Dependency (notes)

The basic principle behind Dependency Injection (DI) is that objects define their dependencies only through constructor arguments, arguments to a factory method, or properties which are set on the object instance after it has been constructed or returned from a factory method. Then, it is the job of the container to actually inject those dependencies when it creates the bean. This is fundamentally the inverse, hence the name Inversion of Control (IoC).

Dependency without Beans

```java
public class Company {
    private Address address;

    public Company(Address address) {
        this.address = address;
    }

    // getter, setter and other properties
}

public class Address {
    private String street;
    private int number;

    public Address(String street, int number) {
        this.street = street;
        this.number = number;
    }

    // getters and setters
}

Address address = new Address("High Street", 1000);
Company company = new Company(address);
```

Assume we have declaration of class Company.
This class needs a collaborator of type Address.
Normally, we create objects with their classes constructors.
There's nothing wrong with this approach, but wouldn't it be nice to manage the dependencies in a better way?
Imagine an application with dozens or even hundreds of classes. Sometimes we want to share a single instance of a class across the whole application, other times we need a separate object for each use case, and so on.
Managing such a number of objects is nothing short of a nightmare. This is where Inversion of Control comes to the rescue.
Instead of constructing dependencies by itself, an object can retrieve its dependencies from an Inversion of Control container. All we need to do is to provide the container with appropriate configuration metadata.

Dependency with Beans

```java
@Component
public class Company {
    private Address address;

    public Company(Address address) {
        this.address = address;
    }

    // getter, setter and other properties
}

@Configuration
@ComponentScan(basePackageClasses = Company.class)
public class Config {
    @Bean
    public Address getAddress() {
        return new Address("High Street", 1000);
    }
}

ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
Company company = context.getBean("company", Company.class);
```

First off, we need to decorate the Company class with the @Component annotation. And write configuration class supplying bean metadata to an Inversion of Control container.

The configuration class produces a bean of type Address. It also carries the @ComponentScan annotation, which instructs the container to looks for beans in the package containing the Company class.

When a Spring Inversion of Control container constructs objects of those types, all the objects are called Spring beans as they are managed by the Inversion of Control container.

Since we defined beans in a configuration class, we'll need an instance of the AnnotationConfigApplicationContext class to build up a container.

Constructor-based Injection

```
1   public class ConstructorDI {
2
3   DemoBean demoBean = null;
4
5   public ConstructorDI (DemoBean demoBean) {
6           this.demoBean = demoBean;
7       }
8   }
```

Constructor-based Dependency Injection is realized by invoking a constructor with a number of arguments, each representing a collaborator. Additionally, calling a static factory method with specific arguments to construct the bean, can be considered almost equivalent, and the rest of this text will consider arguments to a constructor and arguments to a static factory method similarly.

Setter-based Injection

```
1  public class TestSetterDI {
2
3  DemoBean demoBean = null;
4
5  public void setDemoBean(DemoBean demoBean) {
6          this.demoBean = demoBean;
7      }
8  }
```

Setter-based Dependency Injection is realized by calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

Constructor-based and setter-based types of injection can be combined for the same bean.

The Spring documentation recommends using constructor-based injection for mandatory dependencies, and setter-based injection for optional ones.

Field-Based Injection

```
1   public class FieldDI {
2       @Autowired
3       private DemoBean demoBean;
4   }
```

In case of Field-Based DI, we can inject the dependencies by marking them with an @Autowired annotation.

While constructing the FieldDI object, if there's no constructor or setter method to inject the DemoBean bean, the container will use reflectiocn to inject DemoBean into FieldDI.

This approach might look simpler and cleaner but is not recommended to use because it has a few drawbacks such as:

This method uses reflection to inject the dependencies, which is costlier than constructor-based or setter-based injection

It's really easy to keep adding multiple dependencies using this approach. If you were using constructor injection having multiple arguments would have made us think that the class does more than one thing which can violate the Single Responsibility Principle.