Tight Coupling

```
// Animal.java
public class Animal {
    // Constructor that prints something
    public Animal() {
        System.out.println("Animal constructor called. Creating a generic animal.");
    }

    // play() method that prints something
    public void play() {
        System.out.println("The animal is playing around.");
    }
}



// Person.java
public class Person {
    // Tight coupling: Directly creating an instance of Animal using 'new'
    private Animal animal = new Animal();

    // Method to play with the animal
    public void playWithAnimal() {
        animal.play();  // Calling the play() method on the tightly coupled animal object
    }
}
```

```
// App.java (The main Spring Boot application class)
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class App {
    public static void main(String[] args) {
        // Manually creating Person object (not using Spring's dependency injection yet)
        Person p = new Person();
        // Calling the method to demonstrate the behavior
        p.playWithAnimal();

        // Starting the Spring Boot application (though in this basic example, it's not using Spring features yet)
        SpringApplication.run(App.class, args);
    }
}
```

In the code you provided, the Person class directly creates an Animal object using Animal animal = new Animal();. This means Person is tightly coupled to the specific Animal class. Tight coupling implies that Person depends heavily on the exact implementation of Animal. Any changes to Animal could directly impact Person, even if you don't modify Person.java.

Every time Animal changes (constructor, methods, or behavior), you may need to modify Person.java to accommodate those changes. This creates a ripple effect where changes in one class force changes in others, making maintenance time-consuming and error-prone.

Explanation of Tight Coupling in This Example

In this code, tight coupling is demonstrated between the Person and Animal classes. Here's how it's implemented and why it's considered "tight":

- **Direct Instantiation with 'new' Keyword**: Inside the Person class, we hardcoded the creation of an Animal object using Animal animal = new Animal();. This makes Person directly dependent on the concrete Animal class. The Person class "knows" exactly what type of animal it's dealing with and is responsible for creating it itself.
- **Dependency Hardcoding**: If you want to change the animal to something specific like a Cat or Dog (assuming Cat and Dog are subclasses of Animal or implement a similar interface), you would have to modify the code inside Person.java. For example:
    - To use a Cat, you'd change it to Animal animal = new Cat();.
    - To use a Dog, it'd be Animal animal = new Dog();. This requires editing and recompiling the Person class every time a different animal is needed, which is inefficient and violates principles like the Open-Closed Principle (classes should be open for extension but closed for modification).
- **Lack of Flexibility**: The Person class is rigidly tied to Animal. Any changes to Animal's constructor, methods, or internal behavior could break Person without you even touching Person.java. This high dependency makes the code harder to maintain, test, or extend.

In loose coupling:

- Person would depend on the interface, not the concrete class.
- Spring would handle creating and injecting the specific implementation (e.g., Cat or Dog) at runtime, without hardcoding 'new' in Person.
- This allows swapping implementations easily (e.g., via configuration) without changing Person.java.

Loose Coupling

Loose Coupling

```
// Animal.java (now an interface for loose coupling)
public interface Animal {
    void play();  // Abstract method for playing
}



// Dog.java (implements Animal interface)
public class Dog implements Animal {
    // Constructor that prints something
    public Dog() {
        System.out.println("Dog constructor called. Creating a dog.");
    }

    // Implementation of play() method
    @Override
    public void play() {
        System.out.println("The dog is playing fetch.");
    }
}




// Cat.java (implements Animal interface)
```

```
public class Cat implements Animal {
    // Constructor that prints something
    public Cat() {
        System.out.println("Cat constructor called. Creating a cat.");
    }

    // Implementation of play() method
    @Override
    public void play() {
        System.out.println("The cat is playing with a yarn ball.");
    }
}
```

```
// Person.java (now uses loose coupling via interface and constructor injection)
public class Person {
    // Loose coupling: Depends on the Animal interface, not a concrete class
    private Animal animal;

    // Constructor injection: Animal is provided from outside, no 'new' here
    public Person(Animal animal) {
        this.animal = animal;
    }

    // Method to play with the animal
    public void playWithAnimal() {
```

```
        animal.play();  // Calls play() on whatever Animal implementation was injected
    }
}
```

```
// App.java (The main Spring Boot application class)
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class App {
    public static void main(String[] args) {
        // Loose coupling demo: Create a specific Animal implementation
        // You can easily swap Dog for Cat without changing Person.java
        Animal dog = new Dog();  // Or use: Animal cat = new Cat();

        // Inject the Animal into Person via constructor
        Person p = new Person(dog);  // Pass the animal here

        // Call the method to demonstrate
        p.playWithAnimal();

        // Starting the Spring Boot application (not using Spring DI in this basic demo)
        SpringApplication.run(App.class, args);
    }
}
```

In the provided code, **dependency injection (DI)** is used, but it is implemented **manually** rather than using Spring's automated dependency injection. Let's break down where and how dependency injection is applied in the code and clarify the concept in the context of your example.

Is Dependency Injection Used?

Yes, dependency injection is used in the Person class. Specifically, it is achieved through **constructor injection**, where the dependency (Animal) is passed to Person from outside rather than being created inside Person.

Where and How Is It Used?

1. **Where**: In the Person class and the App class's main method.
    - **Person.java**:

      java

```java
public class Person {
    private Animal animal;

    // Constructor injection: Animal is passed from outside
    public Person(Animal animal) {
        this.animal = animal;
    }

    public void playWithAnimal() {
        animal.play();
    }
}
```

    - **App.java**:

      java

```java
public class App {
    public static void main(String[] args) {
        Animal dog = new Dog();  // Create the dependency
        Person p = new Person(dog);  // Inject it into Person
        p.playWithAnimal();
        SpringApplication.run(App.class, args);
```

```
    }
}
```

2. **How**:
    - **Constructor Injection**: The Person class does not create its own Animal object using new Animal(). Instead, it receives an Animal (e.g., a Dog or Cat) through its constructor. This is dependency injection because the dependency (Animal) is "injected" into Person from an external source (in this case, the main method).
    - **Manual Dependency Injection**: In the main method of App.java, we manually create a Dog (Animal dog = new Dog();) and pass it to the Person constructor (Person p = new Person(dog);). This is manual DI because we explicitly write code to create and pass the dependency, rather than relying on a framework like Spring to do it automatically.
    - **Loose Coupling Enabled by DI**: By injecting the Animal dependency, Person is no longer tightly coupled to a specific implementation like Dog or Cat. You can pass any class that implements the Animal interface (e.g., Dog, Cat, or a future Bird), and Person will work without modification.

In our above loose coupling code, my teacher said, we were still creating object of animal and person... So we are creating their objects, what is wrong about it? Then my teacher went on to say to use beans which will be managed by Spring - what did he mean by that - give the explanation along with code. Then my professor went ahead and said that we needed to remember 3 things to manage beans - 1. About the beans 2. About the dependencies 3. Where (which package) to search for brans. Then he went ahead to make our previous Person class as a Bean, where he used @Component annotation - what did that mean? He said Person become a component now which will be managed by Spring. He made Dog a component also. Then he added that we needed Component annotation to declare beans. Now he used Autowired dependency before Animal animal() in Person class . Then he added that if we make extra beans then make sure to add it in @ComponentScan(basePackage={}} Then he said everything was packaged inside Spring container now. Now, was dependency injection used here?- where and how?

Why Creating Objects Manually is Still "Problematic" in Loose Coupling

In the loose coupling code we had earlier, we achieved some flexibility by using an interface (Animal) and constructor injection in Person. This means Person no longer hardcodes the creation of a specific animal inside itself (no more new Animal() in Person.java). Instead, the animal is passed in from outside, allowing you to swap implementations (e.g., Dog or Cat) without modifying Person.

However, your teacher pointed out that we're **still manually creating objects** in the main() method of App.java:

- Animal dog = new Dog();
- Person p = new Person(dog);

What's "wrong" about this? It's not strictly wrong for small demos, but in real Spring applications, it defeats the purpose of Spring's core features:

- **Manual Management**: You're still responsible for creating, wiring (connecting dependencies), and managing the lifecycle of objects yourself. If your app grows (e.g., more classes, complex dependencies), this becomes error-prone and tedious.
- **No Inversion of Control (IoC)**: Spring's philosophy is IoC—let the framework handle object creation and dependency wiring, not you. Manual new calls mean you're in control, not Spring.

- **Limited Spring Benefits**: You miss out on automatic dependency resolution, lifecycle management (e.g., init/destroy methods), scoping (singleton vs. prototype), and easy configuration changes.
- **Tight Coupling to Creation Logic**: In main(), you're hardcoded to create a Dog. To switch to Cat, you still edit code and recompile, instead of configuring it externally (e.g., via XML or annotations).

Your teacher introduced **Spring beans** to fix this: Beans are objects managed by Spring's IoC container. Spring creates them, injects dependencies automatically, and handles their lifecycle. This makes your code more modular, testable, and configurable without code changes.

Using Beans Managed by Spring: Explanation and Code

"Beans managed by Spring" means you declare your classes as beans using annotations (or XML), and Spring's **IoC container** (a runtime environment) takes over:

- **Creating** instances (no more manual new).
- **Injecting** dependencies (wiring objects together automatically).
- **Managing lifecycle** (e.g., initializing beans when the app starts).
- **Storing beans** in a container for easy retrieval.

This shifts control from your code to Spring (IoC). Dependencies are injected automatically (Dependency Injection, or DI), promoting loose coupling.

Here's the updated code using Spring beans. I've assumed we're using Dog as the default animal implementation, but you could swap it via configuration. Add these to your Spring Boot project (ensure you have Spring Boot dependencies in pom.xml or build.gradle).

java

```java
// Animal.java (interface remains the same)
public interface Animal {
    void play();  // Abstract method for playing
}
```

java

```java
// Dog.java (now a Spring bean)
import org.springframework.stereotype.Component;

@Component  // Declares this as a Spring bean, managed by the container
public class Dog implements Animal {
    public Dog() {
        System.out.println("Dog constructor called. Creating a dog.");
    }
```

```
    @Override
    public void play() {
        System.out.println("The dog is playing fetch.");
    }
}
```

java

```
// Cat.java (optional extra bean, also managed by Spring if needed)
import org.springframework.stereotype.Component;

// @Component  // Uncomment if you want Cat as a bean; otherwise, Spring won't create it automatically
public class Cat implements Animal {
    public Cat() {
        System.out.println("Cat constructor called. Creating a cat.");
    }

    @Override
    public void play() {
        System.out.println("The cat is playing with a yarn ball.");
    }
}
```

java

```
// Person.java (now a Spring bean with autowired dependency)
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component  // Declares this as a Spring bean, managed by the container
public class Person {
    // Loose coupling: Depends on Animal interface
    @Autowired  // Tells Spring to automatically inject a matching bean (e.g., Dog) here
    private Animal animal;

    // No constructor needed for injection now; Spring handles it via field injection
```

```
    public void playWithAnimal() {
        animal.play();  // Uses the injected animal
    }
}
```

java

```
// App.java (Main Spring Boot application; now uses Spring to get beans)
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication  // Enables auto-configuration and component scanning
// If beans are in different packages, add: @ComponentScan(basePackages =
{"com.example.package1", "com.example.package2"})
public class App {
    public static void main(String[] args) {
        // Start Spring app and get the IoC container (ApplicationContext)
        ApplicationContext context = SpringApplication.run(App.class, args);

        // Retrieve the Person bean from the container (no manual 'new')
        Person p = context.getBean(Person.class);

        // Use it
        p.playWithAnimal();
    }
}
```

When you run this:

- Spring scans for @Component classes, creates beans (e.g., Dog and Person), and injects Dog into Person's animal field automatically.
- Output: "Dog constructor called..." and "The dog is playing fetch."
- No manual object creation in main()—Spring manages everything.

To use Cat instead: Add @Component to Cat.java, but since there would be two Animal beans (Dog and Cat), Spring needs help choosing. Add @Primary to one (e.g., on Dog) or use @Qualifier("dog") on the @Autowired field.

The 3 Things to Remember for Managing Beans

Your professor outlined the key aspects of Spring bean management:

1. **About the Beans**: Declare which classes are beans (e.g., using @Component, @Service, @Repository, etc.). This tells Spring to manage them. Beans are the core units in the IoC container—POJOs (Plain Old Java Objects) enhanced with Spring features.
2. **About the Dependencies**: Specify how beans depend on each other (e.g., Person depends on Animal). Use annotations like @Autowired to let Spring inject them automatically. This handles wiring without manual code.
3. **Where (Which Package) to Search for Beans**: Spring scans packages for annotated beans. By default, it scans the package of the main class (@SpringBootApplication) and subpackages. For beans in other packages, use @ComponentScan(basePackages = {"package1", "package2"}) on the main class to tell Spring where to look.

Explanation of @Component Annotation

- **What it Means**: @Component is a Spring stereotype annotation that marks a class as a Spring-managed bean. When Spring starts, it scans for these, creates an instance, and adds it to the IoC container.
- On Person: Person becomes a bean, meaning Spring creates and manages it (instead of you using new Person()). It's now "a component" in the Spring ecosystem—part of the app's modular structure.
- On Dog: Same thing—Dog is now a bean. Spring creates it automatically when needed.
- **Declaring Beans**: Yes, @Component (or similar like @Bean in config classes) is a common way to declare beans. Alternatives include XML config or @Configuration classes with @Bean methods.

@Autowired on the Animal Field in Person

- **What it Does**: @Autowired enables automatic dependency injection. Placed before private Animal animal;, it tells Spring: "Find a bean that matches the Animal type (e.g., Dog) and inject it here when creating the Person bean."
- This is **field injection** (directly on the field). Alternatives: constructor injection (@Autowired public Person(Animal animal) { ... }) or setter injection.
- If multiple matching beans exist (e.g., Dog and Cat), use @Qualifier("beanName") to specify which one.

Handling Extra Beans with @ComponentScan

- If you add more beans in different packages (e.g., Cat in a separate package), Spring might not find them during default scanning.
- Add @ComponentScan(basePackages = {"com.example.animals", "com.example.people"}) to App.java to explicitly tell Spring which packages to scan for @Component classes.
- This ensures all beans are discovered and managed.

Everything Packaged Inside the Spring Container

- The **Spring IoC Container** (implemented by ApplicationContext) is like a registry/box that holds all beans. It:
    - Scans for annotations.
    - Creates beans.
    - Resolves and injects dependencies.
    - Manages bean scopes (e.g., singleton by default—one instance shared).
- Once annotated and scanned, your classes (Person, Dog) are "packaged" inside this container—Spring oversees their creation, wiring, and destruction. You access them via context.getBean(...) instead of new.

Was Dependency Injection Used? Where and How?

Yes, dependency injection (DI) is used here—it's a key part of IoC.

- **Where**: In Person.java, via @Autowired private Animal animal;. This injects the Animal dependency (e.g., Dog bean) into Person.
- **How**: When Spring creates the Person bean, it automatically looks for a bean matching Animal (scans the container), finds Dog, and sets it on the field. No manual passing like in the old constructor. This is **autowiring by type** (matches the interface type). If ambiguous, it fails or uses qualifiers.
- DI promotes loose coupling: Person doesn't create or know *how* to get its dependency—Spring provides it at runtime.

This setup makes your app more scalable. For further config (e.g., switching animals without code changes), you could use @Configuration classes or properties files.