# Java Basic Web Project: Phonebook

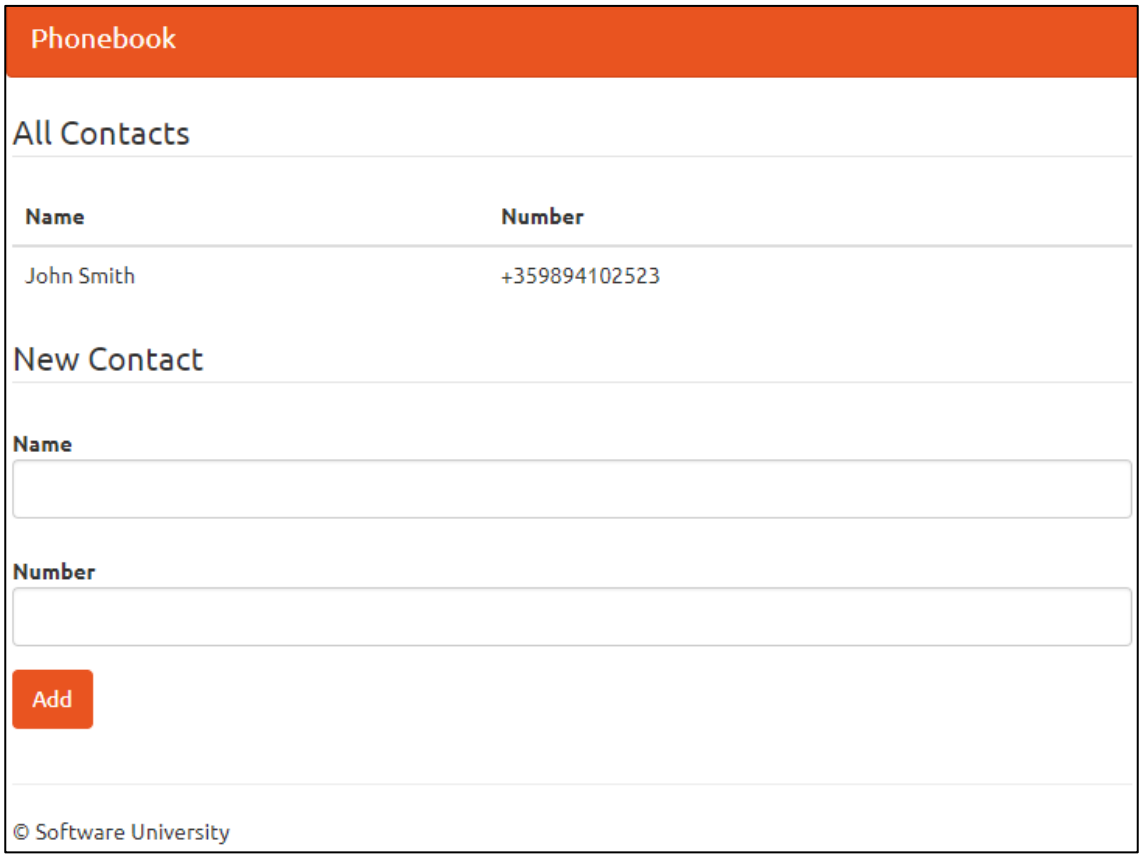
Problems for exercises for the "Programming Fundamentals" course @ SoftUni

## 1. Problem

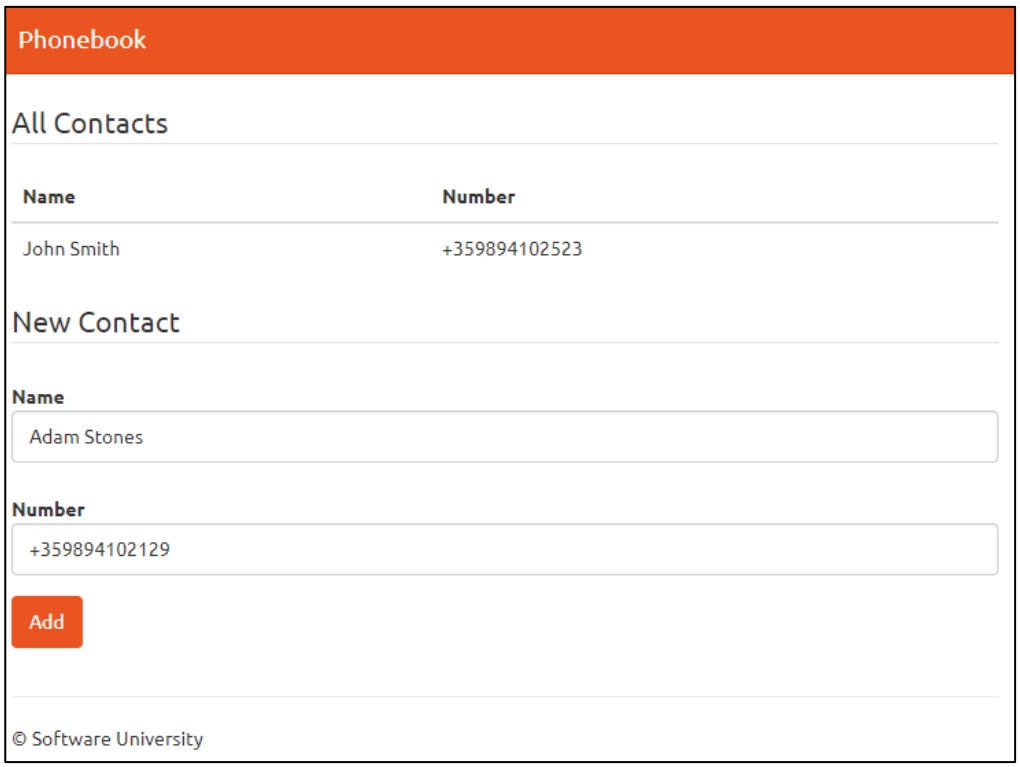
You have been tasked to create a simple **Phonebook** application. The application should hold **contacts**, which are the main app **entity**.

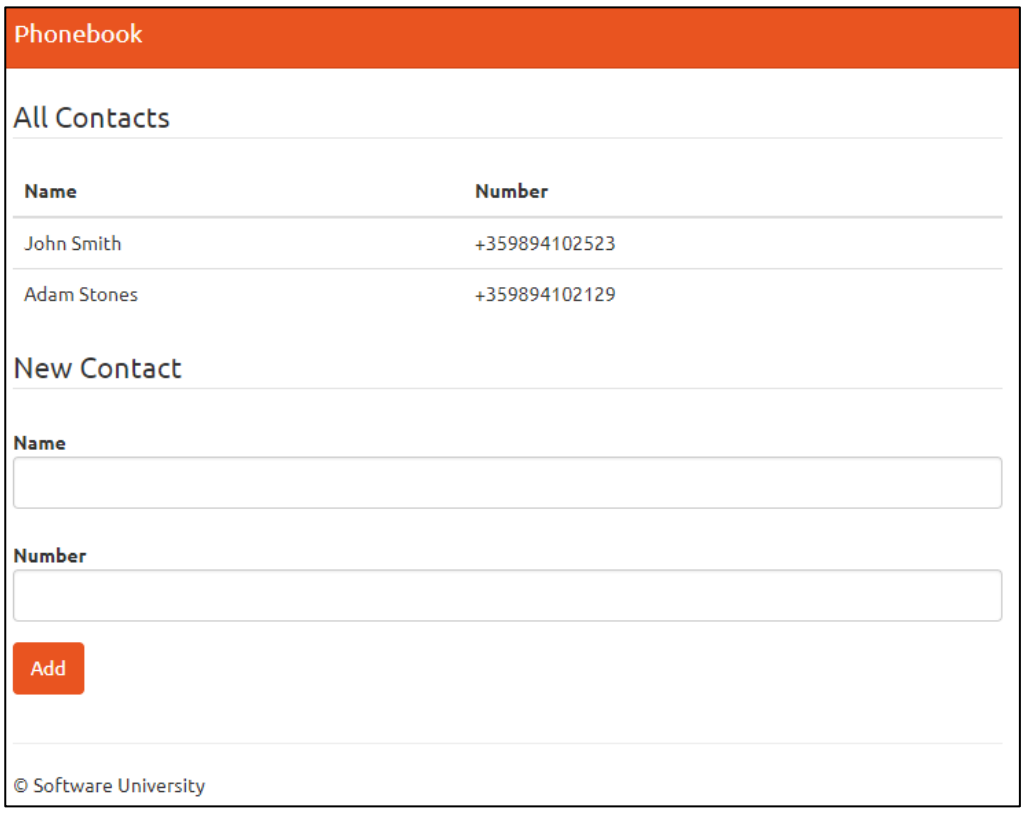
The functionality of the application should support:

- **Listing contacts**



- **Add Contact**




# 2. Overview

## Requirements

- **Spring** framework (**Spring MVC** + **Spring Boot** + **Spring Data**)
- **Thymeleaf** view engine

## Data Model

The **`Contact`** entity holds **2 properties**:

- **`name`** – non-empty text
- **`number`** – non-empty text

## Project Skeletons

You will be given the applications' **`skeletons`**, which holds about **90%** of the logic. You'll be given some **files**. The files will have **partially implemented logic**, so you'll need to write some code for the application to **function properly**.

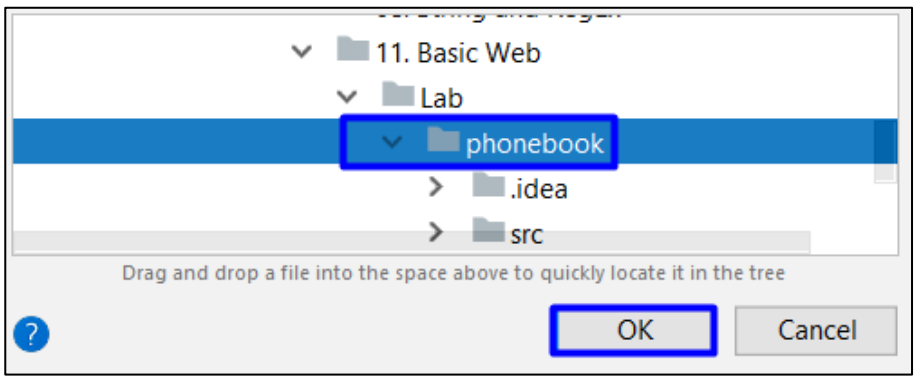
The application's views will be given to you fully implemented. You only need to include them in your business logic.

Everything that has been given to you inside the skeleton is **correctly implemented** and if you write your code **correctly**, the application should work just fine. You are free to change anything in the Skeleton on your account.

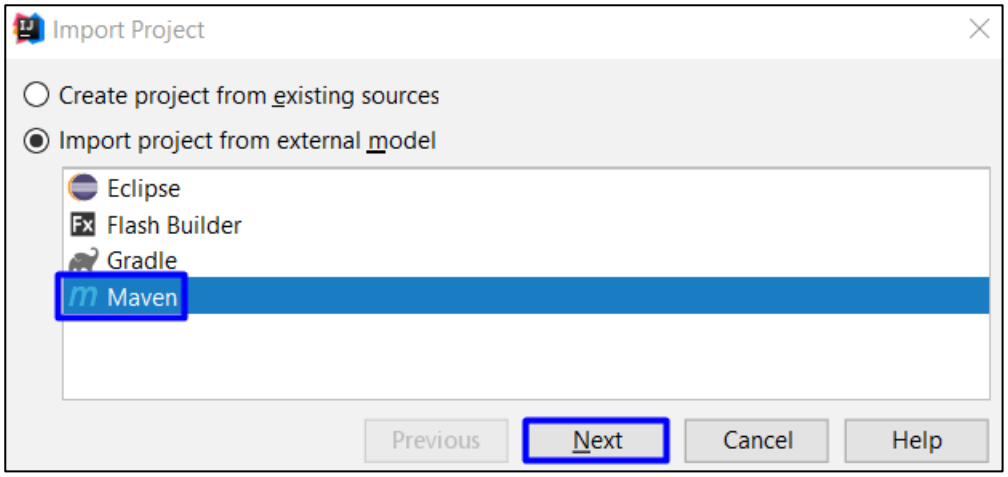
# 3. Setting Up IntelliJ Idea Configuration

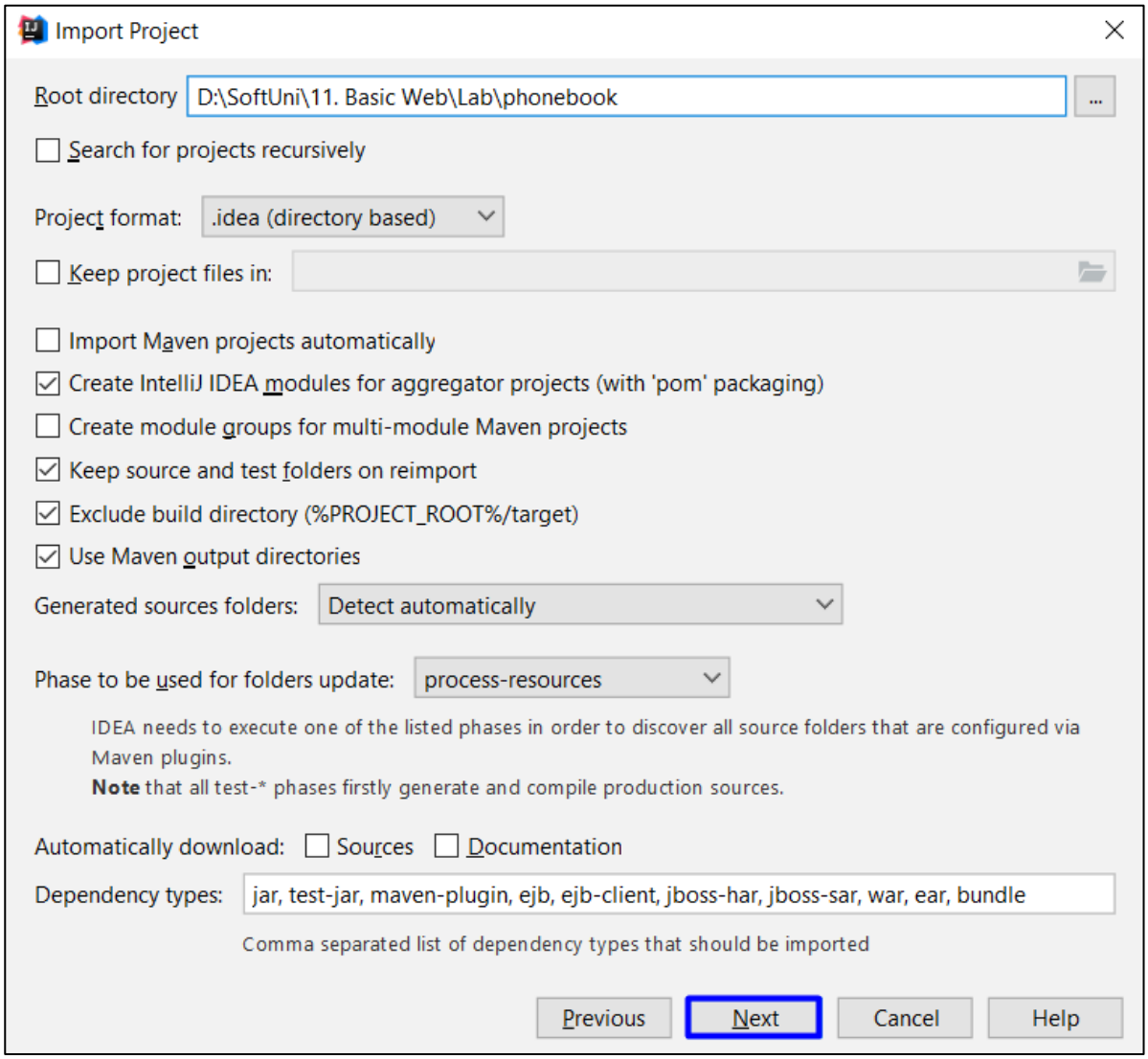
Start **Intellij** and **import** the skeleton.

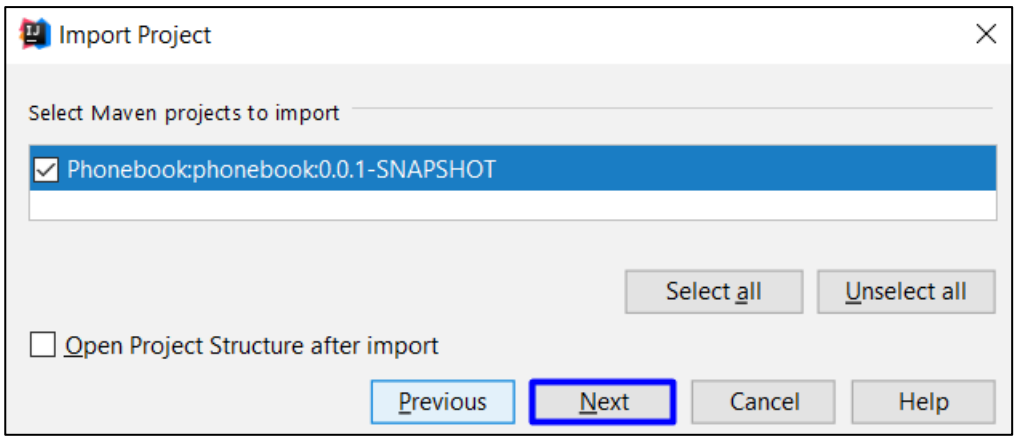

Now we need to **setting up** our **project. Choose** the directory you've download your skeleton and **click OK.**



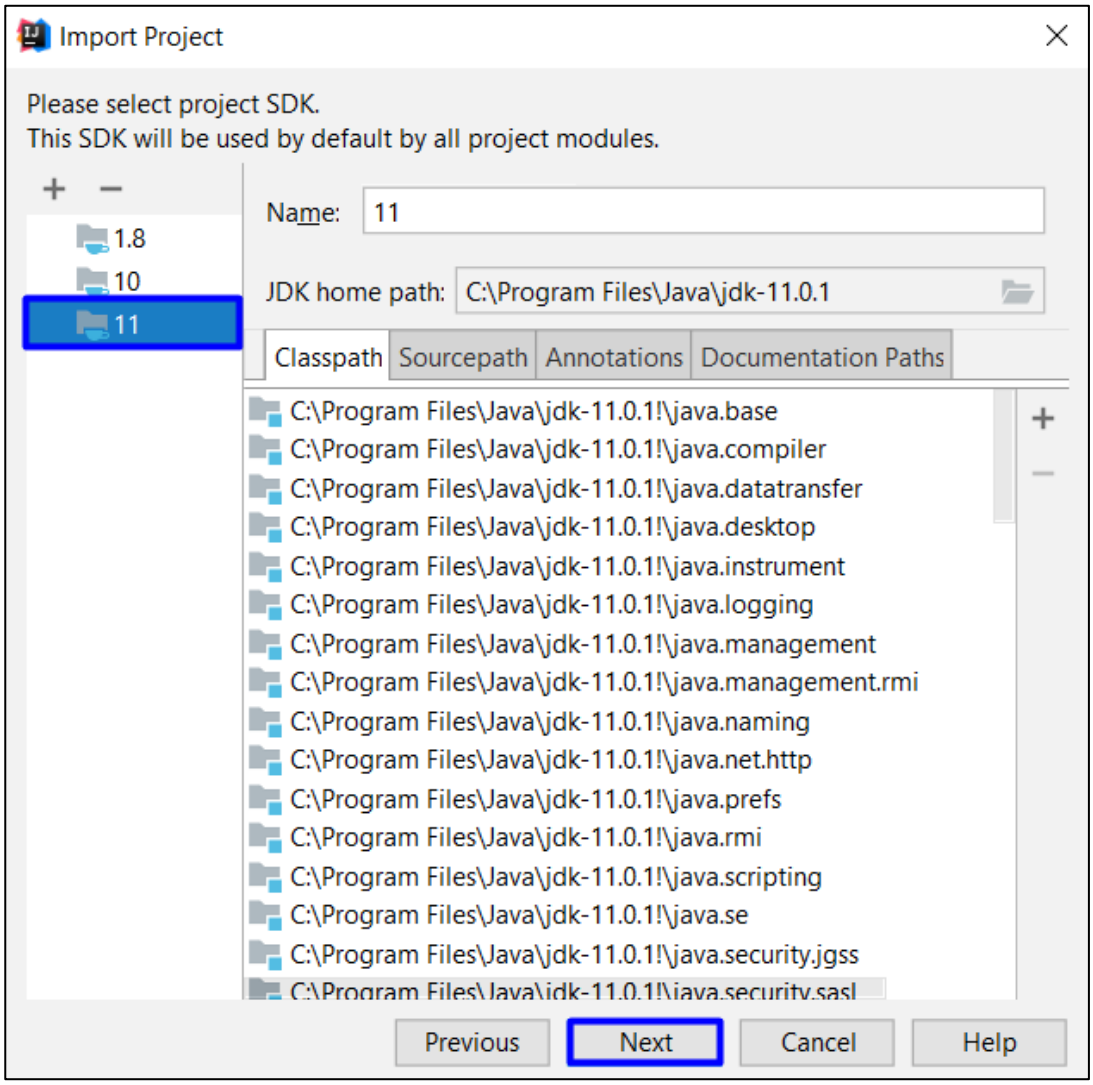
Then you should **click on** Import project from external model, and choose **Maven.**



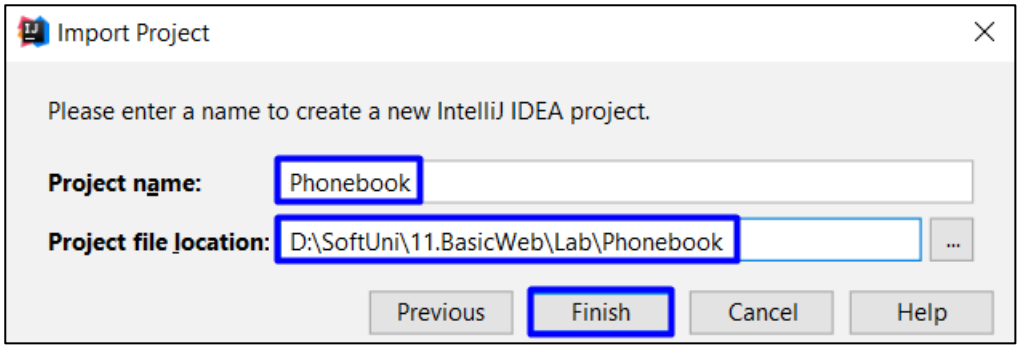






**Choose JDK 11** for your project, and click next.

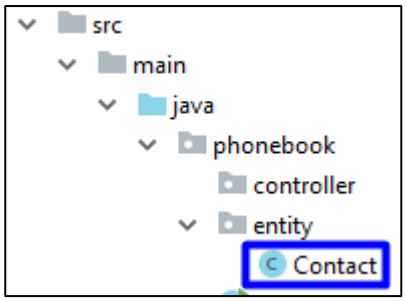


**Finally** just give a name to your project, choose directory and click **finish.**




# 4. Contact Entity

It's time to create our first **`entity`**. In the **`src/main/java/phonebook`** **package** you can see few packages that **define our project**. A **package** is a **folder containing Java** files. The one we are interested in is the "**`entity`**" package. Inside, create a **new java class** called "**`Contact`**":



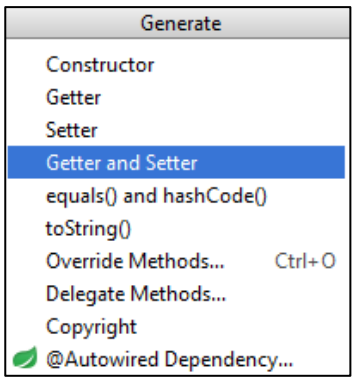

The file should look like this:

```
package phonebook.entity;

public class Contact {
}
```

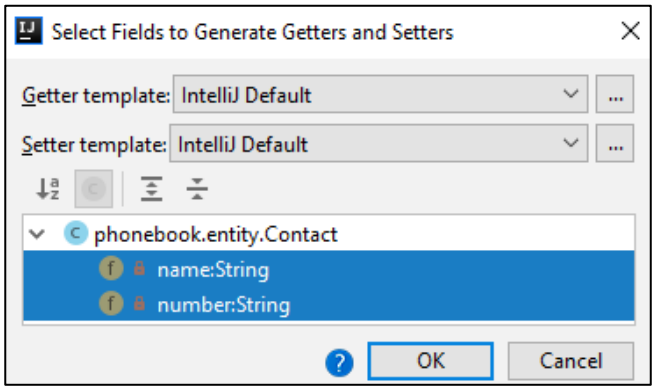
We need to define our **`contact`** entity. Create the following private fields:

```
public class Contact {
    private String name;

    private String number;
}
```

Let's create getters and setters for our fields. You should already be familiar with them. If you are curious why we are doing that, you can read more here. There is a **simple way to create them** in **IntelliJ Idea**. If you press **`"Alt + Insert"`**, you should see that context menu:




Choosing the "**Getter and Setter**" option will **open new window**. You should select **all private fields** from there:




When you **click** "**OK**", you should **receive this code**:

```
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getNumber() {
    return number;
}

public void setNumber(String number) {
    this.number = number;
}
```

It's the time to **create our constructor**:

```
public Contact(String name, String number) {
    this.name = name;
    this.number = number;
}
```

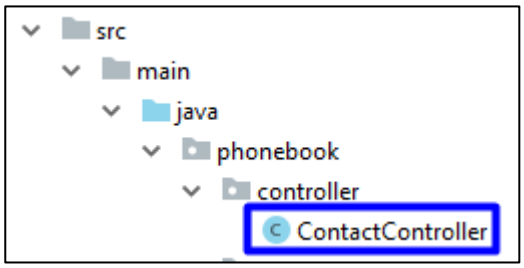
We will **use this constructor** to **create tasks** easily. However, we need to create another **empty constructor** for **Spring**:

```
public Contact() {
}
```

And this is pretty much everything. Our **`Contact`** entity is ready.

# 5. Contact Controller

We have reached the point in which we can create our **controller**. In the "**`controller`**" package create a new class called "**`ConctactController`**":




Add the following annotation:

```
import org.springframework.stereotype.Controller;

@Controller
public class ContactController {

}
```

This class will **list** and **save** contacts. That means that it will use **routes**. In order to let **Spring,** know that this class will be controller, we need to use the "**`@Controller`**" annotation. This annotation also gives us **access** to **requests** and gives us the ability to respond to them. Now, we need to store contacts somewhere. We will use a **list**, later in the course we will learn how to store information in **database**.

```
private List<Contact> contacts;

public ContactController() {
    this.contacts = new ArrayList<>();
}
```

We need to **initialize** our list, so we are going to do it through the **constructor**.

## Listing All Tasks

First, we need to list all articles to the given route. Go to the **`ContactController`** and create method **`index`**.

```
@GetMapping("/")
public ModelAndView index(ModelAndView modelAndView){
}
```

The "**`@GetMapping`**" annotation tells **Spring** that this method **cannot be called** if the user wants to **submit data**. It should be **only used** for **viewing data**. **`ModelAndView`** allows us to pass all the information required by **Spring MVC** in one return:

We have only one **view** which we will use in our application, called "**`index`**". If you look the view, you will find:

```
<tr th:each="contact : ${contacts}">
    <td th:text="${contact.name}"></td>
    <td th:text="${contact.number}"></td>
</tr>
```

Which means that our view expects a collection called "**contacts**", so we have to add **contacts** to the **modelAndView** and return it as a result:

```
@GetMapping("/")
public ModelAndView index(ModelAndView modelAndView) {
    modelAndView.setViewName("index");
    modelAndView.addObject( attributeName: "contacts", contacts);
    return modelAndView;
}
```

We are ready with the listing all **contacts**.

## Adding Contact

As we have mentioned, we have only one view. If you go back to the view and take a look once again, you will find a **form**:

```
<form class="form-horizontal" th:method="POST">
    <fieldset>
        <legend>New Contact</legend>
        <div class="form-group">
            <label for="name" class="col-lg-2 control-label">Name</label>
            <div class="col-lg-10">
                <input type="text" autofocus="autofocus" name="name" title="Name" class="form-control"
                       id="name"/>
            </div>
        </div>
        <div class="form-group">
            <label for="number" class="col-lg-2 control-label">Number</label>
            <div class="col-lg-10">
                <input type="text" autofocus="autofocus" name="number" title="Number" class="form-control"
                       id="number"/>
            </div>
        </div>
        <div class="form-group">
            <div class="col-lg-10 col-lg-offset-2">
                <button type="submit" class="btn btn-primary">Add</button>
            </div>
        </div>
    </fieldset>
</form>
```

Thanks to the form, we can process post requests.

```
@PostMapping("/")
public String add(Contact contact) {

}
```

**Spring** will automatically map the data into **contact** object, if we've created it correctly.

With "**@PostMapping**" annotation, we told **Spring** that **this method expects data** that it needs to **autofill in contact object**. The annotation handles "**POST**" **request** that are usually what the **HTML forms** are using as a "**method**" of the **request**. In summary, **this method** will be called **when users submit the data**.

Finally, we need to **add the contact in our list** and give a **response** to the user.

```
@PostMapping("/")
public String add(Contact contact) {
    this.contacts.add(contact);
    return "redirect:/";
}
```

We redirect the user back to index page as response.

With that we finished our **Java Phonebook**. Feel free to **build on your project even further**. ☺