Dependency Injection

Tan Cher Wah (cherwah@nus.edu.sg)

Dependency Injection

- Dependency Injection is a software design to make code more modular and testable
- If component A needs component B to function, then component B is a dependency to component A
- In Dependency Injection, component B (the dependency) is injected into component A during runtime

Dependency

JPG_Renderer is a **dependency** to JPGViewer, as JPGViewer **depends** on the Rendering functionality from JPG_Renderer

```
public class JPGViewer
{
    private readonly JPG_Renderer renderer;

    public JPGViewer()
    {
        renderer = new JPG_Renderer();
    }

    public void Show(string path)
    {
        byte[] bytes = File.ReadAllBytes(path);

        renderer.Render(bytes);
    }
}
```

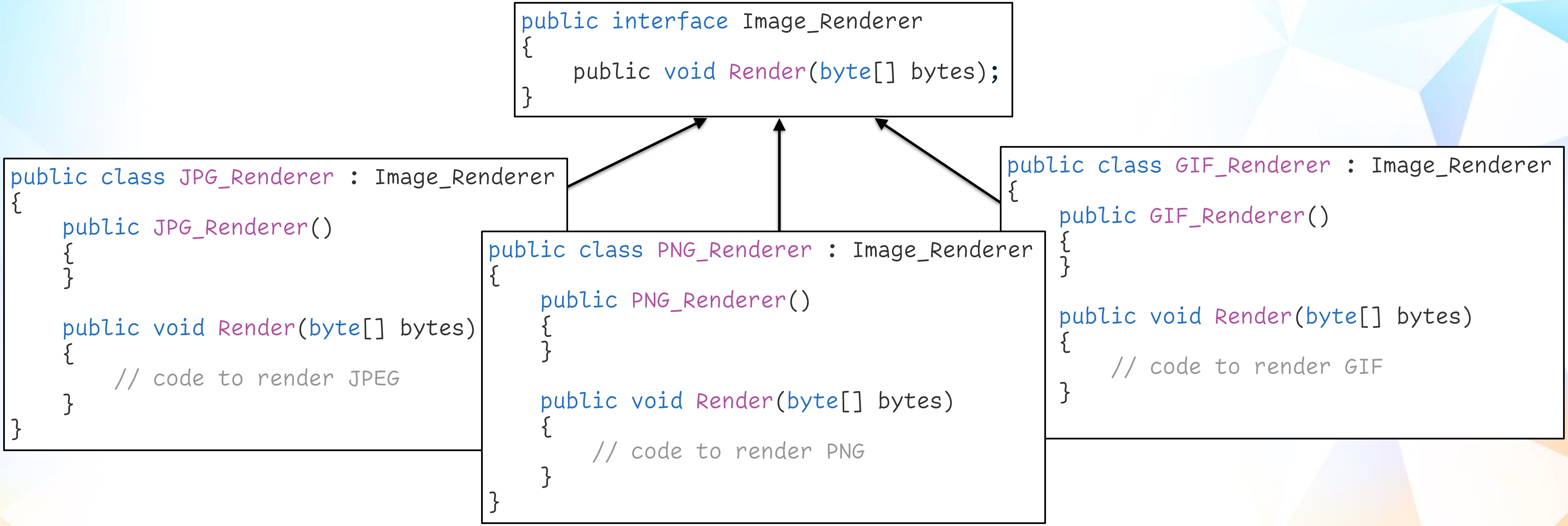
JPG_Renderer is **tightly-coupled** to JPGViewer as the render is **instantiated** within the JPGViewer class

```
public class JPG_Renderer
{
    public JPG_Renderer()
    {
    }

    public void Render(byte[] bytes)
    {
        // code to render JPG
    }
}
```

```
class Program
{
  static void Main(string[] args)
  {
    JPGViewer viewer = new JPGViewer();
    viewer.Show("sunrise.jpg");
  }
}
```

Defining an Interface for Renderers

Our viewer can be made **less coupled** with the renderer by first defining an **Interface** for different image-renderers to implement



```
public interface Image_Renderer
{
    public void Render(byte[] bytes);
}
```

```
public class JPG_Renderer : Image_Renderer
{
    public JPG_Renderer()
    {
    }

    public void Render(byte[] bytes)
    {
        // code to render JPEG
    }
}
```

```
public class PNG_Renderer : Image_Renderer
{
    public PNG_Renderer()
    {
    }

    public void Render(byte[] bytes)
    {
        // code to render PNG
    }
}
```

```
public class GIF_Renderer : Image_Renderer
{
    public GIF_Renderer()
    {
    }

    public void Render(byte[] bytes)
    {
        // code to render GIF
    }
```

Design using Dependency Injection

Next, we re-design our viewer to take in **an instance** of a **created** Image Renderer, instead of **hardcoding an instantiation**

```
public class JPGViewer
{
    private JPG_Renderer renderer;

    public JPGViewer()
    {
        renderer = new JPG_Renderer();
    }

    public void Show(string path)
    {
        byte[] bytes = File.ReadAllBytes(path);

       renderer.Render(bytes);
    }
}
```

→

```
public class Viewer
{
    private Image_Renderer renderer;

    public Viewer(Image_Renderer renderer)
    {
        this.renderer = renderer;
    }

    public void Show(string path)
    {
        byte[] bytes = File.ReadAllBytes(path);

        renderer.Render(bytes);
    }
}
```

Dependency Injection in action

Our viewer is now **loosely-coupled** with its **dependencies** (different image renderers) as they can now be **injected** into our viewer as needed

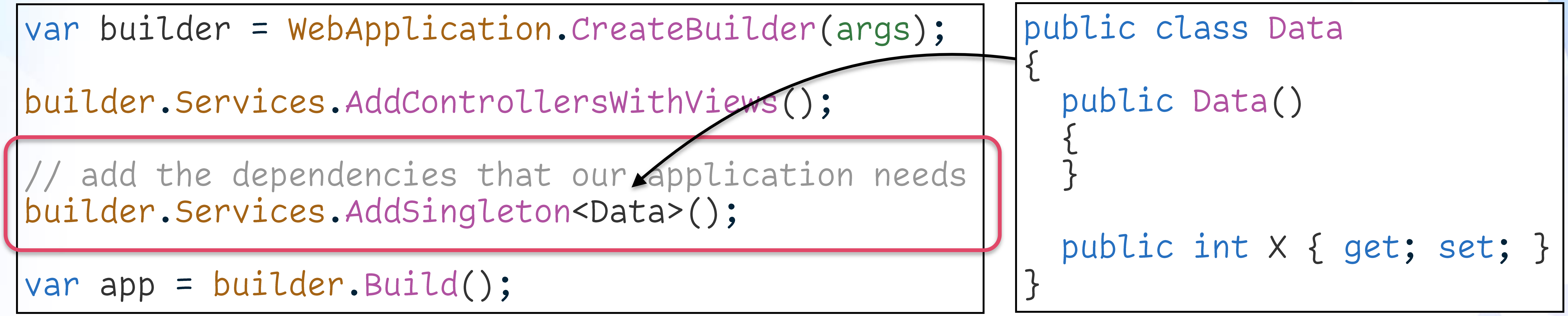
```
class Program
{
  static void Main(string[] args)
  {
      JPG_Renderer jpg_renderer = new JPG_Renderer();
      Viewer view1 = new Viewer(jpg_renderer);

      PNG_Renderer png_renderer = new PNG_Renderer();
      Viewer view2 = new Viewer(png_renderer);

      GIF_Renderer gif_renderer = new GIF_Renderer();
      Viewer view3 = new Viewer(gif_renderer);

      // same viewer class; able to display different formats
      view1.Show("sunrise.jpg");
      view2.Show("sunrise.png");
      view3.Show("sunrise.gif");
  }
}
```

Injecting Dependencies

Dependency Injection in ASP.NET

Registering Dependencies

In .NET, **dependencies** are **registered** with **builder.Services**; and must be **before** builder.Build() is called



```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

// add the dependencies that our application needs
builder.Services.AddSingleton<Data>();

var app = builder.Build();
```

```
public class Data
{
   public Data()
   {
   }

   public int X { get; set; }
}
```

Injecting Dependencies

One way to inject dependencies in .NET is to specify dependencies as **input parameters** in our Controllers' **Constructors**

```
public class HomeController : Controller
{
    private readonly Data data;

    // "data" is injected via dependency-injection
    public HomeController(Data data)
    {
        this.data = data;
    }

    public IActionResult Index()
    {
        Debug.WriteLine("X = " + data.X);

        return View();
    }
}
```

Injecting a **dependency** into our **Controller**, via the **Constructor**

Parameterising our Dependencies

To parameterise our dependencies, instantiate via a **Lambda expression** (which has the form _ **=> <expression>**)

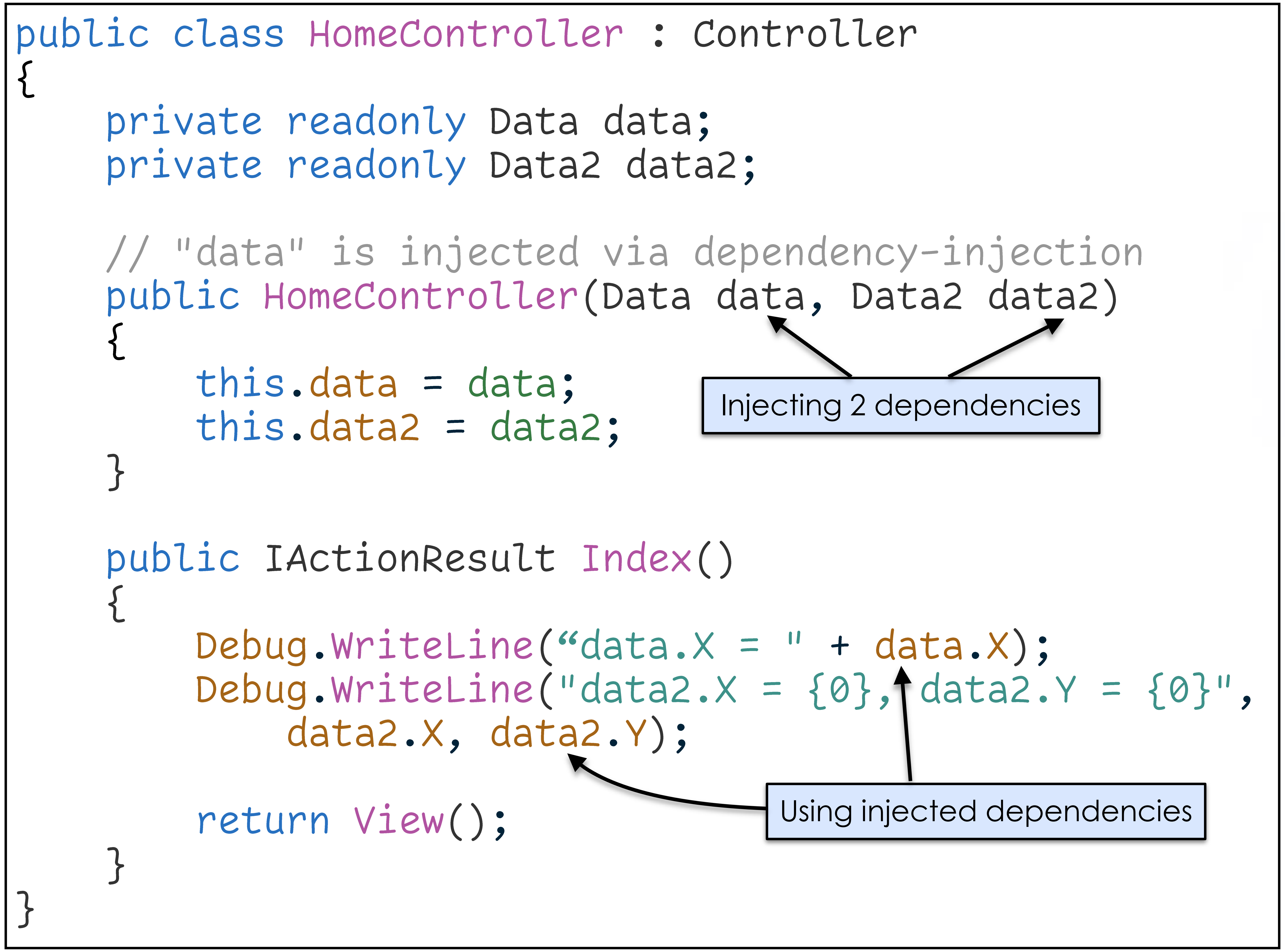
```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

// add the dependencies that our application needs
builder.Services.AddSingleton<Data>();
builder.Services.AddSingleton<Data2>(_ =>
{
    // parameterising our dependencies
    return new Data2(1, 2);
});

var app = builder.Build();
```

```
public class Data
{
  public Data()
  {
  }

  public int X { get; set; }
}
```

```
public class Data2
{
  public Data2(int x, int y)
  {
    X = x;
    Y = y;
  }

  public int X { get; set; }
  public int Y { get; set; }
}
```

Injecting Multiple Dependencies

An example of injecting **multiple dependencies** into a Controller



```
public class HomeController : Controller
{
    private readonly Data data;
    private readonly Data2 data2;

    // "data" is injected via dependency-injection
    public HomeController(Data data, Data2 data2)
    {
        this.data = data;
        this.data2 = data2;
    }

    public IActionResult Index()
    {
        Debug.WriteLine(“data.X = " + data.X);
        Debug.WriteLine("data2.X = {0}, data2.Y = {0}",
            data2.X, data2.Y);

        return View();
    }
}
```

Injecting 2 dependencies

Using injected dependencies

Injection via Controller's Methods

Another way to inject dependencies in .NET is via **Action Methods**, using the **[FromServices]** attribute

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

// add the dependencies that our application needs
builder.Services.AddSingleton<Data>();

var app = builder.Build();
```

```
public class HomeController : Controller
{
    public HomeController()
    {
    }

    public IActionResult Index([FromServices] Data data)
    {
        Debug.WriteLine("X = " + data.X);

        return View();
    }
}
```

Add **[FromServices]** when injecting dependencies via **Action Methods**

Dependency Injection occurs here

AddSingleton

A **dependency**, added with **AddSingleton**, **retains its state** throughout an **application's lifetime**

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

// add the dependencies that our application needs
builder.Services.AddSingleton<Data>();

var app = builder.Build();
```

```
public class HomeController : Controller
{
    private readonly Data data;

    public HomeController(Data data)
    {
        this.data = data;
    }

    public IActionResult Index()
    {
        Debug.WriteLine("X = “ + data.X);

        data.X++;

        return View();
    }
}
```

Injecting a dependency

Value changes every time we check

AddScoped

A **dependency**, added with **AddScoped**, **retains its state** only for a **single HTTP request/response cycle**

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

// add the dependencies that our application needs
builder.Services.AddScoped<Data>();

var app = builder.Build();
```

```
public class HomeController : Controller
{
    private readonly Data data;

    public HomeController(Data data)
    {
      this.data = data;
    }

    public IActionResult Index()
    {
        Debug.WriteLine("X = “ + data.X);

        data.X++;

        return View();
    }
}
```

Injecting a dependency

Value will always be the same, for the next request web request, regardless our update here

IMemoryCache

Caching

- Caching is the process of storing data in a temporary storage location (a cache) to speed up access
  - Reduce fetch-frequencies on data (from slower, farther location)
  - Reuse previously computed results

Benefits of Caching

- Faster system performance
- Lower latency for user requests
- Reduced load on slower data sources (e.g. databases)
- Reduced network traffic

Cache Eviction Policies

- Which items should be removed from the cache when it is full?
  - FIFO: First In First Out
  - LFU: Least Frequently Used
  - LRU: Least Recently Used
  - TTL: Time to Live

IMemoryCache

- IMemoryCache is a caching interface in that provides an easy way to add in-memory caching to our .NET application
- The lifetime of IMemoryCache is tied to the lifetime of the .NET application that it is used in
- The namespace Microsoft.Extensions.Caching.Memory needs to be imported into our .NET application

Using IMemoryCache

As IMemoryCache is built-in .NET, it can be readily injected as a dependency into our Controllers

```
public class HomeController : Controller
{
    private readonly IMemoryCache cache;

    public HomeController(IMemoryCache cache)
    {
        this.cache = cache;
    }

    ...
```

Injecting IMemoryCache as dependency

Using IMemoryCache

IMemoryCache can then be Get and Set in any action methods of that Controller

```
public class HomeController : Controller
{
    private readonly IMemoryCache cache;

    public HomeController(IMemoryCache cache)
    {
        this.cache = cache;
    }

    public IActionResult Index()
    {
        Debug.WriteLine("last_timestamp: " + cache.Get("last_timestamp"));

        cache.Set("last_timestamp", DateTimeOffset.UtcNow.ToUnixTimeSeconds());

        return View();
    }
...
```

Getting the last timestamp this method was accessed

Setting the timestamp for this latest access

THE END