Module 03:

"Dependency Injection"

Agenda

- Classifying Dependencies
- Dependency Injection Containers
- Summary

Volatile Dependencies

- Out-of-process or unmanaged resources
- Nondeterministic resources
- Resources to be
  - Replaced
  - Intercepted
  - Decorated
  - Mocked

Examples of Volatile Dependencies

- Databases
- File system
- Web services
- Security contexts
- Message Queues
- **`System.Random`** (or similar)

Stable Dependencies

- A dependency is *stable* if it's not volatile...!

```
interface IUserRoleParser
{
    bool Parse(string role);
}
```

```
class DisplayState
{
    public string Name { get; }
    public string DisplayText { get; }

    public DisplayState(MovieDto movie) { ... }
}
```

Discussion Point:

Which type of dependency is **`Computation`**?

Dependency Injection applies exclusively to Volatile Dependencies.

Don't inject Stable Dependencies!

Agenda

- Classifying Dependencies
- **Dependency Injection Containers**
- Summary

Pattern: Pure DI (a.k.a. Poor Man's DI)

- *Pure DI is the practice of applying DI without a DI Container.*

- Outline
  - This is essentially what we have been doing throughout the workshop
  - Compose object graphs manually at Composition Root

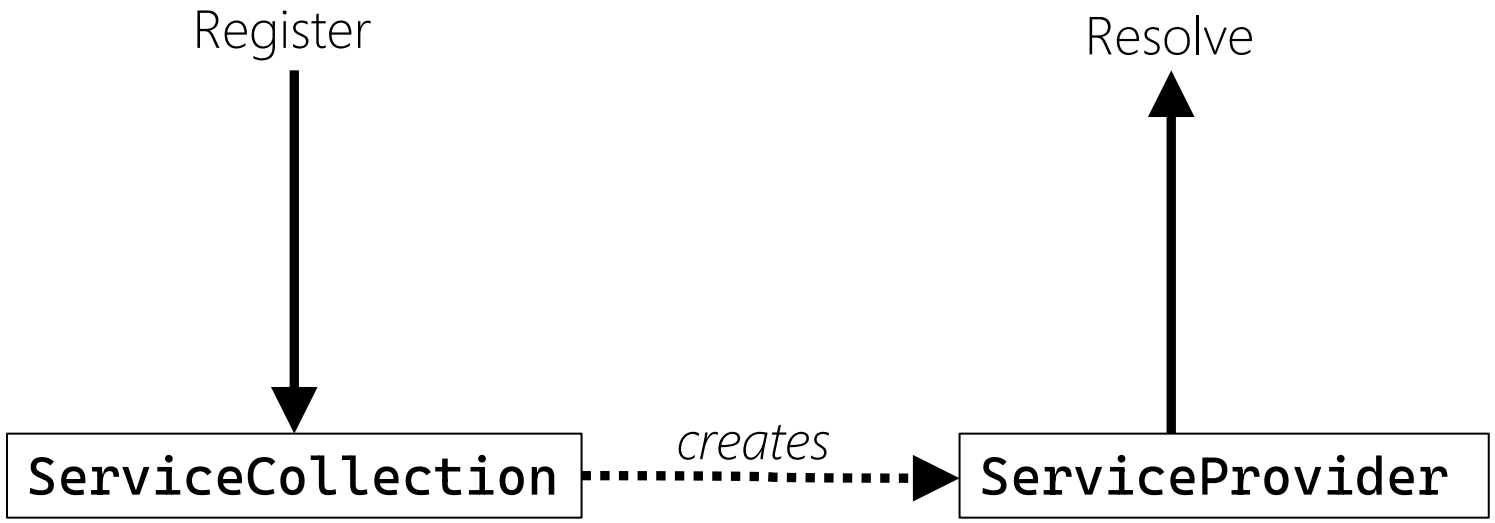
- See:
  "Dependency Injection Principles, Practices, and Patterns"
  Steven van Deursen and Mark Seemann (2019)

Dependency Injection Containers

Autofac
Simple Injector
Microsoft.Extensions.DependencyInjection
Unity
Ninject
Castle Windsor
Spring.NET
SimpleIoc
…

Don’t create your own! ☺

Microsoft.Extensions.DependencyInjection



Register and Resolve

```
IServiceCollection services = new ServiceCollection();
services
    .AddTransient<IParser, JsonParser>()
    .AddTransient<IFormatter, JsonFormatter>()
    .AddSingleton<IWriteStorage, ConsoleStorage>()
    .AddSingleton<IReadStorage>(sp ⇒
        new WebStorage(Url)
    )
    .AddSingleton<StockAnalyzer>()
    ;
```

```
IServiceProvider serviceProvider = services.BuildServiceProvider(true);
StockAnalyzer analyzer = serviceProvider.GetRequiredService<StockAnalyzer>();
```

Definition: Lifestyles

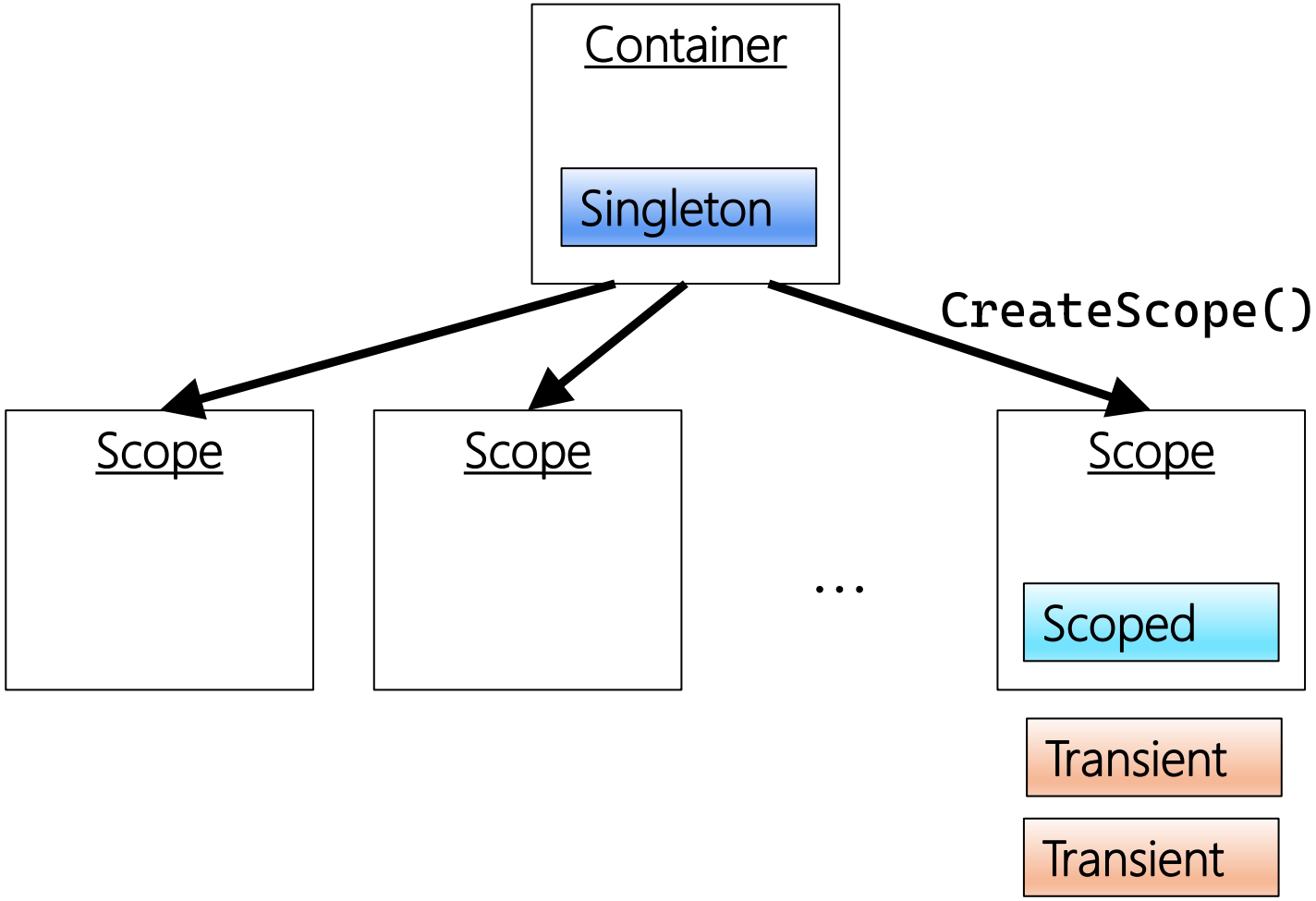
- *A Lifestyle is a formalized way of describing the intended lifetime of a dependency.*

- Transient ~ New instance at every resolve
- Singleton ~ Only one instance exists per container (*)
- Scoped ~ New instance at every scope

- Note: Lifetime is associated with registrations

Containers and Scopes

- Dependencies should be resolved from Scopes – not the Container itself



Creating Scopes

```
IServiceProvider serviceProvider = services.BuildServiceProvider(true);

using IServiceScope scope = serviceProvider.CreateScope();
StockAnalyzer analyzer = scope.serviceProvider.GetRequiredService<StockAnalyzer>();
```

Pattern: Register-Resolve-Release

- *Always do a sequence of three things with a container:*
  - *Register components with the container*
  - *Resolve root components*
  - *Release components from the container.*

- Outline
  - RRR captures the best practice of container use in the Composition Root only!

- See:
  https://blog.ploeh.dk/2010/09/29/TheRegisterResolveReleasepattern/
  Mark Seemann (2010)

Summary

- Classifying Dependencies
- Dependency Injection Containers


WINCUBATE
Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD
Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net
Ringgårdsvej 4A
8270 Højbjerg
Denmark