Dependency Inversion is one of the last principles we are going to look at. The principle states that:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Let's get started with some code that violates that principle.

Say you are working as part of a software team. We need to implement a project. For now, the software team consists of:

A BackEnd Developer

```
package mypack;
public class BackEndDeveloper {
    public void writeJava() {
    }
}
```

And a FrontEnd developer:

```
package mypack;
public class FrontEndDeveloper {
    public void writeJavascript() {
    }
}
```

And our project uses both throughout the development process:

```
package mypack;
public class Project { // high level
    private BackEndDeveloper
backEndDeveloper = new
BackEndDeveloper();     // low level
    private FrontEndDeveloper
frontEndDeveloper = new
FrontEndDeveloper();
    public void implement() {
        backEndDeveloper.writeJava();

frontEndDeveloper.writeJavascript();
    }
}
```

As we can see, the **Project class is a high-level module, and it depends on low-level modules such as BackEndDeveloper and FrontEndDeveloper**. We are actually violating the first part of the dependency inversion principle.

Also, by inspecting the **implement** function of Project class, we realize that the methods **writeJava** and **writeJavascript** are methods bound to the corresponding classes. Regarding the project scope, those are **details** since, in both cases, they are forms

of development. Thus, the second part of the dependency inversion principle is violated.

In order to tackle this problem, we shall implement an interface called the Developer interface:

```
package mypack;
public interface Developer {
    void develop();
}
```

Therefore, we introduce an abstraction.

The BackEndDeveloper shall be refactored to:

```
package mypack;
public class BackEndDeveloper
implements Developer {
    @Override
    public void develop() {
        writeJava();
    }
    private void writeJava() { //
details
    }
}
```

And the FrontEndDeveloper shall be refactored to:

```
package mypack;
```

```
public class FrontEndDeveloper
implements Developer {
    @Override
    public void develop() {
        writeJavascript();
    }
    private void writeJavascript() {
    } // details
}
```

The next step, in order to tackle the violation of the first part, would be to refactor the Project class so that it will not depend on the FrontEndDeveloper and the BackendDeveloper classes.

```
package mypack;
import java.util.List;
public class Project {
    private List<Developer>
developers;
    public Project(List<Developer>
developers) {
        this.developers = developers;
    }
    public void implement() {
        developers.forEach(d-
>d.develop());
    }
}
```

The outcome is that the **Project class does not depend on lower level modules, but rather abstractions.**(Project class has “Developer” as a member [program to interface] Also, **low-level modules (BackEndDeveloper and FrontEndDeveloper)and their details (writeJava and writeJavaScript) depend on abstractions** (develop() method of Developer)**.**