What is Dependency Inversion Principle (DIP) ?

The Dependency Inversion Principle (DIP) states that **high-level modules should not depend upon low-level modules; they should depend on abstractions. Secondly, abstractions should not depend upon details; details should depend upon abstractions.**
This way, instead of high-level modules depending on low-level modules, both will depend on abstractions. Every dependency in the design should target an interface or an abstract class. No dependency should target a concrete class.

The idea is that we isolate our class behind a boundary formed by the abstractions it depends on. If all the details behind those abstractions change, then our class is still safe. This helps keep coupling low and makes our design easier to change. DIP also offers us to test things in isolation, details like database are plugins to our system.

Robert Martin equated the Dependency Inversion Principle, as a first-class combination of the Open Closed Principle and the Liskov Substitution Principle.

Example: Code that violates Dependency Inversion Principle

Suppose a book store asked us to build a new feature that enables customers to put their favorite books on a shelf.

In order to implement the new functionality, we create a lower-level Book class and a higher-level Shelf class. The Book class will allow users to see reviews and read a sample of each book they store on their shelves. The Shelf class will let them add a book to their shelf and customize the shelf. For example, observe the below code.

```
public class Book {

   void seeReviews() {
        ...
   }
```

```
    void readSample() {
        ...
    }
}


public class Shelf {

     Book book;

     void addBook(Book book) {
         ...
     }

     void customizeShelf() {
         ...
     }
}
```

Everything looks fine, but as the high-level Shelf class depends on the low-level Book, the above code violates the Dependency Inversion Principle. This becomes clear when the store asks us to enable customers to add DVDs to their shelves, too. In order to fulfil the demand, we create a new DVD class

```
public class DVD {

    void seeReviews() {
        ...
    }

    void watchSample() {
        ...
    }
}
```

Now, we should modify the Shelf class so that it can accept DVDs, too. However, this would clearly break the Open/Closed Principle too.

Example: Code that follows Dependency Inversion Principle

The solution is to create an abstraction layer for the lower-level classes (Book and DVD).

We’ll do so by introducing the Product interface, both classes will implement it. For example, below code demonstrates the concept.

```
public interface Product {

    void seeReviews();

    void getSample();

}

public class Book implements Product {

    @Override
    public void seeReviews() {
          ...
    }

    @Override
    public void getSample() {
          ...
    }
}
```

```
public class DVD implements Product {

    @Override
    public void seeReviews() {
        ...
    }

    @Override
    public void getSample() {
        ...
    }
}
```

Now, Shelf can reference the Product interface instead of its implementations
(Book and DVD). The refactored code also allows us to later introduce new product types (for instance, Magazine) that customers can put on their shelves, too.

```
public class Shelf {
    Product product;

    void addProduct(Product product) {
        ...
    }

    void customizeShelf() {
        ...
    }
}
```

The above code also follows the Liskov Substitution Principle, as the Product type can be substituted with both of its subtypes (Book and DVD) without breaking the program. At the same time, we have also implemented the Dependency Inversion Principle, as in the refactored code, high-level classes don’t depend on low-level classes, either.

How is DIP related to Dependency Injection of Spring Framework?

It would be correct if you think that Dependency Inversion Principle is related to **Dependency Injection** as it applies to the Spring Framework. Uncle Bob Martin introduced the concept of Dependency Inversion before Martin Fowler introduced the term Dependency Injection. These both concepts are extremely related. Dependency Inversion is more concentrated on the structure of your code. Moreover, its focus is keeping your code loosely coupled. On the other hand, Dependency Injection is about how the code functionally works.

Dependency Inversion Principle has been very well implemented in Spring framework, the beauty of this design principle is that any class which is injected by DI framework is easy to test with the mock object and easier to maintain because object creation code is centralized in the framework and client code is not messed up with that.

What is the benefit of Dependency Inversion Principle?

Below are some of the benefits when we apply this principle in our code.

1) Keeps your code loosely coupled
2) Easier Maintenance
3) Better Code Reusability

What is the disadvantage of Dependency Inversion Principle?

When we use this principle, it results in an increased effort. We need to maintain more classes and interfaces, in a few words more complex code, but more flexible. If someone applies this principle blindly for every class or every module, it may not provide any benefit.

When should we not use Dependency Inversion Principle?

If our java class has functionality that is more likely to remain unchanged in the future, there is no need to apply this principle as it will not provide us any benefit.