DZone

SOLID Principles: Dependency Inversion Principle

by Emmanouil Gkatziouras MVB · May. 04, 18 · Java Zone · Tutorial

Up until now, we had a look at the Single Responsibility, Open/Closed, Liskov Substitution, and Interface Segregation principles.

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 com.gkatzioura.solid.di;

public class BackEndDeveloper {

    public void writeJava() {
    }
}
```

And a FrontEnd developer:

```
package com.gkatzioura.solid.di;

public class FrontEndDeveloper {

    public void writeJavascript() {
    }

}
```

And our project uses both throughout the development process:

```
package com.gkatzioura.solid.di;

public class Project {

    private BackEndDeveloper backEndDeveloper = new BackEndDeveloper();
    private FrontEndDeveloper frontEndDeveloper = new FrontEndDeveloper();

    public void implement() {

        backEndDeveloper.writeJava();
        frontEndDeveloper.writeJavascript();
    }

}
```

So as we can see, the Project class is a high-level module, and it depends on low-level modules such as BackEndDeveloper and

So 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 com.gkatzioura.solid.di;

public interface Developer {

    void develop();
}
```

Therefore, we introduce an abstraction.

The BackEndDeveloper shall be refactored to:

```
package com.gkatzioura.solid.di;

public class BackEndDeveloper implements Developer {

    @Override
    public void develop() {
        writeJava();
    }

    private void writeJava() {
    }

}
```

And the FrontEndDeveloper shall be refactored to:

```java
package com.gkatzioura.solid.di;

public class FrontEndDeveloper implements Developer {

    @Override
    public void develop() {
        writeJavascript();
    }

    public void writeJavascript() {
    }

}
```

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.

```java
package com.gkatzioura.solid.di;

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. Also, low-level modules and their details depend on abstractions.

You can find the source code on GitHub.

Like This Article? Read More From DZone

Perfecting Your SOLID Meal With DIP

SOLID Design Principles Explained: Dependency Inversion

The Dependency Inversion Principle for Beginners, Without Diagrams

Free DZone Refcard
Java Containerization

Topics: JAVA , SOLID , DEPENDENCY INVERSION PRINCIPLE , REFACTORING , TUTORIAL

Running a Kotlin Class as a Subprocess

by Dan Newton MVB · May 29, 19 · Java Zone · Tutorial

Last week, I wrote a post on running a Java class as a subprocess. That post was triggered by my need to run a class from within a test without prebuilding a jar. The only difference between what I wrote in that post and what actually happened was the language. I used Kotlin to write that test, not Java. Therefore, I have decided to write this follow up post that builds upon what I previously wrote and focuses on starting a Kotlin subprocess instead of a Java subprocess.

Let's start with an equivalent implementation of the `exec` from the previous article, Running a Java Class as a Subprocess:

```
@Throws(IOException::class, InterruptedException::class)
fun exec(clazz: Class<*>, args: List<String> = emptyList(), jvmArgs: List<String> = emptyList()): Int {
  val javaHome = System.getProperty("java.home")
  val javaBin = javaHome + File.separator + "bin" + File.separator + "java"
  val classpath = System.getProperty("java.class.path")
  val className = clazz.name

  val command = ArrayList<String>()
  command.add(javaBin)
  command.addAll(jvmArgs)
  command.add("-cp")
  command.add(classpath)
  command.add(className)
  command.addAll(args)

  val builder = ProcessBuilder(command)
  val process = builder.inheritIO().start()
  process.waitFor()
  return process.exitValue()
}
```

Here is a short explanation since, again, I covered everything else in my previous post.

The path to the Java executable is retrieved and stored in `javaBin`. Using this along with the passed in `clazz`, `args`, `jvmArgs`, and the process's classpath, a command is created and executed by the `ProcessBuilder`. The class is now successfully running as a subprocess.

The code above is pretty much a copy and paste of the Java implementation. The differences are the order of the function parameters. I decided to switch `args` and `jvmArgs` around so I can make full use of their default values. This is based on the assumption that you are more likely to provide `args` instead of `jvmArgs`. It isn't really that much of a concern when running from Kotlin since you can name the parameters when calling the function. But if any Java code needs to call this function, then the order of the parameters might be helpful (as well as adding on `@JvmOverloads`).

Below are some ways of calling `exec`:

```
exec(MyProcess::class.java, listOf("3"), listOf("-Xmx200m"))
// jvmArgs = emptyList()
exec(MyProcess::class.java, listOf("3"))
// args = emptyList()
exec(MyProcess::class.java, jvmArgs = listOf("-Xmx200m"))
// both args and jvmArgs = emptyList()
exec(MyProcess::class.java)
```

A noticeable difference when writing in Kotlin instead of Java is the number of ways that you can define a `main` function. This is due to Kotlin providing many different routes to create a static function.

The function I provided above will run a `main` function in the two following scenarios:

- Inside a companion object and annotated with `@JvmStatic`

```
class MyProcess {
  companion object {
    @JvmStatic
    fun main(args: Array<String>) {
      // do stuff
    }
  }
}
```

- Inside an object and annotated with `@JvmStatic`

```
object MyProcess {
  @JvmStatic
  fun main(args: Array<String>) {
    // do stuff
  }
}
```

Another way to create a static function in Kotlin is to define it outside of a class:

```
@file:JvmName("MyProcess")

package dev.lankydan

fun main(args: Array<String>) {
  // do stuff
}
```

A name needs to be provided so Java knows what to do with it. Executing the `main` function in this snippet is one situation where this is needed. Unfortunately, the `exec` function won't work in this situation, since the `MyProcess` class doesn't actually exist, preventing it from compiling. Funnily enough, calling `MyProcess` from Java code will compile without any changes.

To resolve the compilation error found in the Kotlin version, `exec` ’s parameters need to change very slightly:

```
@Throws(IOException::class, InterruptedException::class)
fun exec(className: String, args: List<String> = emptyList(), jvmArgs: List<String> = emptyList()): Int {
  // className passed into command
}
```

This change sidesteps the compilation error the original `exec` function presents. But, this comes with the downside of not really being type-safe and might lead to a few errors here and there where an invalid string is passed in. As you can see below:

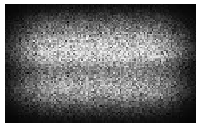
```
exec("dev.lankydan.MyProcess", listOf("argument"), listOf("-Xmx200m"))
```

Something about this change really rustles my jimmies. Therefore, providing both overloads is probably the sanest thing to do.

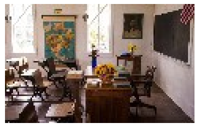
To wrap up, although most of what I have to say about executing Kotlin classes as subprocesses was already covered in Running a Java Class as a Subprocess, there are a few differences due to Kotlin’s flexibility. A `main` function in Kotlin can be defined in multiple ways and unfortunately, one of one those does not get along with `exec` function that the rest are happy with. Luckily, the solution is a small code change that can work well if there are overloads for both `String` and `Class` class names.

If you enjoyed this post or found it helpful (or both), then please feel free to follow me on Twitter at @LankyDanDev and remember to share with anyone else who might find this useful!

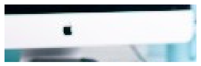
Like This Article? Read More From DZone



Kotlin: Static Methods

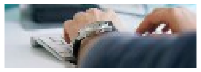


Learning Kotlin: Data Classes



Learning Kotlin: Operators

Free DZone Refcard



Java Containerization

Topics: KOTLIN, JAVA, SUBPROCESS, CLASS, KOTLIN CLASS, TUTORIAL, STATIC

IN PROGRESS