

Dependency Injection

<https://www.tutorialsteacher.com/ioc/dependency-injection>

&

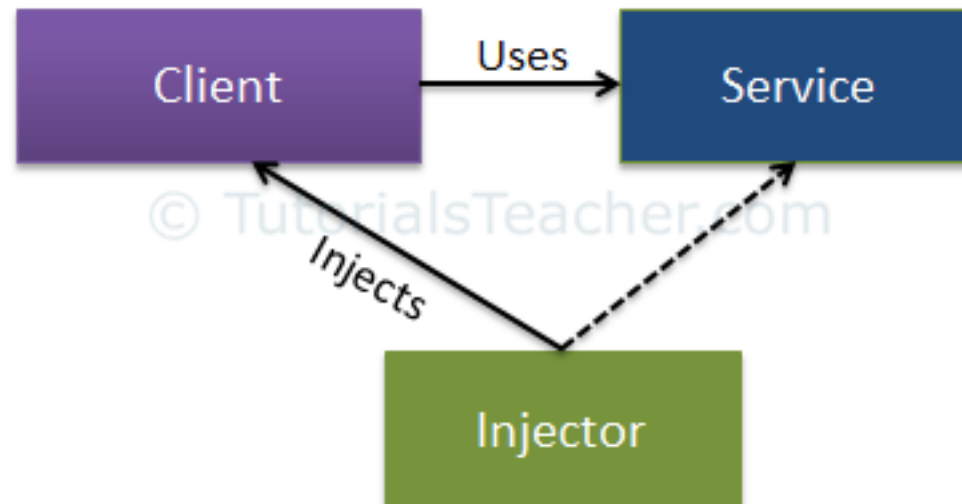
<https://www.javagists.com/dependency-injection-design-pattern>

Dependency Injection

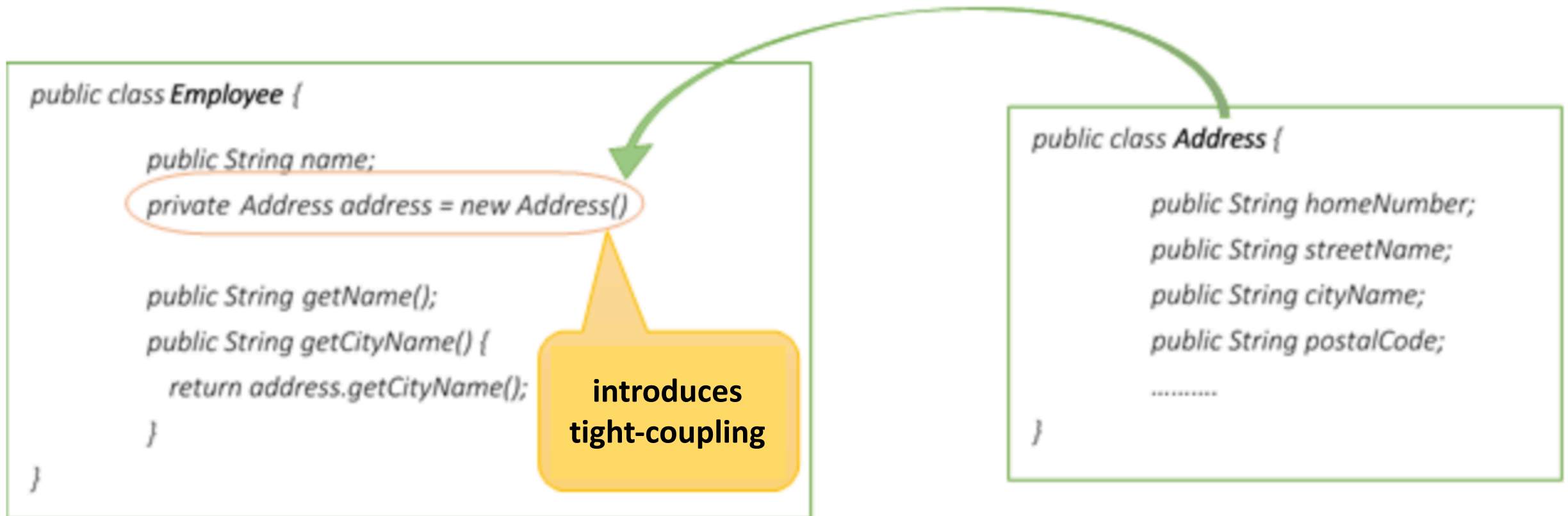
- Dependency Injection (DI) is a design pattern.
- It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways.
- Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

Dependency Injection

- The Dependency Injection pattern involves 3 types of classes:
 - Client Class: The client class (dependent class) is a class which depends on the service class
 - Service Class: The service class (dependency) is a class that provides service to the client class.
 - Injector Class: The injector class injects the service class object into the client class.



Problem



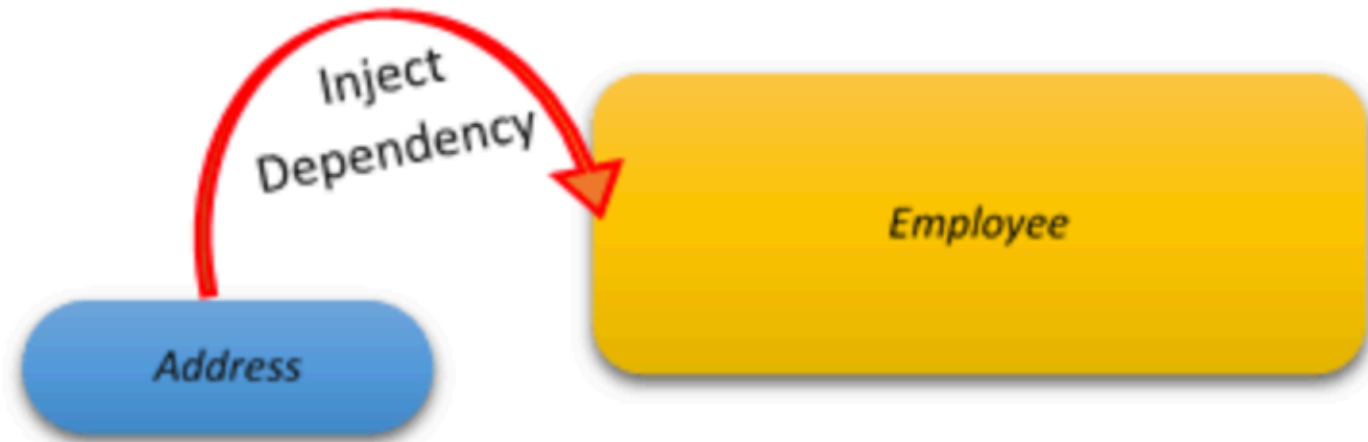
Problem

In any case, if the 'Address' class was modified as to insert another part to the address employee class should reflect necessary changes accordingly and re-compile again to function properly. Also, if address object gets to depend on any other external class that also will have to consider when managing the employee class. So, this makes the maintenance of the system very difficult.



Solution

Find a way to decouple the Address object from the Employee object. Separating each class responsibilities independently, by introducing a way to inject the service from Address object to Employee object. So that Employee object can behave independently.



Types of Dependency Injection

- **Constructor Injection:** In the constructor injection, the injector supplies the service (dependency) through the client class constructor.
- **Property (Setter) Injection:** In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.
- **Method (Interface) Injection:** In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

Traditional Approach

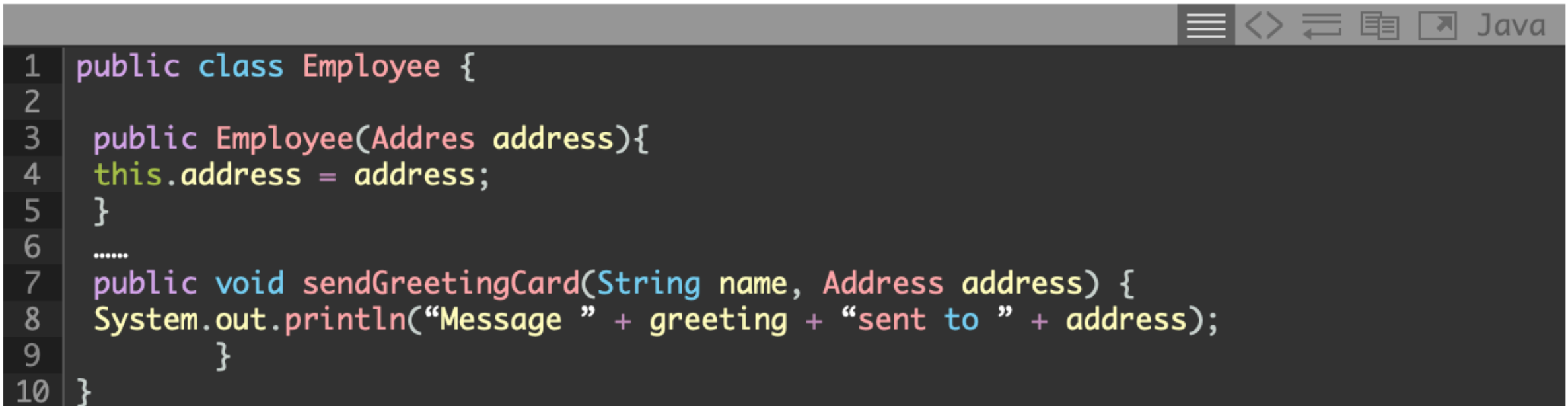
```
1 public class Employee {  
2     private Address address = new Address();  
3     .....  
4     public void sendGreetingCard(String greeting, Address address) {  
5         System.out.println("Message " + greeting + "sent to " + address);  
6     }  
7 }
```

Testing class contains below code,

```
1 public class EmployeeTest {  
2     public static void main(String[] args) {  
3         Employee emp = new Employee();  
4         emp.sendGreetingCard ("Amy", "489-A/Flower Av./Readhive/Leeds");  
5     }  
6 }
```


Constructor Injection

In this approach, the dependencies are provided via the constructor arguments. Basically, an argument is an object reference of the dependency class. The constructor will be called only upon class compilation and it ensures that the dependency won't change during the object lifetime. Hence, this is good for mandatory dependencies, ensuring the object is fully ready to be used upon construction. Here the 'Address' object is created outside and injected to 'Employee' object as a constructor parameter, so 'Employee' don't have to worry about address object creation.

A screenshot of a Java IDE window. The title bar shows a hamburger menu icon, a code editor icon, a run icon, a search icon, and the text 'Java'. The code is as follows:

```
1 public class Employee {  
2  
3     public Employee(Address address){  
4         this.address = address;  
5     }  
6     .....  
7     public void sendGreetingCard(String name, Address address) {  
8         System.out.println("Message " + greeting + "sent to " + address);  
9     }  
10 }
```

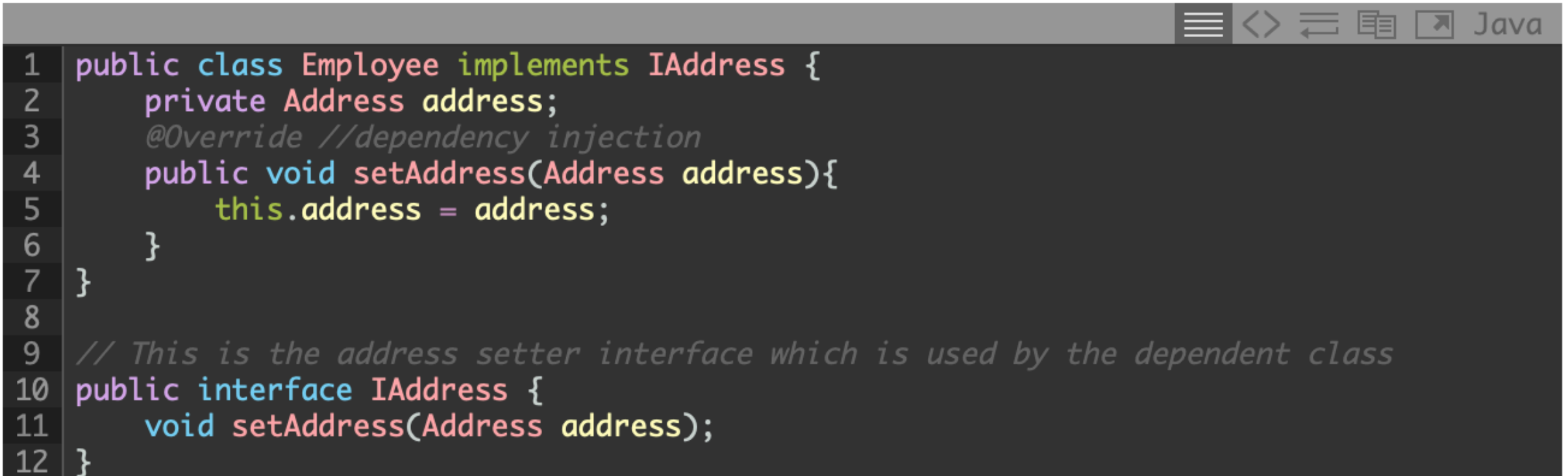
Setter Injection

In this approach, setter method is used to pass the externally instantiated object to the required class. Also, it is encouraging to apply optional dependencies in setter injection. The dependent class should be able to function even without the setter arguments. These dependencies can be changed even after the object is instantiated and it won't create a new instance always like with constructor injection. Hence, setter injection is flexible than constructor injection.

```
1 public class Employee {
2     private Address address = null;
3
4     public void setAddress(Address address){
5         this.address = address;
6     }
7
8     public Address getAddress() {
9         return this.address;
10    }
11    public void sendGreetingCard(String name, Address address) {
12        System.out.println("Message " + greeting + "sent to " + address);
13    }
14 }
```

Interface Injection

This approach takes the use of an interface to clearly separate the responsibility. Here, the client is implementing the interface with the dependency and override its method to achieve the required functionality. Here, the dependency is given a chance to control its own injection.

A screenshot of a Java IDE window. The title bar shows a hamburger menu icon, a code editor icon, a search icon, a list icon, and the text "Java". The code is as follows:

```
1 public class Employee implements IAddress {  
2     private Address address;  
3     @Override //dependency injection  
4     public void setAddress(Address address){  
5         this.address = address;  
6     }  
7 }  
8  
9 // This is the address setter interface which is used by the dependent class  
10 public interface IAddress {  
11     void setAddress(Address address);  
12 }
```