39. What is the Dependency Injection design pattern?

Brief summary: Dependency Injection is providing the objects some class needs (its dependencies) from the outside, instead of having it construct them itself.

Dependency Injection means providing the objects that some class needs (its dependencies) from the outside, instead of having it construct them itself.

Let's see this in practice. First, the code that does **not** use the Dependency Injection:

```
class PersonalDataFormatter
{
    0 references
    public string Format()
    {
        var peopleDataReader = new PeopleDataReader();

        var people = peopleDataReader.ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }
}
```

The PersonalDataFormatter needs to use the PepleDataReader - it means, the PeopleDataReader is its **dependency**. In this code, the PersonalDataFormatter creates the PepleDataReader object itself using the **new** operator.

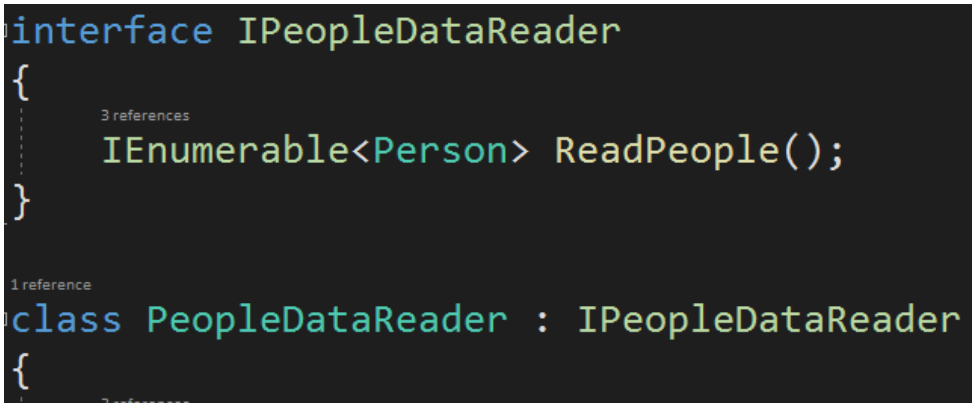
There are a couple of issues with this design:

- PersonalDataFormatter depends on a very particular implementation of people's data reading logic. What if we wanted to use a different data source? We would not have any way of doing this, as this class commits to using the specific PeopleDataReader object by creating it with the **new** operator. Now those two classes are **tightly coupled**.
- This is particularly problematic when we want to unit test this code. Let's assume the PeopleDataReader connects to a real database and sources

people's information from there. If we created the PersonalDataFormatter in tests, it would instantiate the PeopleDataReader, which would try to access the database. This is not acceptable in unit tests. We must have a way of providing a mock implementation instead. With the current design, it's not possible. We will learn more about mocks in the "**What are mocks?**" lecture.

- We are breaking the Single Responsibility Principle here. The PersonalDataFormatter should only be responsible for formatting personal data, but now it is also responsible for creating a PeopleDataReader object. In this simple code this may not seem like an issue, but keep in mind that in real-life applications it's often much more complicated to create an object, as it may have many dependencies of its own.

All right. Let's refactor this code to use Dependency Injection. First of all, let's make the PeopleDataReader implement an interface:



And now, let's inject this dependency to the PersonalDataFormatter, instead of creating it right in it:

```
class PersonalDataFormatter
{
    private readonly IPeopleDataReader _peopleDataReader;

    0 references
    public PersonalDataFormatter(
        IPeopleDataReader peopleDataReader)
    {
        _peopleDataReader = peopleDataReader;
    }

    0 references
    public string Format()
    {
        var people = _peopleDataReader.ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }
}
```

This **solves** all problems mentioned before:

- The classes are now **loosely coupled**. We can easily switch the object we pass to the PersonalDataFormatter's constructor to any other object implementing IPeopleDataReader interface. We can also do it at runtime. PersonalDataFormatter doesn't know anything about the concrete PeopleDataReader class. All it cares about is that it's being provided a dependency that can retrieve people's data - it doesn't care how it is done exactly.
- Because of that, we can easily provide a **mock** of the IPeopleDataReader in tests, to avoid connecting to a real database.
- The PersonalDataFormatter is no longer responsible for creating PeopleDataReader object. The creation of this object and using it are separated. The Single Responsiblilty Principle is not broken and we maintain the separation of concerns.

As you can see, the Dependency Injection is a straightforward pattern, yet it solves a lot of problems.

In C#, we most typically use the constructor injection - so the dependency is injected to a class via its constructor. It is also possible to inject dependency via a setter, but this is much less popular (as, in general, having a public setter is a risky business):

```
class PersonalDataFormatter
{
    1 reference
    public IPeopleDataReader PeopleDataReader { get; set; }

    0 references
    public string Format()
    {
        var people = PeopleDataReader.ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }
}
```

Before we mentioned that classes should not be responsible for creating their dependencies. Well, who should be responsible for it, then? Most typically we have two places where we construct objects, depending on our needs:

- if we need objects that can be constructed right at the program start (for example a logger that will be reused throughout the application) we can create them at the **entry point** of the application, like the Main method, and then pass them down to whatever class that need them:

```
public static void Main(string[] args)
{
    //creation of objects
    var logger = new Logger();
    var peopleDataReader = new PeopleDataReader(logger);
    var personalDataFormatter = new PersonalDataFormatter(
        peopleDataReader, logger);

    //actual run of the application
    Console.WriteLine(personalDataFormatter.Format());

    Console.ReadKey();
}
```

Please notice that in many real-life projects the creation of objects is not done manually, but with **Dependency Injection frameworks**. They are mechanisms that automatically create dependencies and inject them into objects that need them. Dependency Injection frameworks are configurable, so we can decide what concrete types will be injected into objects. They can also be configured to reuse one instance of some type or to create separate

instances for each object that needs them. Some of the popular Dependency Injection frameworks in C# are Autofac or Ninject.

- If we are not sure what objects exactly we need (a concrete type may depend on some parameter or configuration provided at runtime), or whether we will need them at all, we can use a **factory**. Let's say that in PersonalDataFormatter we can either use the default formatting or formatting provided from the outside. The decision which one will be used is done at runtime, and it depends on the value of a parameter of the Format method:

```
public string Format(bool isDefaultFormatting)
{
    _logger.Log("Formatter running...");
    var people = _peopleDataReader.ReadPeople();

    if (isDefaultFormatting)
    {
        return string.Join("\n",
          people.Select(p => $"{p.Name} born in" +
          $" {p.Country} on {p.YearOfBirth}"));
    }
    var formatter = _formatterFactory.Create();
    return formatter.Format(people);
}
```

If the isDefaultFormatting parameter is set to true, we don't need to create a Formatter object at all. In other words, the action of creating an object must happen right in this class, so this object can't be injected. But we don't want to lose the benefits of dependency injection.

A factory allows us to achieve this. Please note that the factory returns an interface, so for testing purposes, we can provide a mock of the factory, that will create a mock of an actual Formatter.

```
interface IFormatterFactory
{
    2 references
    IFormatter Create();
}

1 reference
class FormatterFactory : IFormatterFactory
{
    2 references
    public IFormatter Create()
    {
        return new NameOnlyFormatter();
    }
}
```

```
interface IFormatter
{
    2 references
    string Format(IEnumerable<Person> people);
}

1 reference
class NameOnlyFormatter : IFormatter
{
    2 references
    public string Format(IEnumerable<Person> people)
    {
        return string.Join("\n",
            people.Select(p => p.Name));
    }
}
```

This way we don't need to create the Formatter object upfront. Maybe it will not be created at all if the Format method is never called with the isDefaultFormatting parameter set to false. Of course, in this sample code it wouldn't matter that much, but again: in a real-life application the creation of an object might be more complicated and performance-costly.

All right. We learned that Dependency Injection is a design pattern, according to which we should provide the dependencies that an object needs instead of having it construct them itself.

Dependency Injection is a specific kind of Inversion of Control, which we will learn about in the next lecture.

Bonus questions:

- "**What are Dependency Injection frameworks?**"
  *Dependency Injection frameworks are mechanisms that automatically create dependencies and inject them into objects that need them. They are configurable, so we can decide what concrete types will be injected into objects depending on some abstractions. They can also be configured to reuse one instance of some type or to create separate instances for each object that needs them. Some of the popular Dependency Injection frameworks in C# are Autofac or Ninjec*t.

- "**What are the benefits of using Dependency Injection?**"
  Dependency Injection decouples a class from its dependencies. The class doesn't make the decision of what concrete type it will use, it only declares in the constructor what interfaces it will need. Thanks to that, we can easily switch the dependencies according to our needs, which is particularly useful when injecting mock implementations for testing purposes.