ASP.NET Core Activity 7

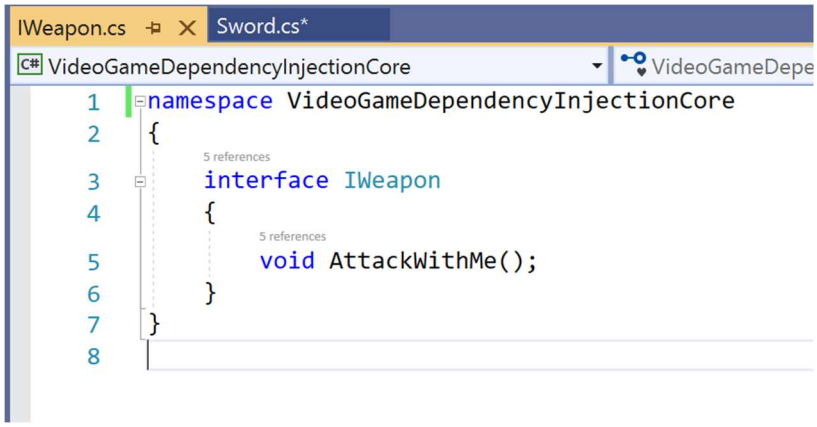
Part 1 dependency injection

Overview

In the following activity, we are going to build a console application that purposely includes many class dependencies. The Hero will have a weapon. The gun (weapon) will have bullets. We will first create dependencies by creating instances of the items inside of its parent, and then show an example of dependency injection.

1. Create a new **Console App (Core)** application**.**

2. Create a new interface class called **IWeapon**.



3. Create an interface called **IHero**.

IHero.cs | IWeapon.cs | Sword.cs*

VideoGameDependencyInjectionCore

```
namespace VideoGameDependencyInjectionCore
{
    2 references
    interface IHero
    {
        // hero only needs to do one thing.
        2 references
        void Attack();
    }
}
```

4. Create a Sword class with two constructors, default and parameterized.

Sword.cs | HeroThatOnlyUsesSwords.cs | HeroThatCanUseAnyWeapon.cs

VideoGameDependencyInjectionCore

```
namespace VideoGameDependencyInjectionCore
{
    4 references
    internal class Sword
    {
        2 references
        public string SwordName { get; set; }
        1 reference
        public Sword(string swordName)
        {
            SwordName = swordName;
        }
        0 references
        public Sword()
        {
            SwordName = "Lame name sword";
        }

    }
}
```

5. Add the **IWeapon** interface and print something catchy in the **AttackWithMe** method. Show the **SwordName**.

Sword.cs* | HeroThatOnlyUsesSwords.cs | HeroThatCanUseAnyWeapon.cs | IHero.cs | IWeapon.cs

VideoGameDependencyInjectionCore | VideoGameDependencyInjectionCore.Sword | SwordName

```
1  namespace VideoGameDependencyInjectionCore
2  {
3      internal class Sword : IWeapon
4      {
5          public string SwordName { get; set; }
6          public Sword(string swordName)
7          {
8              SwordName = swordName;
9          }
10         public Sword()
11         {
12             SwordName = "Lame name sword";
13         }
14
15         public void AttackWithMe()
16         {
17             System.Console.WriteLine(SwordName + " slices through the air, devestating all enemies");
18         }
19
20
21     }
22 }
```

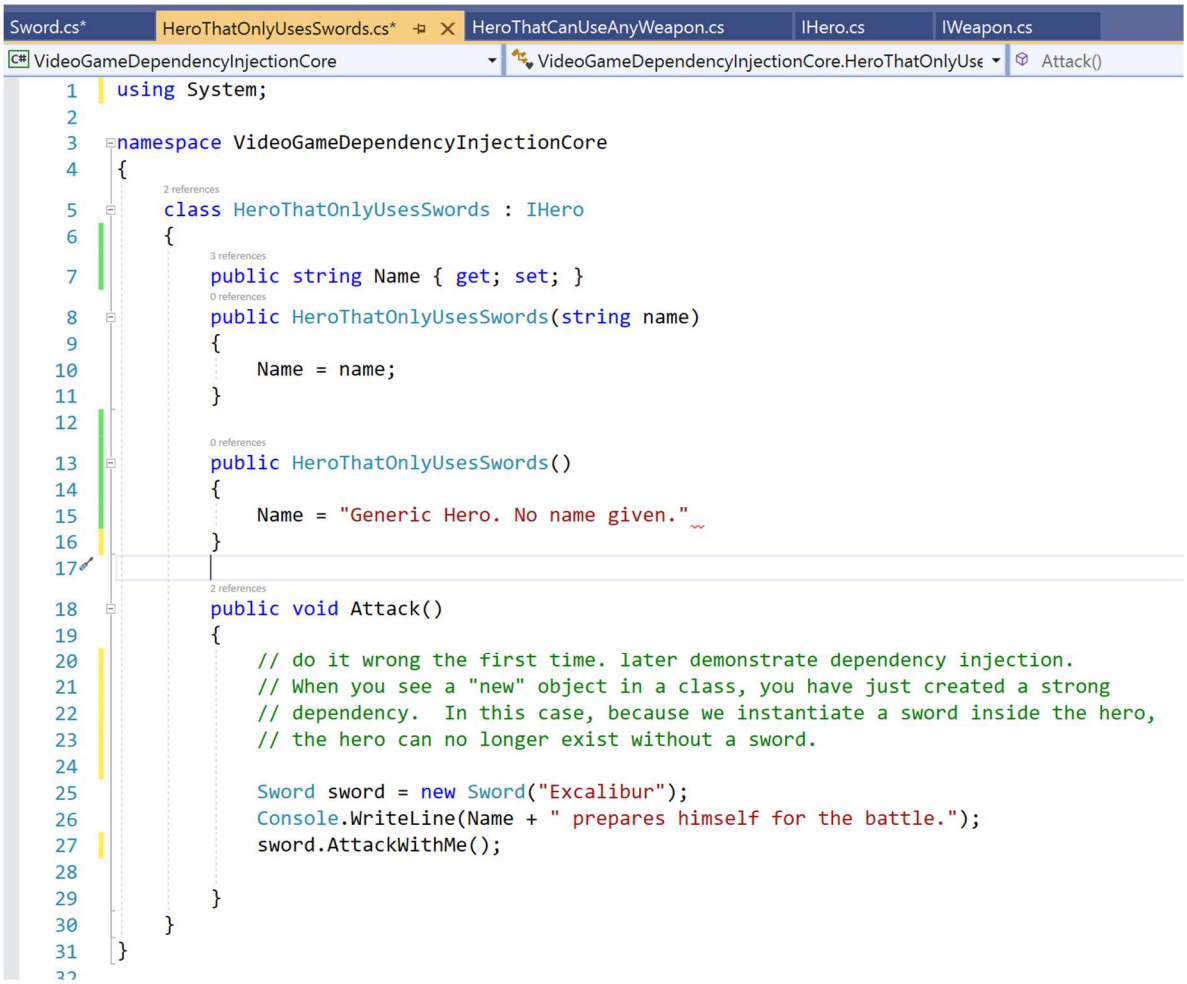
6. Create a class called **HeroThatOnlyUsesSwords**

Sword.cs* | HeroThatOnlyUsesSwords.cs* | HeroThatCanUseAnyWeapon.cs

VideoGameDependencyInjectionCore | VideoGameDependencyI

```
1  using System;
2
3  namespace VideoGameDependencyInjectionCore
4  {
5      class HeroThatOnlyUsesSwords
6      {
7          public string Name { get; set; }
8          public HeroThatOnlyUsesSwords(string name)
9          {
10             Name = name;
11         }
12
13         public HeroThatOnlyUsesSwords()
14         {
15             Name = "Generic Hero. No name given.";
16         }
17
18     }
19 }
20
```

7. Implement the **IHero** interface. In the new **Attack** method, **instantiate a new sword** and attack with it.



8. In the **Program.cs** file, create a new hero and use his **attack** method.

```
Program.cs*  Sword.cs*  HeroThatOnlyUsesSwords.cs*  HeroThatCanUseAnyWeapon.cs  IHero.cs
VideoGameDependencyInjectionCore  VideoGameDependencyInjectionCore.Program  Main(str
1   using Microsoft.Extensions.DependencyInjection;
2   using System;
3   using System.Collections.Generic;
4
5   namespace VideoGameDependencyInjectionCore
6   {
        0 references
7       class Program
8       {
            0 references
9           static void Main(string[] args)
10          {
11              HeroThatOnlyUsesSwords hero1 = new HeroThatOnlyUsesSwords("Ultraman");
12              hero1.Attack();
13              Console.WriteLine();
14
15              Console.ReadLine();
16
17          }
18      }
19  }
```

9. Run the program.

```
C:\Users\shadsluiter\source\repos\VideoGameDependencyInjectionCore\bin\Del
Ultraman prepares himself for the battle.
Excalibur slices through the air, devestating all enemies
```

Everything works properly. However, we are going to modify the program to utilize dependency injection, so don't celebrate yet.

10. Create another weapon, a **Grenade**.

Grenade.cs* | Gun.cs | Program.cs | Sword.cs | HeroThatOnlyUsesSwords.cs

VideoGameDependencyInjectionCore | VideoGameDependencyInjectionCore.Gre

```
1  namespace VideoGameDependencyInjectionCore
2  {
       2 references
3      class Grenade
4      {
           0 references
5          public Grenade(string weaponName)
6          {
7              WeaponName = weaponName;
8          }
           0 references
9          public Grenade()
10         {
11             WeaponName = "Pathetic granede. No name provided.";
12         }
13
           2 references
14         public string WeaponName { get; set; }
15
16     }
17 }
```

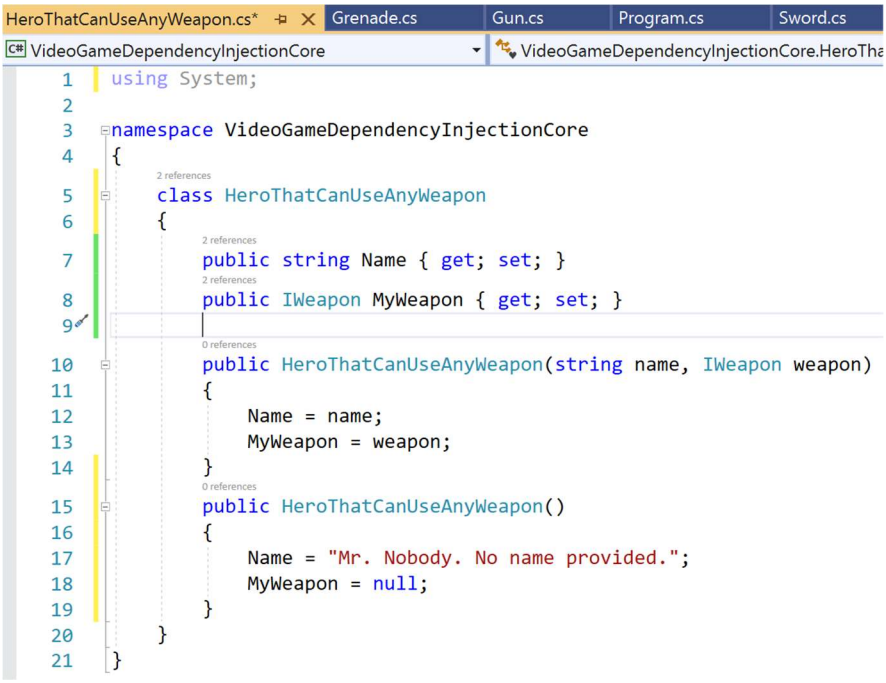
11. Implement the **IWeapon** interface and print something clever about how the grenade works. Print the grenade's name.

Grenade.cs | Gun.cs | Program.cs | Sword.cs | HeroThatOnlyUsesSwords.cs | HeroThatCanUseAnyWeapon.cs | IHero.cs

VideoGameDependencyInjectionCore | VideoGameDependencyInjectionCore.Grenade | WeaponName

```
1  using System;
2
3  namespace VideoGameDependencyInjectionCore
4  {
       2 references
5      class Grenade : IWeapon
6      {
           3 references
7          public string WeaponName { get; set; }
           0 references
8          public Grenade(string weaponName)
9          {
10             WeaponName = weaponName;
11         }
           0 references
12         public Grenade()
13         {
14             WeaponName = "Pathetic granede. No name provided.";
15         }
16
           5 references
17         public void AttackWithMe()
18         {
19             Console.WriteLine(WeaponName + " sizzles for a moment and then explodes into a shower of deadly metal
                 fragments");
20         }
21     }
22 }
```

12. Create a second Hero. This person takes **Iweapon** as a parameter.



13. Implement the IHero interface and print a status message in the **Attack** method.

HeroThatCanUseAnyWeapon.cs* | Grenade.cs | Gun.cs | Program.cs | Sword.cs

VideoGameDependencyInjectionCore

```
1  using System;
2
3  namespace VideoGameDependencyInjectionCore
4  {
5      class HeroThatCanUseAnyWeapon : IHero
6      {
7          public string Name { get; set; }
8          public IWeapon MyWeapon { get; set; }
9
10         public HeroThatCanUseAnyWeapon(string name, IWeapon weapon)
11         {
12             Name = name;
13             MyWeapon = weapon;
14         }
15         public HeroThatCanUseAnyWeapon()
16         {
17             Name = "Mr. Nobody. No name provided.";
18             MyWeapon = null;
19         }
20
21         public void Attack()
22         {
23             Console.WriteLine(Name + " prepares to attack");
24             MyWeapon.AttackWithMe();
25         }
26     }
27 }
```

11. Add two more heroes to the program. Give one a **sword**. Give the other a **Grenade**.

```
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             HeroThatOnlyUsesSwords hero1 = new HeroThatOnlyUsesSwords("Ultraman");
12             hero1.Attack();
13             Console.WriteLine();
14
15             HeroThatCanUseAnyWeapon hero2 = new HeroThatCanUseAnyWeapon("Eregon", new Sword("Brisinger"));
16             hero2.Attack();
17             Console.WriteLine();
18
19             HeroThatCanUseAnyWeapon hero3 = new HeroThatCanUseAnyWeapon("The Joker", new Grenade("The Pineapple"));
20             hero3.Attack();
21             Console.WriteLine();
22
23
24
25             Console.ReadLine();
26
```

14. Run the program and verify the results.

```
C:\Users\shadsluiter\source\repos\VideoGameDependencyInjectionCore\bin\Debug\netcoreapp3.1\VideoGameDependencyInjec
Ultraman prepares himself for the battle.
Excalibur slices through the air, devestating all enemies

Eregon prepares to attack
Brisinger slices through the air, devestating all enemies

The Joker prepares to attack
The Pineapple sizzles for a moment and then explodes into a shower of deadly metal fragments
```

15. Just to extend the dependencies a little further, let's create a **Gun**. Of course, the Gun needs **bullets**, which are also a class.

Bullet.cs | HeroThatCanUseAnyWeapon.cs | Grenade.cs | Gun.cs

VideoGameDependencyInjectionCore

```
1   namespace VideoGameDependencyInjectionCore
2   {
        3 references
3       public class Bullet
4       {
            2 references
5           public string Name { get; set; }
            1 reference
6           public int GramsOfPowder { get; set; }
7
            0 references
8           public Bullet(string name, int gramsOfPowder)
9           {
10              Name = name;
11              GramsOfPowder = gramsOfPowder;
12          }
13      }
14  }
```

HeroThatCanUseAnyWeapon.cs | Grenade.cs | Gun.cs | Program.cs | Sword.cs | HeroThatOnlyUsesSwords.cs | IHero.cs

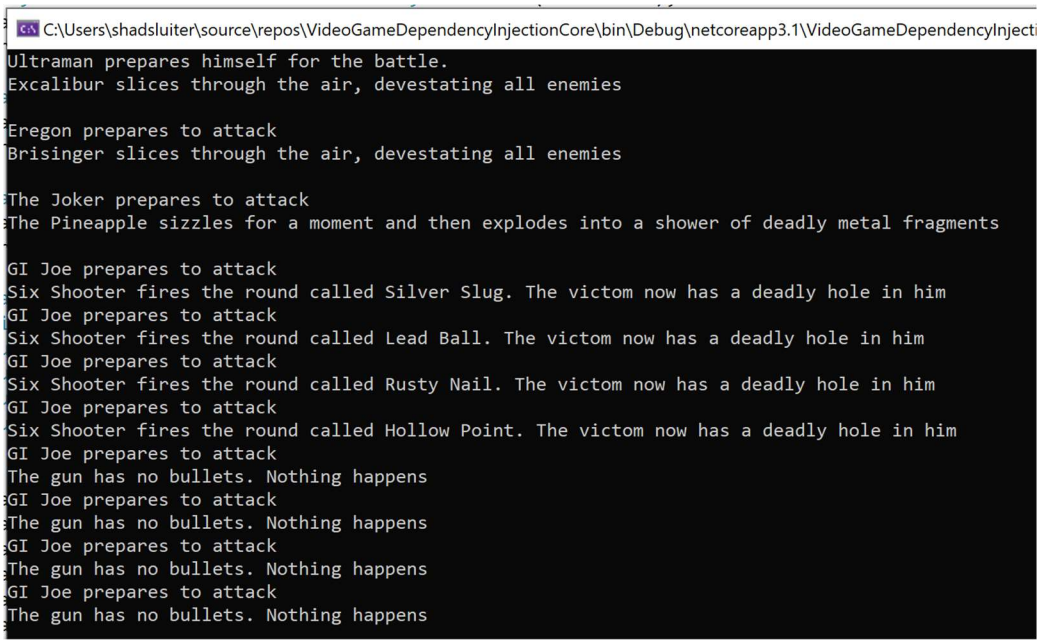
VideoGameDependencyInjectionCore | VideoGameDependencyInjectionCore.Gun | Name

```
1   using System.Collections.Generic;
2
3   namespace VideoGameDependencyInjectionCore
4   {
        1 reference
5       internal class Gun : IWeapon
6       {
            2 references
7           public string Name { get; set; }
            4 references
8           public List<Bullet> Bullets { get; set; }
9
10
            0 references
11          public Gun(string name, List<Bullet> bullets)
12          {
13              this.Name = name;
14              this.Bullets = bullets;
15          }
16
            5 references
17          public void AttackWithMe()
18          {
19
20              if (Bullets.Count > 0)
21              {
22                  System.Console.WriteLine(Name + " fires the round called " + Bullets[0 ].Name + ". The victom now
                      has a deadly hole in him");
23                  Bullets.RemoveAt(0);
24              }
25              else
26              {
27                  System.Console.WriteLine("The gun has no bullets. Nothing happens");
28              }
29
30          }
31      }
32  }
```

16. Add a new Hero that uses the gun with four bullets.

```
{
    0 references
    static void Main(string[] args)
    {
        HeroThatOnlyUsesSwords hero1 = new HeroThatOnlyUsesSwords("Ultraman");
        hero1.Attack();
        Console.WriteLine();

        HeroThatCanUseAnyWeapon hero2 = new HeroThatCanUseAnyWeapon("Eregon", new Sword("Brisinger"));
        hero2.Attack();
        Console.WriteLine();

        HeroThatCanUseAnyWeapon hero3 = new HeroThatCanUseAnyWeapon("The Joker", new Grenade("The Pineapple"));
        hero3.Attack();
        Console.WriteLine();

        HeroThatCanUseAnyWeapon hero4 = new HeroThatCanUseAnyWeapon("GI Joe", new Gun("Six Shooter",
            new List<Bullet> {
            new Bullet("Silver Slug", 10),
            new Bullet("Lead Ball", 20),
            new Bullet("Rusty Nail", 3),
            new Bullet("Hollow Point", 5)
        }));

        hero4.Attack();
        hero4.Attack();
        hero4.Attack();
        hero4.Attack();
        hero4.Attack();
        hero4.Attack();
        hero4.Attack();
        hero4.Attack();

        Console.ReadLine();
```

17. Run the program to verify that everything is wor king.



Dependency Injection.

The point of all this nonsense is to set up a situation where we can create a configuration for a hero who has dependencies, giving him a specific type of hero class and weapon. When we instantiate him, we can do it in only one line. But first, let's investigate the **motivations** behind Dependency Injection.

The Dependency Problem

In the two versions of the hero we created earlier, the **HeroThatCanUseAnyWeapon** was much more flexible than the **HeroThatOneUsesSword**. We could pass a weapon to his constructor. The **HeroThatOnlyUsesSword** has the problem of creating an instance of the Sword within one of his methods. We passed the dependency of a weapon as a parameter to the **HeroThatCanUseAnyWeapon** constructor using an IWeapon **interface**.

In software engineering, dependency injection is a technique in which an object receives other objects that it depends on. These other objects are called dependencies. In the typical "using" relationship, the receiving object is called a client and the passed (that is, "injected") object is called a **service**. The code that passes the service to the client can be many kinds of things and is called the **injector**. Instead of the client specifying which service it will use, the injector tells the client what service to use. The "injection" refers to the passing of a dependency (a service) into the object (a client) that would use it.

The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

The intent behind dependency injection is to achieve separation of concerns of construction and use of objects. This can increase readability and code reuse.

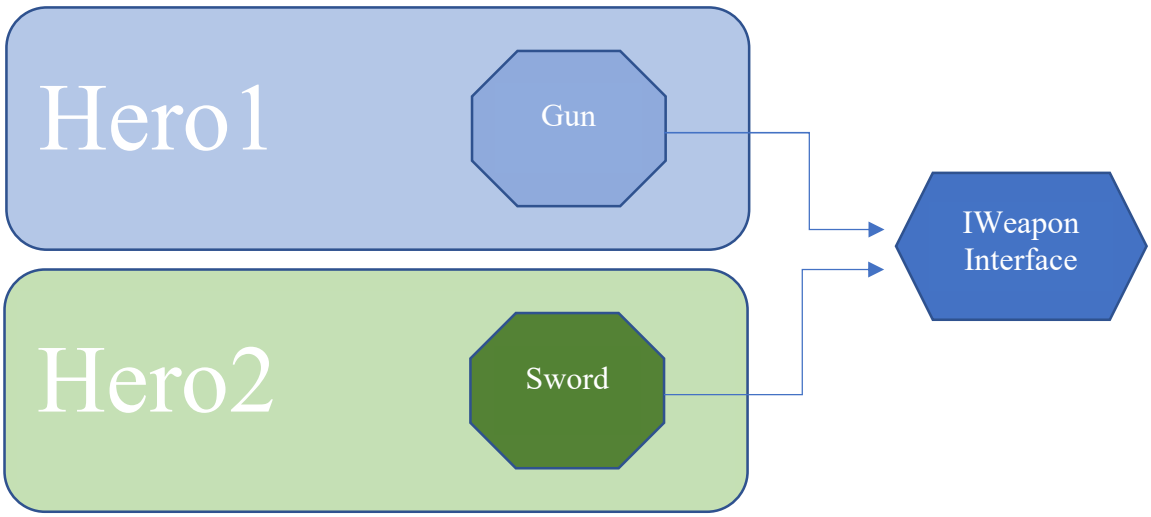
Dependency injection is one form of the broader technique of **inversion of control**. A client who wants to call some services should not have to know how to construct those services. Instead, the client delegates the responsibility of providing its services to external code (the injector). The client is not allowed to call the injector code; it is the injector that constructs the services. The injector then passes (injects) the services, which might already exist or may also be constructed by the injector, into the client. The client then uses the services. This means the client does not need to know about the injector, how to construct the services, or even which actual services it is using. The client only needs to know about the intrinsic **interfaces** of the services because these define how the client may use the services. This separates the responsibility of "use" from the responsibility of "construction."

The Dependency Inversion Principle

The Dependency Inversion Principle is a strategy to mitigate the dependency problem and make it more manageable.

- *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
- *Abstractions should not depend on details. Details should depend on abstractions.*

You can translate the two formal recommendations as follows: In the typical layered architecture of an application, a high-level component should not directly depend on a lower-level component. You should create an interface and make both components depend on this abstraction.



Dependency Terminology

Here are definitions of some common Dependency Injection terms.

- **Dependency Inversion Principle** – A software design principle. It suggests a solution to the dependency problem, but does not say how to implement it or which technique to use.
- **Inversion of Control (IoC) –** A way to apply the Dependency Inversion Principle. Inversion of Control is the actual mechanism that allows your higher-level components to depend on abstraction rather than the concrete implementation of lower-level components.
- **Dependency Injection** – A design pattern to implement Inversion of Control. It allows you to inject the concrete implementation of a low-level component into a high-level component.
- **IoC Container** – Also known as Dependency Injection (DI) Container, it is a programming framework that provides you with an automatic Dependency Injection of your components.

Dependency Injection approaches

You can implement Dependency Injection using three common approaches:

- **Constructor Injection** – Create an instance of the dependency and pass it as an argument to the constructor of the dependent class. This is the type of injection that we will demonstrate.
- **Method Injection** – Create an instance of the dependency and pass it to a specific method of the dependent class.

```
1 reference
class Sandwich
{
    1 reference
    public List<string> Topping { get; set; }
    0 references
    public Sandwich(IToppings toppings)
    {
        Topping =  toppings;
    }
```

- **Property Injection** – Assign the instance of the dependency to a specific property of the dependent class.

.NET Core and the Dependency Injection

You can implement Dependency Injection manually by using one or more of the three approaches discussed before. However, .NET Core comes with a built-in **IoC Container** that simplifies Dependency Injection management.

The **IoC Container** is responsible for supporting automatic Dependency Injection. Its basic features include:

- **Registration** – The IoC Container needs to know which type of object to create for a specific dependency; so, it provides a way to **map** a type to a class so that it can create the correct dependency instance.

- **Resolution** – This feature allows the IoC Container to resolve a dependency by creating an object and injecting it into the requesting class. Thanks to this feature, you don't have to instantiate objects manually to manage dependencies.
- **Disposition** – The IoC Container manages the lifetime of the dependencies following specific criteria.

Service lifetimes

The IoC Container allows you to control the lifetime of a registered service. When you register a service specifying a lifetime, the container will automatically dispose of it accordingly. You have three service lifetimes:

- **Singleton** – Creates one instance of the service.
- **Transient** –The service will be created each time it will be requested. This means, for example, that a service injected in the constructor of a class will last as long as that class instance exists.
- **Scoped** – Creates an instance of a service for each client request. This is particularly useful in the ASP.NET context since it allows you to share the same service instance for the duration of an HTTP request processing.

Finally, let’s compare some of the advantages and disadvantages to using Dependency Injection.

Dependency Injection Advantages

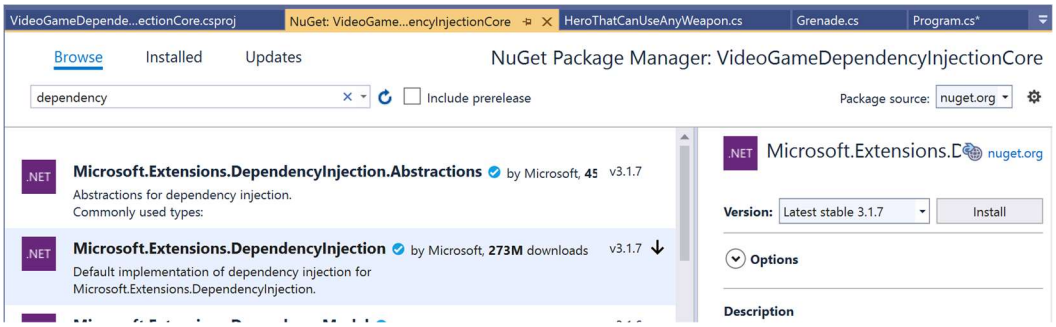
- Dependency injection allows a client to be configurable.
- Dependency injection can be externalized into configuration files, allowing the system to be reconfigured without recompilation. Separate configurations can be written for different situations that require different implementations of components.
- Because dependency injection does not require any change in code, clients are more independent and they are easier to unit test in isolation
- Dependency injection allows a client to remove all knowledge of a concrete implementation that it needs to use. This helps isolate the client from the impact of design changes and defects. It promotes reusability, testability, and maintainability.
- Dependency injection allows concurrent or independent development. Two developers can independently develop classes that use each other, while only needing to know the interface the classes will communicate through.

Dependency Injection Disadvantages

- Dependency injection creates clients that demand configuration details be supplied by construction code. This can be onerous when obvious defaults are available.
- Dependency injection can make code difficult to trace (read) because it separates behavior from construction. This means developers must refer to more files to follow how a system performs.
- Dependency injection typically requires more upfront development effort since one cannot instantiate something right when and where it is needed, but must ask that it be injected and then ensure that it has been injected.
- Dependency injection forces complexity to move out of classes and into the linkages between classes, which might not always be desirable or easily managed.
- Dependency injection can encourage dependence on a dependency injection framework.

Install the Microsoft Dependency Manager

1. Using **NeGet**, add the **Microsoft.Extensions.DependencyInjection** package to the program.

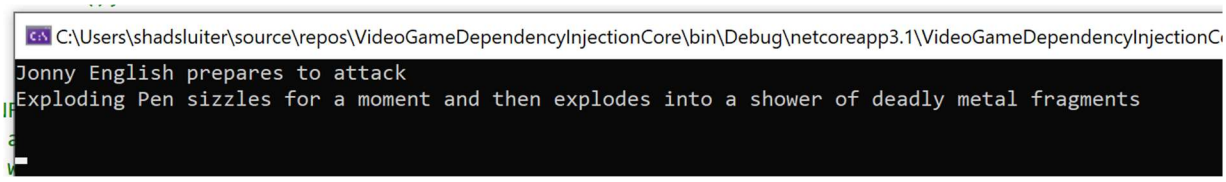


2. In **Program.cs**, comment out all of the heroes, and the laborious work that went into them, that we created in the first part of the tutorial.

3. Add a new hero that is configured with Dependency Injection as shown here. Pay attention to the comments; they explain the strategy and config.

```
41
42          // CONFIGURATION FILE
43          // In an ASP.NET program this secion usually sits in a configuration file like Startup.cs
44          // we will modify the contents of this startup section.
45
46          // ServiceCollection is the "container" of all registered dependencies
47          ServiceCollection services = new ServiceCollection();
48
49          // All new weapons will now be Grenades by default.
50          services.AddTransient<IWeapon, Grenade>(grenade => new Grenade("Exploding Pen"));
51
52          // All new heroes will be "Jonny" by default.
53          services.AddTransient<IHero, HeroThatCanUseAnyWeapon>(
54              hero => new HeroThatCanUseAnyWeapon("Jonny English", hero.GetService<IWeapon>() )
55          );
56          // Sort of a "compile" step to assemble everything listed above.
57          ServiceProvider provider = services.BuildServiceProvider();
58
59          // IMPLEMENTATION
60          // based on all of the configuration above, we can create a new hero in one, small step.
61          var hero5 = provider.GetService<IHero>();
62
63          // let's see if it works...
64          hero5.Attack();
65          Console.WriteLine();
66
67          Console.ReadLine();
68
```

4. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.
5. Run the program. We should see Jonny English blow up his pen grenade.



6. In the dependency configuration, make a change to the weapon. The pen is commented out and a sword is included instead.

```
40          */
41
42          // CONFIGURATION FILE
43          // In an ASP.NET program this secion usually sits in a configuration file like Startup.cs
44          // we will modify the contents of this startup section.
45
46          // ServiceCollection is the "container" of all registered dependencies
47          ServiceCollection services = new ServiceCollection();
48
49          // All new weapons will now be Grenades by default.
50          // services.AddTransient<IWeapon, Grenade>(grenade => new Grenade("Exploding Pen"));
51          services.AddTransient<IWeapon, Sword>(s => new Sword("The Sword of Gryffindor"));
52
53          // All new heroes will be "Jonny" by default.
54          services.AddTransient<IHero, HeroThatCanUseAnyWeapon>(
55              hero => new HeroThatCanUseAnyWeapon("Jonny English", hero.GetService<IWeapon>() )
56          );
57          // Sort of a "compile" step to assemble everything listed above.
58          ServiceProvider provider = services.BuildServiceProvider();
59
60          // IMPLEMENTATION
61          // based on all of the configuration above, we can create a new hero in one, small step.
62          var hero5 = provider.GetService<IHero>();
63
64          // let's see if it works...
65          hero5.Attack();
66          Console.WriteLine();
67
68          Console.ReadLine();
69
```

7. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.
8. Run the program to see Jonny wield a new sword.

```
C:\Users\shadsluiter\source\repos\VideoGameDependencyInjectionCore\bin\Debug\netcoreapp3.1\\
Jonny English prepares to attack
The Sword of Gryffindor slices through the air, devestating all enemies
```

9. Do we dare trust Jonny English with a gun?

```
42              // CONFIGURATION FILE
43              // In an ASP.NET program this secion usually sits in a configuration file like Startup.cs
44              // we will modify the contents of this startup section.
45
46              // ServiceCollection is the "container" of all registered dependencies
47              ServiceCollection services = new ServiceCollection();
48
49              // All new weapons will now be Grenades by default.
50              // services.AddTransient<IWeapon, Grenade>(grenade => new Grenade("Exploding Pen"));
51              // services.AddTransient<IWeapon, Sword>(s => new Sword("The Sword of Gryffindor"));
52              services.AddTransient<IWeapon, Gun>(g => new Gun("Six Shooter",
53                  new List<Bullet> {
54                  new Bullet("Silver Slug", 10),
55                  new Bullet("Lead Ball", 20),
56                  new Bullet("Rusty Nail", 3),
57                  new Bullet("Hollow Point", 5)
58              }));
59
60              // All new heroes will be "Jonny" by default.
61              services.AddTransient<IHero, HeroThatCanUseAnyWeapon>(
62                  hero => new HeroThatCanUseAnyWeapon("Jonny English", hero.GetService<IWeapon>() )
63              );
64              // Sort of a "compile" step to assemble everything listed above.
65              ServiceProvider provider = services.BuildServiceProvider();
66
67              // IMPLEMENTATION
68              // based on all of the configuration above, we can create a new hero in one, small step.
69              var hero5 = provider.GetService<IHero>();
70
71              // let's see if it works...
72              hero5.Attack();
73              Console.WriteLine();
74
75              Console.ReadLine();
```

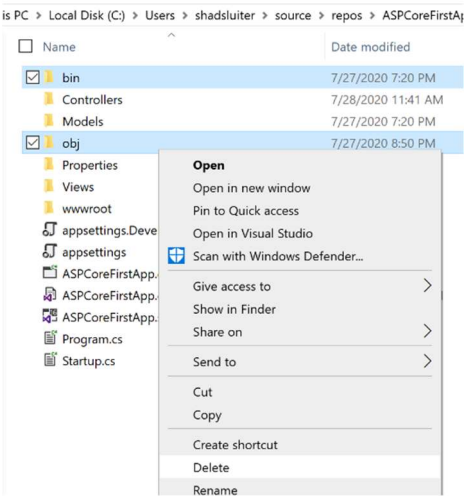
10. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Apparently, we can trust Jonny with a gun…

```
C:\Users\shadsluiter\source\repos\VideoGameDependencyInjectionCore\bin\Debug\netcoreapp3.1\VideoGameDependencyIr
Jonny English prepares to attack
Six Shooter fires the round called Silver Slug. The victom now has a deadly hole in him
```

In the next tutorial, we will return to **ASP.NET Core web applications** where dependency injection is commonly used in the setup file.

Deliverables:



1. Submit a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
2. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
3. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.

Part 2 Dependency Injection in .NET Core

Overview

In the following console application, we are going to put dependency injection into practice using a database service.

1. Open the **Products application** which was created in previous tutorials.

2. In the **ProductsController**, there is a place where the **data repository** is specified. Currently, the programmer can swap the data source by commenting out one line and leaving the other line active. We are going to **refactor** this part of the program to utilize dependency injection.

```
10
11  namespace ASPCoreFirstApp.Controllers
12  {
        1 reference
13      public class ProductsController : Controller
14      {
15          // comment out one of these to choose the database source.
16          // HardCodedSampleDataRepository repository = new HardCodedSampleDataRepository();
17          ProductDAO repository = new ProductDAO();
            0 references
18          public ProductsController()
19          {
20              repository = new ProductDAO();
21          }
22
```

Dependency instantiated from an external class.

3. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

We are going to be using both the **HardCodedDataRepository** and the **ProductDAO**. Both of these implement the interface **IProductDataService**.

In case you missed it, the **HardCodedSampleDataRepository** looks like this.

```
7
8   namespace ASPCoreFirstApp.Services
9   {
        2 references
10      public class HardCodedSampleDataRepository : IProductDataService
11      {
12          List<ProductModel> productList;
            0 references
13          public HardCodedSampleDataRepository()
14          {
15              productList = new List<ProductModel>();
16          }
            6 references
17          public List<ProductModel> AllProducts()
18          {
19
20              productList.Add(new ProductModel(1, "Mouse Pad", 5.99m, "A square piece of plastic to make mousing easier"));
21              productList.Add(new ProductModel(2, "Web Cam", 45.50m, "Enables you to attend more Zoom meetings"));
22              productList.Add(new ProductModel(3, "4 TB USB Hard Drive", 130.00m, "Back up all of your work. You won't regret
                  it."));
23              productList.Add(new ProductModel(4, "Wireless Mouse", 15.99m, "Notebook mice really don't work very well. Do
                  they?"));
24              for (int i = 0; i < 100; i++)
25              {
26                  productList.Add(new Faker<ProductModel>()
27                      .RuleFor(p => p.Id, i + 5)
28                      .RuleFor(p => p.Name, f => f.Commerce.ProductName())
29                      .RuleFor(p => p.Price, f => f.Random.Decimal(100))
30                      .RuleFor(p => p.Description, f => f.Rant.Review())
31                  );
32              }
33              return productList;
34          }
35
            7 references
36          public ProductModel GetProductById(int id)
37          {
38              return productList.FirstOrDefault(p => p.Id == id);
39          }
40
            4 references
41          public List<ProductModel> SearchProducts(string seartchTerm)
42          {
43              return productList.FindAll(p => p.Name.Contains(seartchTerm));
44          }
            2 references
45          public bool Delete(ProductModel product)
46          {
47              throw new NotImplementedException();
48          }
```

Configure Dependencies

1. In the **Startup**.cs file, add a relation between the **IProductDataService** and **HardCodedSampleDataRepository**.

```
15    public class Startup
16    {
          0 references
17        public Startup(IConfiguration configuration)
18        {
19            Configuration = configuration;
20        }
21
          1 reference
22        public IConfiguration Configuration { get; }
23
24        // This method gets called by the runtime. Use this method to add services to the container.
          0 references
25        public void ConfigureServices(IServiceCollection services)
26        {
27            services.AddTransient<IProductDataService, HardCodedSampleDataRepository>();
28            services.AddControllersWithViews();
29        }
30
```

Any reference to **IProductDataService** will mean **HardCodedSampelDataRepository**

2. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

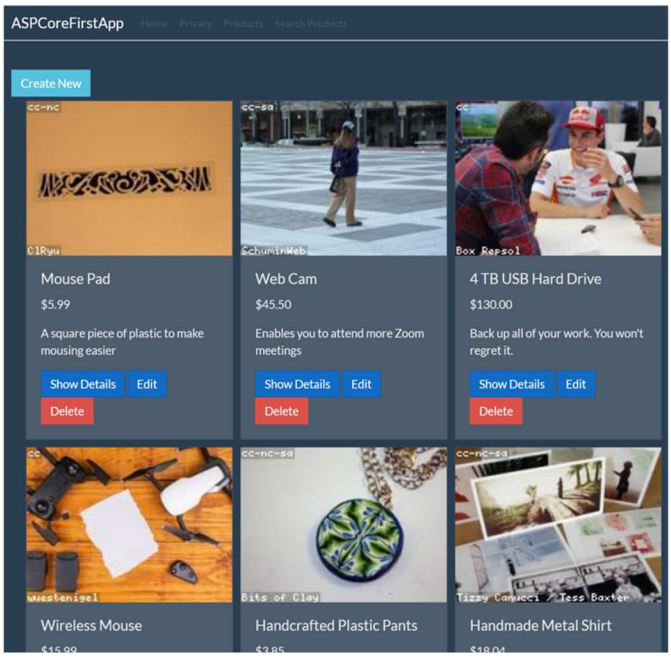
3. In the **ProductsController**, add a parameter of **IProductDataService** to the constructor.

```
11  namespace ASPCoreFirstApp.Controllers
12  {
        1 reference
13      public class ProductsController : Controller
14      {
15          // comment out one of these to choose the database source.
16          // HardCodedSampleDataRepository repository = new HardCodedSampleDataRepository();
17          // ProductDAO repository = new ProductDAO();
18
            9 references
19          public IProductDataService repository { get; set; }
20
            0 references
21          public ProductsController(IProductDataService dataService)
22          {
23              repository = dataService;
24          }
25
            0 references
```

HardCodedSampleDataRepository is now injected (passed to) **dataService**.

4. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

5. Run the program. You should see the four tech products and 100 “Faker” products appear in the app.



6. In the **Startup.cs** file, change the dependency relation to the **ProductDAO**.

```
13  namespace ASPCoreFirstApp
14  {
        2 references
15      public class Startup
16      {
            0 references
17          public Startup(IConfiguration configuration)
18          {
19              Configuration = configuration;
20          }
21
            1 reference
22          public IConfiguration Configuration { get; }
23
24          // This method gets called by the runtime. Use this method to add services to the container.
            0 references
25          public void ConfigureServices(IServiceCollection services)
26          {
27              // services.AddTransient<IProductDataService, HardCodedSampleDataRepository>();
28              services.AddTransient<IProductDataService, ProductDAO>();
29
30              services.AddControllersWithViews();
```

ProductDAO now becomes the item associated with **IProductDataService**

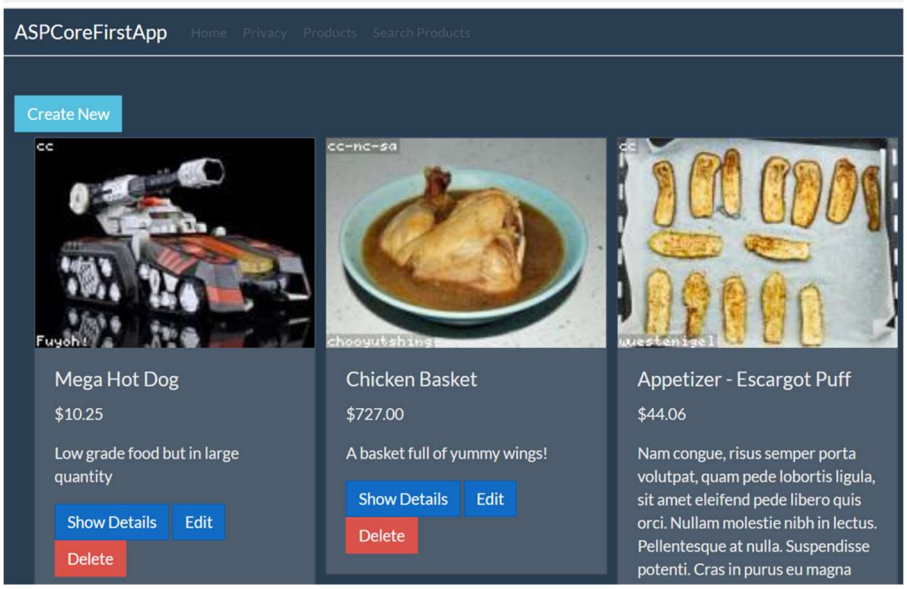
7. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

8. No change is in the **ProductsController** is necessary. However, the repository is now a **different data source**.

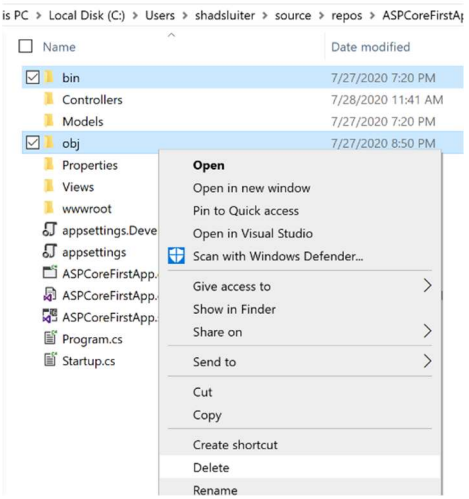
```
11  namespace ASPCoreFirstApp.Controllers
12  {
        1 reference
13      public class ProductsController : Controller
14      {
15          // comment out one of these to choose the database source.
16          // HardCodedSampleDataRepository repository = new HardCodedSampleDataRepository();
17          // ProductDAO repository = new ProductDAO();
18
            9 references
19          public IProductDataService repository { get; set; }
20
            0 references
21          public ProductsController(IProductDataService dataService)
22          {
23              repository = dataService;
24          }
25
            0 references
```

ProductDAO is now injected (passed to) **dataService**.

9. Run the application. The **ProductDAO** data source should now appear in the products list.



Deliverables:



1. Submit a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
2. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
3. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.