Dependency Inversion Principle

This principle of Dependency Inversion is one of the important principles among the SOLID Principles. It defines the dependency of the classes.We can define the formal definition of this principle as per below:

Classes should be dependent on Interfaces rather than Concrete Classes.

We can understand this principle as per below with the example of simplicity:
Suppose we have 2 interfaces called Mouse and Keyboard. It has 2 classes implemented in itself (1) WiredMouse and (2)WirelessMouse. (1) WiredKeyboard and (2) WirelessKeyboard.

```
public interface Keyboard{
    public void WiredKeyboard();
    public void WirelessKeyboard();
}

public interface Mouse{
    public void WiredMouse();
    public void WirelessMouse();
}
```

Now I have a class called MacBook and I want to assemble different components.

```
class MacBook
{
    //we are assigning the concrete class objects
    private WiredKeyboard KB;
    private WiredMouse MS;

    //creating the public class constructor
    public MacBook()
    {
        KB = new WiredKeyboard();
        MS = new WiredMouse();
    }
}
```

Now the problem in this type of implementation is that, It already assigned the WiredKeyboard and WiredMouse class objects. Suppose in future I decide to implement the Wireless classes object, I can not do it , which means I can not extend it and which shows the dependency on the classes.

What should we have done instead? We should have assigned the Interface instead of Concrete Class and should have passed the interface in the parameter which keeps the scope of extending the functional features and classes like Wireless.

```
class MacBook
{
    //we are assigning the interface objects
    private Keyboard KB;
    private Mouse MS;

    //creating the public class constructor
    public MacBook(Keyboard KB, Mouse MS)
    {
        this.Keyboard = KB;
        this.Mouse = MS;
    }
}
```

When I will need to include the feature of Wireless, It will be easy for me to extend the capability as Class MacBook is dependent on the Interface, not on the classes.

This is how we can understand and implement the Dependency Inversion Principle.