‹ Previous Home

Lifetime Managers in Unity Container

Unity container manages the lifetime of objects of all the dependencies that it resolves using lifetime managers.

Unity container includes different lifetime managers for different purposes. You can specify the lifetime manager in the `RegisterType()` method at the time of registering type-mapping. For example, the following code snippet shows how to register a type-mapping with `TransientLifetimeManager`.

```
var container = new UnityContainer()
                      .RegisterType<ICar, BMW>(new TransientLifetimeManager());
```

The following table lists all the lifetime managers:

TransientLifetimeManager	Creates a new object of the requested type every time you call the Resolve or ResolveAll method.
ContainerControlledLifetimeManager	Creates a singleton object first time you call the Resolve or ResolveAll method and then returns the same object on subsequent Resolve or ResolveAll calls.
HierarchicalLifetimeManager	Same as the ContainerControlledLifetimeManager, the only difference is that the child container can create its own singleton object. The parent and child containers do not share the same singleton object.
PerResolveLifetimeManager	Similar to the TransientLifetimeManager, but it reuses the same object of registered type in the recursive object graph.
PerThreadLifetimeManager	Creates a singleton object per thread. It returns different objects from the container on different threads.
ExternallyControlledLifetimeManager	It maintains only a weak reference of the objects it creates when you call the Resolve or ResolveAll method. It does not maintain the lifetime of the strong objects it creates, and allows you or the garbage collector to control the lifetime of the objects. It enables you to create your own custom lifetime manager.

Let's understand each lifetime manager using the following example classes.

Example: C# Copy

```
public interface ICar
{
    int Run();
}

public class BMW : ICar
{
    private int _miles = 0;

    public int Run()
    {
        return ++_miles;
    }
}

public class Ford : ICar
{
    private int _miles = 0;
    public int Run()
    {
        return ++_miles;
    }
}

public class Audi : ICar
{
    private int _miles = 0;

    public int Run()
    {
        return ++_miles;
    }
}

public class Driver
{
    private ICar _car = null;

    public Driver(ICar car)
    {
        _car = car;
    }

    public void RunCar()
    {
        Console.WriteLine("Running {0} - {1} mile ", _car.GetType().Name, _car.Run());
    }
}
```

TransientLifetimeManager

The `TransientLifetimeManager` is the default lifetime manager. It creates a new object of the requested type every time you call the `Resolve()` or ResolveAll() method.

Example: TransientLifeTimeManager - C# Copy

```
var container = new UnityContainer()
                   .RegisterType<ICar, BMW>();

var driver1 = container.Resolve<Driver>();
driver1.RunCar();

var driver2 = container.Resolve<Driver>();
driver2.RunCar();
```

Output:

```
Running BMW - 1 Mile
Running BMW - 1 Mile
```

In the above example, Unity container will create two new instances of the `BMW` class and will inject into the `driver1` and `driver2` objects. This is because the default lifetime manager is `TransientLifetimeManager`, which creates a new dependent object every time you call the `Resolve()` or `ResolveAll()` method. You can specify the lifetime manager when registering the type using the `RegisterType()` method.

The following example will display the same output as in above example because TransientLifetimeManager is the **default** manager, if not specified.

```
var container = new UnityContainer()
                   .RegisterType<ICar, BMW>(new TransientLifetimeManager());

var driver1 = container.Resolve<Driver>();
driver1.RunCar();

var driver2 = container.Resolve<Driver>();
driver2.RunCar();
```

Output:

```
Running BMW - 1 Mile
Running BMW - 1 Mile
```

ContainerControlledLifetimeManager

Use the `ContainerControlledLifetimeManager` when you want to create a singleton instance.

Example: ContainerControlledLifetimeManager - C# Copy

```
var container = new UnityContainer()
                   .RegisterType<ICar, BMW>(new ContainerControlledLifetimeManager());

var driver1 = container.Resolve<Driver>();
driver1.RunCar();

var driver2 = container.Resolve<Driver>();
driver2.RunCar();
```

Output:

```
Running BMW - 1 mile
Running BMW - 2 mile
```

In the above example, we specified `ContainerControlledLifetimeManager` in the `RegisterType()` method. So, Unity container will create a single instance of the `BMW` class and inject it in all the instances of `Driver`.

HierarchicalLifetimeManager

The `HierarchicalLifetimeManager` is the same as the `ContainerControlledLifetimeManager`, except that if you create a child container then it will create its own singleton instance of the registered type and will not share the instance with the parent container.

Example: HierarchicalLifetimeManager - C# Copy

```
var container = new UnityContainer()
            .RegisterType<ICar, BMW>(new HierarchicalLifetimeManager());

var childContainer = container.CreateChildContainer();

var driver1 = container.Resolve<Driver>();
driver1.RunCar();

var driver2 = container.Resolve<Driver>();
driver2.RunCar();
```

```
var driver3 = childContainer.Resolve<Driver>();
driver3.RunCar();

var driver4 = childContainer.Resolve<Driver>();
driver4.RunCar();
```

Output:

```
Running BMW - 1 mile
Running BMW - 2 mile
Running BMW - 1 Mile
Running BMW - 2 Mile
```

As you can see, both container and childContainer have their own singleton instance of `BMW`.

Visit [Understand Lifetime Managers](https://www.tutorialsteacher.com) to learn more about it.

Share Tweet Share Whatsapp

Previous Home