C# 10 Dependency Injection

Getting Started with a Product Importer

Henry Been
Independent DevOps & Azure Architect

@henry_been www.henrybeen.nl

Version Check

This version was created by using:

- C# 10
- .NET 6.0.101
- Visual Studio 2022 Community Edition 17.0.5

Version Check

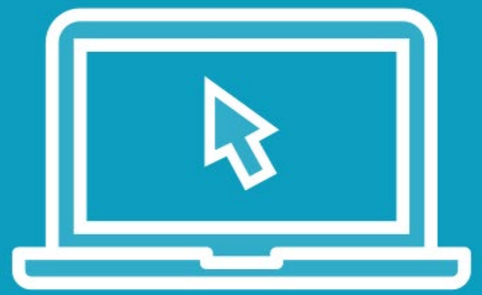
This course is 100% applicable to:

- C# 10
- Any version of .NET 6
- Any version of Visual Studio 2022

Overview

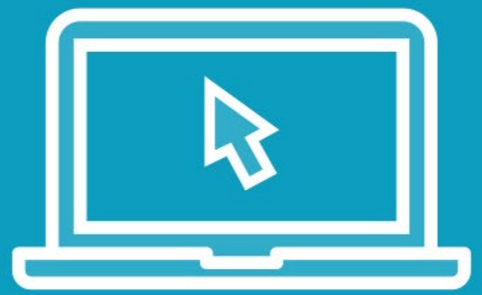
- Why you need dependency injection (DI)
- How to add DI to your application
- Different DI containers available
- Theoretical background

Demo



Why you need dependency injection

Demo



Using a dependency injection container for the Product Importer

No longer calling constructors

Container calls the constructors

Not concerned with ordering registrations

Not concerned with dependencies of each type

About Dependency Injection Containers

Well-known DI Containers

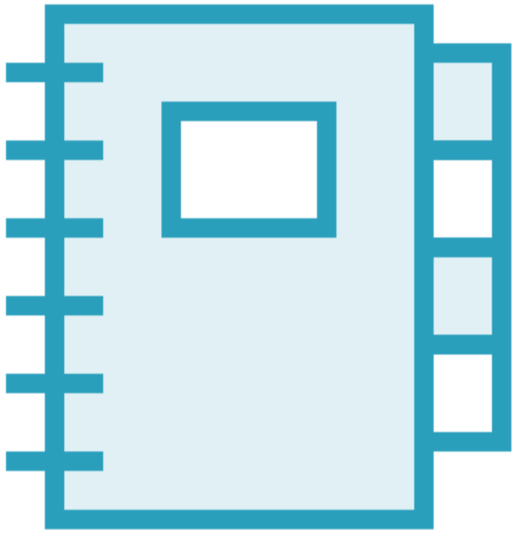
Autofac

Ninject

Unity (discontinued)

Microsoft.Extensions.DependencyInjection (default for .NET Core and .NET 5 and up)

Working with a DI Container

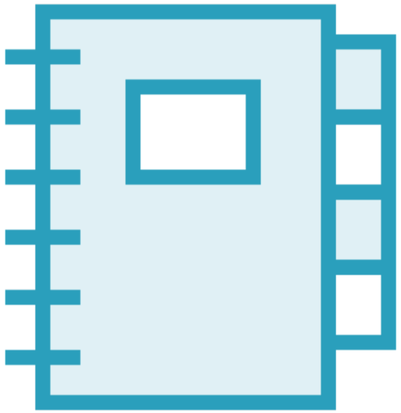


Registration phase

You register types in the container, so it knows of their existence and when to construct them

Resolving phase

The container is responsible for instantiating types and providing them when requested



Register types for later use

Indirection through service type and implementing type

Choose a lifetime

```
serviceCollection.AddTransient<IProductSource, ProductSource>();

serviceCollection.AddSingleton<ProductImporter>();
```

Registering types

When registering types you specify the *lifetime*, the requested *service type*, and the *implementing type*. If these types are the same, you provide it once.

Resolves and creates types directly

Provides dependencies of types you work with

Provides dependencies to dependencies of the types you work with

Manage the lifetimes of types

```
host.Services.GetRequiredService<ProductImporter>();
```

Resolving types

When resolving a type, you request an instance of a *service type*. The container will find the *implementing type*, instantiate it if needed, and return it to you.

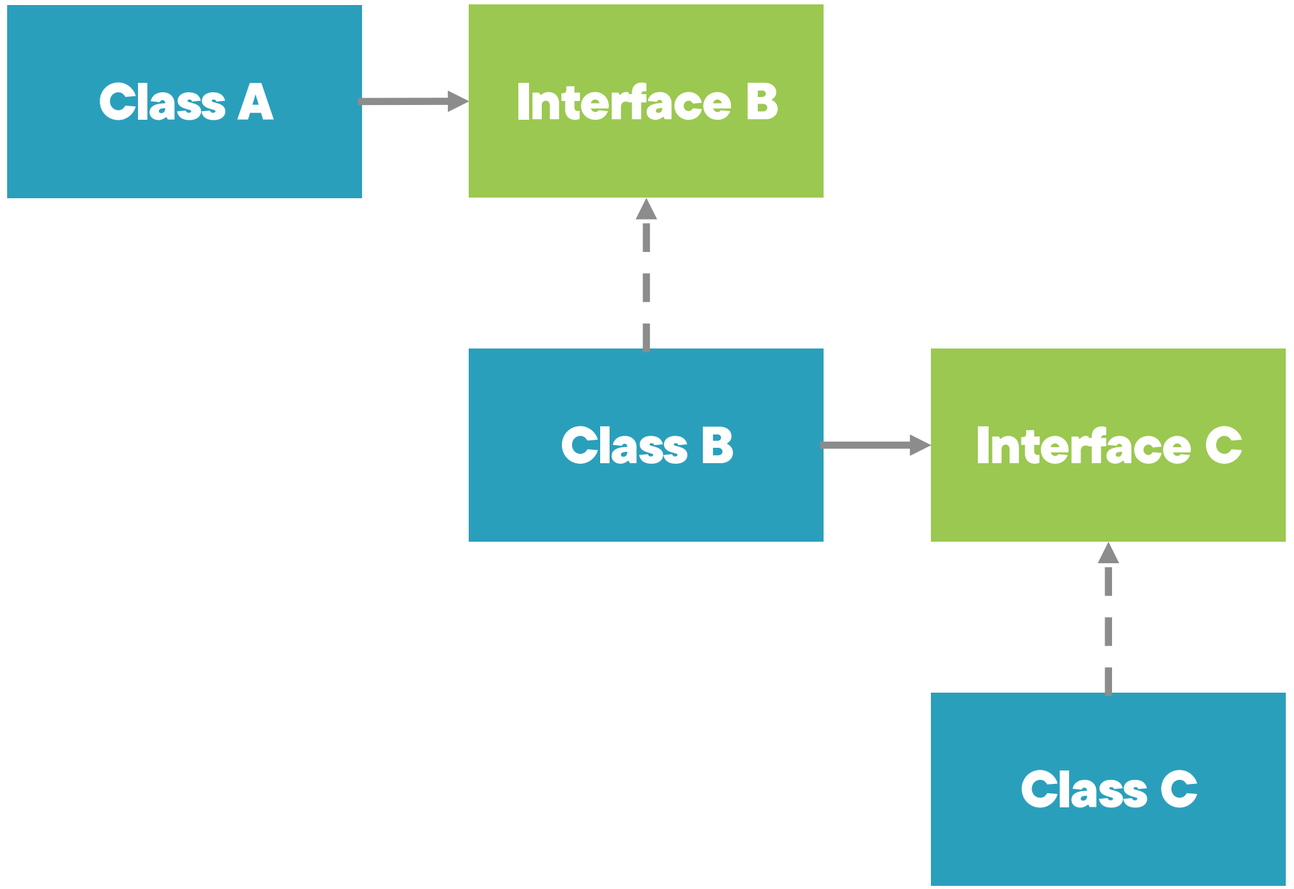
If the *implementing type* has dependencies, they are provided to the *implementing type* as well.

Dependency Inversion and Inversion of Control

Dependency inversion: High-level modules should not depend on low-level modules

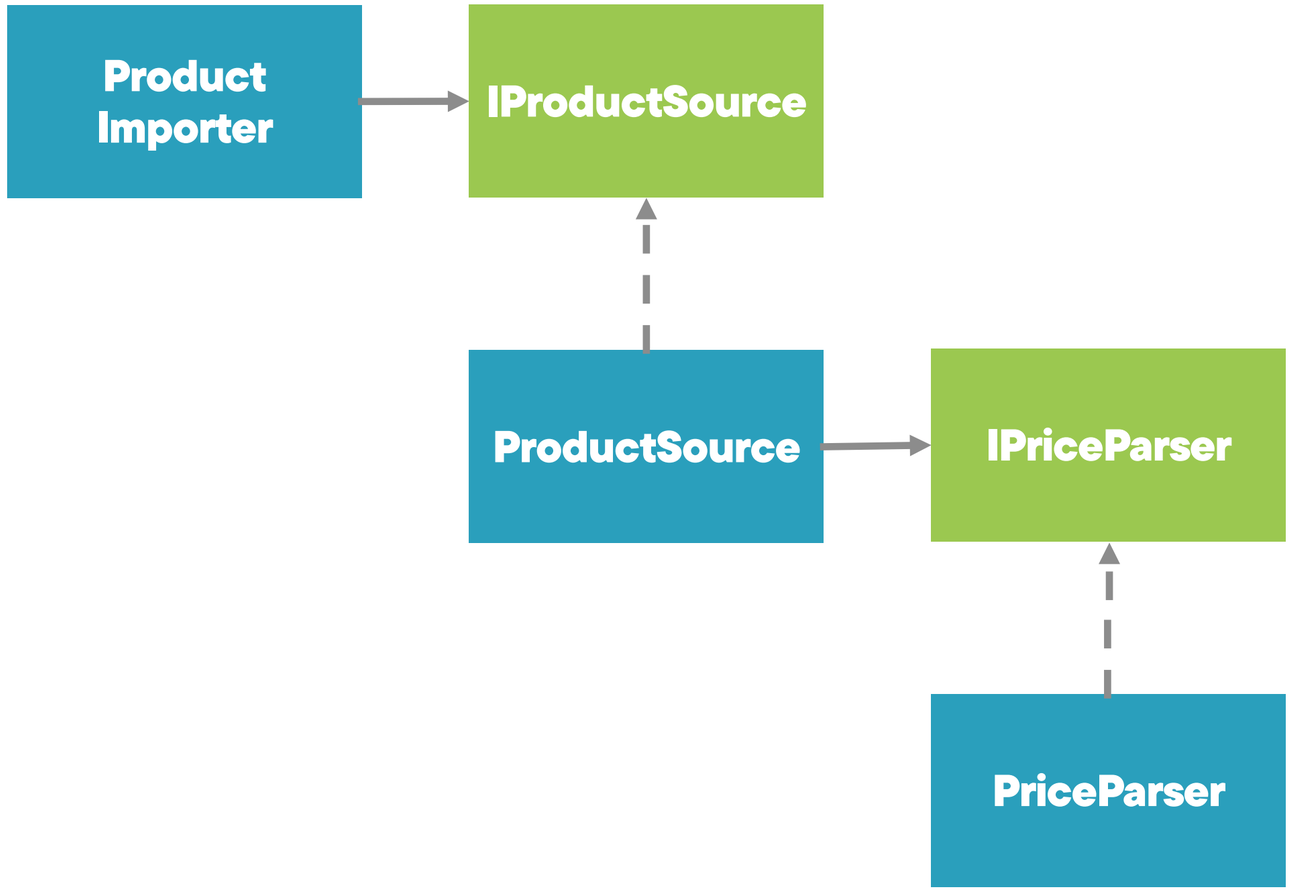
Dependency Inversion

Dependency Inversion



Dependency Inversion - Example

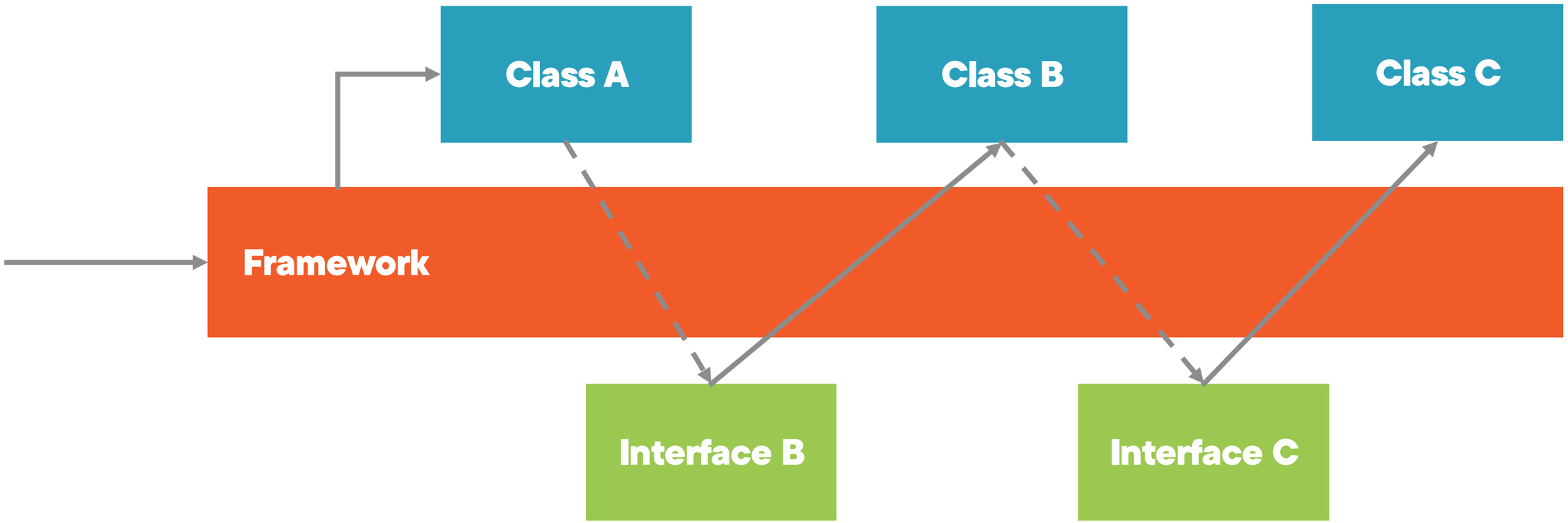
Dependency Inversion - Example



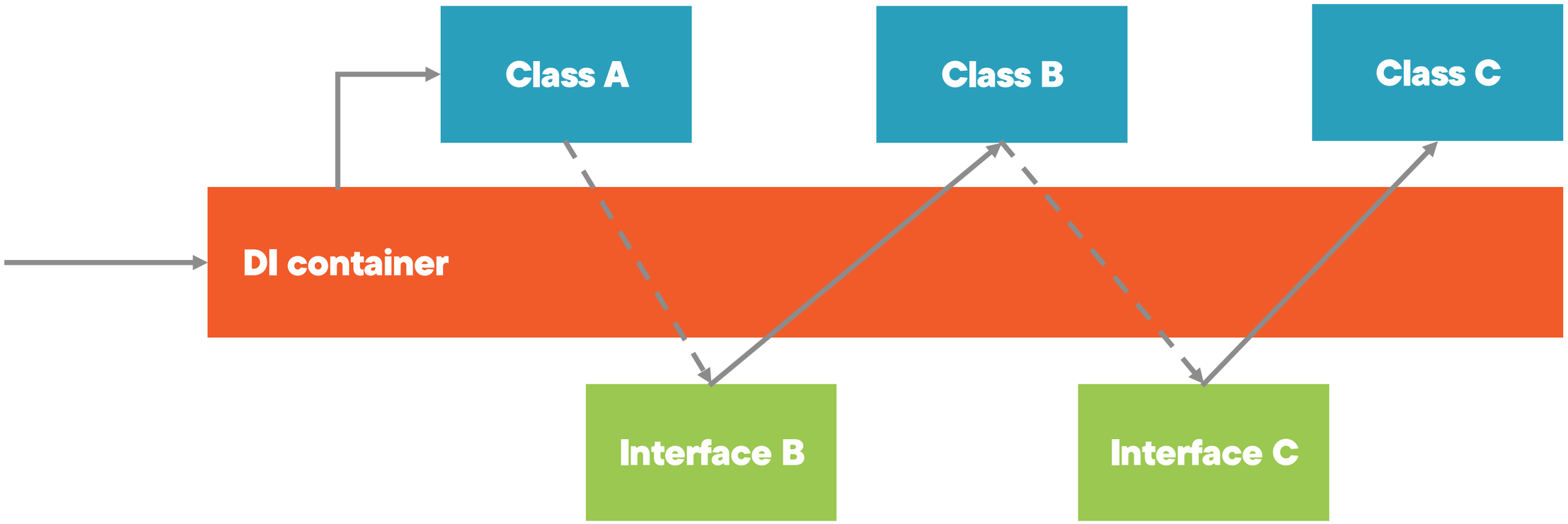
Inversion of Control:
A framework controls which code
is executed next, not your code

Traditional Flow

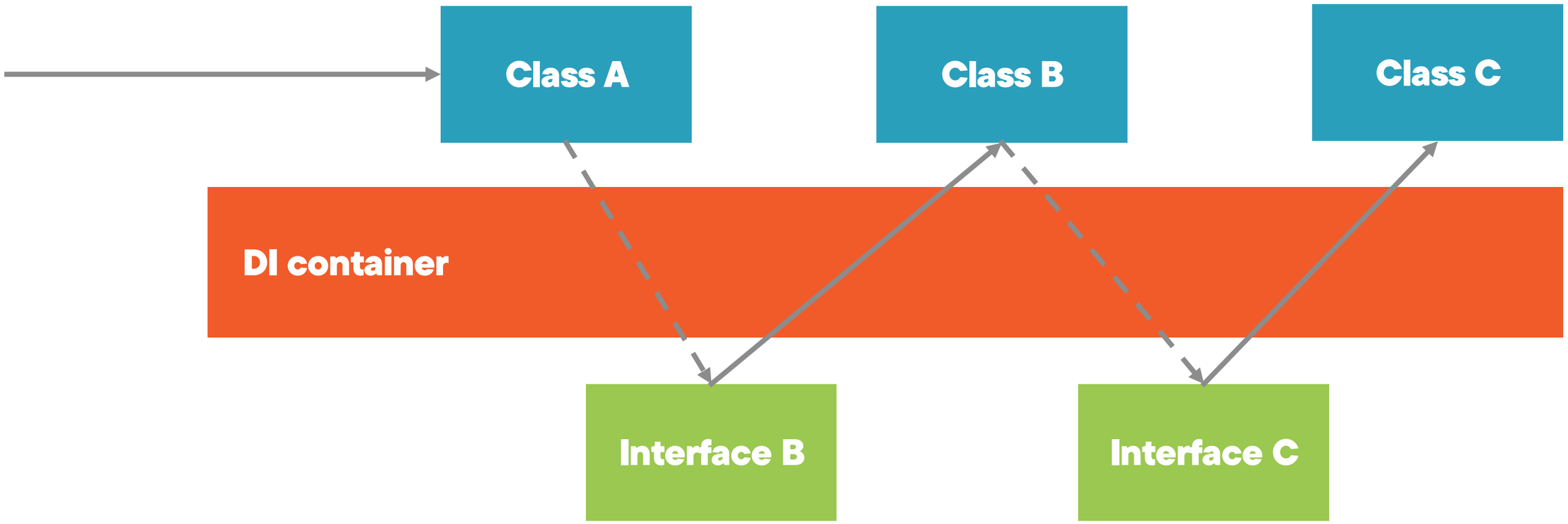
Inversion of Control



Inversion of Control



Inversion of Control



Creating Maintainable Solutions Using Dependency Injection

Creating Maintainable Solutions Using DI

Design classes that have a single responsibility

Depend upon interfaces, not classes

Interfaces are “owned” by the consumer

Have no assumptions about implementations

Appy dependency inversion and IoC

Summary

Why you need dependency injection

Dependency injection container

- Add to your application
- Register types
- Resolve dependencies

Background

- Dependency Inversion
- Inversion of Control
- Decoupling

Up Next:
Lifetime Management