简述

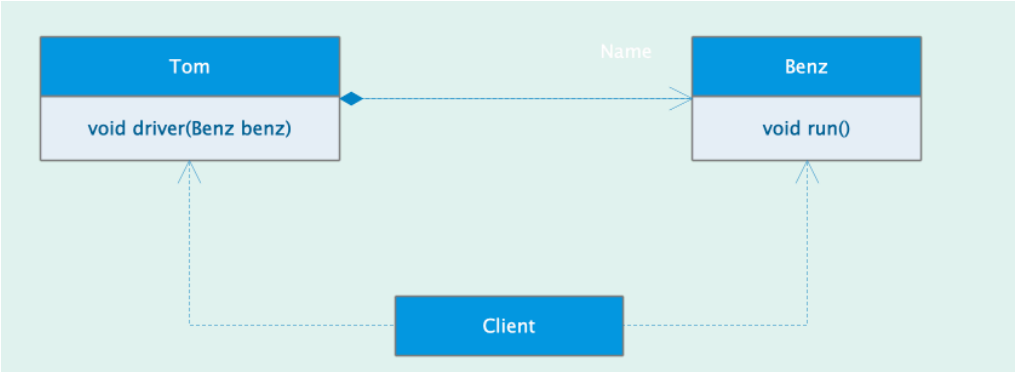
依赖倒置原则的包含如下的三层含义：

- 高层模块不应该依赖低层模块，两者都应该依赖其抽象
- 抽象不应该依赖细节
- 细节应该依赖抽象

举例说明

需求：Tom 开 奔驰

```
public class Benz {
    public void run(){
        System.out.println("benz run...");
    }
}

public class Tom {
    public void driver(Benz benz){
        benz.run();
     }
}

public class M1 {

     public static void main(String[] args) {
         Tom tom = new Tom();
         Benz benz = new Benz();

         tom.driver(benz);
     }

}
```



如果需求改了，Tom 还可以开宝马呢？ 在 Tom类 里面加上一个 重载

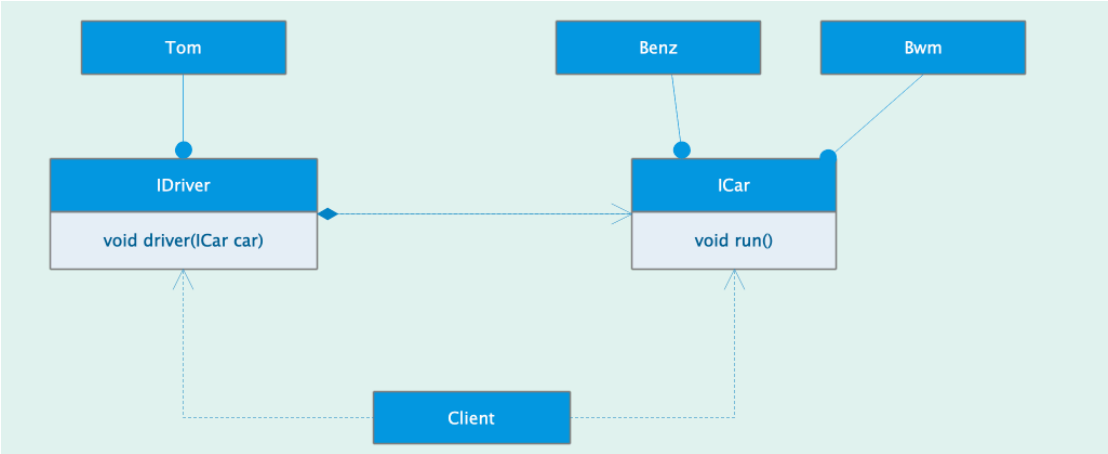
```
public class Tom {
    public void driver(Benz benz){
        benz.run();
    }
```

```
5
6       public void driver(Bmw bmw){
7           bmw.run();
8       }
9   }
```

如果按照上面的方式，肯定也可以实现。但是改动了Tom类， 并且扩展一种类型的汽车 都加上一个 重载肯定不好。

所以将代码进行改造

```
1  public interface ICar {
2      void run();
3  }
4
5  public class Bmw implements ICar{
6      public void run(){
7          System.out.println("bmw is run...");
8      }
9  }
10
11 public class Benz implements ICar{
12     public void run(){
13         System.out.println("benz run...");
14     }
15 }
16
17 public interface IDriver {
18     void driver(ICar car);
19 }
20
21 public class Tom implements IDriver{
22
23     public void driver(ICar car) {
24         car.run();
25     }
26 }
27
28 public class M1 {
29
30     public static void main(String[] args) {
31         IDriver driver = new Tom();
32         ICar car = new Benz();
33
34         driver.driver(car);
35     }
36
37 }
```



DI原则的几种写法

构造注入

```
public interface IDriver {
    void driver();
}

public class Tom implements IDriver{

    private ICar car;

    public Tom(ICar car){
        this.car = car;
    }

    public void driver() {
        car.run();
    }
}
```

setter注入

```
public class Tom implements IDriver{

    private ICar car;

    public Tom(){
    }

    public void setCar(ICar car) {
        this.car = car;
    }

    public void driver() {
        car.run();
    }
}
```