependency** attribute applied, as shown in the following code.

**C#**

```
private ILogger logger = new NullLogger();

[Dependency]
public ILogger Logger
{
  get { return logger; }
  set { logger = value; }
}
```

**Visual Basic**

```
Private _logger As ILogger = New NullLogger()

<Dependency()> _
Public Property Logger As ILogger
  Get
    Return _logger
  End Get
  Set(ByVal value As ILogger)
    _logger = value
  End Set
End Property
```

!!K:UnityQS1!!

# Walkthrough: The Unity Event Broker Extension QuickStart

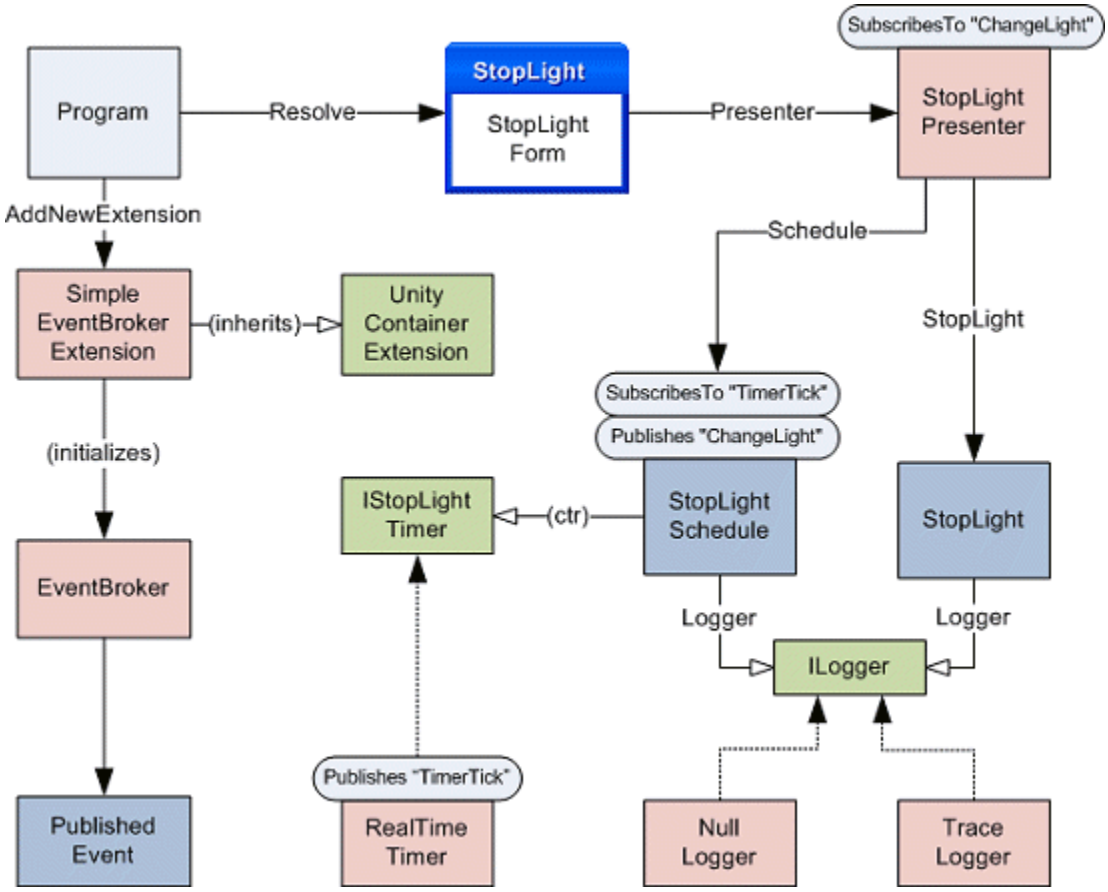
The Event Broker Extension QuickStart demonstrates how you can extend the Unity container by adding a custom extension. The QuickStart implements an event broker for the container as a container extension and demonstrates the new extension using the StopLight application discussed in Walkthrough: The Unity StopLight QuickStart.

The Event Broker Extension QuickStart contains five projects:

- **SimpleEventBroker**. This project implements a simple publish and subscribe mechanism that supports multiple event publishers and multiple subscribers.
- **EventBrokerExtension**. This project implements the custom container extension that allows applications to publish and subscribe to events using attributes or explicitly using code.
- **StopLight**. This project is basically the same as that described in the Unity StopLight QuickStart, but it uses the custom container extension to manage the publishing of, and subscription to, two events within the application.
- **Tests.EventBrokerExtension**. A test fixture for the **EventBrokerExtension.**
- **Tests.SimpleEventBroker**. A test fixture for the **SimpleEventBroker.**

For information about how you can create and use custom container extensions, see Creating and Using Container Extensions.

The following diagram shows the classes and architecture of the Event Broker Extension QuickStart.




If you compare this diagram to the structure of the StopLight QuickStart shown in Walkthrough: The Unity StopLight QuickStart, you can see that the EventBroker Extension QuickStart has the following additional features:

- The **Program** class, which registers the type mappings in the container and calls the **Resolve** method to instantiate the main **StopLight** form, also adds the **SimpleEventBrokerExtension** to the container.
- The **SimpleEventBrokerExtension**, which inherits from the **UnityContainerExtension** base class, creates an instance of the **EventBroker** class that implements the publish and subscribe pattern for distributed events.
- The **EventBroker** class creates an instance of the **PublishedEvent** class that provides the facilities for maintaining a list of event subscriptions and raising events to registered subscribers.
- The **StopLightPresenter**, **StopLightSchedule**, and **RealTimeTimer** classes include attributes that register event publications and subscriptions with the **SimpleEventBrokerExtension** class.

---

The Event Broker Extension QuickStart demonstrates the following features of Unity and the custom container extension mechanism:

- Creating the custom Unity container extension
- Adding an extension to the Unity container at run time

- Using the example Event Broker Extension

## Creating a Custom Unity Container Extension

A Unity container extension is a custom class that inherits from the **UnityContainerExtension** base class and implements extra functionality beyond that available from the container. The extension can access the container and receive notification of registrations taking place within the container. This section describes the significant features of the **EventBrokerExtension** project.

The class **SimpleEventBrokerExtension** inherits **UnityContainerExtension** and creates a new instance of the **EventBroker** class from the **SimpleEventBroker** project. It then overrides the **Initialize** method of the **UnityContainerExtension** base class.

The code in the **Initialize** method of the **SimpleEventBrokerExtension** class, which follows, adds the new **EventBroker** it creates to the container's **Locator** so that other classes can locate and reference it easily, and then it adds two strategies to the Unity build pipeline. It adds a reflection strategy to the **PreCreation** stage that will detect the two attributes that the extension uses (the **PublishesAttribute** and the **SubscribesToAttribute**) and a wire-up strategy to the **Initialization** stage that registers publishers and subscriber delegates with the **EventBroker**. Finally, it exposes the new **EventBroker** it created as a property.

**C#**

```
public class SimpleEventBrokerExtension : UnityContainerExtension,
                                          ISimpleEventBrokerConfiguration
{
  private readonly EventBroker broker = new EventBroker();

  protected override void Initialize()
  {

    Context.Container.RegisterInstance(
            broker, new ExternallyControlledLifetimeManager());

Context.Strategies.AddNew<EventBrokerReflectionStrategy>(UnityBuildStage.PreCreat
ion);

Context.Strategies.AddNew<EventBrokerWireupStrategy>(UnityBuildStage.Initializati
on);
  }

  public EventBroker Broker
  {
    get { return broker; }
  }
}
```

**Visual Basic**

```
Public Class SimpleEventBrokerExtension
    Inherits UnityContainerExtension
```

```
    Implements ISimpleEventBrokerConfiguration

  Private _broker As New EventBroker()

  Protected Overloads Overrides Sub Initialize()
    Context.Container.RegisterInstance( _
            broker, New ExternallyControlledLifetimeManager())
    Context.Strategies.AddNew(Of
EventBrokerReflectionStrategy)(UnityBuildStage.PreCreation)
    Context.Strategies.AddNew(Of
EventBrokerWireupStrategy)(UnityBuildStage.Initialization)
  End Sub

  Public ReadOnly Property Broker() As EventBroker _
         Implements ISimpleEventBrokerConfiguration.Broker
    Get
      Return _broker
    End Get
  End Property

End Class
```

The **EventBroker** class holds a **Dictionary** containing entries that map event names to publishers of that event and exposes methods to register publishers and subscribers (named **RegisterPublisher** and **RegisterSubscriber**). It also exposes methods to unregister publishers and subscribers and to get a list of publishers or subscribers for a specified event name. Much of the functionality for these methods is in the **PublishedEvent** class, which stores and exposes lists of all publishers and subscribers.

The wire-up strategy added to Unity by the **Initialize** method in the previous listing calls the **RegisterPublisher** and **RegisterSubscriber** methods of the **EventBroker** class. This means that, when Unity detects a **PublishesAttribute** or **SubscribesToAttribute** in a class that it creates, it automatically registers the class or member as a publisher or a subscriber in the **EventBroker**.

If the registration is for a publisher, the **EventBroker** calls the **AddPublisher** method of the **PublishedEvent** class. This method adds the new publisher to the list of publishers and wires up an event handler named **OnPublisherFiring** to the published event. Therefore, when the publisher raises the event, the handler in the **PublishedEvent** class can iterate through the list of subscriber delegates and invoke each one, as shown in the following code.

**C#**

```
private void OnPublisherFiring(Object sender, EventArgs e)
{
  foreach(EventHandler subscriber in subscribers)
  {
    subscriber(sender, e);
  }
}
```

**Visual Basic**

```
Private Sub OnPublisherFiring(sender As Object, e As EventArgs)
  For Each subscriber As EventHandler in _subscribers
```

```
    subscriber.Invoke(sender, e)
  Next
End Sub
```

## Adding an Extension to the Unity Container at Run Time

After you create a custom container extension, you must add it to the Unity container. You can do this by compiling the extension and specifying the type and assembly name in the configuration file for Unity. For details on how to configure Unity container extensions, see Configuring Unity.

However, the EventBroker Extension QuickStart adds the custom **SimpleEventBrokerExtension** at run time by calling a method of the Unity container class. The StopLight application **Program** class that initializes the application and loads the main form creates a new UnityContainer instance and registers the concrete types that map to the **ILogger** and **IStoplightTimer** classes. It also calls the **AddNewExtension** method, specifying the **SimpleEventBrokerExtension** class, as shown in the following code.

**C#**

```
IUnityContainer container = new UnityContainer()
           .AddNewExtension<SimpleEventBrokerExtension>()
           .RegisterType<ILogger, TraceLogger>()
           .RegisterType<IStoplightTimer, RealTimeTimer>();
```

**Visual Basic**

```
Dim container As IUnityContainer = New UnityContainer() _
           .AddNewExtension(Of SimpleEventBrokerExtension)() _
           .RegisterType(Of ILogger, TraceLogger)() _
           .RegisterType(Of IStoplightTimer, RealTimeTimer)()
```

The container automatically instantiates the extension and calls the **Initialize** method that you saw in the section Creating a Custom Unity Container Extension of this topic.

## Using the Example Event Broker Extension

As shown in the previous diagram, the EventBroker Extension QuickStart uses the custom **SimpleEventBrokerExtension** container extension to implement the publish and subscribe pattern for two events:

- The **RealTimeTimer** class publishes an event named **TimerTick** that it raises when the timer reaches zero. The **StoplightSchedule** class subscribes to this event. In the event handler, it updates its index to the **lightTimes** array (an array of **TimeSpan** values for the duration of the colors), sets the new duration for the light, and starts the timer running again.
- The **StoplightSchedule** class publishes an event named **ChangeLight** that it raises before it changes the timer duration and restarts the timer. The **StoplightPresenter** class subscribes to this event. In the event handler, it calls the **Next** method of the **StopLight** class to change the color of the light and write a message to the **TraceLogger**.

To indicate that it publishes the **TimerTick** event, the **RealTimeTimer** class uses the **Publishes** attribute, specifying the name of the event publication, as shown in the following code.

**C#**

```
[Publishes("TimerTick")]
public event EventHandler Expired;
private void OnTick(Object sender, EventArgs e)
{
  timer.Stop();
  OnExpired(this);
}
```

**Visual Basic**

```
<Publishes("TimerTick")> _
Public Event Expired As EventHandler Implements IStoplightTimer.Expired
Private Sub OnTick(ByVal sender As Object, e As EventArgs)
  timer.[Stop]()
  OnExpired(Me)
End Sub
```

The **OnExpired** method simply checks that there is an event handler instance and raises the event, as shown here.

**C#**

```
protected virtual void OnExpired(object sender)
{
  EventHandler handlers = Expired;
  if(handlers != null)
  {
    handlers(this, EventArgs.Empty);
  }
}
```

**Visual Basic**

```
Protected Overridable Sub OnExpired(ByVal sender As Object)
  Dim handlers As EventHandler = ExpiredEvent
  If Not handlers Is Nothing Then
    RaiseEvent Expired(Me, EventArgs.Empty)
  End If
End Sub
```

The **StoplightSchedule** class subscribes to the **TimerTick** event by applying the **SubscribesTo** attribute to a suitable event handler.

**C#**

```
[SubscribesTo("TimerTick")]
public void OnTimerExpired(Object sender, EventArgs e)
{
  EventHandler handlers = ChangeLight;
  if(handlers != null)
  {
    handlers(this, EventArgs.Empty);
  }
```

```
  currentLight = ( currentLight + 1 ) % 3;
  timer.Duration = lightTimes[currentLight];
  timer.Start();
}
```

**Visual Basic**

```
<SubscribesTo("TimerTick")> _
Public Sub OnTimerExpired(ByVal sender As Object, ByVal e As EventArgs)
  Dim handlers As EventHandler = ChangeLightEvent
  If Not handlers Is Nothing Then
    RaiseEvent ChangeLight(Me, EventArgs.Empty)
  End If
  currentLight = ( currentLight + 1 ) Mod 3
  _timer.Duration = lightTimes(currentLight)
  _timer.Start()
End Sub
```

The preceding code shows that the **OnTimerExpired** event handler raises an event named **ChangeLight** when it receives the **TimerTick** event. **ChangeLight** is the local name of the event, but the **StoplightSchedule** class also publishes the event using this name, as shown in the following code.

**C#**

```
[Publishes("ChangeLight")]
public event EventHandler ChangeLight;
```

**Visual Basic**

```
<Publishes("ChangeLight")> _
Public Event ChangeLight As EventHandler
```

Finally, the **StoplightPresenter** class subscribes the **ChangeLight** event using an event handler named **OnScheduledLightChange**. Inside the event handler, it calls the **Next** method of the StopLight class, as shown in the following code.

**C#**

```
[SubscribesTo("ChangeLight")]
public void OnScheduledLightChange(Object sender, EventArgs e)
{
  stoplight.Next();
}
```

**Visual Basic**

```
<SubscribesTo("ChangeLight")> _
Public Sub OnScheduledLightChange(ByVal sender As Object, ByVal e As EventArgs) _
  _stoplight.Next()
End Sub
```

The code examples in this section demonstrate how useful the custom **SimpleEventBrokerExtension** is for working with distributed events and how easy it is to publish and subscribe to events when using Unity to generate instances of the classes used in the application.

!!K:UnityQS2!!