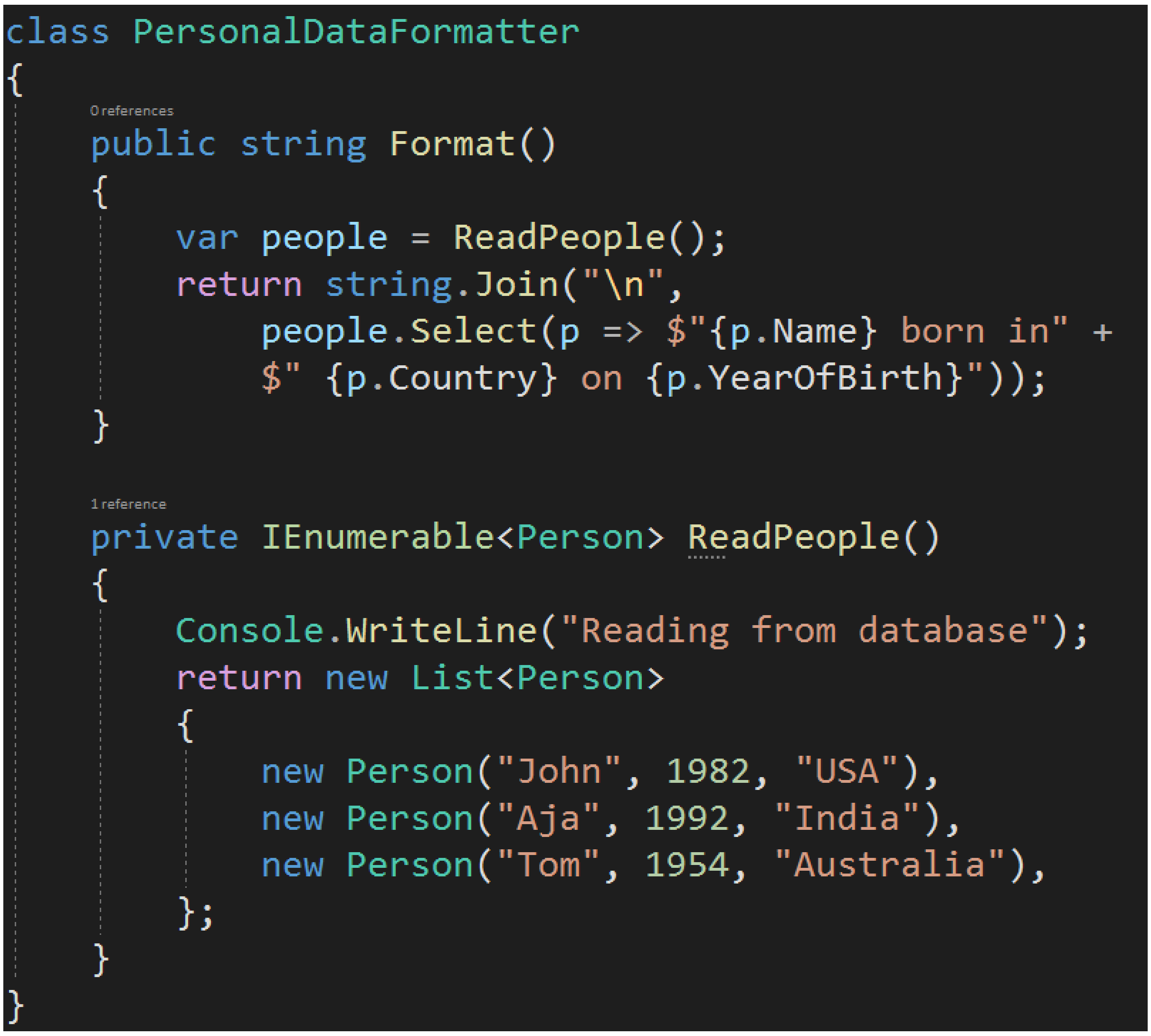
45. What is the “composition over inheritance” principle?

Brief summary: “Composition over inheritance” is a design principle stating that we should favor composition over inheritance. In other words, we should reuse the code by rather containing objects within other objects, than inheriting one from another.

“Composition over inheritance” is a design principle stating that we should favor composition over inheritance. In other words, we should reuse the code by rather containing objects within other objects, than inheriting one from another.
Let’s see a practical example.



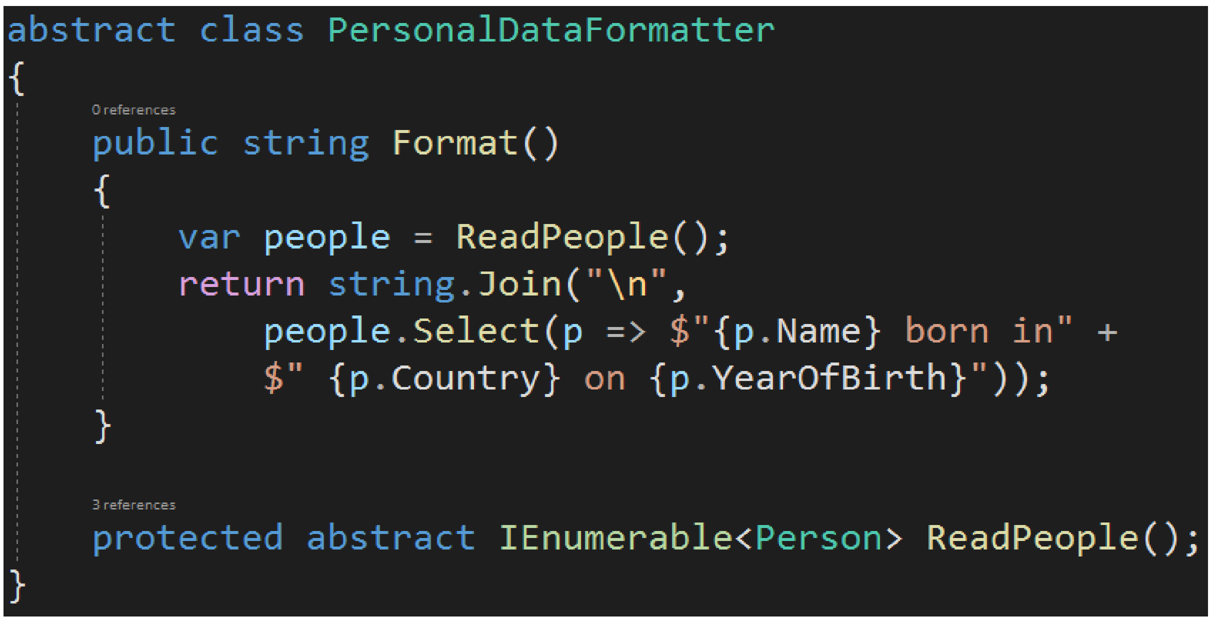
```
class PersonalDataFormatter
{
    public string Format()
    {
        var people = ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }

    private IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from database");
        return new List<Person>
        {
            new Person("John", 1982, "USA"),
            new Person("Aja", 1992, "India"),
            new Person("Tom", 1954, "Australia"),
        };
    }
}
```

This class reads people's data from a database and formats it as a single string. It's obviously breaking the Single Responsibility Principle, but let's by now not focus on that.

One day the business requirements change, and we are told that sometimes the people's information will be read from the database, but sometimes from an Excel file. We want to be able to make this decision at runtime.

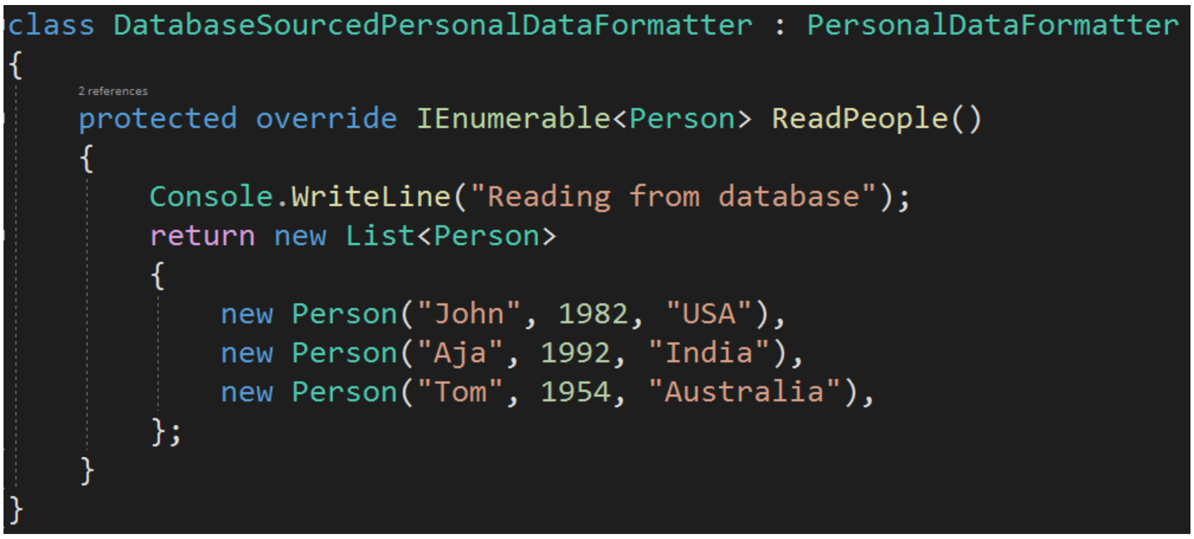
We can solve it in two ways - by either using composition, and injecting an object implementing some IPeopleDataReader interface with this class's constructor, or we can use inheritance.

First, let's solve this with inheritance. I will make the PersonalDataFormatter class abstract:

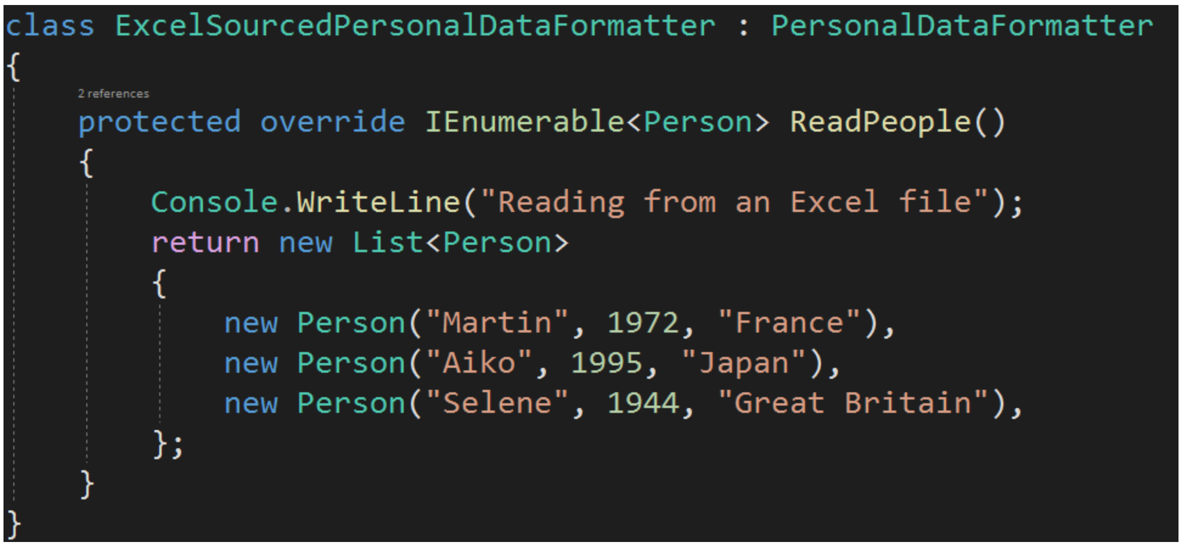


```
abstract class PersonalDataFormatter
{
    0 references
    public string Format()
    {
        var people = ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }

    3 references
    protected abstract IEnumerable<Person> ReadPeople();
}
```

The ReadPeople method is also abstract, so it will have to be overridden in inheritors:

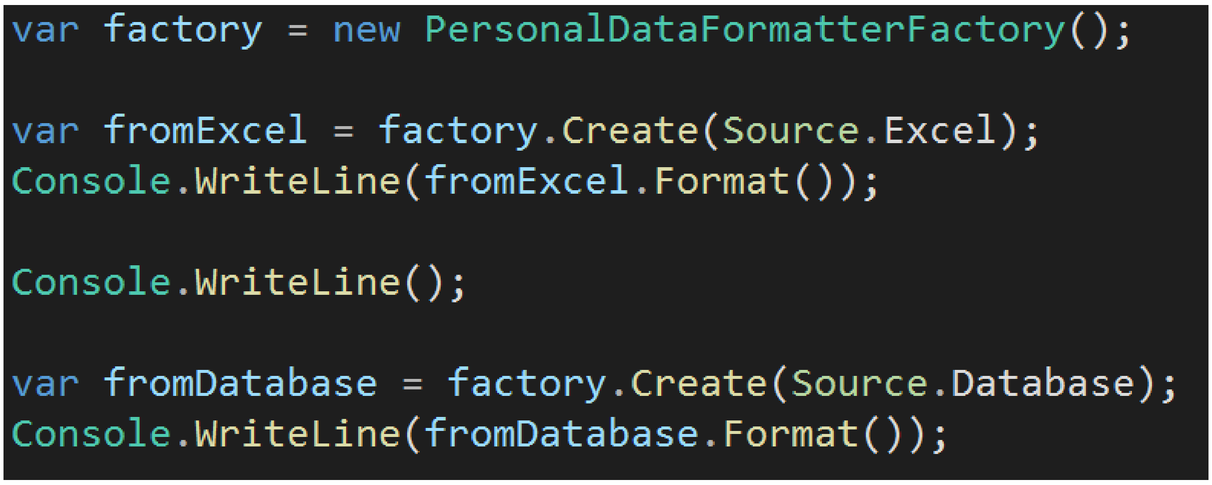


```
class DatabaseSourcedPersonalDataFormatter : PersonalDataFormatter
{
    2 references
    protected override IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from database");
        return new List<Person>
        {
            new Person("John", 1982, "USA"),
            new Person("Aja", 1992, "India"),
            new Person("Tom", 1954, "Australia"),
        };
    }
}
```

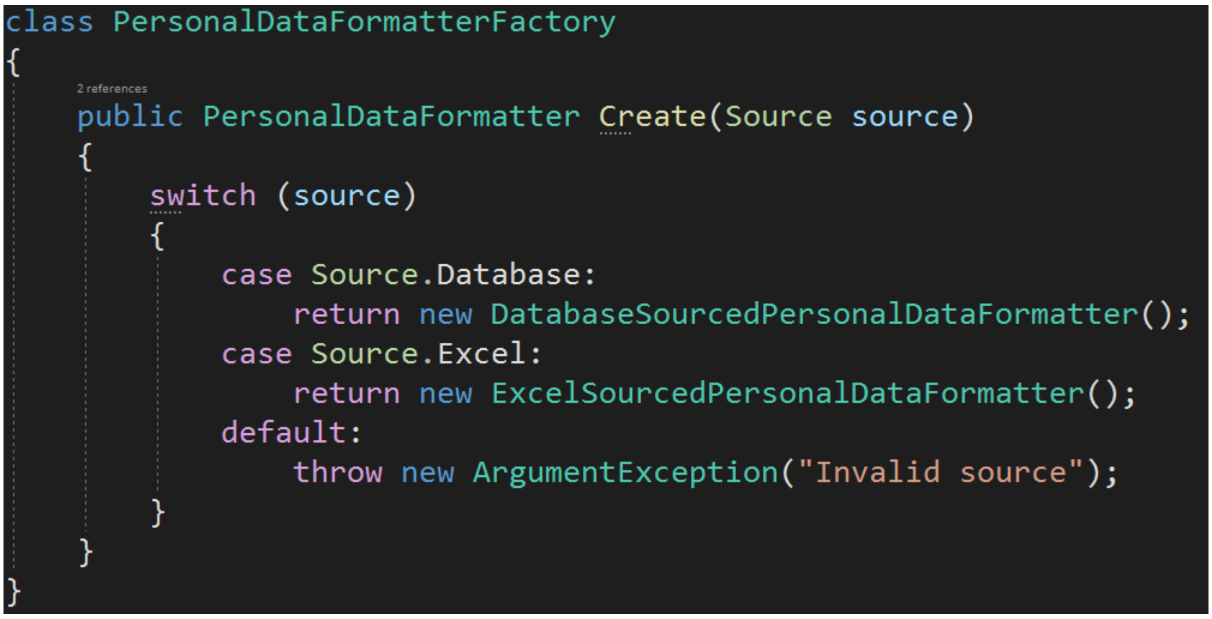


```
class ExcelSourcedPersonalDataFormatter : PersonalDataFormatter
{
    2 references
    protected override IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from an Excel file");
        return new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
        };
    }
}
```

Great. We achieved what we wanted - we can now format the personal data sourced from both databases and Excel files. We don't have any code duplications, and we can decide the type at runtime, using some Factory.



```
var factory = new PersonalDataFormatterFactory();

var fromExcel = factory.Create(Source.Excel);
Console.WriteLine(fromExcel.Format());

Console.WriteLine();

var fromDatabase = factory.Create(Source.Database);
Console.WriteLine(fromDatabase.Format());
```



```
class PersonalDataFormatterFactory
{
    2 references
    public PersonalDataFormatter Create(Source source)
    {
        switch (source)
        {
            case Source.Database:
                return new DatabaseSourcedPersonalDataFormatter();
            case Source.Excel:
                return new ExcelSourcedPersonalDataFormatter();
            default:
                throw new ArgumentException("Invalid source");
        }
    }
}
```

Everything looks good, doesn't it?

Well… not so fast. Let me tell you why using inheritance, in this case, wasn't our brightest idea.

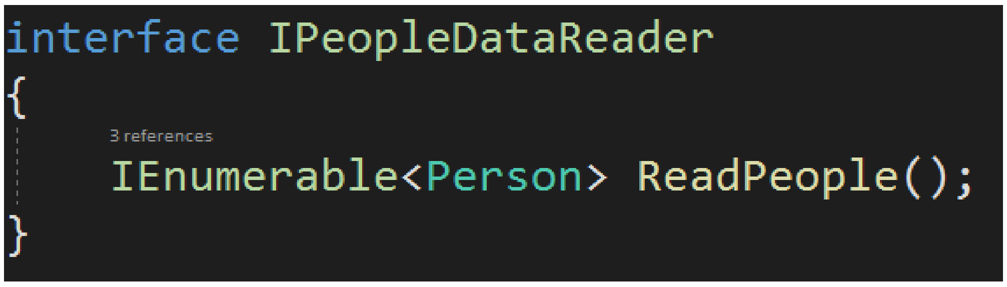
- The PersonalDataFormatter class is tightly coupled with its inheritors now. Any change in the base class will affect the child classes. We can't really use any of those types without the others provided, so if I wanted to use ReadPeople method anywhere else, I would not be able without engaging this entire hierarchy of classes. You can learn more about coupling in the "What is coupling?" lecture.
- The relation between those particular classes is rigid - it is defined at compile time. If I had some other mechanism that can read people's information from some source, I wouldn't be able to use it here without creating another derived type.
- Also, we have all the limitations of inheritance here, especially the fact that we can only inherit from a single base class.
- This example is simple, but if we needed some other changes that would make those classes different, we would have the inheritance hierarchy growing really fast. For example, if the way of formatting the final string would also need to be configurable, we would need to create even more classes, like:
  DatabaseSourcedPersonalData**Short**Formatter,
  DatabaseSourcedPersonalData**Full**Formatter,
  ExcelSourcedPersonalData**Short**Formatter,
  ExcelSourcedPersonalData**Full**Formatter
  Such hierarchy would soon become unmanageable.

- If we wanted to create unit tests for those classes, it would be tricky, and there are actually two approaches for testing abstract classes and their inheritors, both with their own disadvantages:
    - We can test both inheritors, but in both of them, we will also test the common part belonging to the base class. Our tests will be partially duplicated.
    - We can test inheritors ignoring the logic belonging to the base type as much as possible. Then, we can test the base abstract class logic by creating for the testing purposes a special, dedicated concrete type derived from it. In the tests of this class, we would focus on testing the base class logic. This is even worse than the first point - if you need to create special inheritors classes for testing purposes only, it means you messed your design up badly.
- It is often the case that we inherit more than we would actually want. The base class is exposing the implementation details to inheritors.
- There is one more reason for avoiding inheritance that is not really related to this example, but I want to mention it anyway: if we use inheritance in a hierarchy of objects that we intend to store in a database using some Object-Relational Mapping tools like Entity Framework, it may be a challenge to store those objects properly. Databases don't easily "understand" inheritance, so mapping the C#'s hierarchy of inheritance into a flat structure of tables is tricky, and often leads to overcomplicating the model in the database.

So how to solve all of it?

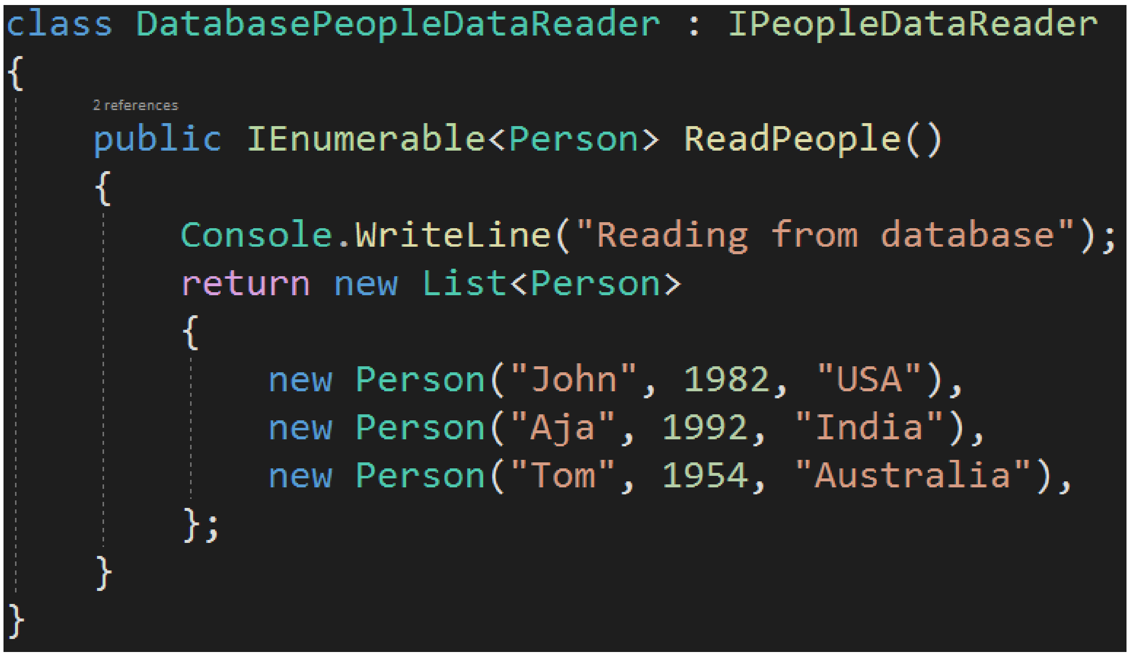
Well, in this case, we should definitely apply the "composition over inheritance" principle. Let's refactor this code:

First of all, I will introduce an interface:

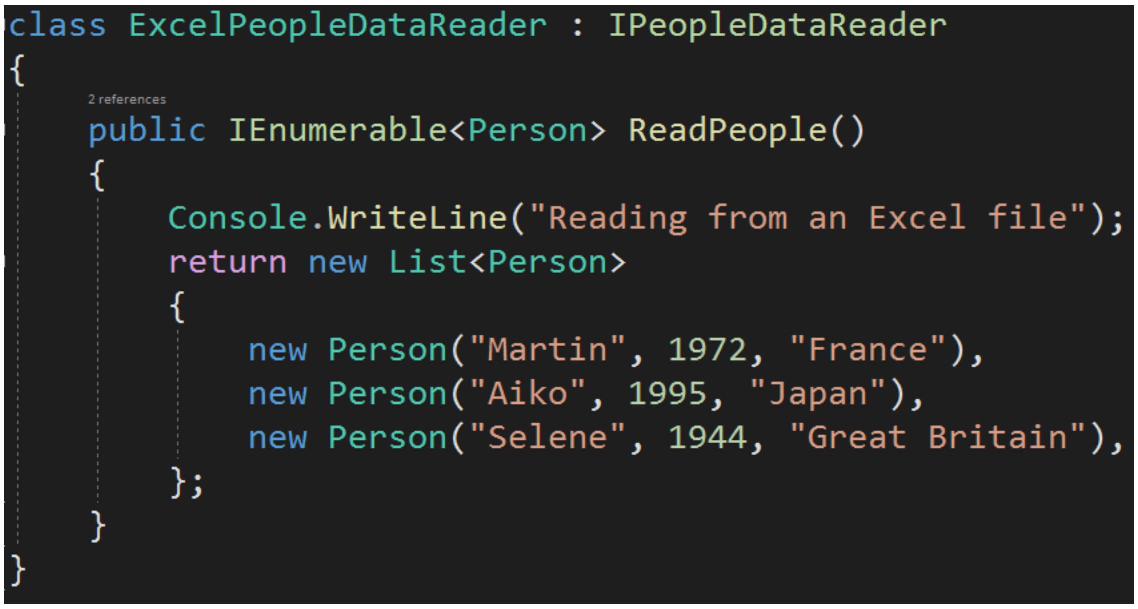


```
interface IPeopleDataReader
{
    3 references
    IEnumerable<Person> ReadPeople();
}
```

I will have two classes implementing it:

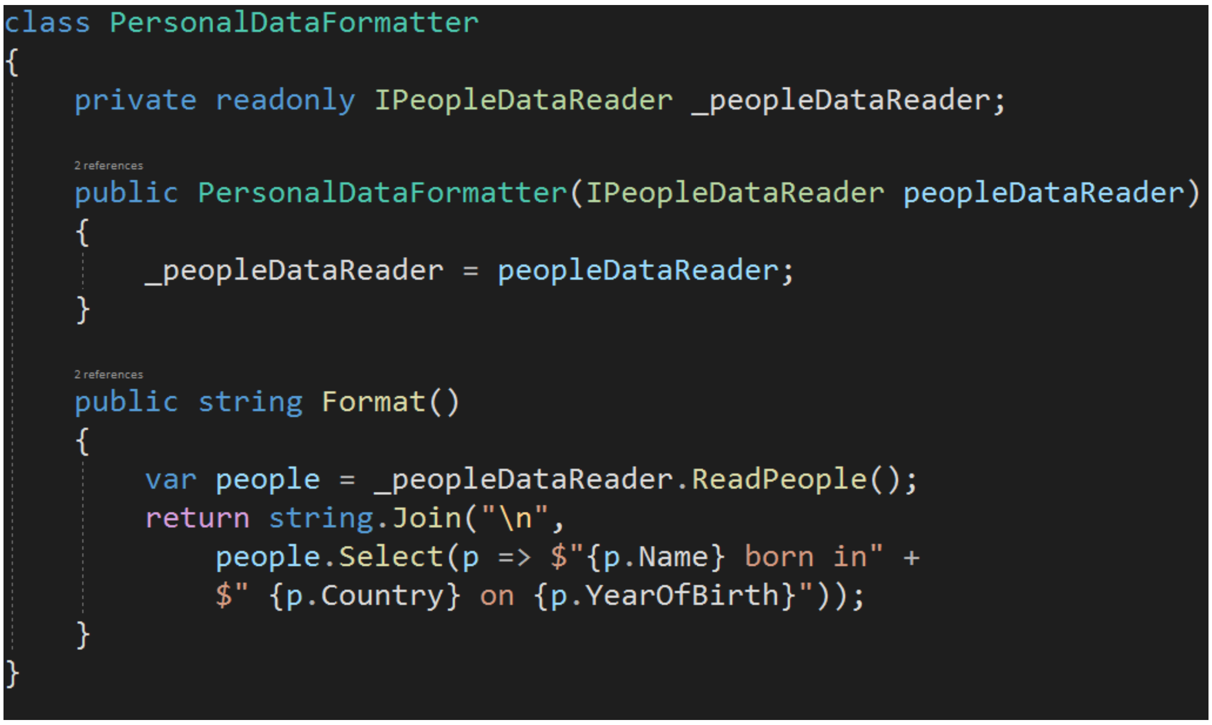


```
class DatabasePeopleDataReader : IPeopleDataReader
{
    2 references
    public IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from database");
        return new List<Person>
        {
            new Person("John", 1982, "USA"),
            new Person("Aja", 1992, "India"),
            new Person("Tom", 1954, "Australia"),
        };
    }
}
```



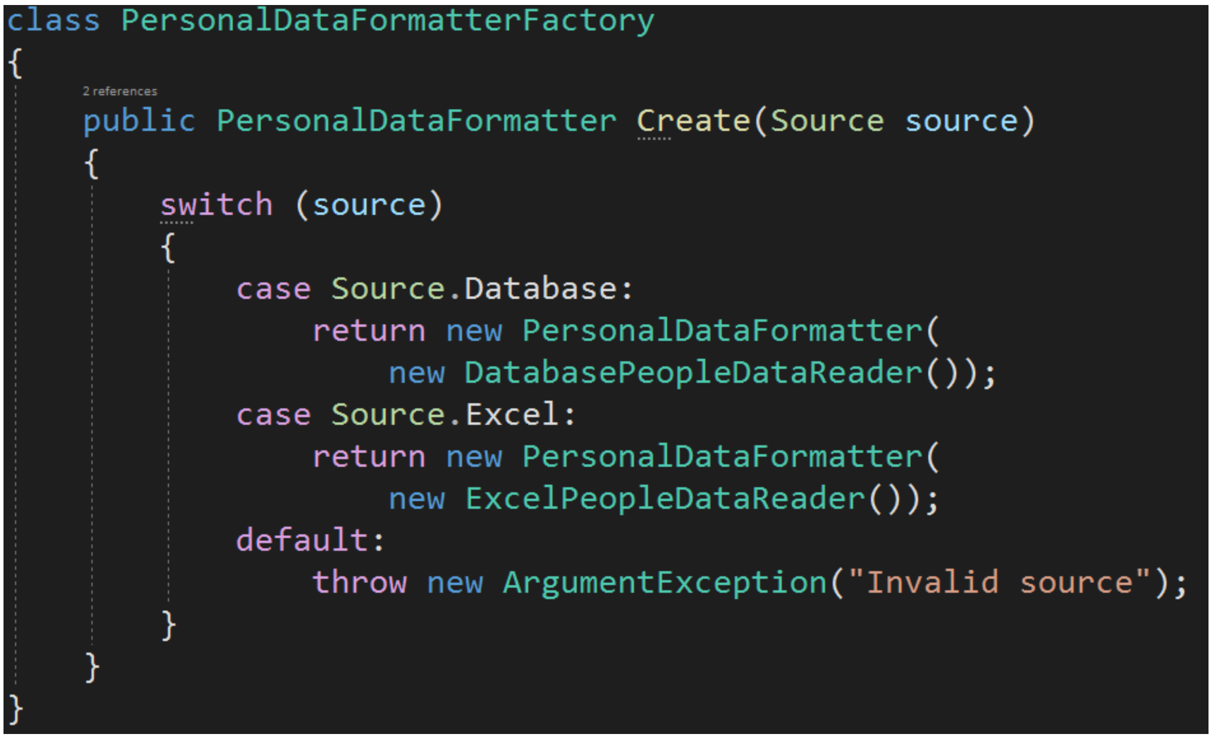
```
class ExcelPeopleDataReader : IPeopleDataReader
{
    2 references
    public IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from an Excel file");
        return new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
        };
    }
}
```

Instead of using inheritance, I will compose The PersonalDataFormatter with a type implementing the IPeopleDataReader interface:



```csharp
class PersonalDataFormatter
{
    private readonly IPeopleDataReader _peopleDataReader;

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

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

Finally, I will adjust the Factory:



```csharp
class PersonalDataFormatterFactory
{
    2 references
    public PersonalDataFormatter Create(Source source)
    {
        switch (source)
        {
            case Source.Database:
                return new PersonalDataFormatter(
                    new DatabasePeopleDataReader());
            case Source.Excel:
                return new PersonalDataFormatter(
                    new ExcelPeopleDataReader());
            default:
                throw new ArgumentException("Invalid source");
        }
    }
}
```

Great. Everything works as before, but no problems mentioned above occur now:

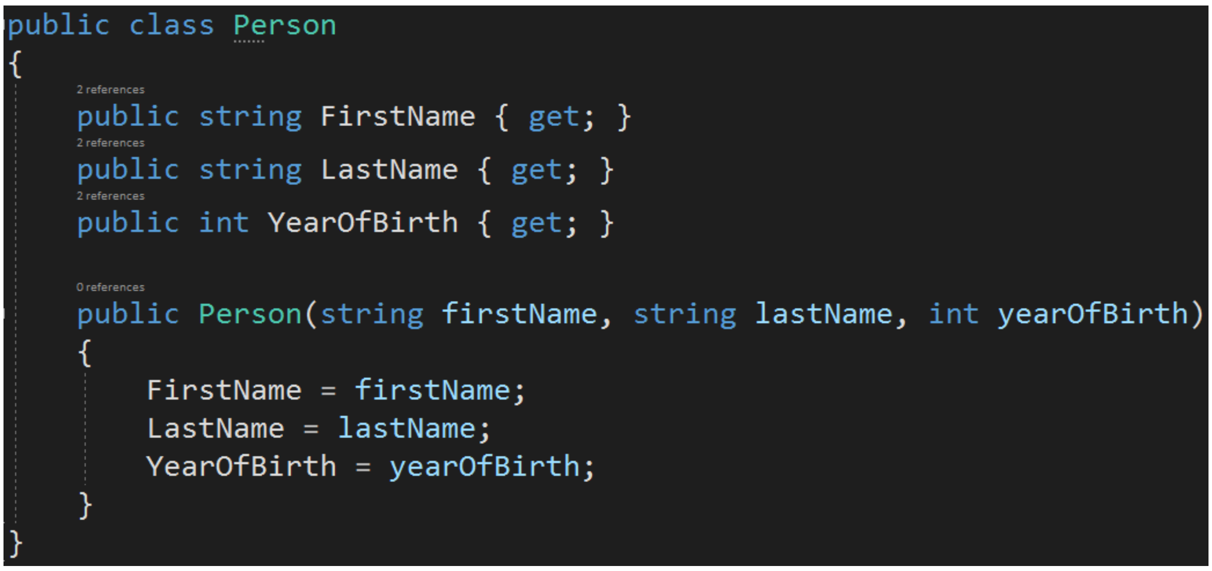
- The classes are **loosely coupled**. They live in complete separation, and they only communicate by an interface
- The relationship between classes is **not rigid** anymore. It is defined at runtime when we actually inject a concrete PeopleDataReader to

PersonalDataFormatter object with the constructor. Before, the relation was defined at compile time.
- If we needed to add more changes, the inheritance hierarchy wouldn't grow. We would only add a new interface and classes implementing it, for example, an IPersonFormatter implemented by PersonShortFormatter and PersonFullFormatter.
- Testing would be simple. We would test each class in separation, and no tests would be duplicated.
- No class exposes any implementation details to another class.

All right. I hope you see now that in this case "composition over inheritance" was a rule worth following. I would say it is in the majority of cases, and when in doubt, you should follow the composition design rather than inheritance. To be honest, at my everyday work I use inheritance extremely rarely.

One more thing before we move on. If you know the Bridge design pattern, this all may sound very familiar to you. This is because the Bridge pattern is simply a way of implementing the composition over inheritance principle. You can read more about the **Junior e-book**.

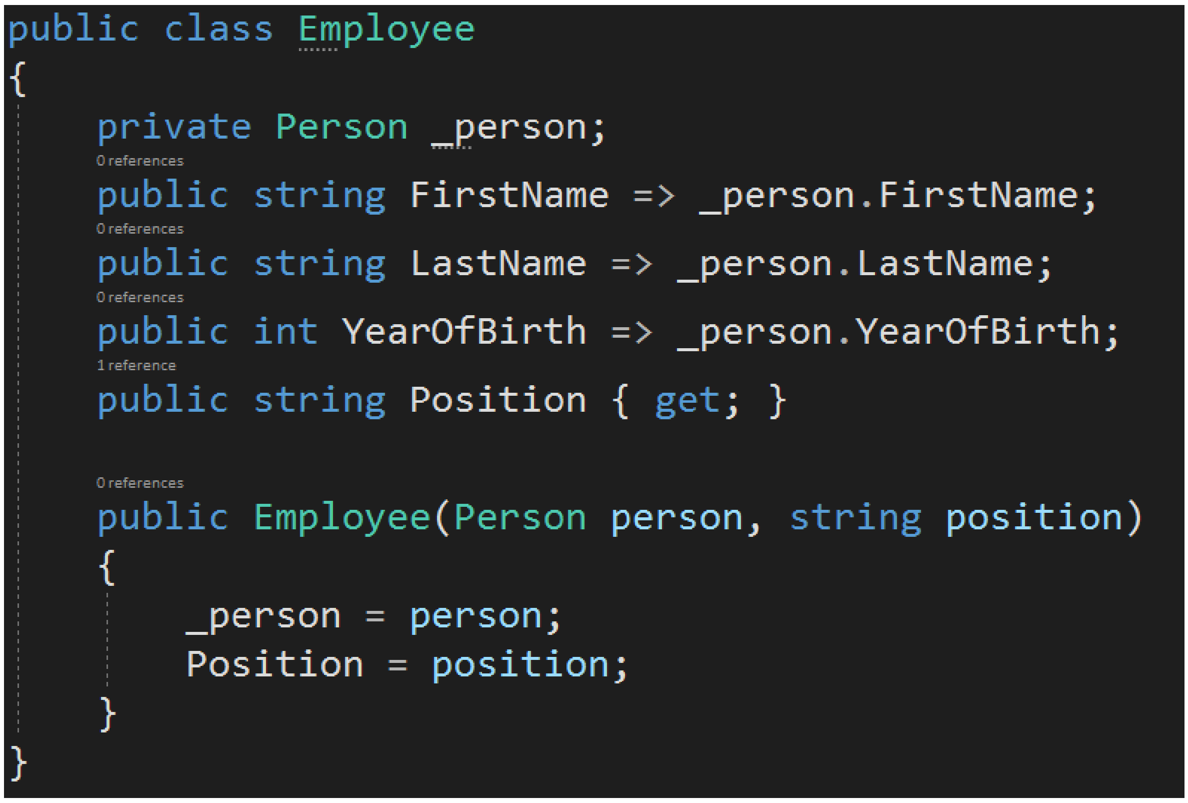
All right. We said that having composition instead of inheritance has a lot of benefits. But it doesn't mean that inheritance should be avoided at any cost. Let's take a look at the Person type:



```
public class Person
{
    2 references
    public string FirstName { get; }
    2 references
    public string LastName { get; }
    2 references
    public int YearOfBirth { get; }

    0 references
    public Person(string firstName, string lastName, int yearOfBirth)
    {
        FirstName = firstName;
        LastName = lastName;
        YearOfBirth = yearOfBirth;
    }
}
```

Now, let's say we want to introduce an Employee type to the project. An Employee is still a Person, and it should have FirstName, LastName, and YearOfBirth properties. Besides that, this type should have a "Position" property.

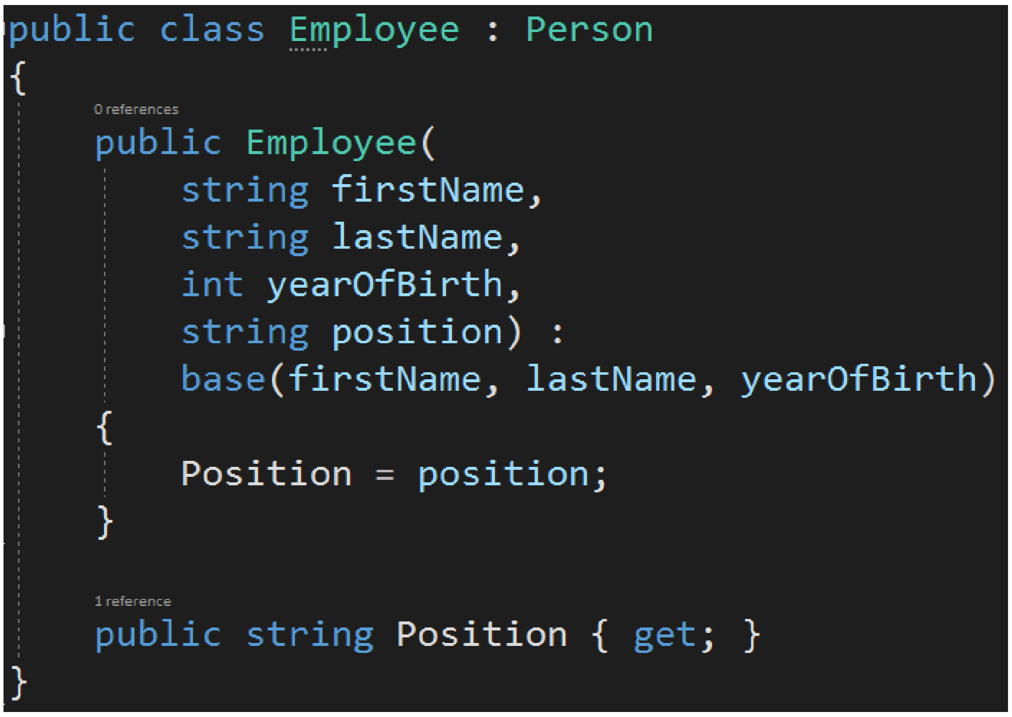
Let’s say we are so excited about using the “composition over inheritance” that we decide not to use inheritance ever again. And this is the code we create:



```
public class Employee
{
    private Person _person;
    0 references
    public string FirstName => _person.FirstName;
    0 references
    public string LastName => _person.LastName;
    0 references
    public int YearOfBirth => _person.YearOfBirth;
    1 reference
    public string Position { get; }

    0 references
    public Employee(Person person, string position)
    {
        _person = person;
        Position = position;
    }
}
```

Is this design good? Well, I wouldn’t say so. What looks a bit fishy are **the forwarding methods** - so the methods that only exist to call methods from some inner object. In our case, those methods are the FirstName, LastName, and YearOfBirth properties (remember that properties are like special kinds of methods).

Let’s see what this code would look like if we used inheritance:



```
public class Employee : Person
{
    0 references
    public Employee(
        string firstName,
        string lastName,
        int yearOfBirth,
        string position) :
        base(firstName, lastName, yearOfBirth)
    {
        Position = position;
    }

    1 reference
    public string Position { get; }
}
```

Well, I think it looks much simpler. The forwarding methods are not there, as the properties we want to have in the Employee class are simply inherited from the Person class. All we need to define is the new Position property that actually makes the Employee different from a Person.

How to decide whether to use composition over inheritance? Well, first of all, you need to answer this question: when thinking about your types, can you say that one of them IS the other one? Do they have the same structure and similar functionality, with only some extended behavior in the derived type? If so, inheritance can be the right choice. Otherwise, you should rather opt for composition. When in doubt, go for composition and in the worst case, you will adjust your design if it turns out it's not working out.

You can read more about the details of making the "composition or inheritance" decision in this article:
https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

Let's summarize. "Composition over inheritance" is a design principle stating that we should favor composition over inheritance. In other words, we should reuse the code by rather containing objects within different objects, than inheriting one from another.

Bonus questions:

- "**What is the problem with using composition only?**"
  If we decide not to use inheritance at all, we make it harder for ourselves to define types that are indeed in an "IS-A" relation - so when one type IS the other one. For example, a Dog IS an Animal, or an Employee IS a person. When implementing such hierarchy with the composition we create very similar types that wrap other types only adding a bit of new functionality, and they mostly contain forwarding methods.

- "**What are forwarding methods?**"
  They are methods that don't do anything else than calling almost identical methods from some other type. Forwarding methods indicate a very close relationship between types, which may mean that one type should be inherited from another.