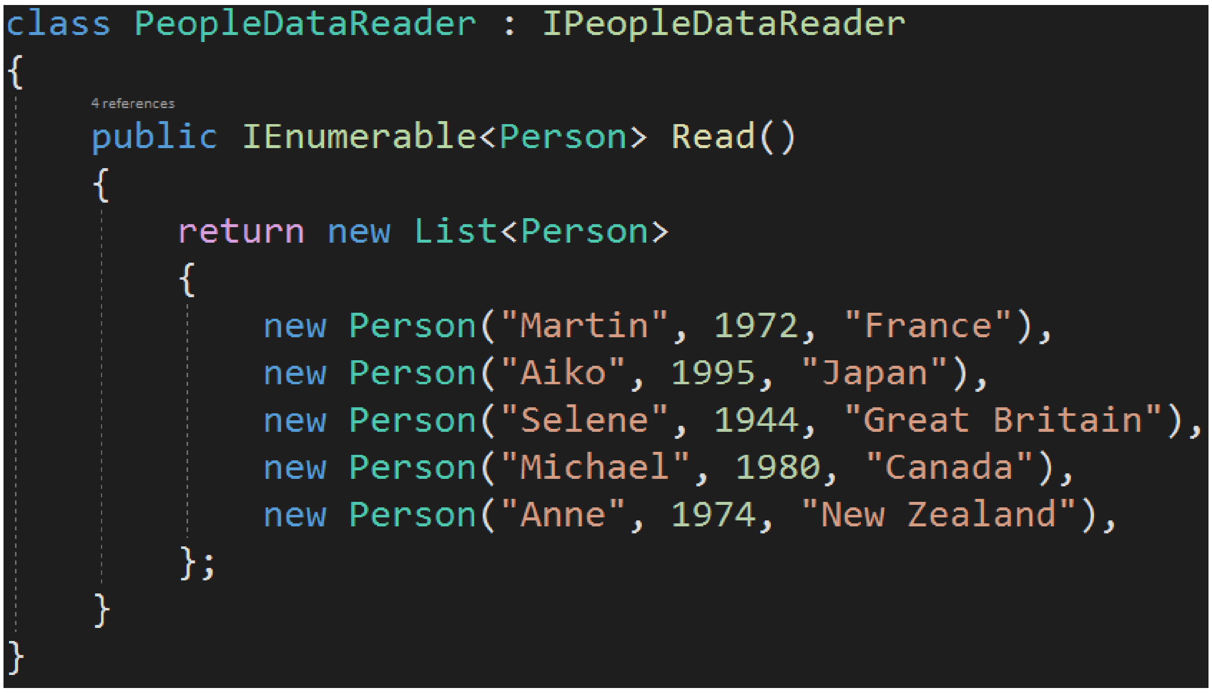
41. What is the Decorator design pattern?

Brief summary: Decorator is a design pattern that dynamically adds extra functionality to an existing object, without affecting the behavior of other objects from the same class.

Decorator is a design pattern that dynamically adds extra functionality to an existing object, without affecting the behavior of other objects from the same class.

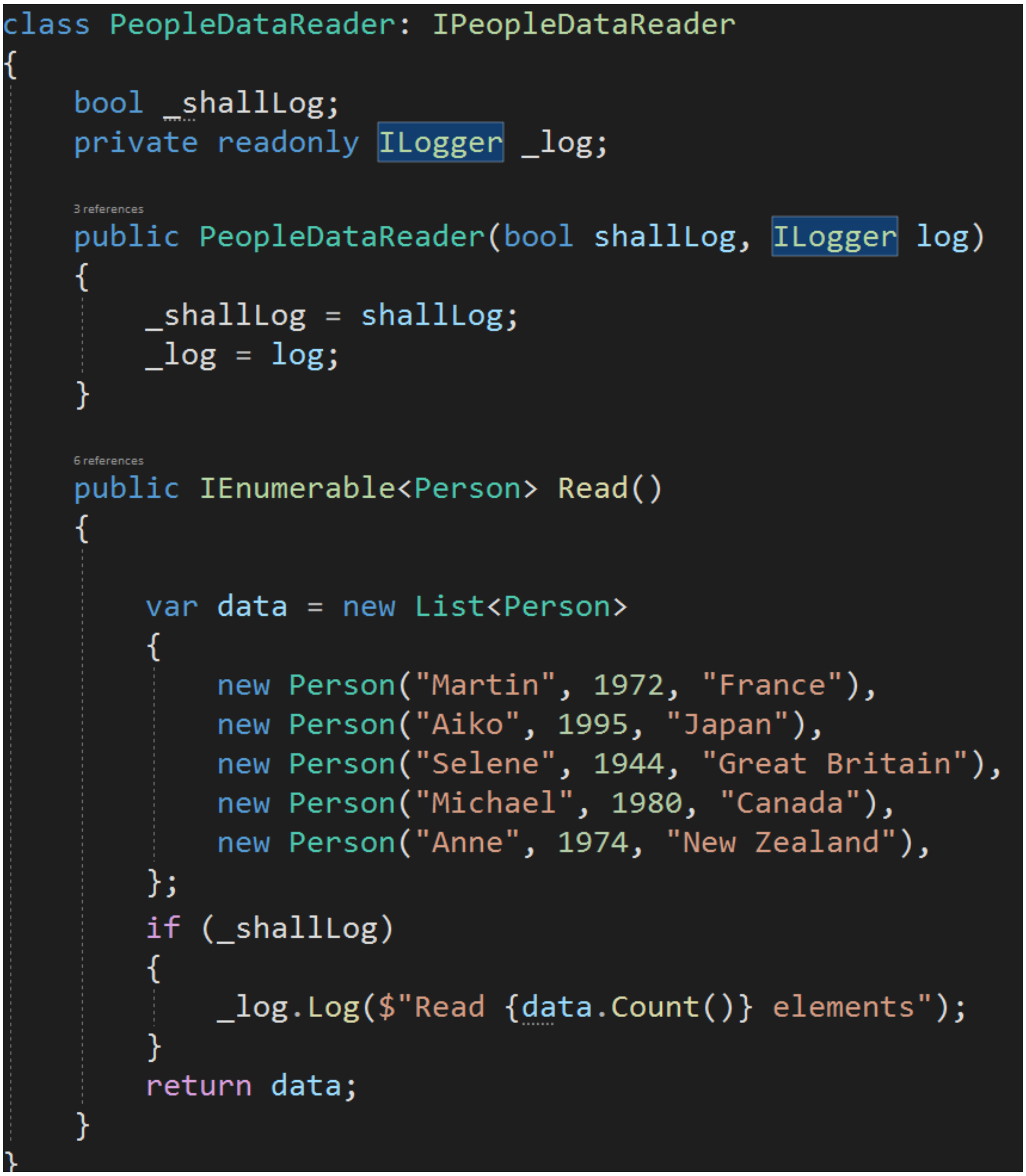
Let's start with something simple. We have a class that reads information about people from some data source:



```
class PeopleDataReader : IPeopleDataReader
{
    4 references
    public IEnumerable<Person> Read()
    {
        return new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
            new Person("Michael", 1980, "Canada"),
            new Person("Anne", 1974, "New Zealand"),
        };
    }
}
```

This class is nice, simple, and focused. *For now.*

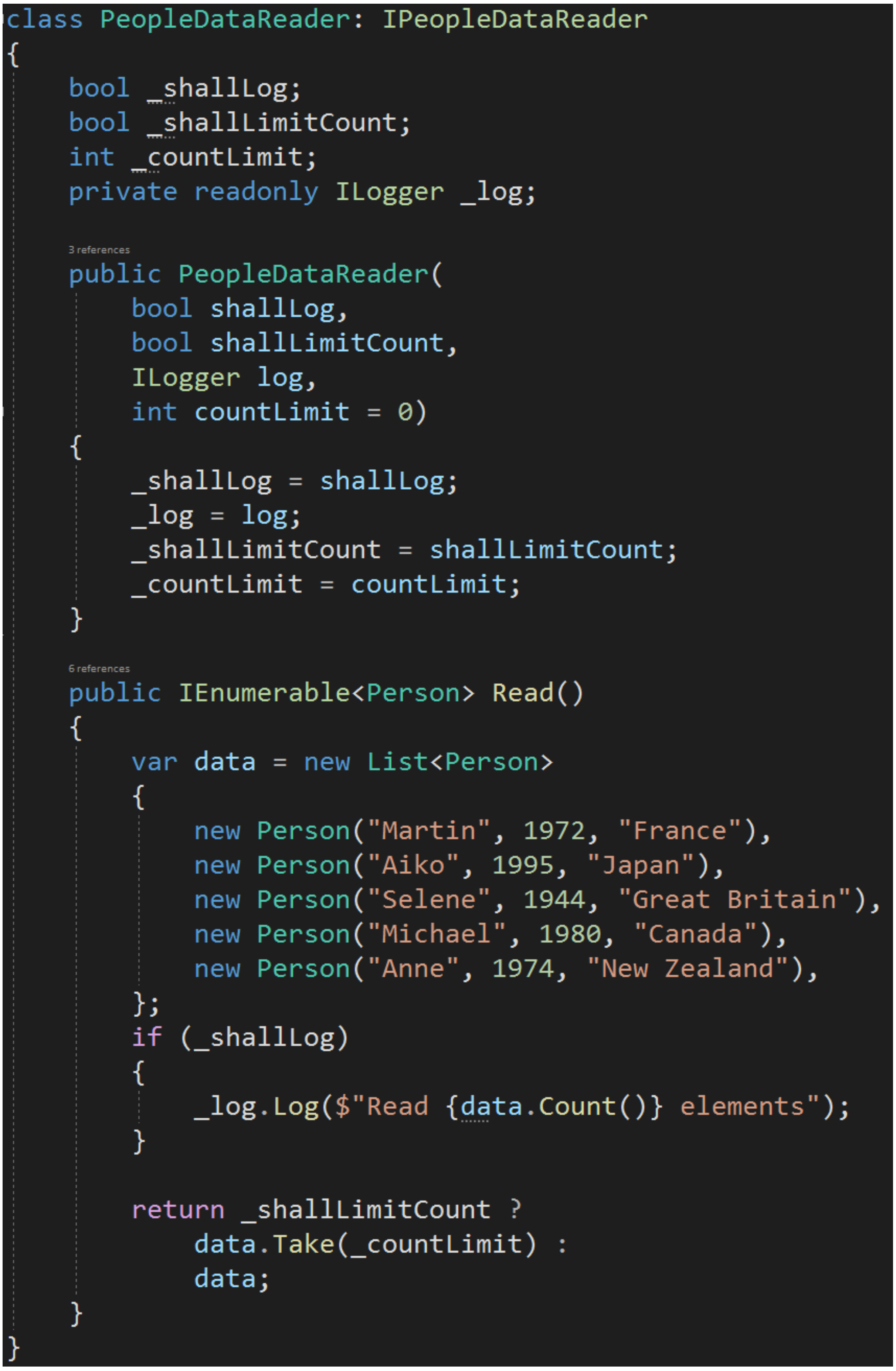
At some point, we are asked to add an optional feature of logging how many elements have been read. Let's add this feature to the class:



```
class PeopleDataReader: IPeopleDataReader
{
    bool _shallLog;
    private readonly ILogger _log;

    3 references
    public PeopleDataReader(bool shallLog, ILogger log)
    {
        _shallLog = shallLog;
        _log = log;
    }

    6 references
    public IEnumerable<Person> Read()
    {

        var data = new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
            new Person("Michael", 1980, "Canada"),
            new Person("Anne", 1974, "New Zealand"),
        };
        if (_shallLog)
        {
            _log.Log($"Read {data.Count()} elements");
        }
        return data;
    }
}
```

Ouch. It was such a pretty class, and now it grew large and ugly. Well, never mind… at least it does what it's supposed to.

Soon after that change, we are asked to add one more optional feature: to be able to limit data to some given count of People. Let's try to add this:

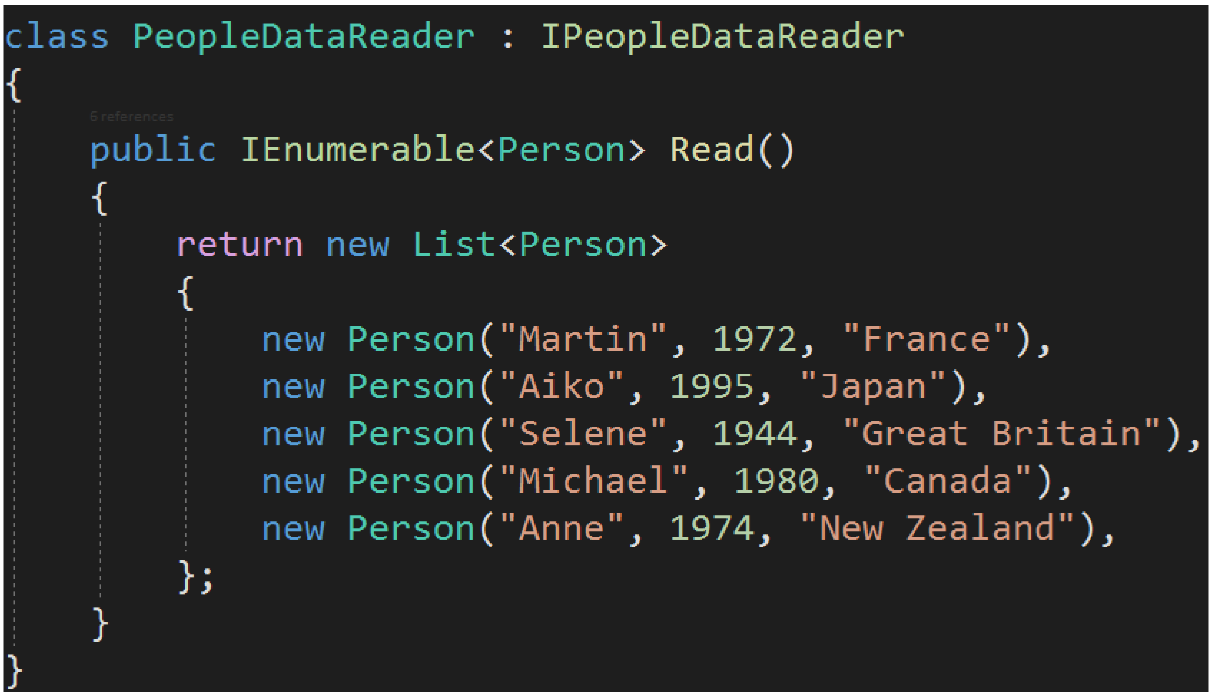

```
class PeopleDataReader: IPeopleDataReader
{
    bool _shallLog;
    bool _shallLimitCount;
    int _countLimit;
    private readonly ILogger _log;

    3 references
    public PeopleDataReader(
        bool shallLog,
        bool shallLimitCount,
        ILogger log,
        int countLimit = 0)
    {
        _shallLog = shallLog;
        _log = log;
        _shallLimitCount = shallLimitCount;
        _countLimit = countLimit;
    }

    6 references
    public IEnumerable<Person> Read()
    {
        var data = new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
            new Person("Michael", 1980, "Canada"),
            new Person("Anne", 1974, "New Zealand"),
        };
        if (_shallLog)
        {
            _log.Log($"Read {data.Count()} elements");
        }

        return _shallLimitCount ?
            data.Take(_countLimit) :
            data;
    }
}
```

This code is terrible. This class has big chunks of logic which will or will not be executed depending on the flags. Its logic, so simple before, is now messy and complex. It takes more parameters than it may need (we don't need a logger if logging is not enabled, and we don't need **countLimit** if limiting is not enabled). It will be a nightmare to test it.

As more and more extra features are required to be added to this class, it will keep growing, becoming an unmanageable mess that no one wants to work with.

It's time to introduce the **Decorator** design pattern. This pattern allows adding some behavior to an object dynamically, without touching its code. If you know the **Open-Closed Principle** from SOLID, you know this is a good thing. It also allows us to keep the **Single Responsibility Principle** happy.

First, let's revert this class to how it was before changes:



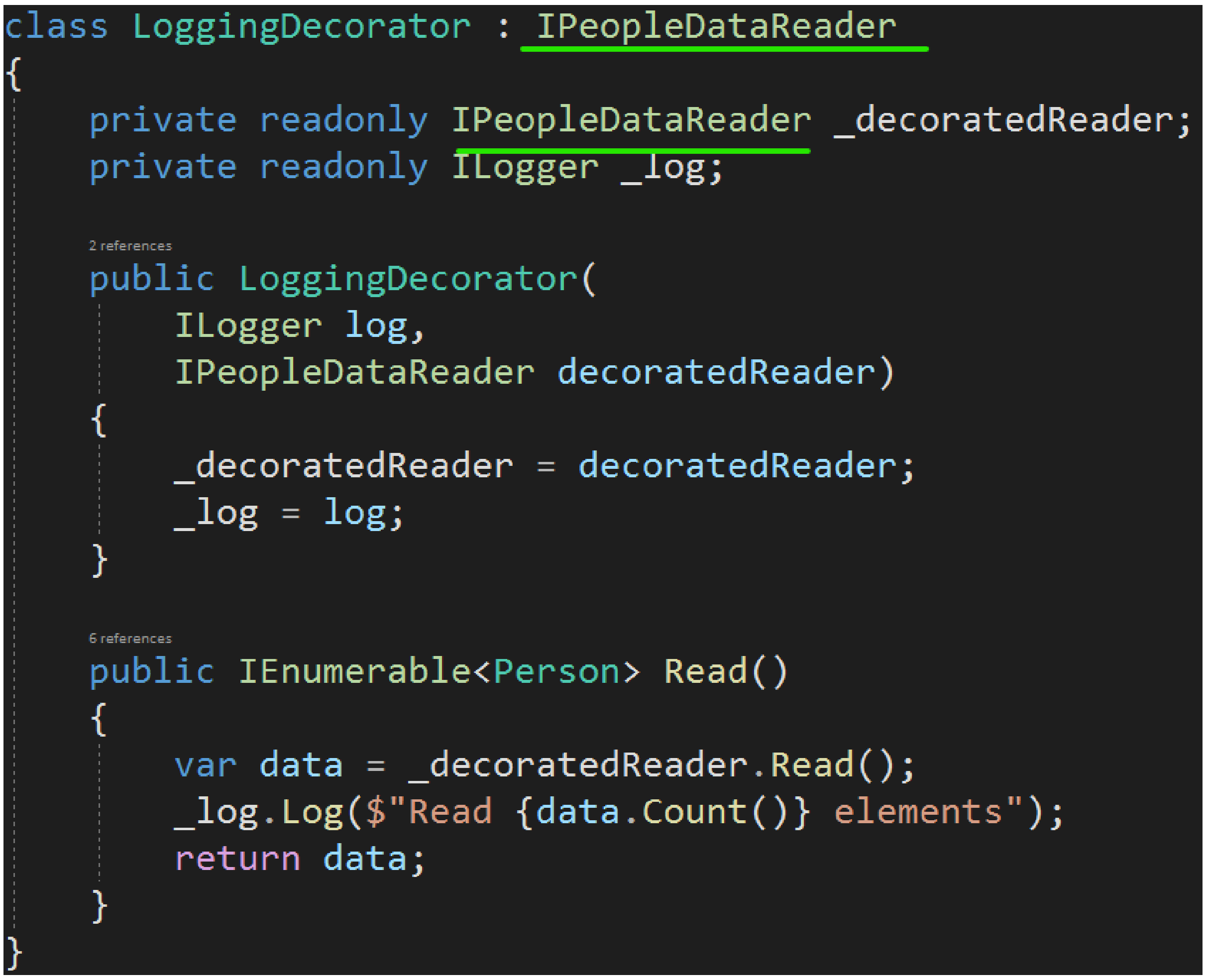
```
class PeopleDataReader : IPeopleDataReader
{
    public IEnumerable<Person> Read()
    {
        return new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
            new Person("Michael", 1980, "Canada"),
            new Person("Anne", 1974, "New Zealand"),
        };
    }
}
```

Beautiful in its simplicity. Now, let's add LoggingDecorator class. It will "decorate" the PeopleDataReader with the ability of logging.

Implementing the Decorator design pattern boils down to two steps:

- making the Decorator **implement the same interface as the decorated object**
- making the Decorator **own an object implementing this interface**. It will be the decorated class itself or another Decorator, which allows us to compose many Decorators together

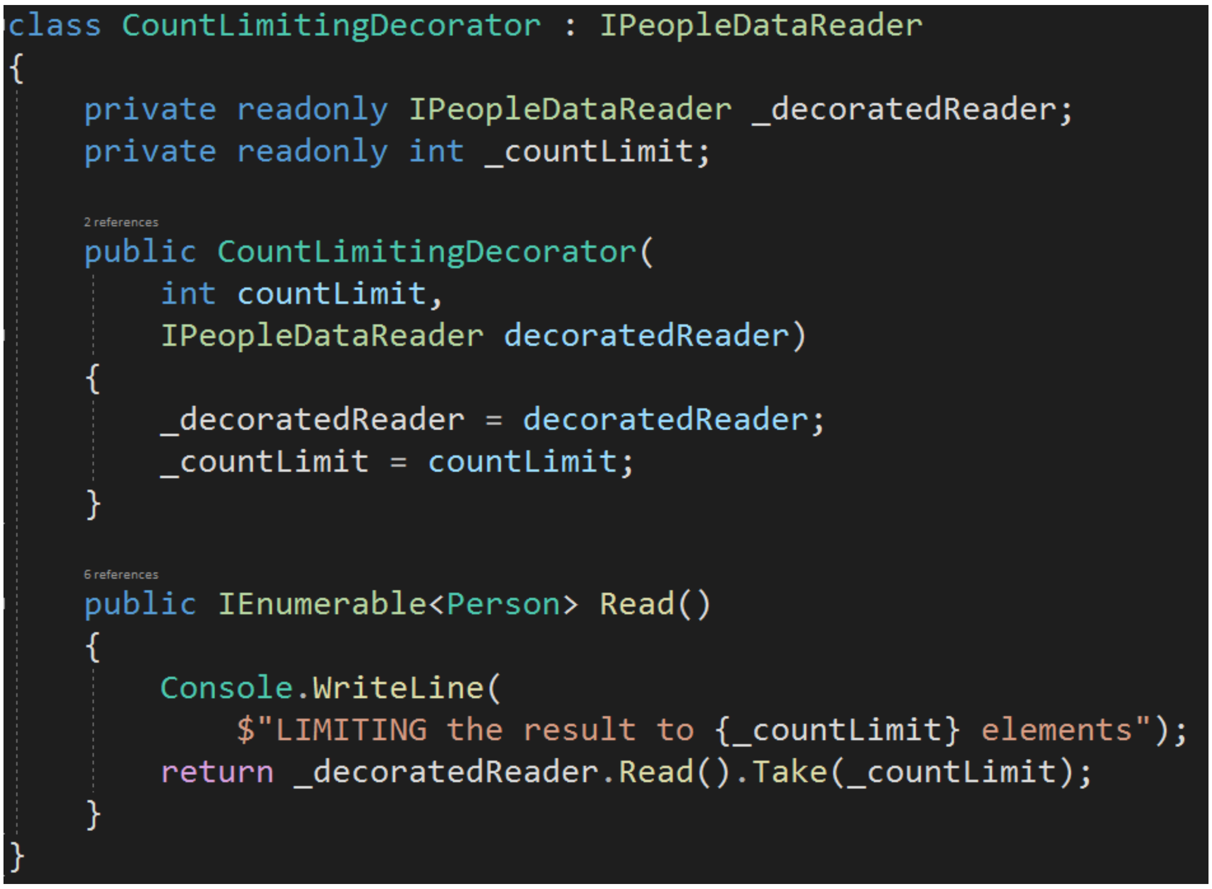
Let’s see how it looks in code:


```
class LoggingDecorator : IPeopleDataReader
{
    private readonly IPeopleDataReader _decoratedReader;
    private readonly ILogger _log;

    2 references
    public LoggingDecorator(
        ILogger log,
        IPeopleDataReader decoratedReader)
    {
        _decoratedReader = decoratedReader;
        _log = log;
    }

    6 references
    public IEnumerable<Person> Read()
    {
        var data = _decoratedReader.Read();
        _log.Log($"Read {data.Count()} elements");
        return data;
    }
}
```

As you can see, the Decorator owns an object that it wants to decorate. It implements the same interface. In the Read method that comes from the interface, it calls whatever implementation is provided, but it adds a little something from itself - in this case, it writes to a log.

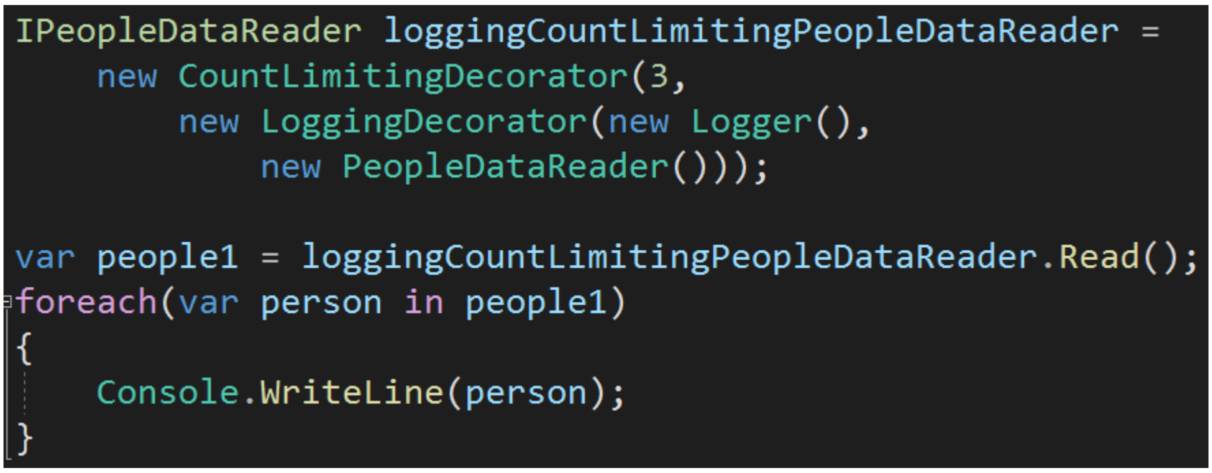
Remember that the **_decoratedReader** doesn’t need to be the plain PeopleDataReader object - it **can be anything implementing the IPeopleDataReader interface, including another Decorator**. Of course, at some point one of the Decorators in this structure must own the basic decorated object of PeopleDataReader type.

Let’s now add the Decorator that will be limiting the count of returned Person objects:



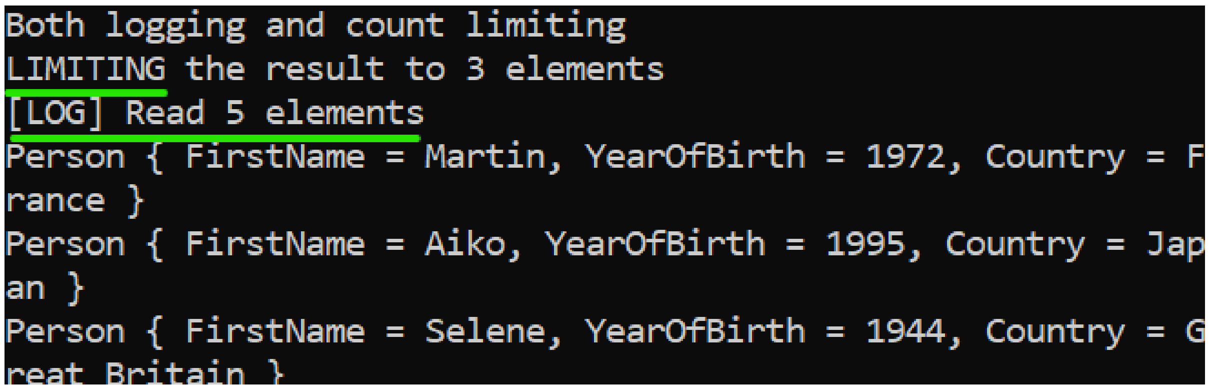
```csharp
class CountLimitingDecorator : IPeopleDataReader
{
    private readonly IPeopleDataReader _decoratedReader;
    private readonly int _countLimit;

    public CountLimitingDecorator(
        int countLimit,
        IPeopleDataReader decoratedReader)
    {
        _decoratedReader = decoratedReader;
        _countLimit = countLimit;
    }

    public IEnumerable<Person> Read()
    {
        Console.WriteLine(
            $"LIMITING the result to {_countLimit} elements");
        return _decoratedReader.Read().Take(_countLimit);
    }
}
```

Great. We can now compose those Decorators to our liking. Let's create an object that reads people data, logs the original count, and then limits it:



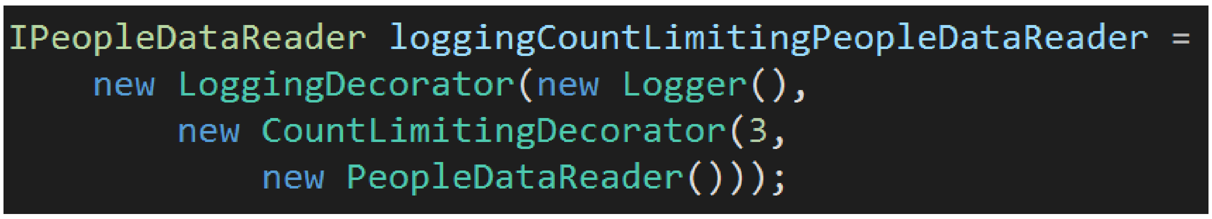
```csharp
IPeopleDataReader loggingCountLimitingPeopleDataReader =
    new CountLimitingDecorator(3,
        new LoggingDecorator(new Logger(),
            new PeopleDataReader()));

var people1 = loggingCountLimitingPeopleDataReader.Read();
foreach(var person in people1)
{
    Console.WriteLine(person);
}
```

Here the real magic happens. Each Decorator takes any object implementing the IPeopleDataReader interface as a parameter but also implements this interface itself. It means, we can pass a Decorator as a parameter to other Decorator, stacking their functionalities. That's why the final object will be able to both log and limit the count of elements:



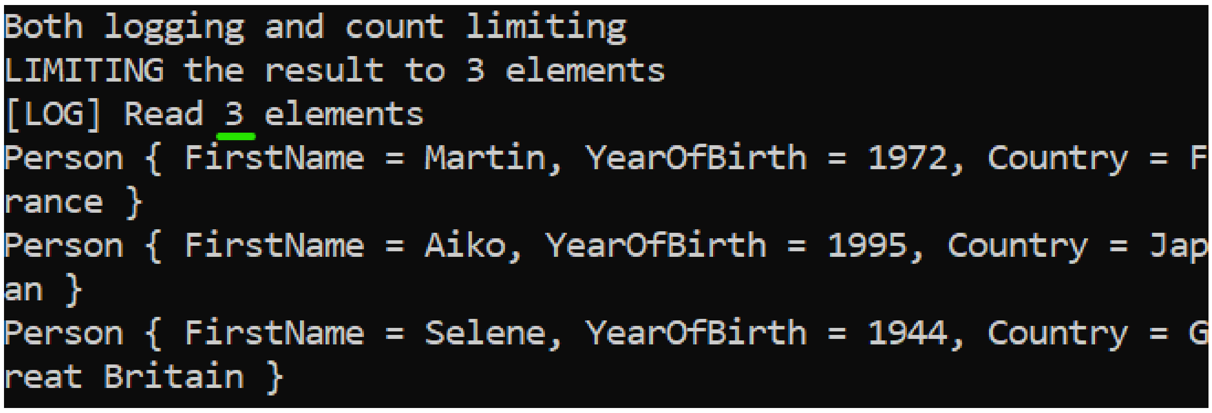
```
Both logging and count limiting
LIMITING the result to 3 elements
[LOG] Read 5 elements
Person { FirstName = Martin, YearOfBirth = 1972, Country = F
rance }
Person { FirstName = Aiko, YearOfBirth = 1995, Country = Jap
an }
Person { FirstName = Selene, YearOfBirth = 1944, Country = G
reat Britain }
```

Please be aware that the order of the Decorators creation matters. If we change this code to this…



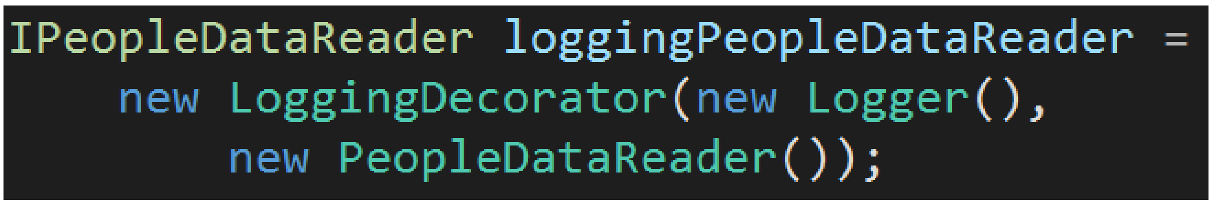
```
IPeopleDataReader loggingCountLimitingPeopleDataReader =
    new LoggingDecorator(new Logger(),
        new CountLimitingDecorator(3,
            new PeopleDataReader()));
```

…the result will be different because the limiting Decorator's Read method will be executed before the logging Decorator's Read method. From the point of view of the LoggingDecorator the count of data will be 3, not 5.



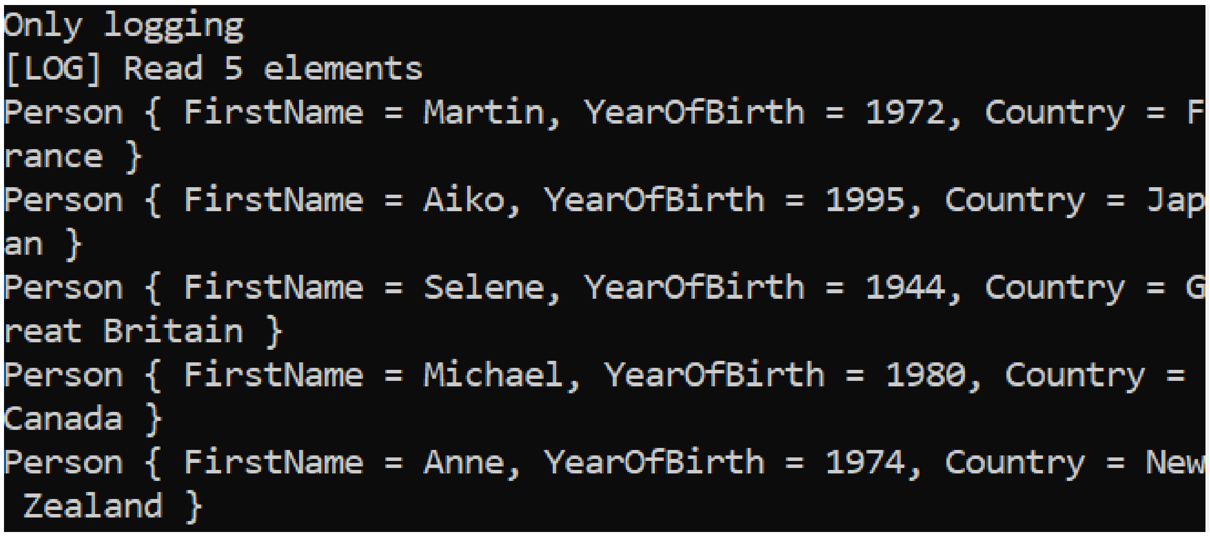
```
Both logging and count limiting
LIMITING the result to 3 elements
[LOG] Read 3 elements
Person { FirstName = Martin, YearOfBirth = 1972, Country = F
rance }
Person { FirstName = Aiko, YearOfBirth = 1995, Country = Jap
an }
Person { FirstName = Selene, YearOfBirth = 1944, Country = G
reat Britain }
```

The features of logging and limiting are optional, but with the Decorator pattern, it's easy to choose what we need. Let's create a PeopleDataReader that only logs some information, but does not limit the count:



```
IPeopleDataReader loggingPeopleDataReader =
    new LoggingDecorator(new Logger(),
        new PeopleDataReader());
```

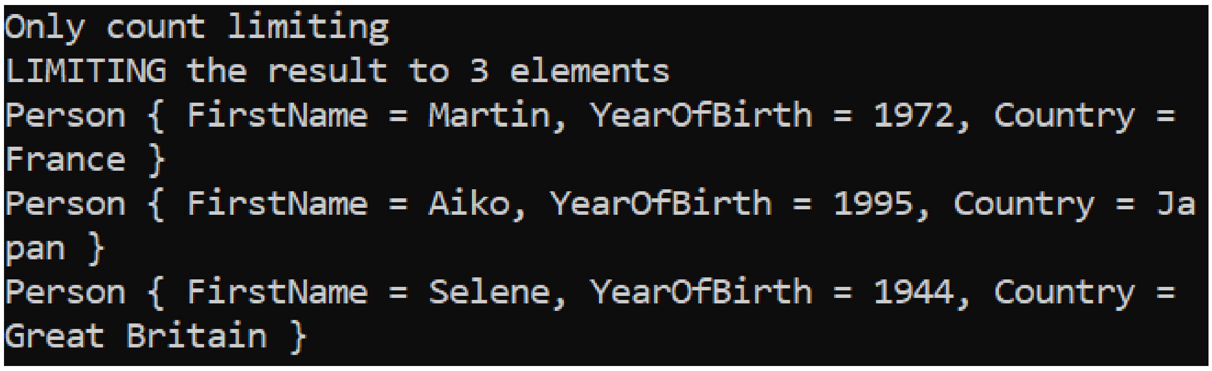
In the result we will see all 5 elements:



```
Only logging
[LOG] Read 5 elements
Person { FirstName = Martin, YearOfBirth = 1972, Country = F
rance }
Person { FirstName = Aiko, YearOfBirth = 1995, Country = Jap
an }
Person { FirstName = Selene, YearOfBirth = 1944, Country = G
reat Britain }
Person { FirstName = Michael, YearOfBirth = 1980, Country =
Canada }
Person { FirstName = Anne, YearOfBirth = 1974, Country = New
 Zealand }
```

And now, let's create an object that does not log, but it does limit the data:

```
IPeopleDataReader countLimitingPeopleDataReader =
    new CountLimitingDecorator(3,
        new PeopleDataReader());
```



```
Only count limiting
LIMITING the result to 3 elements
Person { FirstName = Martin, YearOfBirth = 1972, Country =
France }
Person { FirstName = Aiko, YearOfBirth = 1995, Country = Ja
pan }
Person { FirstName = Selene, YearOfBirth = 1944, Country =
Great Britain }
```

As you can see, there is no "[LOG]" string in this result.

All right. As you can see the Decorator pattern allows us to easily add functionality to objects, without touching the original classes, so it's very much in line with the Open-Closed Principle. It allows us to keep classes simple. It also helps us to be in line with the Single Responsibility Principle, as each class now has a very focused responsibility. They would be easy to test, maintain, and generally pleasant to works with.

Bonus questions:

- "**What are the benefits of using the Decorator design pattern?**"
  The Decorator pattern allows us to easily add functionality to objects, without touching the original classes, so it’s very much in line with the Open-Closed Principle. *It allows us to keep classes simple. It makes it easy to stack functionalities together, building complex objects from simple classes. It also helps us to be in line with the Single Responsibility Principle, as each class now has a very focused responsibility. They would be easy to test, maintain, and generally pleasant to works with.*