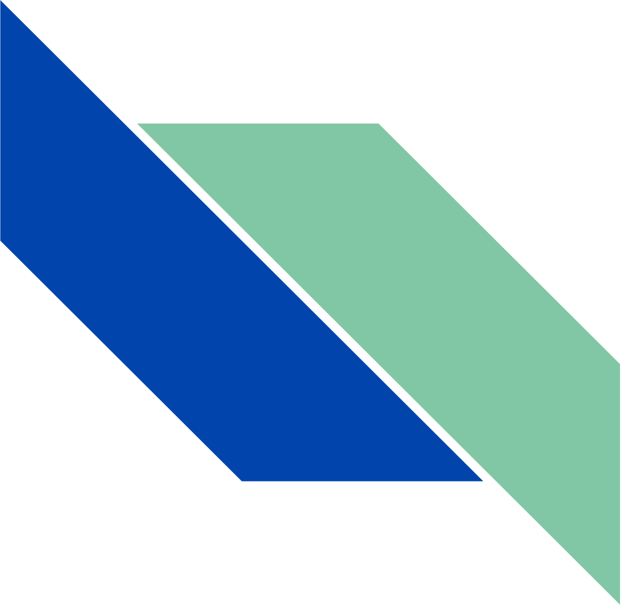
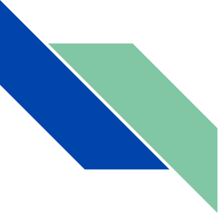


Dependency Injection

Milad Ebrahimi
Saeed Tahmasbi
Hamed Safaei



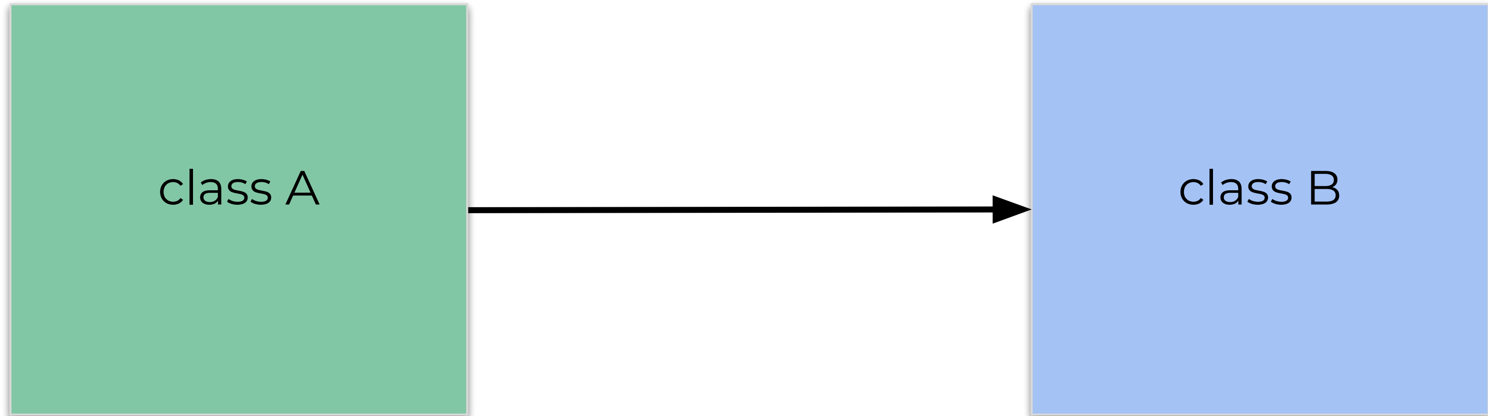
What is Dependency Injection?

Wikipedia:

> *In software engineering, dependency injection is a technique whereby one object (or static method) supplies the dependencies of another object.*

What is Dependency

When class A uses some functionality of class B, then it's said that class A has a dependency of class B.



How should we use Dependencies

Before we can use methods of other classes, we first need to create the object of that class.

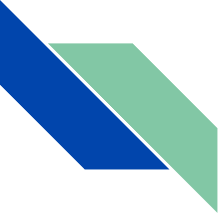
```
class A
{
      B b = new B();
}
```

What is Dependency Injection actually

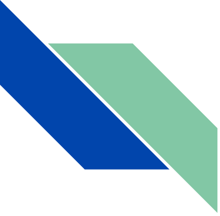
Transferring the task of creating the dependency object to someone else and directly using the dependency is called dependency injection.

Abstract example of Dependency Injection

Abstract example of Dependency Injection



Dependency Injection
in Practice



unity

Zenject

Main ideas of DI in unity

- Injection
    1. Constructor Injection

```
public class Foo
{
    IBar _bar;

    public Foo(IBar bar)
    {
        _bar = bar;
    }
}
```

Main ideas of DI in unity

- Injection

  2. Field Injection

```
public class Foo
{
    [Inject] IBar _bar;
}
```

Main ideas of DI in unity

- Injection

3. Property Injection

```
public class Foo
{
    [Inject]
    public IBar Bar
    {
        get;
        private set;
    }
}
```

- Injection

4. Method Injection

```
public class Foo
{
    IBar _bar;
    Qux _qux;

    [Inject]
    public Init(IBar bar, Qux qux)
    {
        _bar = bar;
        _qux = qux;
    }
}
```

Main ideas of DI in unity

- Binding

Every dependency injection framework is ultimately just a framework to bind types to instances.

```
Container.Bind<Foo>().AsSingle();
Container.Bind<IBar>().To<Bar>().AsSingle();
```

Main ideas of DI in unity

- Binding

```
Container.Bind<ContractType>()
     .WithId(Identifier)
     .To<ResultType>()
     .FromConstructionMethod()
     .AsScope()
     .WithArguments(Arguments)
     .OnInstantiated(InstantiatedCallback)
     .When(Condition)
     .(Copy|Move)Into(All|Direct)SubContainers()
     .NonLazy()
     .IfNotBound();
```

- Installer

  Often, there is some collections of related bindings for each sub-system and so it makes sense to group those bindings into a re-usable object.

Main ideas of DI in unity

- Installer

```
public class FooInstaller : MonoInstaller
{
    public override void InstallBindings()
    {
        Container.Bind<Bar>().AsSingle();
        Container.BindInterfacesTo<Foo>().AsSingle();
        // etc...
    }
}
```

```
public class Samurai : MonoBehaviour
{
    private Sword _sword;

    private void Awake()
    {
        Instantiate(_sword);
    }

    private void Update()
    {
        if (Input.GetKey(KeyCode.Space))
        {
            _sword.Hit();
        }
    }
}
```

```
public class Sword : MonoBehaviour
{
    public void Hit()
    {
        Debug.Log("Hit!");
    }
}
```

```
public interface IWeapon
{
    void Hit();
}
```

```
public enum WeaponType
{
    Sword,
    Makarov
}
```

```
public class Sword : MonoBehaviour
{
    public void Hit()
    {
        Debug.Log("Sword Hit!");
    }
}
```

```
public class Makarov: MonoBehaviour, IWeapon
{
    private int _bullets = 30;

    public void Hit()
    {
        Debug.Log("Makarov Hit!");
        _bullets -= 1;
    }
}
```

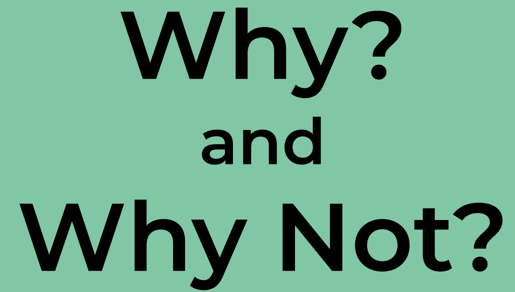
```
public class Samurai : MonoBehaviour
{
    private IWeapon _weapon;
    private WeaponType _weaponType;

    private void Awake()
    {
        switch (_weaponType)
        {
            case WeaponType.Sword:
                Instantiate(_weapon as Sword);
                break;
            case WeaponType.Makarov:
                Instantiate(_weapon as Makarov);
                break;
            default:
                throw new ArgumentOutOfRangeException();
        }
    }

    private void Update()
    {
        if (Input.GetKey(KeyCode.Space))
        {
            _weapon.Hit();
        }
    }
}
```

```
public class PlayerData : ScriptableObject
{
    public int Level;
}
```

```
public class WeaponFactory : PlaceholderFactory<IWeapon>
{

}
```

```
public class Installer : MonoInstaller
{
    [Inject] private PlayerData _playerData;

    public override void InstallBindings()
    {
        if (_playerData.Level >= 30)
        {
            Container.BindFactory<IWeapon, WeaponFactory>().To<Makarov>();
        }
        else
        {
            Container.BindFactory<IWeapon, WeaponFactory>().To<Sword>();
        }
    }
}
```

```
public class Samurai : MonoBehaviour
{
    private WeaponFactory _weaponFactory;
    private IWeapon _weapon;

    [Inject]
    public void Construct(WeaponFactory weaponFactory)
    {
        _weaponFactory = weaponFactory;
    }

    private void Awake()
    {
        _weapon = _weaponFactory.Create();
    }

    private void Update()
    {
        if (Input.GetKey(KeyCode.Space))
        {
            _weapon.Hit();
        }
    }
}
```



Advantages and Disadvantages

- Runtime Changes
- Easier Unit Testing
- Boilerplate Code Reduction
- Easier Extending (Open Closed principle)
- Loose Coupling (Open Closed principle)
- Depending on Abstraction

- Injecting dependencies can be done by configuration files. runtime changes with DI would be super easy.

- dependencies can be injected according to conditions and states of the Application.

- Consider a situation that you want to test your application with **sqlite3** database but your production database is **PostgreSQL**. since client libraries are different, you can inject database client library.

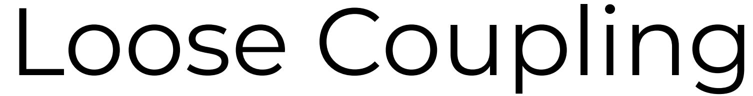
- If you need to **mock** some data you can add a mock class and inject it in testing mode.

- Because Instantiating of dependencies are not done in classes and DI framework does it, boilerplate code is reduced.

- Code duplication is reduced since instantiating dependencies and some logics implemented once.

- Consider that we want to add a new type of weapon to our example. we just add a Class for new Weapon (e.g AK47) and register it to DI Component.

- Hooray! No need to change Samurai class.

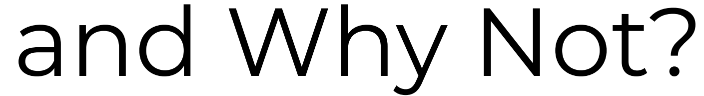


- Removing dependencies through DI leads to loose Coupling for classes in Application.

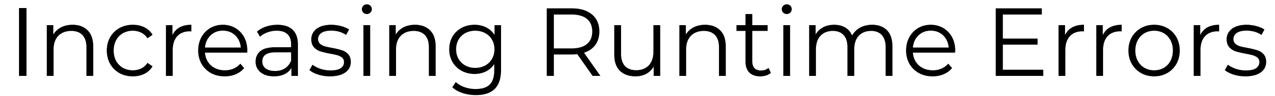
Depending on Abstraction

- Using DI enforces developers to follow the principle of "Depending on Abstraction"

- Abstraction causes easier extending and changing Applications.



- Increasing Runtime Errors
- IDE Automation? Forget about it
- Management Issues
- Slow Initialization
- Same libraries, Different APIs



- Consider a situation when a developer types **PostgrSQL** instead of **PostgreSQL** in configuration file**.** Program will compile but will fail at runtime.

- Having **Exception** in Conditions for Injecting Objects can lead to runtime Errors.

IDE Automation? Forget About it

- DI frameworks are Implemented with **Reflection** and **Dynamic Programming** in most of programming languages.

- Many IDE Automations like Code Completion, safe refactors, find references, show call hierarchy and etc will be out of service.

- Using DI for a **Constant** part of a program adds complexity and Code Base Management Issues.

- Overusing Using DI and Adding redundant complexity to Code Base, increases time of developing new features.

Slow Initialization

- DI will slow down Initialization of Applications. Reflection and DP have some overheads sometimes.

Same libraries, Different APIs

- We should have same interface for two object or library to inject them in applications interchangably.

- Some famous libraries with the same functionality, expose different APIs. so in order to use DI we should use wrapper classes for libraries. Considering and tracking library changes could spare time

Resources

- https://en.wikipedia.org/wiki/Dependency_injection
- https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/
- https://softwareengineering.stackexchange.com/questions/371722/criticism-and-disadvantages-of-dependency-injection
- https://martinfowler.com/articles/injection.html