

# Dependency Injection



# Agenda

- SOLID Principles of Object Oriented Programming
- Creating a Spring DOI Example Project
- The Spring Context
- Basics of Dependency Injection
- Dependency Injection without Spring

# SOLID Principles of OOP



**SOLID**

Software development is not a Jenga game.

# History of SOLID

- The SOLID principles date back to March 1995
- The principles originate from “Uncle Bob” Martin
- Started as writings, which ultimately were turned into the book “Agile Software Development: Principles, Patterns, and Practices”
- Michael Feathers is credited with coming up with the SOLID acronym

# Why SOLID?

- Object Oriented Programming is a powerful concept
  - BUT OOP does not always lead to quality software
- The 5 principles focus on dependency management
- Poor dependency management leads to code that is brittle, fragile and difficult to change.
- Proper dependency management leads to quality code that is easy to maintain.

# 5 SOLID Principles

## 1. Single-Responsibility Principle

- *A class should only have a single responsibility, that is, only changes to one part of the software' specification should be able to affect the specification of the class.*



**SINGLE RESPONSIBILITY PRINCIPLE**

Just Because You Can, Doesn't Mean You Should

# Single-Responsibility Principle

- Every class should have a single responsibility
- There should never be more than one reason for a class to change
- Your classes should be small. No more than a screen full of code.
- Avoid thick “global” like classes
- Split big classes into smaller classes

# 5 SOLID Principles...

## 2. Open-Closed Principle

- *Software entities, should be open for extension, but closed for modification.*

**OPEN**  
**FOR EXTENSION**  
**(CLOSED FOR MODIFICATION)**



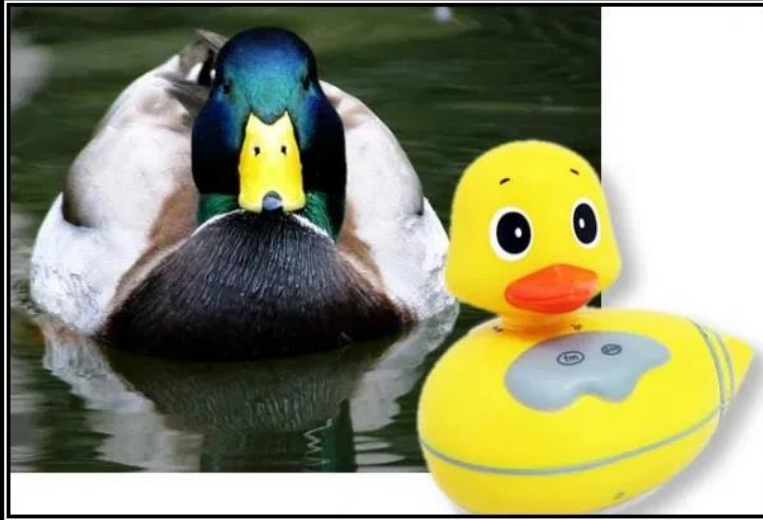
# Open-Closed Principle

- Your classes should be open for extension, but closed for modification
- You should be able to extend a classes behavior without modifying it
- Use private variable with getter and setters – ONLY when you need them.
- User abstract base classes.

# 5 SOLID Principles...

## 3. Liskov Substitution Principle

- *Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.*



### LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

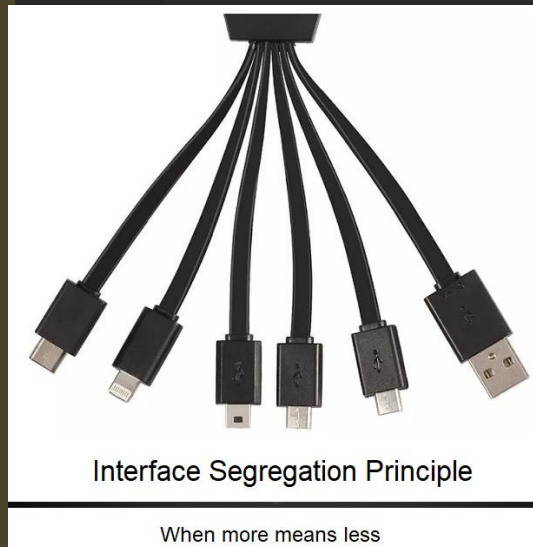
# Liskov Substitution Principle

- Founded back by Barabara Liskov in 1998
- Objects in a program should be replaceable with instances of their subtypes WITHOUT altering the correctness of the program.
- Violations will often fail the “Is a ” test.
  - A Square “Is a” Rectangle
  - However, a Rectangle “Is Not” a Square

# 5 SOLID Principles...

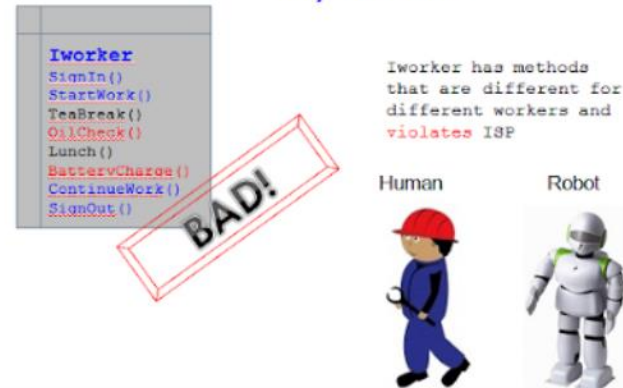
## 4. Interface Segregation Principle

- Many client-specific interfaces are better than one general-purpose interface.



### Interface Segregation Principle

"Clients should not be forced to depend upon interfaces that they don't use"



### Interface Segregation Principle

"Clients should not be forced to depend upon interfaces that they don't use"



# Interface Segregation Principle

- Make fine grained interfaces that are client specific
- Many client specific interfaces are better than one “general purpose” interface
- Keep your components focused and minimize dependencies between them.
- Notice relationship to the Single Responsibility Principle?
  - i.e. → avoid big / complex interfaces.

# 5 SOLID Principles...

## 5. Dependency Inversion Principle

- One should “depend upon abstractions, [not] concretions”.



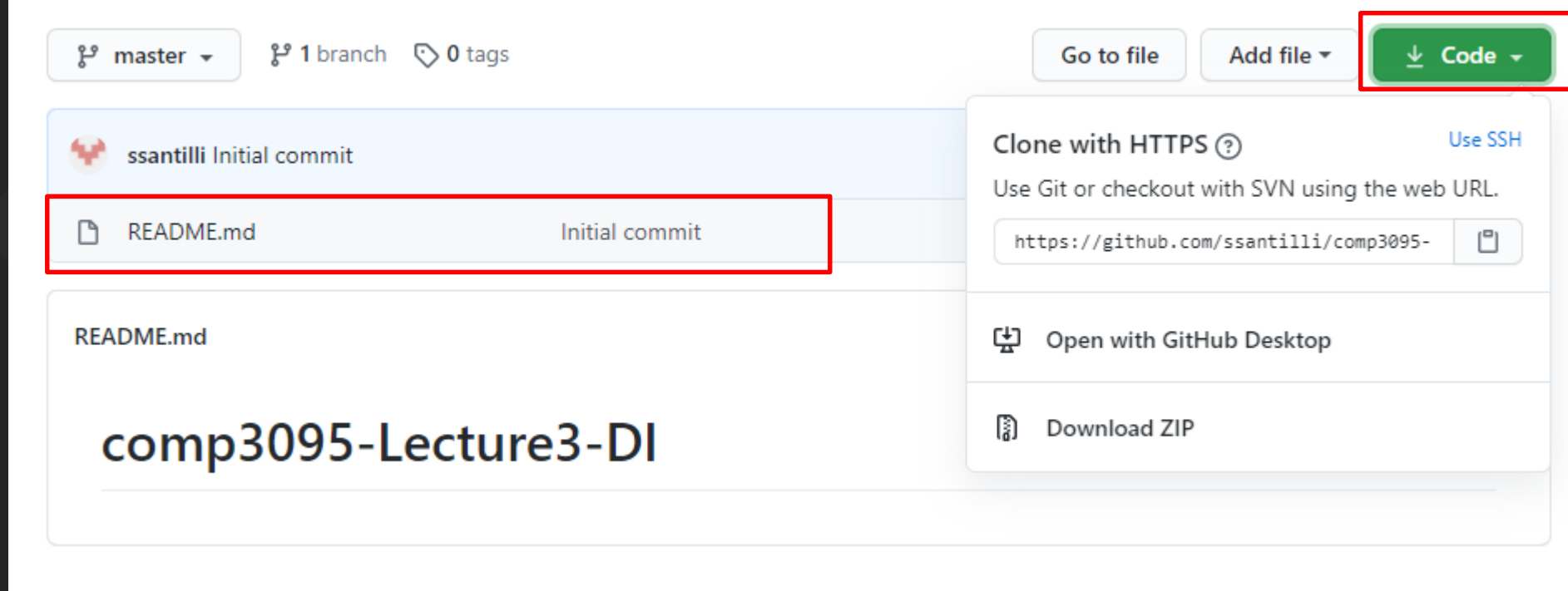
# Dependency Inversion Principle

- Abstractions should not depend upon details
- Details should depend upon abstractions
- Important that higher level and lower level objects depend on the same abstract interaction.
- This is not the same as dependency injection – which is how objects obtain dependent objects.

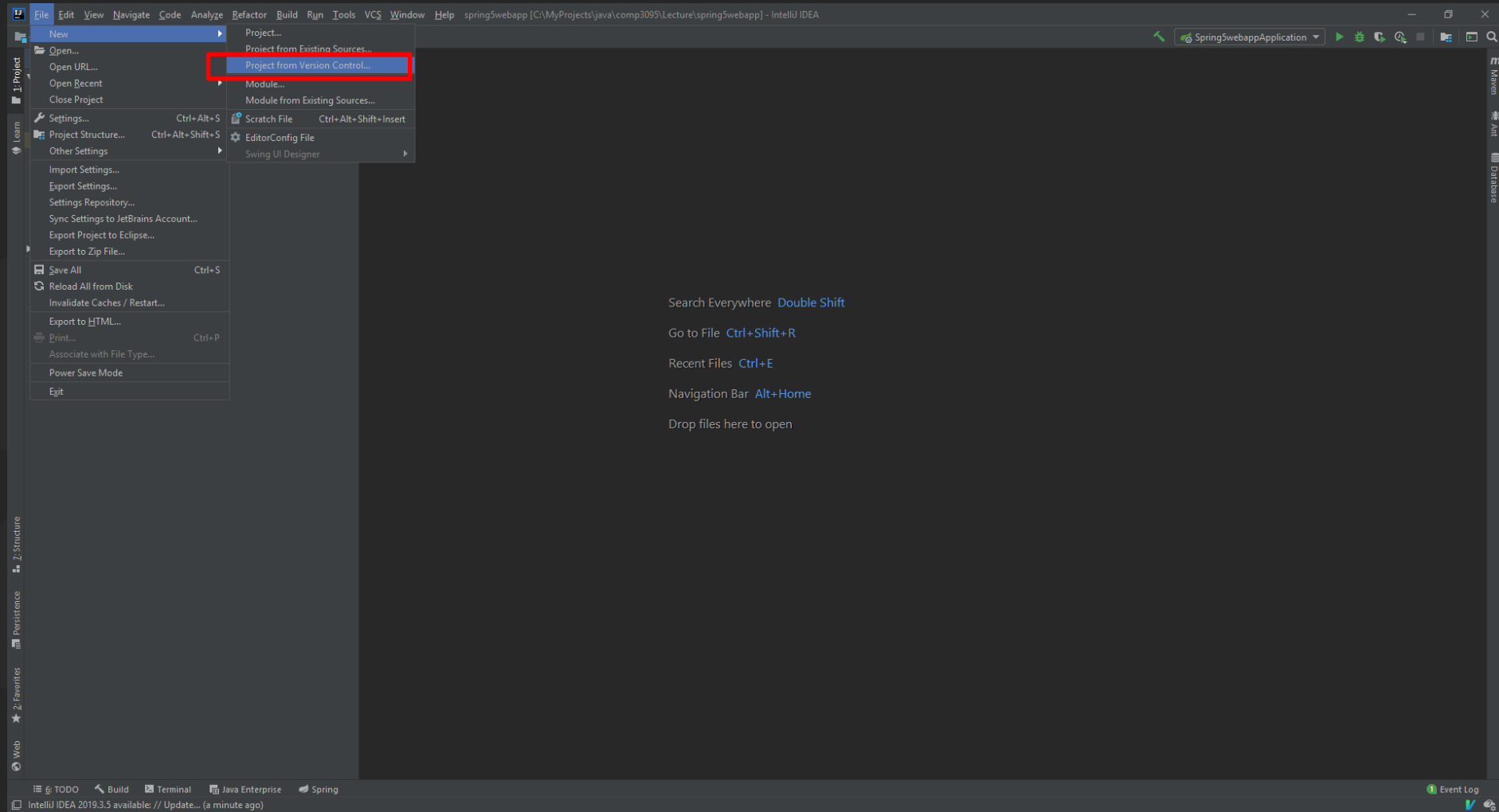
# Create a Spring DI Example Project



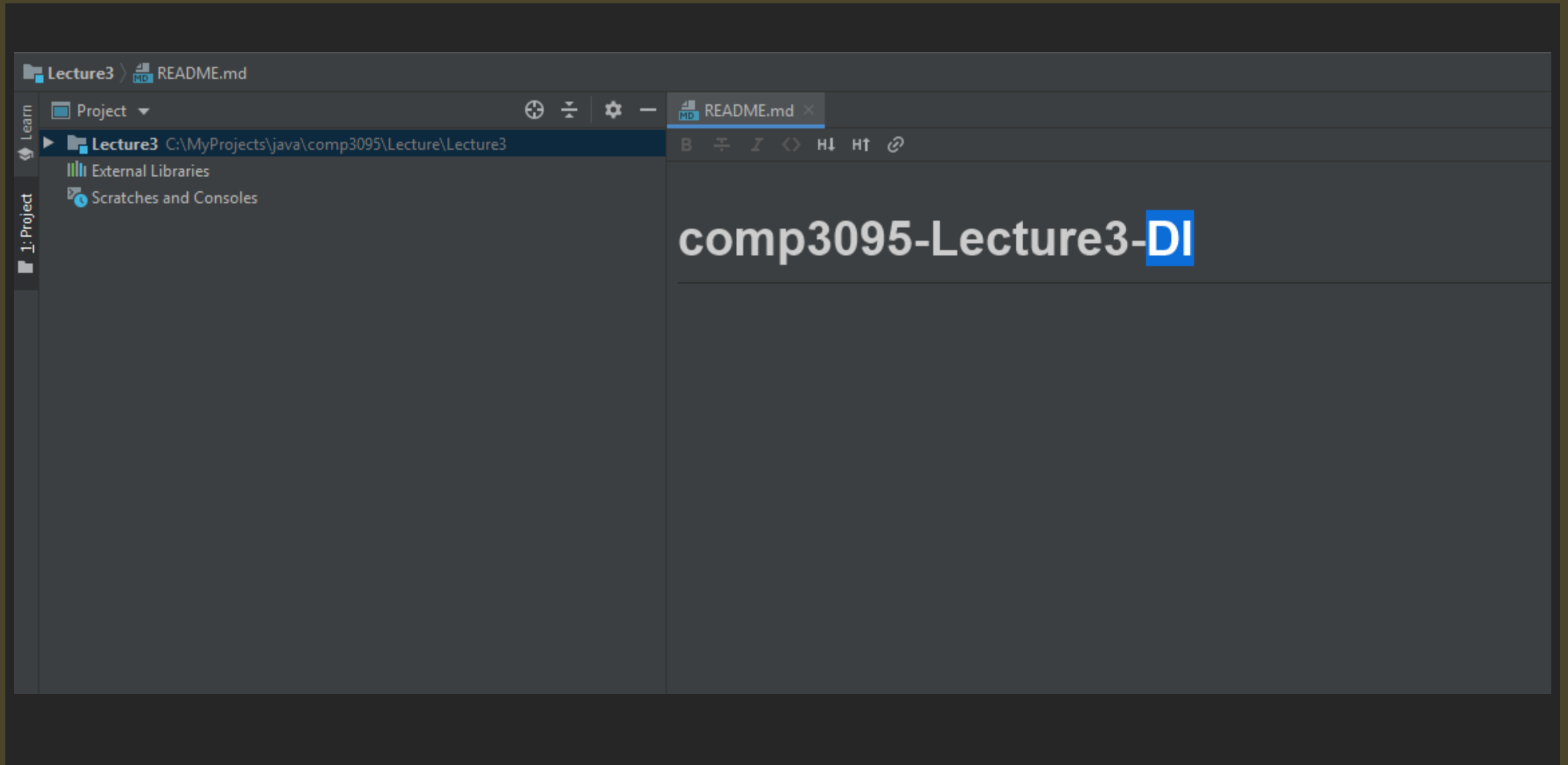
# Create and Clone GIT Project



# Create and Clone GIT Project...




# Create and Clone GIT Project...



# Create and Clone GIT Project...

- Navigate to start.spring.io and create project

 **spring** initializr

**Project**  
☒ Maven Project ☐ Gradle Project

**Language**  
☒ Java ☐ Kotlin ☐ Groovy

**Spring Boot**  
☐ 2.4.0 (SNAPSHOT) ☐ 2.4.0 (M2) ☐ 2.3.4 (SNAPSHOT) ☒ 2.3.3  
☐ 2.2.10 (SNAPSHOT) ☐ 2.2.9 ☐ 2.1.17 (SNAPSHOT) ☐ 2.1.16

**Project Metadata**

Group

Artifact

Name

Description

Package name

**Packaging** ☒ Jar ☐ War

Java ☐ 14 ☒ 11 ☐ 8

**Dependencies** ADD DEPENDENCIES... CTRL + B

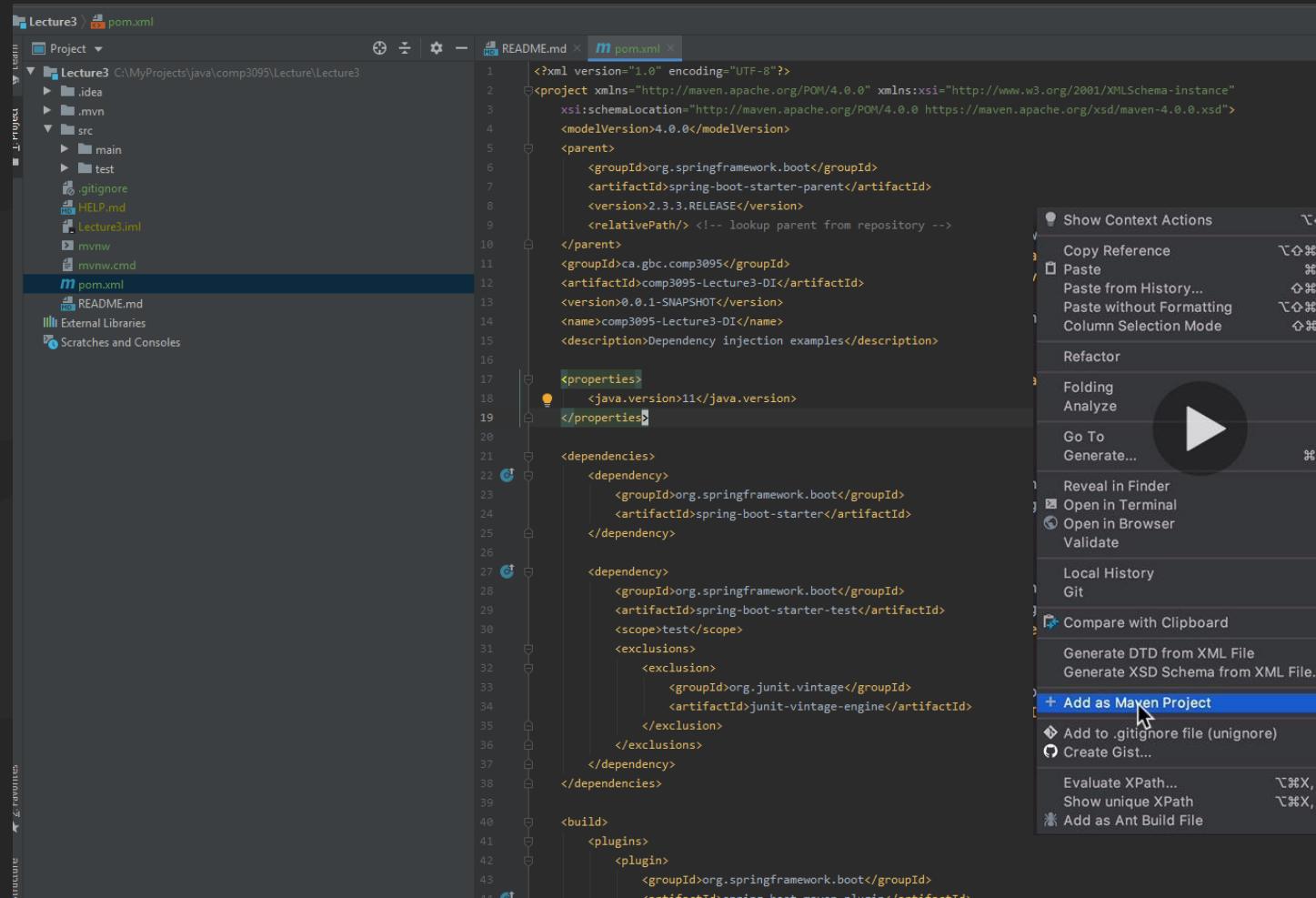
No dependency selected

**Buttons:** GENERATE CTRL + G EXPLORE CTRL + SPACE SHARE...

No additional dependencies  
required for this project

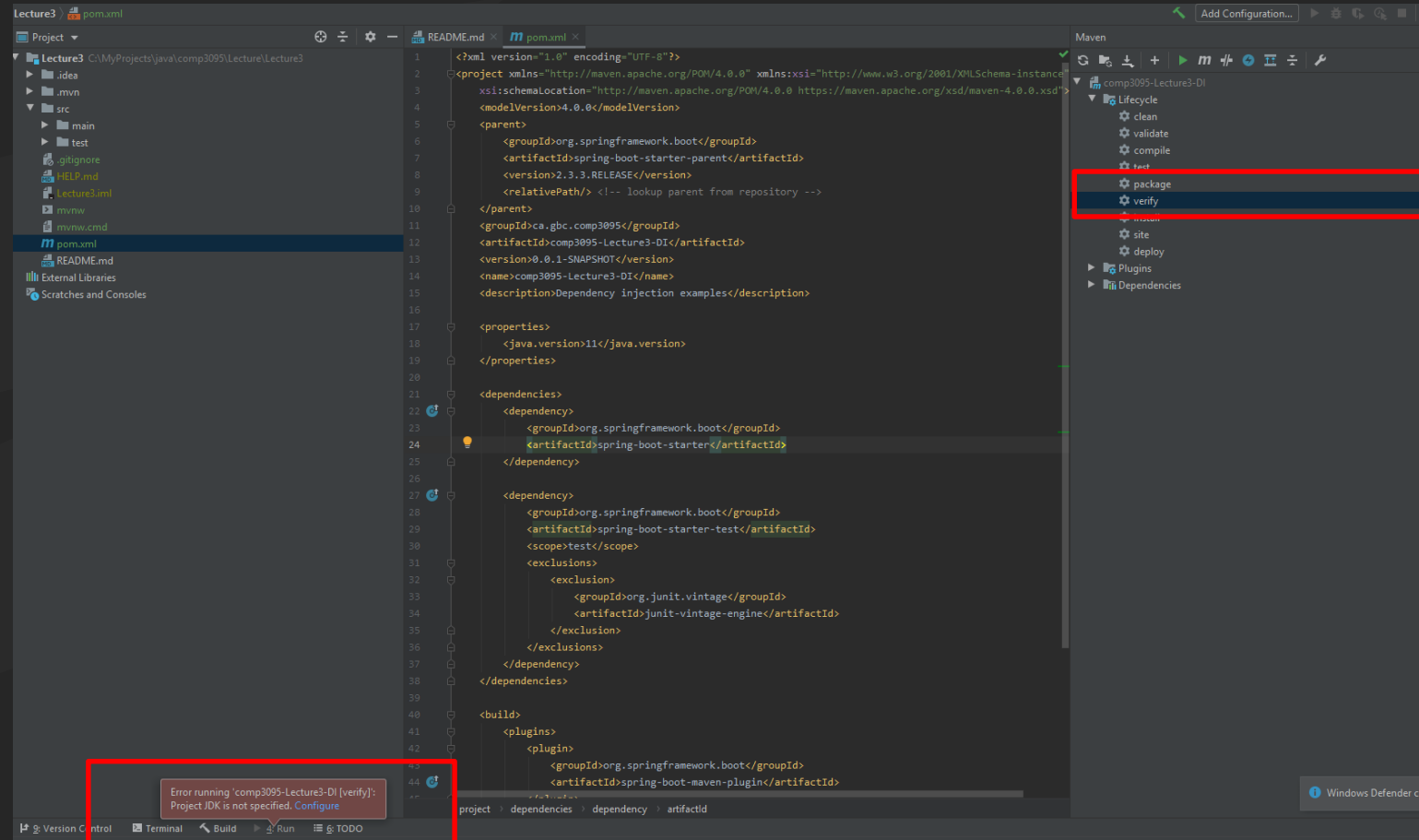
# Extract Project and drag & drop into IntelliJ

- Click on pom.xml file, and “Add as Maven Project”



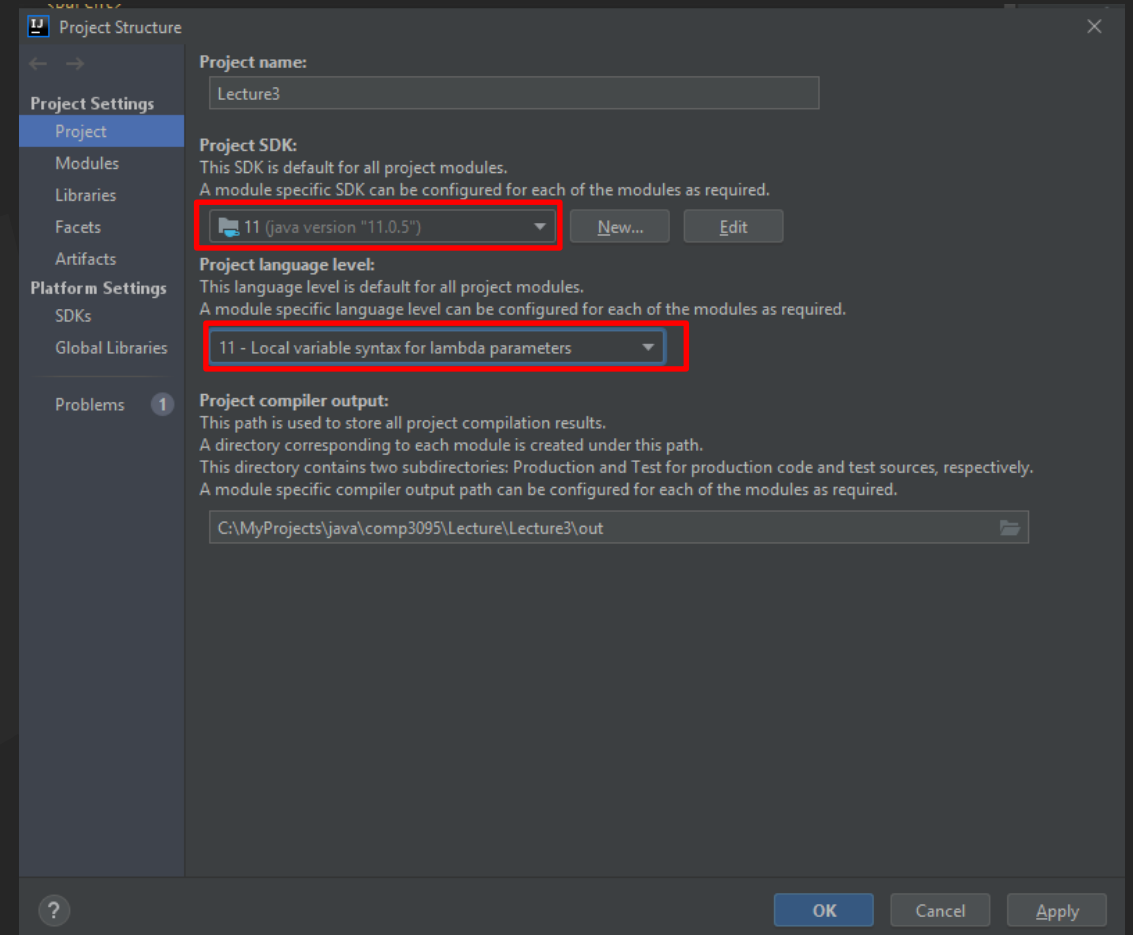
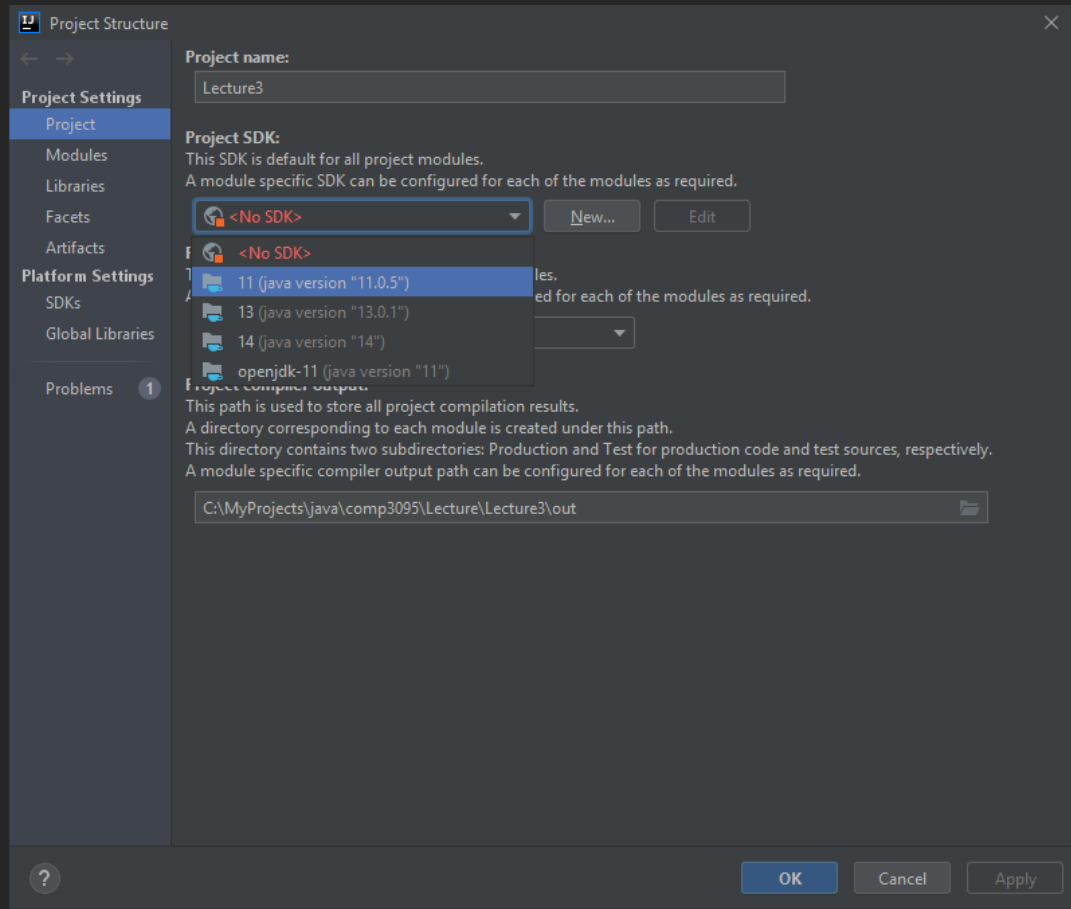
# Extract Project and drag & drop into IntelliJ

- Maven → Lifecycle → Verify
- Configure JDK (if necessary)



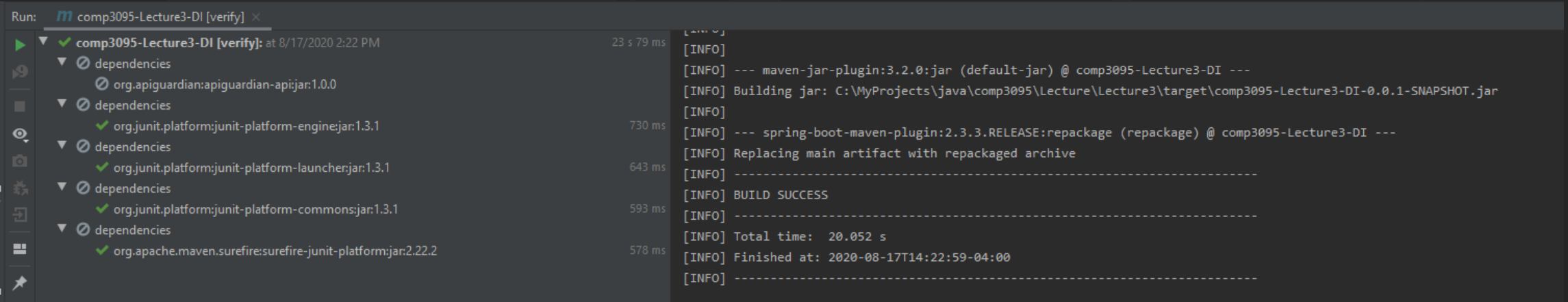
# Extract Project and drag & drop into IntelliJ

- Configure JDK



# Extract Project and drag & drop into IntelliJ

- Maven verify → Should work okay now



The screenshot displays the Run console in IntelliJ IDEA, showing the execution of a Maven `verify` goal for the project `comp3095-Lecture3-DI`. The console is divided into two main sections: a left pane showing the Maven lifecycle and dependencies, and a right pane showing the Maven log output.

**Left Pane (Maven Lifecycle and Dependencies):**

- Run:** `m comp3095-Lecture3-DI [verify]` (23 s 79 ms)
- Dependencies:**
  - `org.apiguardian:apiguardian-api:jar:1.0.0`
  - `org.junit.platform:junit-platform-engine:jar:1.3.1` (730 ms)
  - `org.junit.platform:junit-platform-launcher:jar:1.3.1` (643 ms)
  - `org.junit.platform:junit-platform-commons:jar:1.3.1` (593 ms)
  - `org.apache.maven.surefire:surefire-junit-platform:jar:2.22.2` (578 ms)

**Right Pane (Maven Log Output):**

```
[INFO] --- maven-jar-plugin:3.2.0:jar (default-jar) @ comp3095-Lecture3-DI ---
[INFO] Building jar: C:\MyProjects\java\comp3095\Lecture\Lecture3\target\comp3095-Lecture3-DI-0.0.1-SNAPSHOT.jar
[INFO] --- spring-boot-maven-plugin:2.3.3.RELEASE:repackage (repackage) @ comp3095-Lecture3-DI ---
[INFO] Replacing main artifact with repackaged archive
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 20.052 s
[INFO] Finished at: 2020-08-17T14:22:59-04:00
[INFO] -----
```



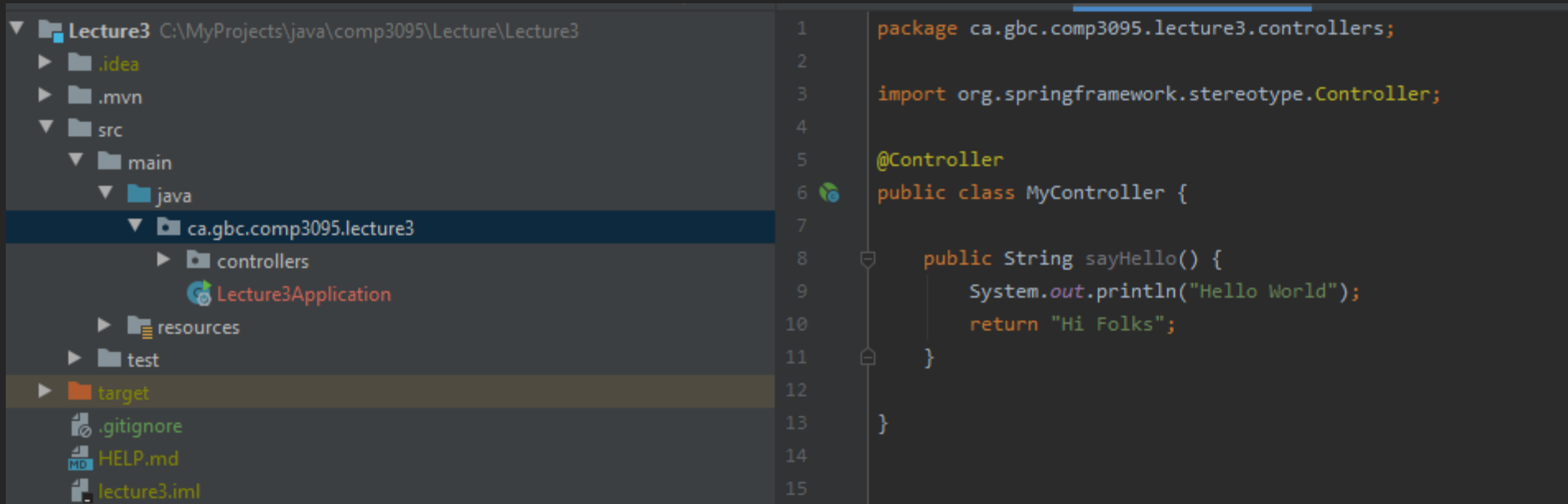
# Spring Context



# Spring Context

- Spring context are also called spring IoC (Inversion of Control) containers
- Responsible for instantiating, configuring and assembling beans by reading metadata from XML, Java annotations and/or Java Code in configuration files.

# Create Controller



The screenshot shows an IDE with a project structure on the left and a code editor on the right. The project structure is for a project named 'Lecture3' located at 'C:\MyProjects\java\comp3095\Lecture\Lecture3'. The structure includes folders for '.idea', '.mvn', 'src', 'main', 'java', 'controllers', 'resources', 'test', and 'target'. The 'java' folder is expanded, showing the package 'ca.gbc.comp3095.lecture3'. The 'controllers' folder is also expanded, showing the 'Lecture3Application' class. The 'target' folder is highlighted. The code editor on the right shows the following Java code:

```
1 package ca.gbc.comp3095.lecture3.controllers;  
2  
3 import org.springframework.stereotype.Controller;  
4  
5 @Controller  
6 public class MyController {  
7  
8     public String sayHello() {  
9         System.out.println("Hello World");  
10        return "Hi Folks";  
11    }  
12  
13 }  
14  
15
```

# Obtaining a Handle on Spring Controller

This program effectively starts and stop since there is no web context to this application (runs like a standalone application)

Notice in the source we never explicitly instantiate a new controller (using "new") ... rather the Spring Context Framework has actually done this for us already.

Spring is managing and constructing the creation of the beans / controllers.

The screenshot shows an IDE with a project named 'Lecture3' at the path 'C:\MyProjects\java\comp3095\Lecture\Lecture3'. The project structure includes 'src/main/java/ca.gbc.comp3095.lecture3' and 'target'. The source code for 'Lecture3Application.java' is displayed, showing a Spring Boot application with a main method that runs the application context, retrieves a 'MyController' bean, and prints a greeting. The console output at the bottom shows the application starting and printing 'Hello World' and 'Hi Folks'.

```
package ca.gbc.comp3095.lecture3;

import ...

@SpringBootApplication
public class Lecture3Application {

    public static void main(String[] args) {

        ApplicationContext ctx = SpringApplication.run(Lecture3Application.class, args);

        MyController myController = (MyController) ctx.getBean("myController");

        String greeting = myController.sayHello();

        System.out.println(greeting);

    }

}
```

Run: Lecture3Application

Console

```
2020-08-17 17:29:59.535 INFO 14652 --- [main] c.g.c.lecture3.Lecture3Application : Started Lecture3Application in 1.36 seconds (JVM running for 3.735s)

Hello World
Hi Folks
```

# Basics of Dependency Injection



# Dependency Injection (DI)

- Design pattern used to implement IoC.
- It allows the creation of dependent objects of a class and provides those objects to a class through various ways.
- Where a needed dependency is injected by another object
- The class being injected has no responsibility in instantiating the object being injected.
- We move the creation and binding of the dependent objects outside of the class that depends on them.

# Types of Dependency Injection

## 1. Property Injection

- Least preferred
- Can be public or private properties
- Using private is discouraged / BAD practice

## 2. Setters Injection

- Area of some debate

## 3. Constructor Injection

- Most preferred

# Concrete Classes vs Interfaces

- DI can be done with Concrete Classes or with Interfaces
- Generally DI with Concrete should be avoided
- DI with interfaces is highly preferred
  - Allows runtime to decide implementation to inject
  - Follows interface Segregation Principle of SOLID
  - Also makes your code more testable



# Inversion of Control (IoC)

- Is a technique to allow dependencies to be injected at runtime
- Dependencies are not predetermined
- Methods defined by the user will often be called by the framework itself, rather than the users application code.
- The framework often plays the role of the main program, coordinating and sequencing the application activity.

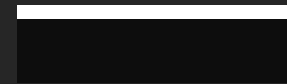
# IoC vs Dependency Injection

- IoC and Di are easily confused
- DI refers much to the composition of your classes
  - i.e. → You compose your classes with DI in mind
- IoC is the runtime environment of your code
  - i.e → Spring Frameworks IoC container
  - Spring is in control of the injection of dependencies

# Best Practices with Dependency Injection

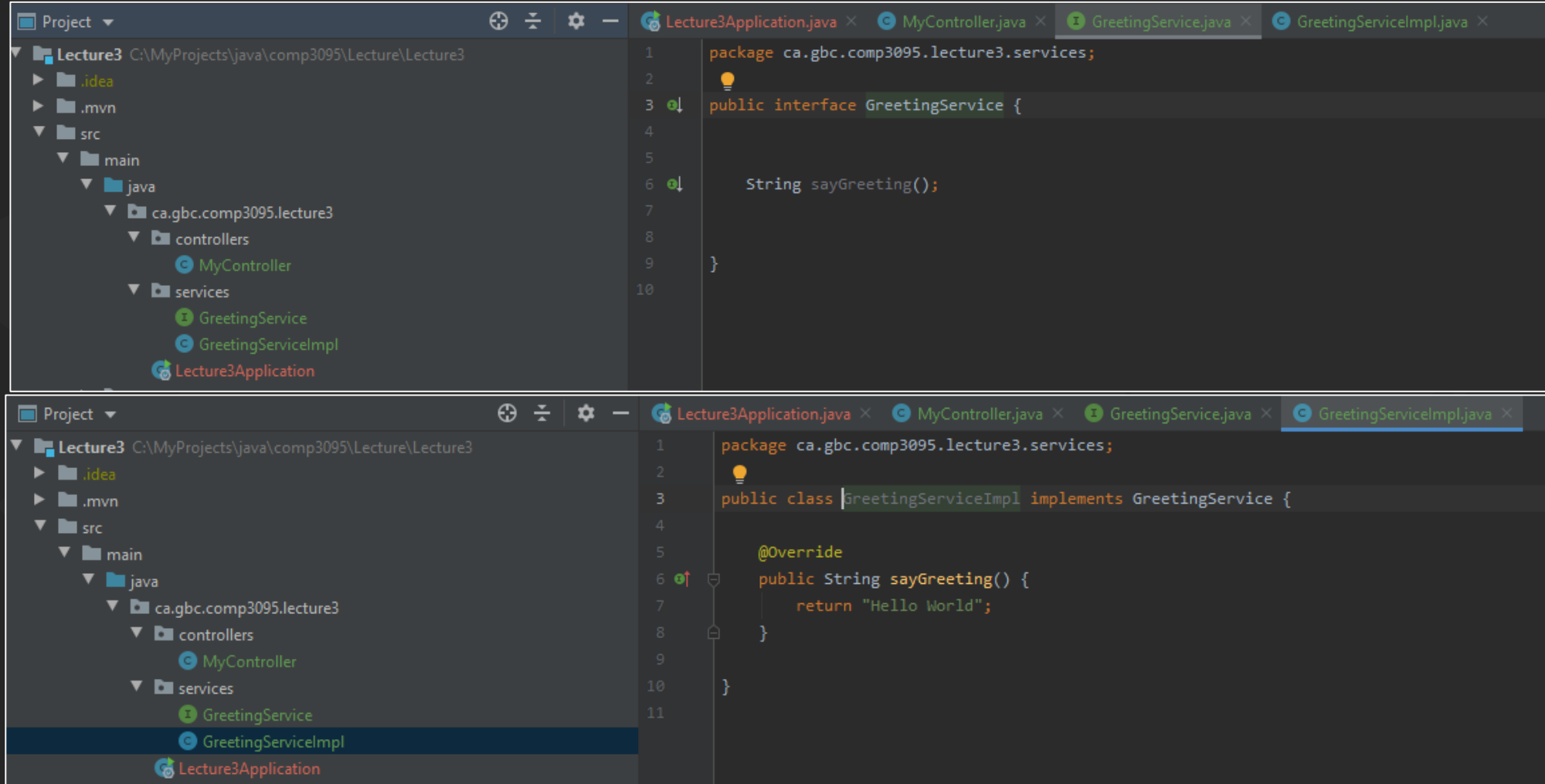
1. Favor using Constructor Injection over Setter Injection
2. Use final properties for injected components
3. Whenever practical, code to an interface

# Dependency Injection without Spring



# Adding a Service

- Add GreetingService Interface and Implementation

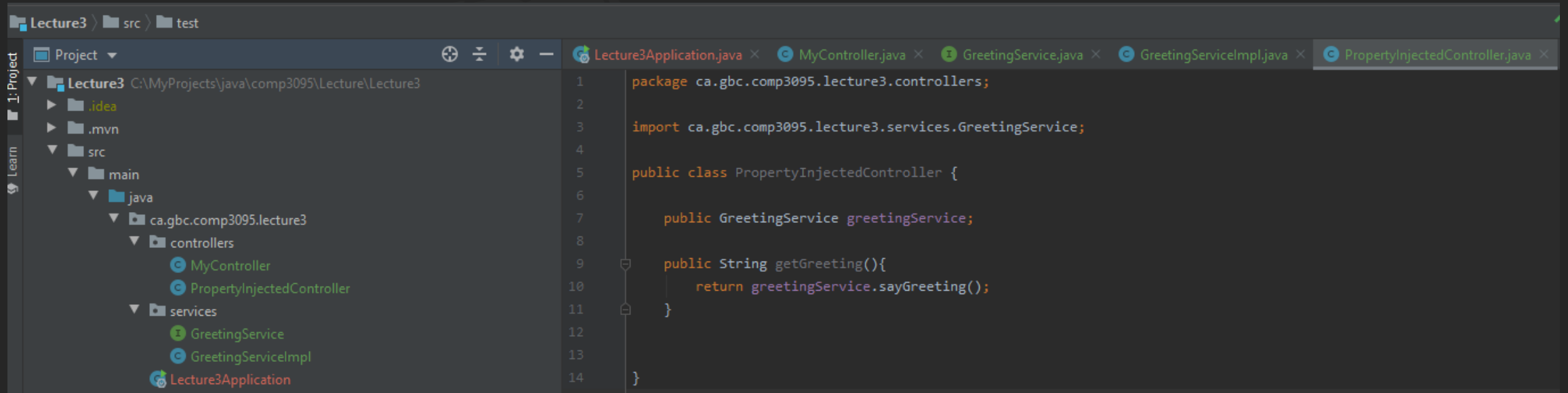


# Property Injection



# Property Injected Controller

- Add New Controller



# Create Junit Test

- Add Junit Test

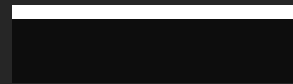
The screenshot illustrates the process of creating and running a JUnit test in IntelliJ IDEA. It is divided into four main sections:

- Create Test Dialog:** Located on the left, this dialog shows the configuration for a new test. The 'Testing library' is set to 'JUnit5'. The 'Class name' is 'PropertyInjectedControllerTest', and the 'Destination package' is 'ca.gbc.comp3095.lecture3.controllers'. The 'Generate' options for '@Before', '@After', and 'Show inherited methods' are all unchecked. The 'Member' list shows 'getGreeting():String' is selected.
- Project Structure:** The 'Project' view in the center shows the file hierarchy. The 'test' directory is expanded, showing the 'java' subdirectory where the new test class 'PropertyInjectedControllerTest' has been created.
- Source Code:** The 'PropertyInjectedControllerTest.java' file is open on the right. It contains the following code:

```
1 package ca.gbc.comp3095.lecture3.controllers;
2
3 import ...
4
5 class PropertyInjectedControllerTest {
6
7     PropertyInjectedController controller;
8
9     //perform this before each unit test
10    @BeforeEach
11    void setUp(){
12        controller = new PropertyInjectedController(); // mimicking what spring context would be doing for us.
13        controller.greetingService = new GreetingServiceImpl();
14    }
15
16    @Test
17    void getGreeting() {
18        System.out.println(controller.getGreeting());
19    }
20 }
```
- Run Results:** The bottom panel shows the execution of the test. The command 'PropertyInjectedControllerTest.getGreeting' was run. The results show 'Tests passed: 1 of 1 test - 38 ms'. The output of the test is 'Hello World', and the process finished with exit code 0.



# Setter Injection



# Setter Injected Controller

- Add New Controller

The screenshot shows an IDE with a project named 'Lecture3' at the path 'C:\MyProjects\java\comp3095\Lecture\Lecture3'. The project structure is visible in the left sidebar, showing folders for '.idea', '.mvn', 'src', 'main', 'java', 'controllers', 'services', and 'resources', along with files like 'MyController', 'PropertyInjectedController', 'SetterInjectedController', 'GreetingService', 'GreetingServiceImpl', and 'Lecture3Application'. The 'SetterInjectedController' file is selected in the 'controllers' folder.

The main editor window displays the code for 'SetterInjectedController.java'. The code is as follows:

```
1 package ca.gbc.comp3095.lecture3.controllers;
2
3 import ca.gbc.comp3095.lecture3.services.GreetingService;
4
5 public class SetterInjectedController {
6
7     private GreetingService greetingService;
8
9     public void setGreetingService(GreetingService greetingService) {
10         this.greetingService = greetingService;
11     }
12
13     public String getGreeting(){
14         return greetingService.sayGreeting();
15     }
16
17 }
```

# Create Junit Test

- Add Junit Test

Create Test

Testing library: JUnit5

Class name: SetterInjectedControllerTest

Superclass:

Destination package: ca.gbc.comp3095.lecture3.controllers

Generate: ☒ setUp/@Before ☐ tearDown/@After

Generate test methods for: ☐ Show inherited methods

Member

☐ setGreetingService(greetingService:GreetingService):void

☒ getGreeting():String

OK Cancel

Project

Lecture3 C:\MyProjects\java\comp3095\Lecture\Lecture3

src

main

java

ca.gbc.comp3095.lecture3

controllers

MyController

PropertyInjectedController

SetterInjectedController

services

GreetingService

GreetingServiceImpl

Lecture3Application

resources

test

java

ca.gbc.comp3095.lecture3

```
package ca.gbc.comp3095.lecture3.controllers;

import ...

class SetterInjectedControllerTest {

    SetterInjectedController controller;

    @BeforeEach
    void setUp() {
        controller = new SetterInjectedController();
        controller.setGreetingService(new GreetingServiceImpl());
    }

    @Test
    void getGreeting() {
        System.out.println(controller.getGreeting());
    }
}
```

Tests passed: 1 of 1 test – 68 ms

Test Results

SetterInjectedControllerTest

getGreeting()

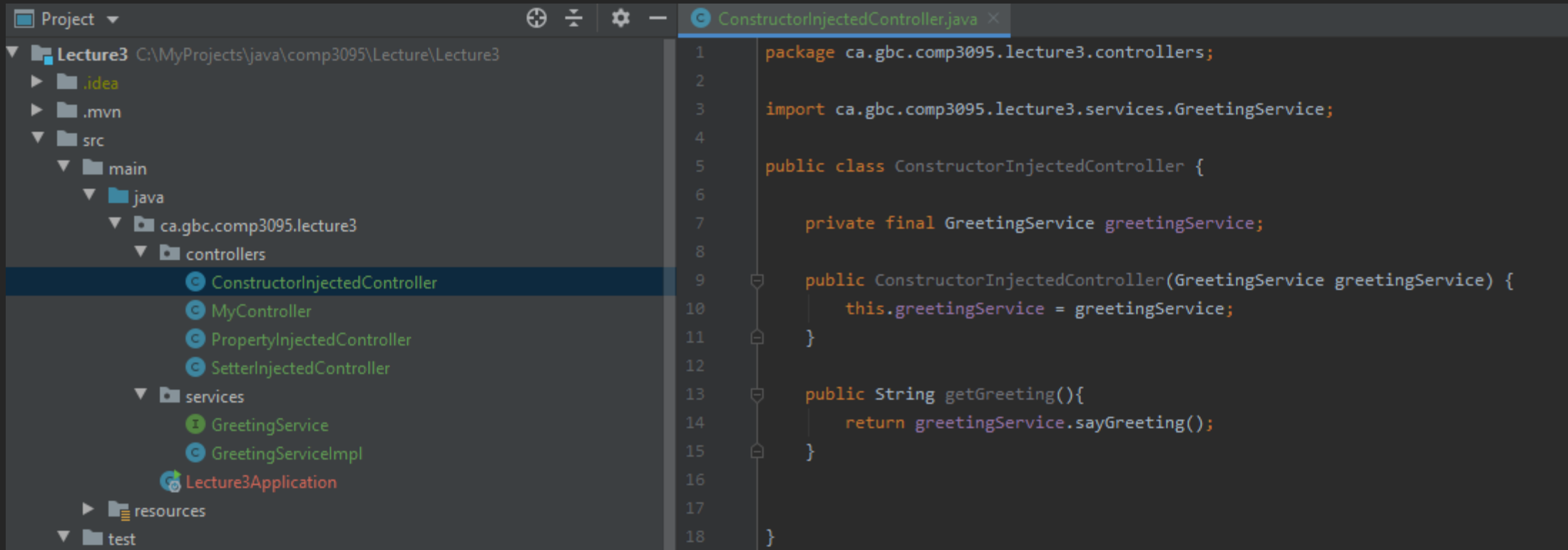
Process finished with exit code 0

# Constructor Injection



# Constructor Injected Controller

- Add New Controller



The screenshot shows an IDE interface with a project explorer on the left and a code editor on the right. The project explorer displays the project structure for 'Lecture3', including folders like .idea, .mvn, src, and test. The 'src/main/java' folder is expanded, showing the package 'ca.gbc.comp3095.lecture3' and its sub-packages 'controllers' and 'services'. The 'controllers' package contains four classes: 'ConstructorInjectedController' (selected), 'MyController', 'PropertyInjectedController', and 'SetterInjectedController'. The 'services' package contains 'GreetingService' and 'GreetingServiceImpl'. The 'Lecture3Application' class is also visible. The code editor on the right shows the implementation of 'ConstructorInjectedController.java'. The code includes a package declaration, an import for 'GreetingService', and the class definition with a constructor and a 'getGreeting()' method.

```
1 package ca.gbc.comp3095.lecture3.controllers;
2
3 import ca.gbc.comp3095.lecture3.services.GreetingService;
4
5 public class ConstructorInjectedController {
6
7     private final GreetingService greetingService;
8
9     public ConstructorInjectedController(GreetingService greetingService) {
10         this.greetingService = greetingService;
11     }
12
13     public String getGreeting(){
14         return greetingService.sayGreeting();
15     }
16
17
18 }
```

# Create Junit Test

- Add Junit Test

Create Test

Testing library: JUnit5

Class name: ConstructorInjectedControllerTest

Superclass:

Destination package: ca.gbc.comp3095.lecture3.controllers

Generate: ☒ setUp/@Before ☐ tearDown/@After

Generate test methods for: ☐ Show inherited methods

Member

☒ ☐ ☐ getGreeting():String

OK Cancel

Project

Lecture3 C:\MyProjects\java\comp3095\Lecture\Lecture3

src

main

java

ca.gbc.comp3095.lecture3

controllers

ConstructorInjectedController

MyController

PropertyInjectedController

SetterInjectedController

services

GreetingService

GreetingServiceImpl

Lecture3Application

resources

test

ConstructorInjectedController.java

ConstructorInjectedControllerTest.java

```
1 package ca.gbc.comp3095.lecture3.controllers;
2
3 import ...
4
5
6
7
8
9 class ConstructorInjectedControllerTest {
10
11     ConstructorInjectedController controller;
12
13     @BeforeEach
14     void setUp() {
15         controller = new ConstructorInjectedController(new GreetingServiceImpl());
16     }
17
18     @Test
19     void getGreeting() {
20         System.out.println(controller.getGreeting());
21     }
22 }
```

Run: ConstructorInjectedControllerTest.getGreeting

Tests passed: 1 of 1 test – 47 ms

Test Results

ConstructorInjectedControllerTest

getGreeting()

47 ms

47 ms

47 ms

"C:\Program Files\Java\OpenJDK\jdk-11\bin\java.exe" ...

Hello World

Process finished with exit code 0

Questions?