Lab 08: Unit Testing, Working with Maven

Aims:

1. Add extra functionality to the Zoo code – add your own classes using the class diagram.
2. Add a collection of unit tests – understand how to use **JUnit**, and to test some common issues.
3. Understand why test suites are important to ensure software quality during future maintenance.
4. Using Maven in IntelliJ.

---

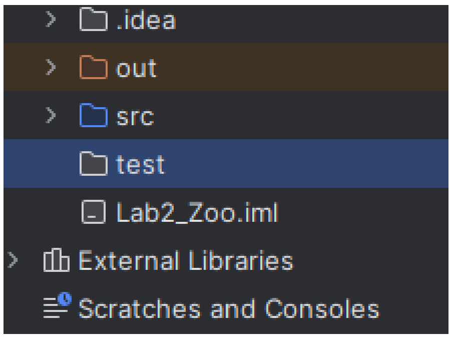
1. Testing the Employee Class

In this section we practice the **Junit Test**. As shown below, we will add a new JUnit Test Case to the project, in which you will test some functionalities of the `Employee` class.

===

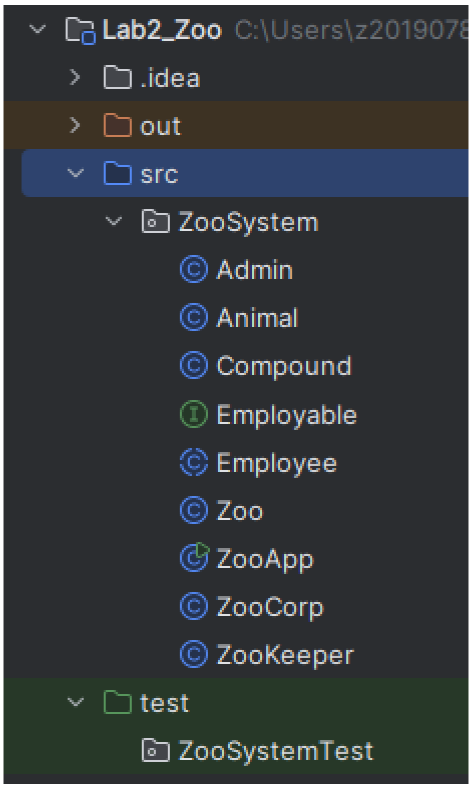
Preparation Task

1. Download **Lab2_Zoo.zip** from "Week 04" section in Moodle, and use it as a starting point.
2. Right click the project name Lab2_Zoo, then new | Directory, create a new directory named '*test*' as below. Notice that it is in grey color.



```
> .idea
> out
> src
  test
  Lab2_Zoo.iml
> External Libraries
  Scratches and Consoles
```

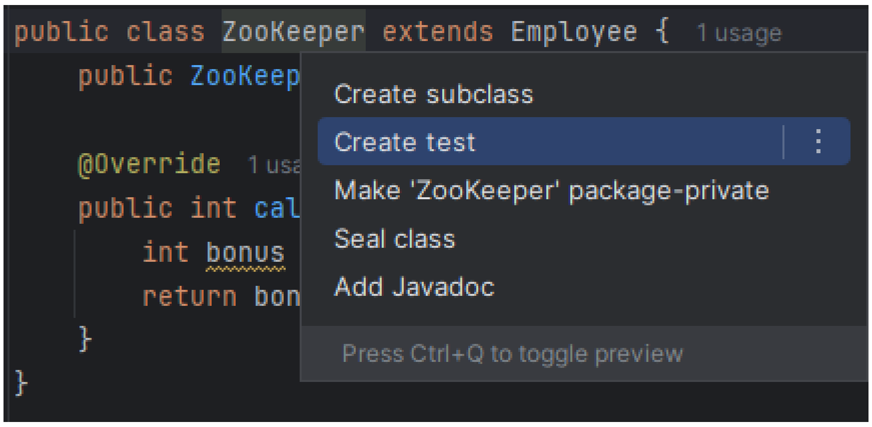
3. Right click on this folder, choose **Mark Directory as**, then **Test Sources Root**. Observe that it now turns green.
4. Right click this folder again, choose **New**, then **Package**, then name this package as "*ZooSystemTest*".
5. Don't forget to change the package name under src to "ZooSystem", to avoid possible compilation errors. (Hint: this is done by right click the current package name | Refactor | Rename…)



```
Lab2_Zoo C:\Users\z2019078
  .idea
  out
  src
    ZooSystem
      Admin
      Animal
      Compound
      Employable
      Employee
      Zoo
      ZooApp
      ZooCorp
      ZooKeeper
  test
    ZooSystemTest
```

Navigate to the '*src*' folder, and open the *ZooKeeper.java* code. Now let's create Junit test classes for this **ZooKeeper** class.

1. Put the cursor on the class name, 'ZooKeeper', then right click, select **Show Context Actions** (using 'Alt+Enter' keyboard shortcut would give the same). Then select **Create Test**.



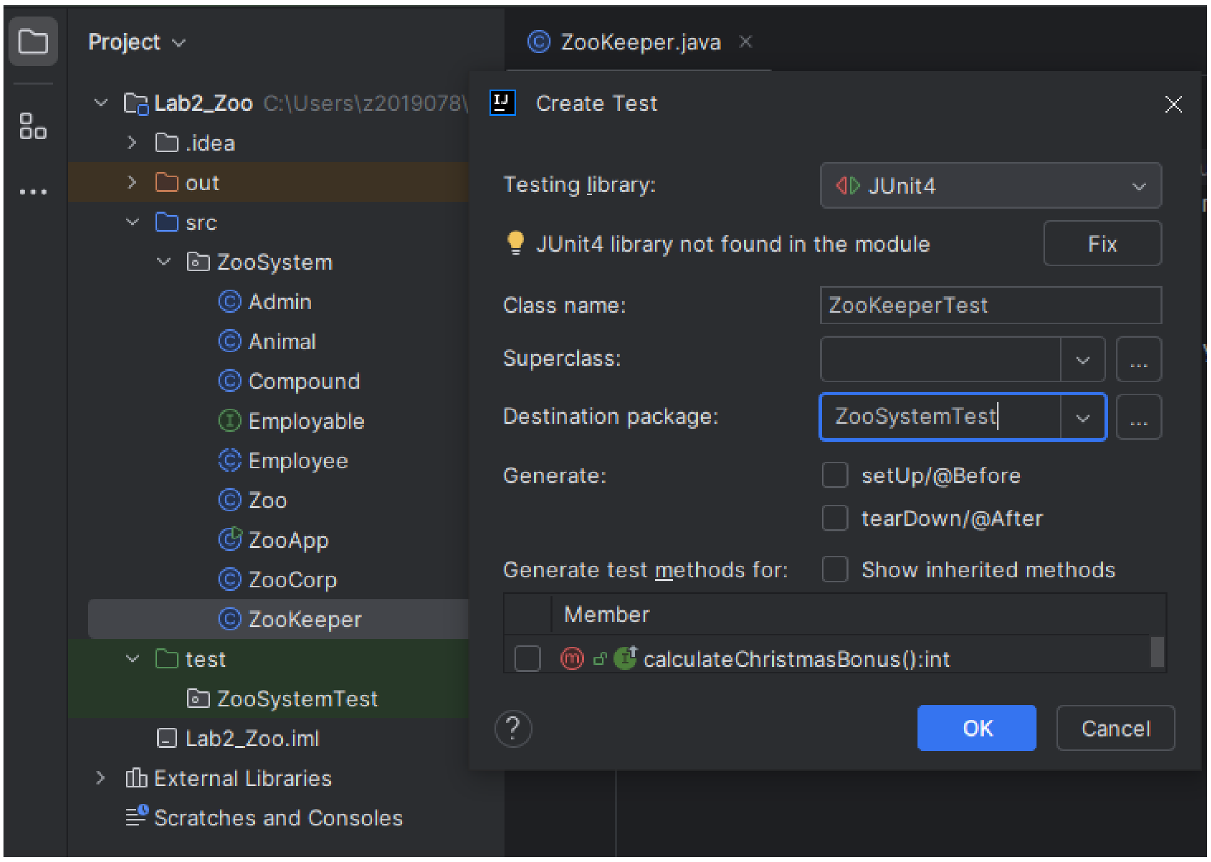
```
public class ZooKeeper extends Employee {  1 usage
    public ZooKeep
    
    @Override  1 usa
    public int cal
        int bonus
        return bon
    }
}
```

Create subclass
Create test
Make 'ZooKeeper' package-private
Seal class
Add Javadoc
Press Ctrl+Q to toggle preview

2. In the pop-up window, configure it as shown below. Note that:
   a) If the library (such as JUnit 4 or 5*) is not installed, click on the "Fix" button to let IntelliJ automatically download and configure for you. (After clicking "Fix" button, wait a while for Maven to find resources, then click "OK".)
   b) It is a good practice to separate the test classes from the source code, thus we choose the destination package as **ZooSystemTest** in the **test** directory, and throw the test classes in.

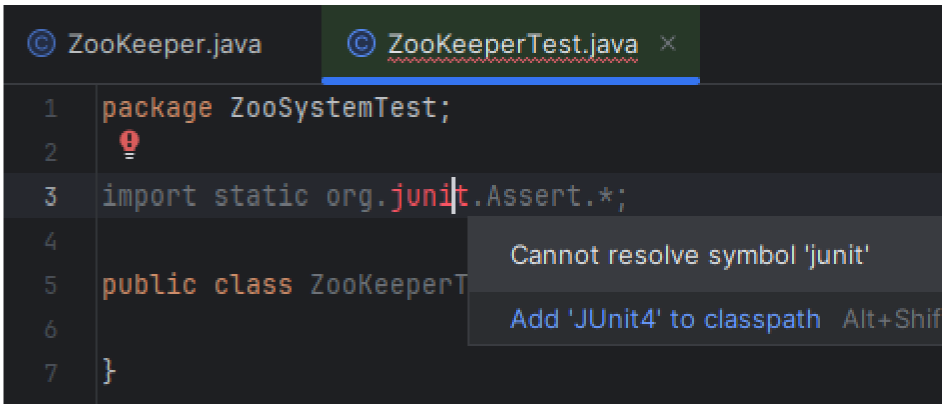
c) To understand the usages for @before and @after annotations, refer to this – https://www.guru99.com/junit-test-framework.html

* In this lab, using JUnit 4 is strongly recommended, as some of the @Test features are specific to this version. JUnit 5 may need some modifications beyond the scope of this manual, but you can still use it with your coursework, if preferred.



Testing library: JUnit4
JUnit4 library not found in the module
Class name: ZooKeeperTest
Superclass:
Destination package: ZooSystemTest
Generate: setUp/@Before
tearDown/@After
Generate test methods for: Show inherited methods
calculateChristmasBonus():int

d) Once clicked OK, a ZooKeeperTest.java file is created in the **test** directory. In the generated test class, if the junit library is not properly imported, click on the red-colored text, then “Add ‘Junit4’ to classpath”.

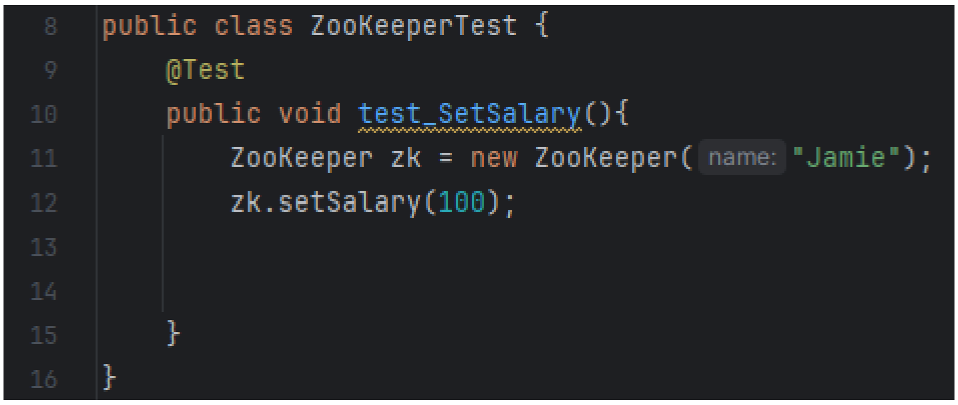


```
package ZooSystemTest;

import static org.junit.Assert.*;

public class ZooKeeperT

}
```

Cannot resolve symbol 'junit'
Add 'JUnit4' to classpath Alt+Shif

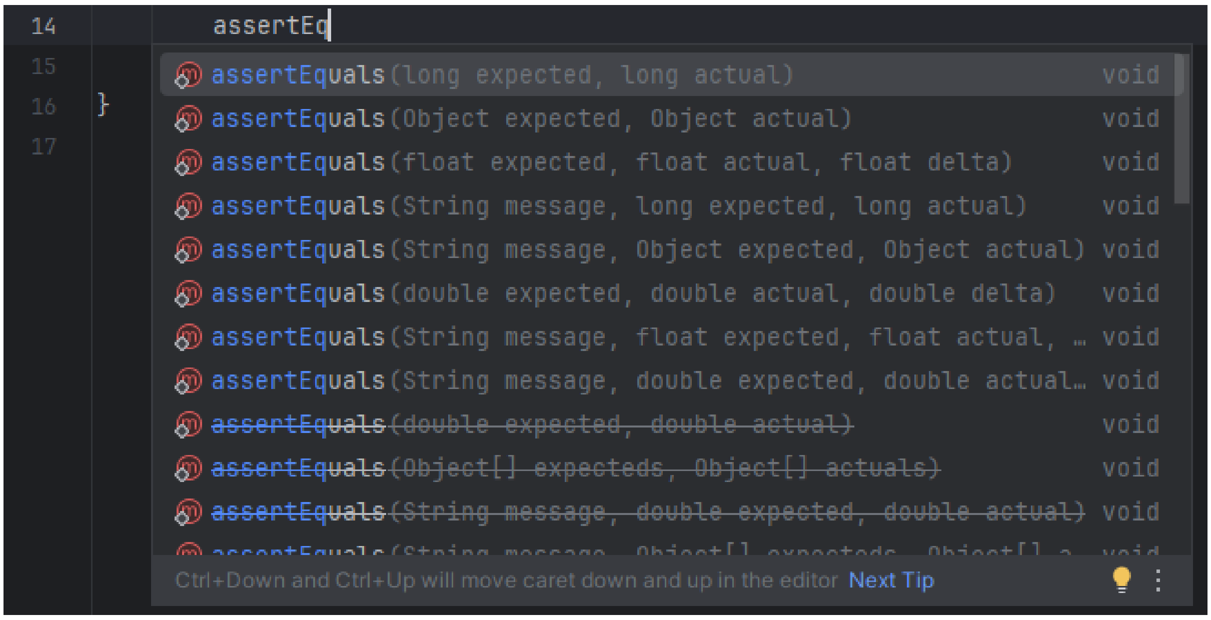
3. The ZooKeeper class inherits a “*setSalary*” method. Let’s now test it. Add the following lines into the ZooKeeperTest class, to initialize a Zookeeper object and execute its

"*setSalary*" once. If any classes are not recognizable, follow system prompt to import the classes.



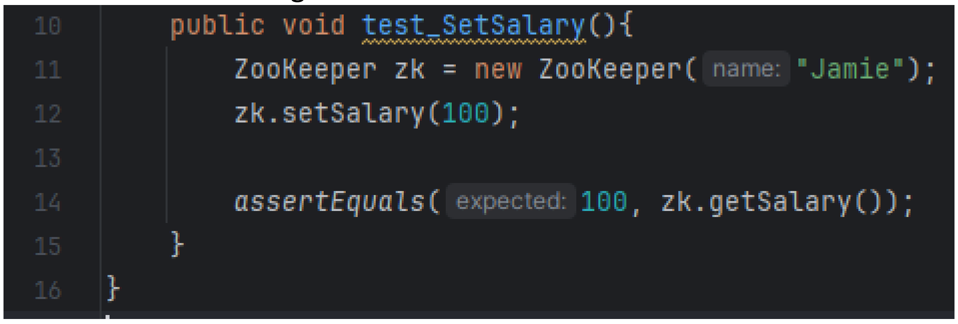
```
public class ZooKeeperTest {
    @Test
    public void test_SetSalary(){
        ZooKeeper zk = new ZooKeeper( name: "Jamie");
        zk.setSalary(100);


    }
}
```

4. Now, type *"assertEquals"* in line 14, then IntelliJ will pop-up many choices for you to select your intended test methods. Here we would like to test whether the *setSalary* (also *getSalary*) method really works, so we compare 100 with the *getSalary()* value. Choose the *long* format listed in the top, which is a general form of *int* format.



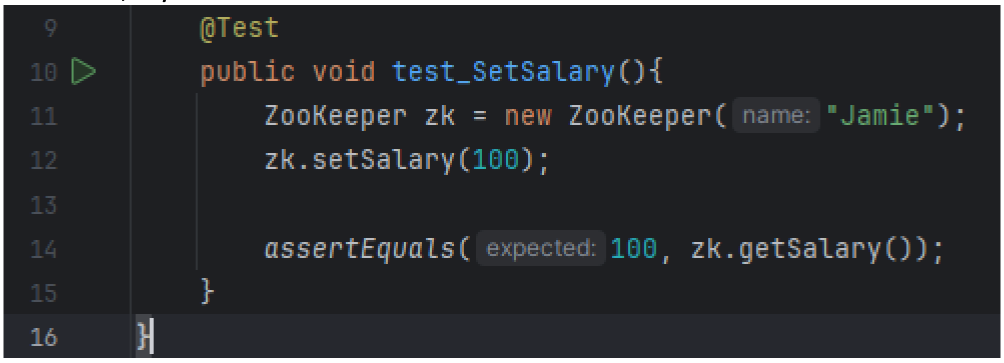
```
        assertEq
    }
assertEquals(long expected, long actual)                        void
assertEquals(Object expected, Object actual)                    void
assertEquals(float expected, float actual, float delta)         void
assertEquals(String message, long expected, long actual)        void
assertEquals(String message, Object expected, Object actual)    void
assertEquals(double expected, double actual, double delta)      void
assertEquals(String message, float expected, float actual, …    void
assertEquals(String message, double expected, double actual…    void
~~assertEquals(double expected, double actual)~~                    void
~~assertEquals(Object[] expecteds, Object[] actuals)~~              void
~~assertEquals(String message, double expected, double actual)~~    void
Ctrl+Down and Ctrl+Up will move caret down and up in the editor Next Tip
```

5. Insert the following line.



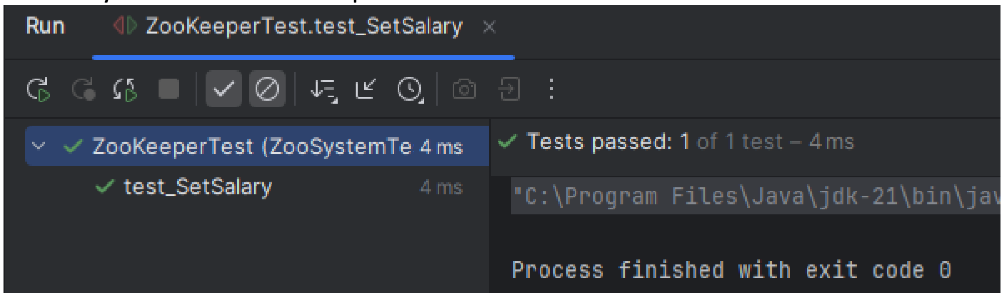
```
    public void test_SetSalary(){
        ZooKeeper zk = new ZooKeeper( name: "Jamie");
        zk.setSalary(100);

        assertEquals( expected: 100, zk.getSalary());
    }
}
```

6. Check if there is a green-colored right triangle ▶ alongside the method name, as below. If not, try restart the IntelliJ.



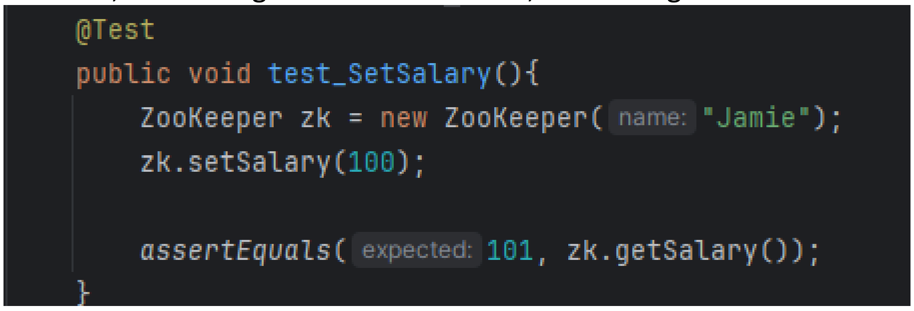
```
    @Test
    public void test_SetSalary(){
        ZooKeeper zk = new ZooKeeper( name: "Jamie");
        zk.setSalary(100);

        assertEquals( expected: 100, zk.getSalary());
    }
}
```

7. Click the green button to build and run this test case. If everything is fine, IntelliJ should tell you that the test is passed.



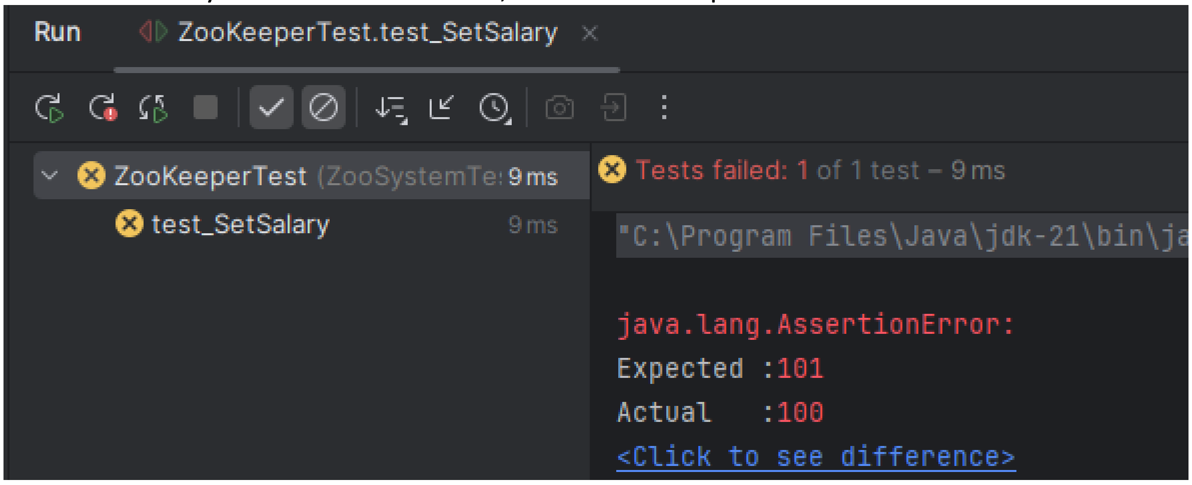
```
Run   ZooKeeperTest.test_SetSalary ×

✓ ZooKeeperTest (ZooSystemTe 4 ms    ✓ Tests passed: 1 of 1 test – 4 ms
   ✓ test_SetSalary          4 ms    "C:\Program Files\Java\jdk-21\bin\jav

                                     Process finished with exit code 0
```

8. Now, let's change the line like below, and run again.



```
    @Test
    public void test_SetSalary(){
        ZooKeeper zk = new ZooKeeper( name: "Jamie");
        zk.setSalary(100);

        assertEquals( expected: 101, zk.getSalary());
    }
```

9. IntelliJ tells you that the test failed, and lists the expected value and actual value.



```
Run   ZooKeeperTest.test_SetSalary ×

⊗ ZooKeeperTest (ZooSystemTe: 9 ms    ⊗ Tests failed: 1 of 1 test – 9 ms
   ⊗ test_SetSalary          9 ms    "C:\Program Files\Java\jdk-21\bin\ja

                                     java.lang.AssertionError:
                                     Expected :101
                                     Actual   :100
                                     <Click to see difference>
```

===
Task

- Think about boundary conditions when you are a designer of the *setSalary* method. E.g., what input values should be accepted (Negative salary? Huge salary?)
- At the end of the Week 04 Lab, you should have `ZooKeeper` and `Admin` classes, which have their own ways of calculating a Christmas bonus:
    - For a `ZooKeeper`, the bonus is *(salary $\times$ 0.05 + 100)*.
    - For an `Admin`, the bonus is *(salary $\times$ 0.08 + 90)*.
- Verify that the bonus calculation is working as expected.

===

2. Testing the Compound Class

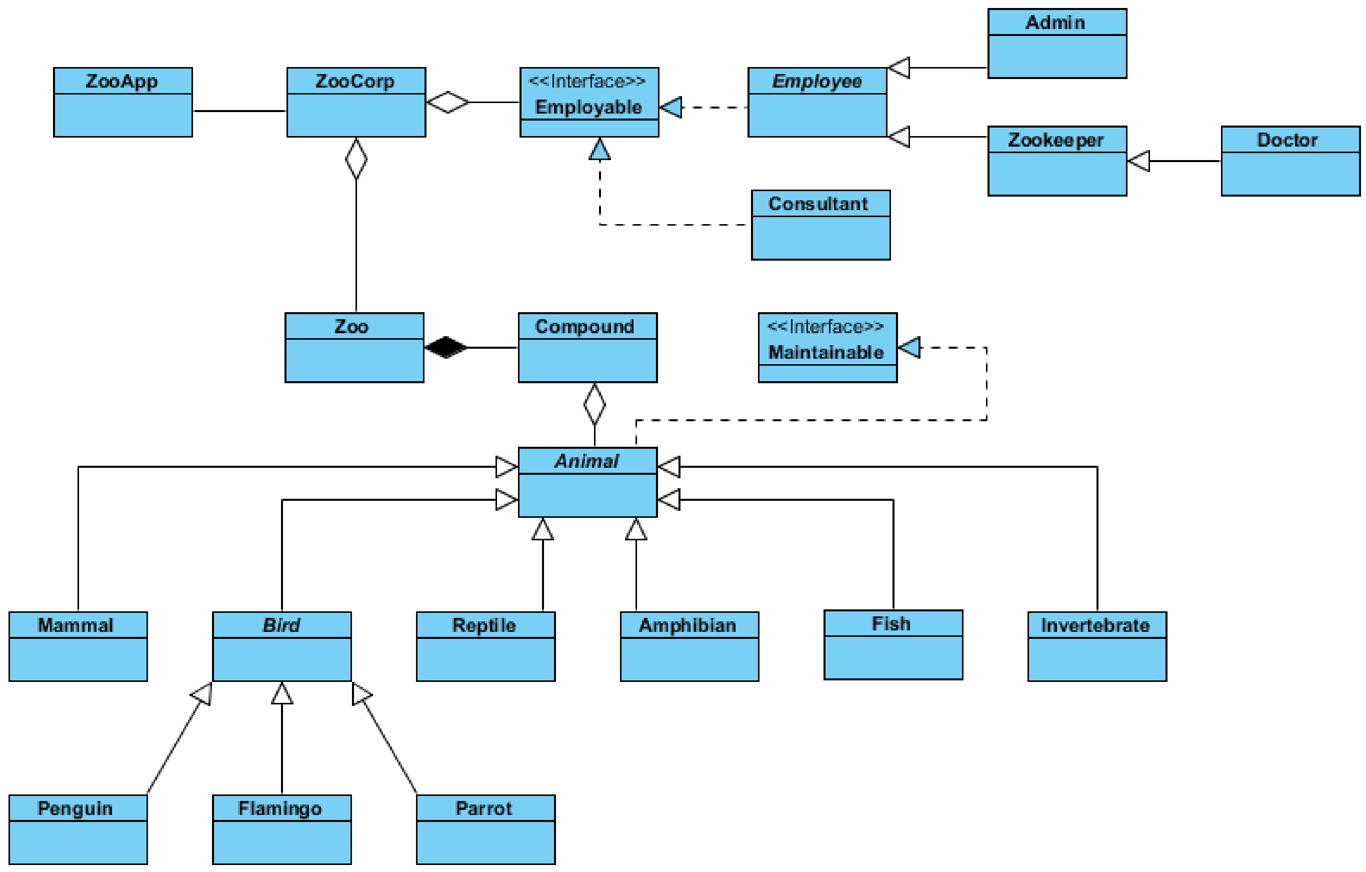
===
Task
Go to *Compound.java*, and check that a variable named **animals** is defined inside. Let's suppose we want to make sure that our **animal** ArrayList is empty when we first create a **compound**.

- Write a test to do the check. You may have to add a method to **Compound** to return the '`animals`' ArrayList. You may find `assertTrue` to be useful in this test.

===

Now, let's modify the code with some animal classes.

- Refactor Animal to be **abstract**, if it is not already (it doesn't make sense to create an instance of a generic Animal as we always want to create a specific type (bat, lion, monkey etc.)).
- Create some bird classes which inherit from the abstract class **Bird** which is inherited from the class **Animal** as shown in the diagram below.



- Add some appropriate fields and methods to the classes just added.
  - E.g., the abstract class Bird may have a lengthOfBeak field and setters/getters.
    - In the abstract class Bird – all birds have beaks.
  - E.g., the class Parrot may have speech method, for phrases it can say.
    - In the class Parrot – only Parrots can talk.

Note: Remember that you can automatically create getters and setters from a private field by right clicking the class name, then choose "Show Context Actions".

Now you should have some specific animal classes that you can create as objects, and store in a **Compound**.

Another sensible test would be that when we add animals, they will in fact be stored.
===
Task

- Write a test to check that when you add two **Parrots**, the class `Compound` stores *two* objects in its **animal** ArrayList.

===

What are other classes of behavior that need to be tested? One may be that we can store A LOT of animals in a compound.
===
Task

- Write a test to automatically store 1,000,000 instances of Parrots in a compound and make sure this amount is successfully stored.

===

This number is chosen on purpose. Note that this test takes more than 1 second to run. This is really too slow for a unit test. If you have 100 or 1000 tests to run (easily the case with a big project), you need to keep them as fast as possible. If you have to test it, it is fine, but always make your test run as quickly as possible.

- This gives a chance to demonstrate another feature as well. Add a line below the `@Test` tag as:

```
@Test(timeout = 100)
```

Now run the test. You should see it fails because it takes too long to run (more than 100ms).

Moreover, You can test for exceptions being thrown in the test method, such as an IndexOutofBoundsException.

```
@Test(expected=IndexOutOfBoundsException.class)
Public void methodThatMayThrowIOBException(){…}
```

The test will only pass if the an IOB exception is thrown in the below method that @Test annotates.

===

Task

- Write a test method (name it **testIOBException**), to automatically store 10 instances of Parrots in a compound. Then retrieve Parrot#20. Check whether the test is a pass. How about retrieving Parror#5? Will the test pass? Why?

===

For more test features of Junit 5, visit https://junit.org/junit5/docs/current/user-guide/#overview . Junit 4 is unfortunately missing in junit.org

3. Why Regression Testing Matters?

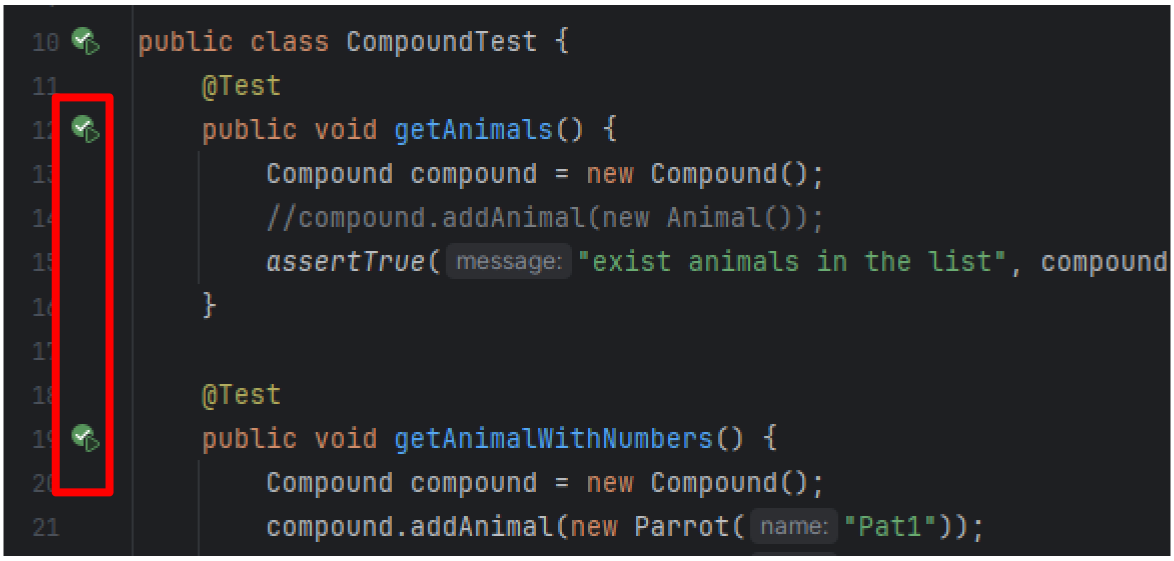
Regression test is a good practice to regularly or on-demand test the functionalities, after the modifications of the source code is done. The modification may be due to a bug being located and fixed, or a new updated requirement of the system being implemented. At this time, changes made to one part of the system may bring bugs to other parts of the system. Let's do the following practice in the context of IntelliJ.

- You've been asked to alter the Zoo system so that when you set the salary of the employee, the salary is reduced by an amount, representing a pension contribution.

- So, you might think about modifying the `Employee.setSalary` method:
  - Alter this method to set the salary as only 90% of the input parameter value.

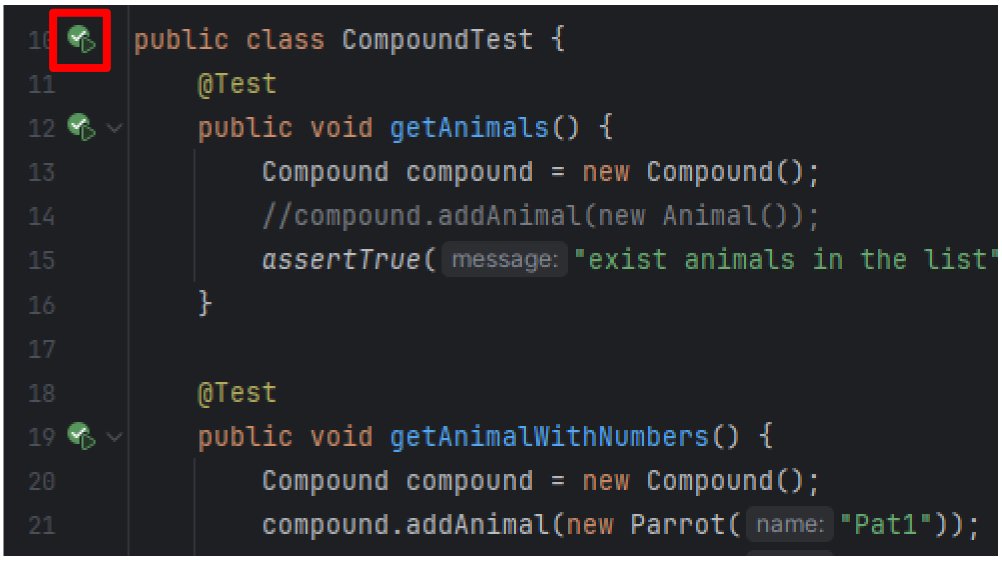
- Re-run the tests – now we choose to re-run all the tests that we write in the previous section of this lab. To check whether this modification has created any system inconsistency, we now run all the test cases developed above. Note that due to efficiency reasons, there could be various regression testing strategies, such as test case prioritization, here we choose to run all the tests.

- In IntelliJ, you have several options to select which test case to run.
  - Run only the selected test case: Click the green "Run" triangle aligning to the specific method/test.



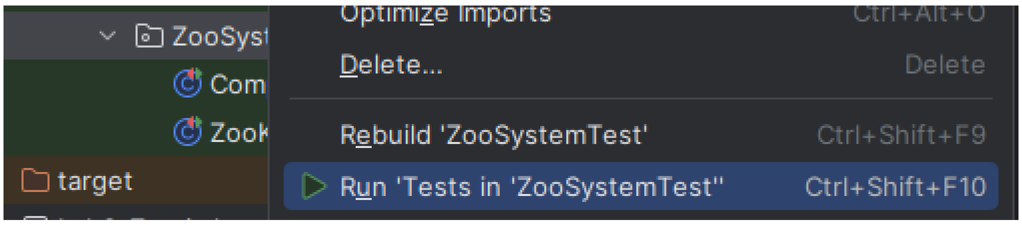
```java
public class CompoundTest {
    @Test
    public void getAnimals() {
        Compound compound = new Compound();
        //compound.addAnimal(new Animal());
        assertTrue( message: "exist animals in the list", compound
    }

    @Test
    public void getAnimalWithNumbers() {
        Compound compound = new Compound();
        compound.addAnimal(new Parrot( name: "Pat1"));
```

  - Run the all test cases for a class: Click the green "Run" triangle aligning to the class name for a certain test class. E.g., the **CompoundTest** class.



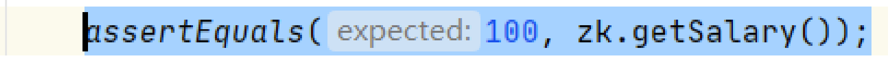
```java
public class CompoundTest {
    @Test
    public void getAnimals() {
        Compound compound = new Compound();
        //compound.addAnimal(new Animal());
        assertTrue( message: "exist animals in the list"
    }

    @Test
    public void getAnimalWithNumbers() {
        Compound compound = new Compound();
        compound.addAnimal(new Parrot( name: "Pat1"));
```

- Run all the tests defined inside a package: Right click the package name "ZooSystemTest", and find the "Run Tests in ZooSystemTest" option. This should run all the methods with annotation `@Test` inside this package.


ZooSyst
Com
Zook
target
Optimize Imports Ctrl+Alt+O
Delete... Delete
Rebuild 'ZooSystemTest' Ctrl+Shift+F9
Run 'Tests in 'ZooSystemTest'' Ctrl+Shift+F10

- After you modify the `setSalary` method, apply the last option to re-run all the tests. Then, what happens?

The test about the `getSalary` fails, after you alter the legacy code. It implies that you have altered something which may affect another component of the system. It suggests that perhaps you need a different way of handling this pension contribution.



```
assertEquals( expected: 100, zk.getSalary());
```

4. Adding More Tests

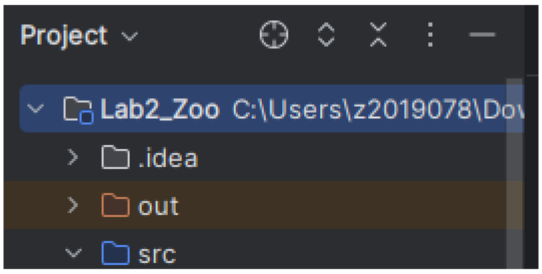
- Add more tests to your test suits. Think about what else it would make sense to test in this Zoo example. Use the resources at the end of this worksheet to find and try out different assert statements, and some optional tags you can use.

- You may have to add more functionality to the Zoo classes first – think about adding and testing the following:
  - Verify that employee names starting with a capital letter when retrieved using the getter.
  - Verify a default Zoo has the correct number of Compounds (Total 30 had been added into the constructor before).
  - Add and test some code to store ticket prices for each Zoo.
  - Implement the class Doctor. Add an `emergencyPhoneNumber` field and appropriate getters and setters.
    - The setter for the number should validate the phone number. Write some unit tests to ensure the validation takes place.
    - The validation rules are:
      - Number should start with a 0.
      - Number should be 11 digits long.

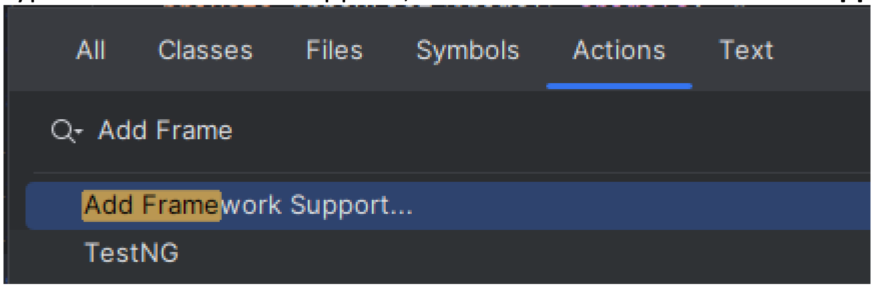
5. Build with Maven in IntelliJ

In this section, let’s look at using Maven to build the project, and apply the Junit test from Maven.

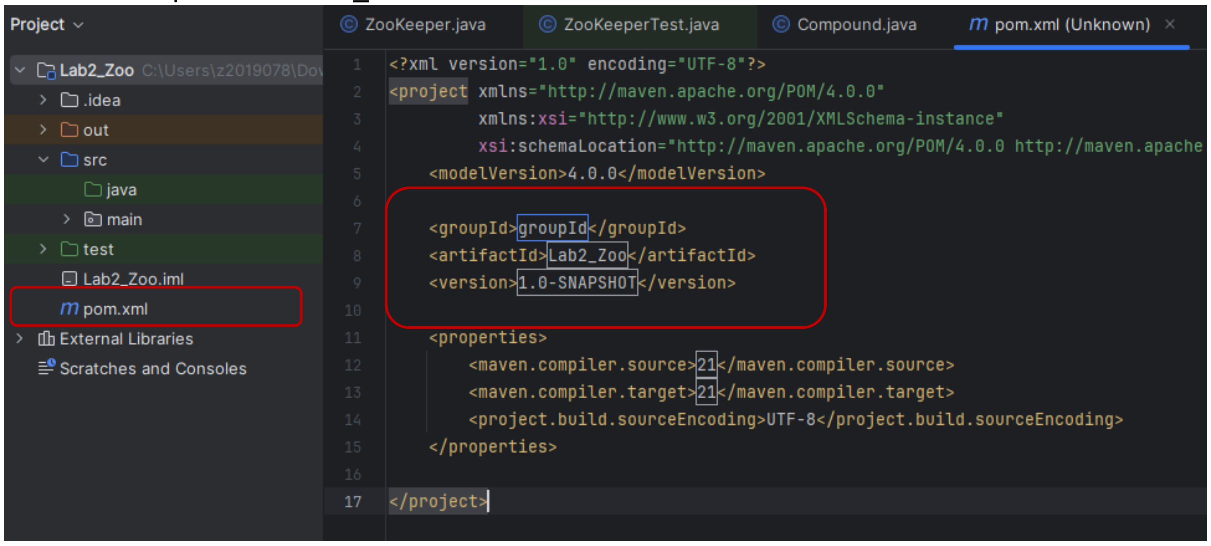
Continuing form the project, let’s add Maven support to our Zoo project. Click on the Project name as below.



Double press the ‘Shift’ key (or press Shift + Alt + A) in your keyboard. In the pop-up window, type ‘Add Framework Support’, then choose **Add Framework Support…**



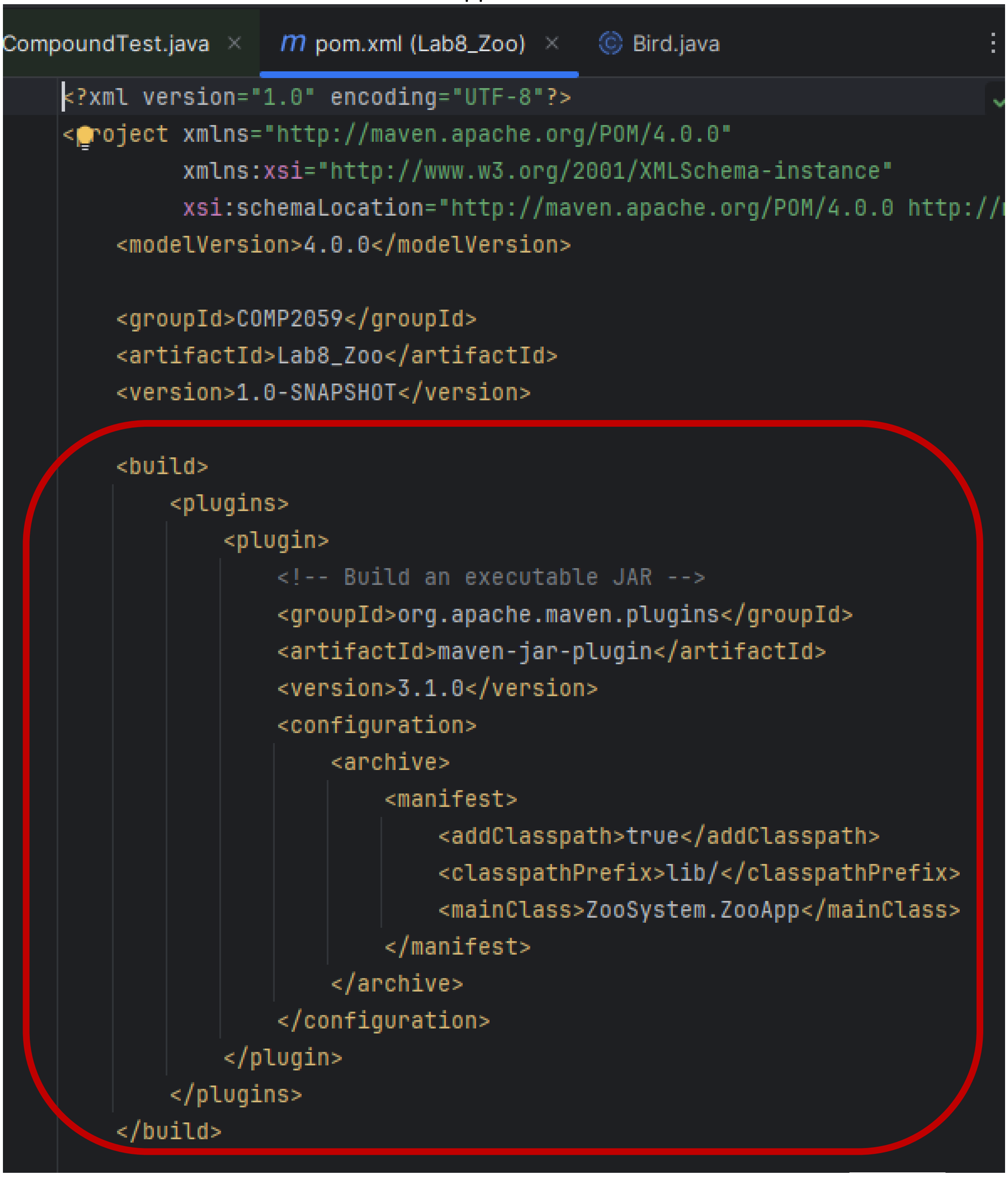
Then select **Maven** from the options, then click **OK**. You can find that a default *pom.xml* appears, which contains the standard Maven formalities. Specify your own groupID, such as COMP2059. *Close and re-open the project to make the Maven effective.* Then, you should see the below window. Note, you can refactor the project name as Lab8_Zoo now, to avoid conflict with previous Lab2_Zoo.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache
    <modelVersion>4.0.0</modelVersion>

    <groupId>groupId</groupId>
    <artifactId>Lab2_Zoo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

[Hint: In case you would like to create the Maven project from scratch, please follow the link here.]

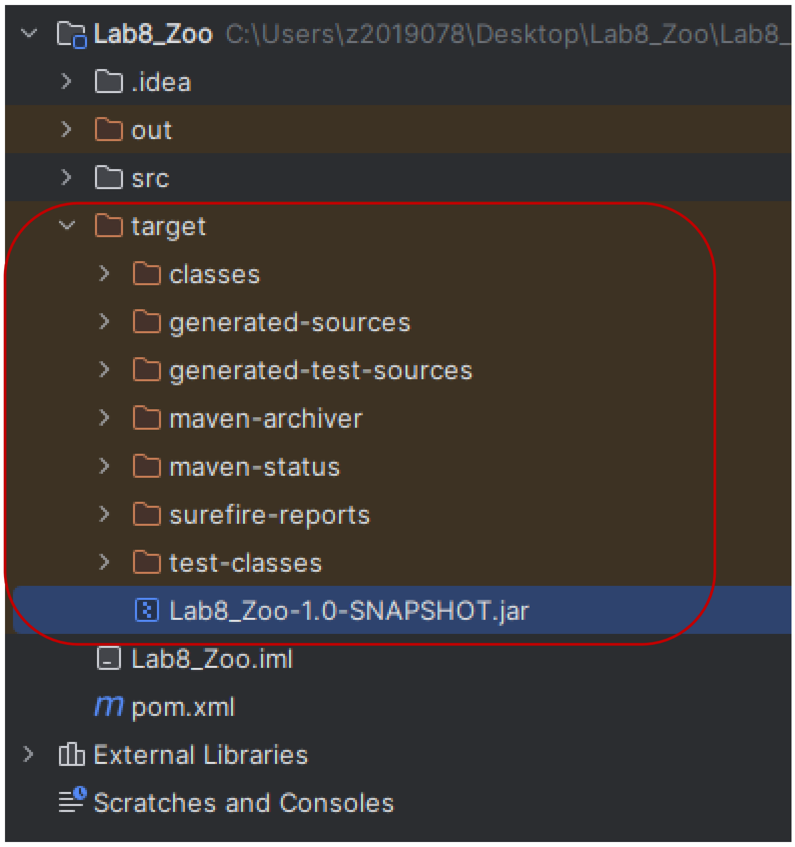
In the pom.xml, add the following lines enclosed in red rectangle. This is needed to build a jar file that can be executed. Note that “ZooApp” is where the main method can be found.



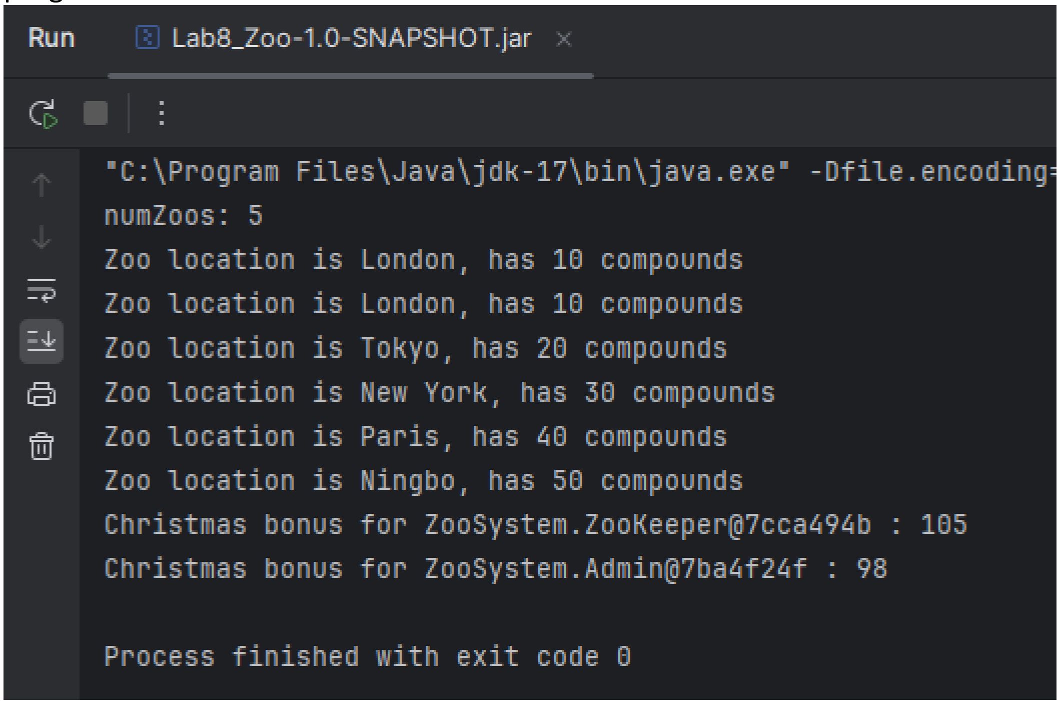
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
    <modelVersion>4.0.0</modelVersion>

    <groupId>COMP2059</groupId>
    <artifactId>Lab8_Zoo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <build>
        <plugins>
            <plugin>
                <!-- Build an executable JAR -->
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.1.0</version>
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <classpathPrefix>lib/</classpathPrefix>
                            <mainClass>ZooSystem.ZooApp</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

Note that whenever you change the content of the pom.xml, the refresh sign should appear. Click it to let Maven automatically configure what you update.

Open the Maven tab on the right, and double click “install” in the lifecycle. This should build the project, and you will see a target folder appears.

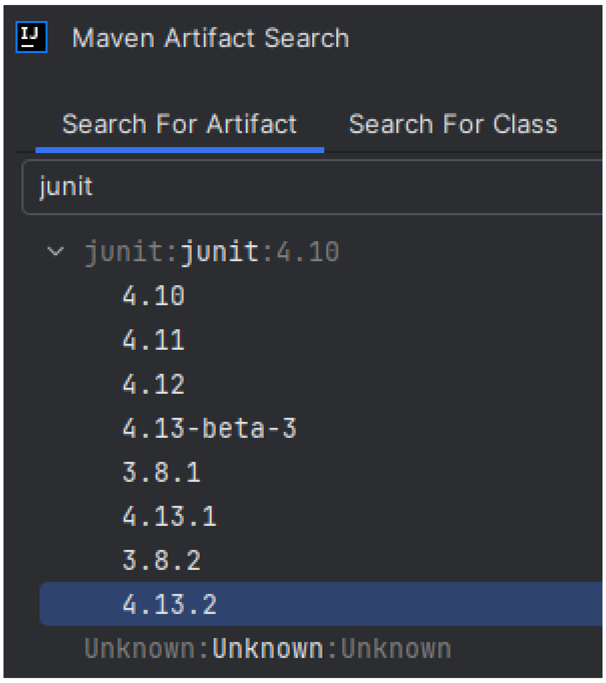


Right click the .jar file, and click Run 'Lab8_Zoo-1.0-SNAPSHOT.jar'. You should see the program run.



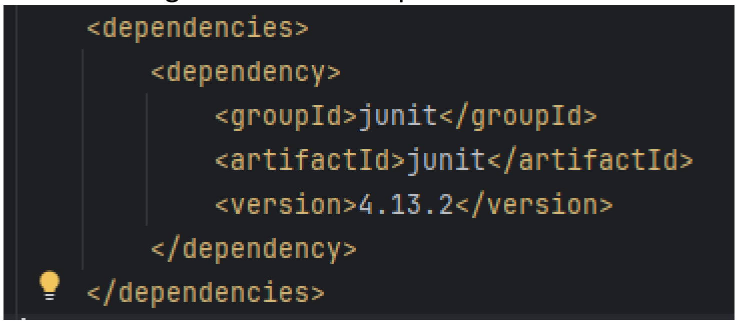
```
"C:\Program Files\Java\jdk-17\bin\java.exe" -Dfile.encoding=
numZoos: 5
Zoo location is London, has 10 compounds
Zoo location is London, has 10 compounds
Zoo location is Tokyo, has 20 compounds
Zoo location is New York, has 30 compounds
Zoo location is Paris, has 40 compounds
Zoo location is Ningbo, has 50 compounds
Christmas bonus for ZooSystem.ZooKeeper@7cca494b : 105
Christmas bonus for ZooSystem.Admin@7ba4f24f : 98

Process finished with exit code 0
```

Now let's use Maven to run the Junit test. In the pom.xml editing page, right click an empty place, choose **Generate…**, then dependency. A search dialog pops up, then search for 'junit', then choose 4.13.2



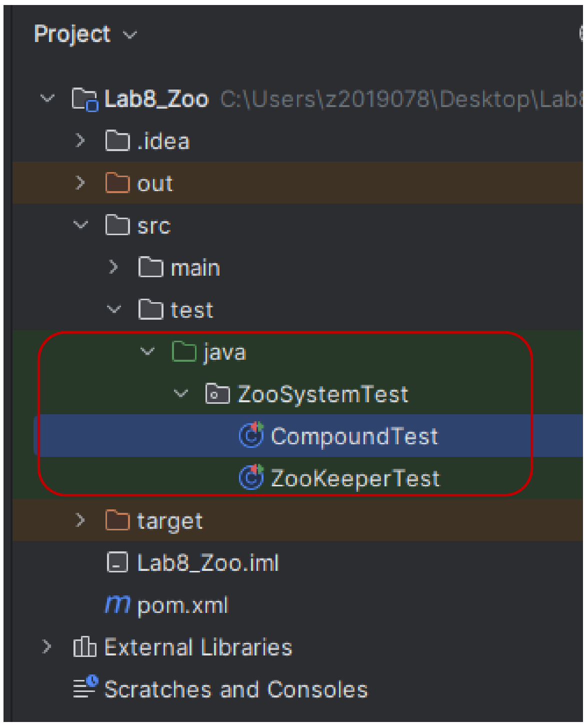
```
junit
junit:junit:4.10
    4.10
    4.11
    4.12
    4.13-beta-3
    3.8.1
    4.13.1
    3.8.2
    4.13.2
Unknown:Unknown:Unknown
```

The following will show in the pom.xml.

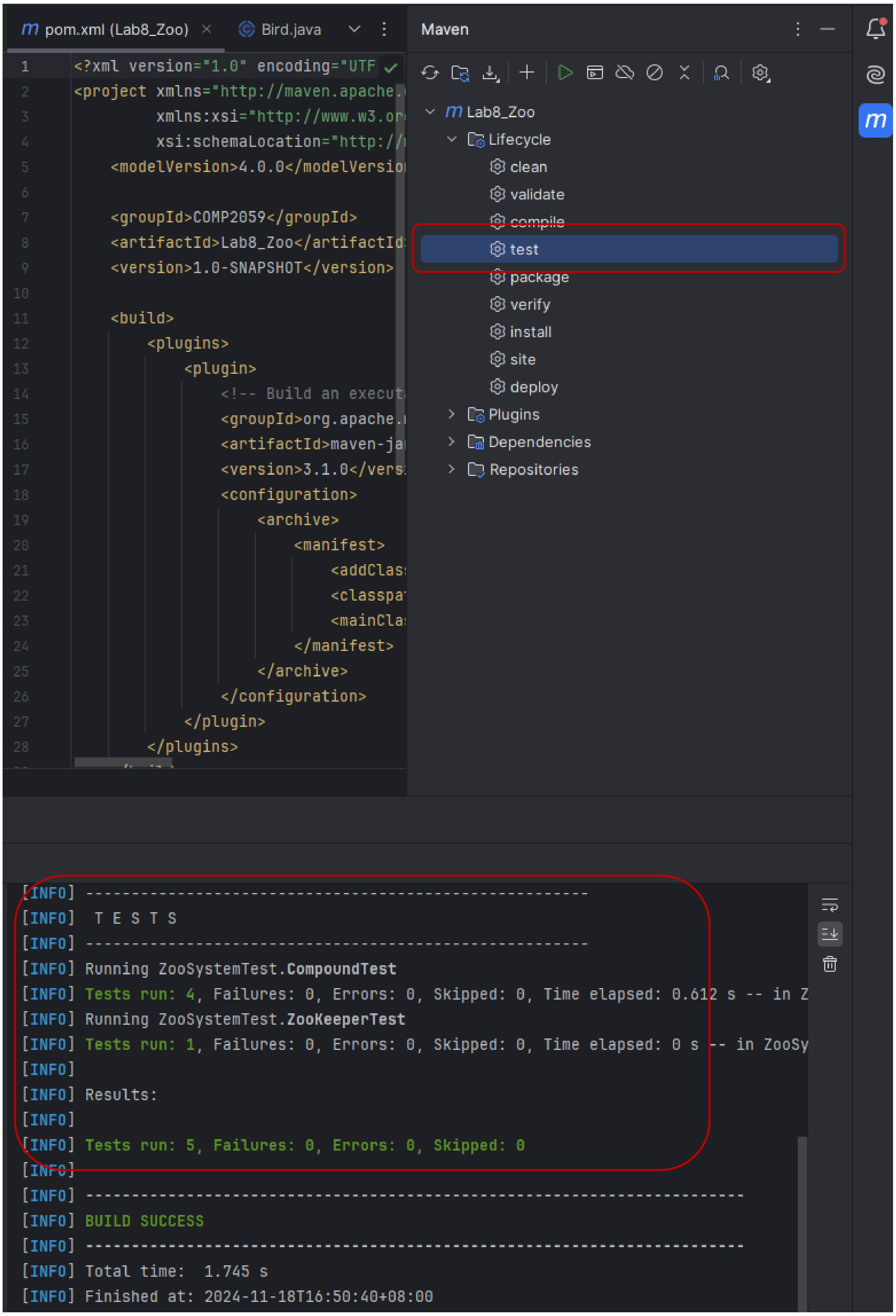


```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
    </dependency>
</dependencies>
```

Also, remember to move the CompoundTest.java and ZooKeeperTest.java, and their package ZooSystemTest, together into the test/java path.



Use the “test” method in the Maven for testing. The results will be shown below.



```
<?xml version="1.0" encoding="UTF
<project xmlns="http://maven.apache.
         xmlns:xsi="http://www.w3.or
         xsi:schemaLocation="http://
    <modelVersion>4.0.0</modelVersio

    <groupId>COMP2059</groupId>
    <artifactId>Lab8_Zoo</artifactId
    <version>1.0-SNAPSHOT</version>

    <build>
        <plugins>
            <plugin>
                <!-- Build an execut
                <groupId>org.apache.
                <artifactId>maven-ja
                <version>3.1.0</vers
                <configuration>
                    <archive>
                        <manifest>
                            <addClas
                            <classpa
                            <mainCla
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running ZooSystemTest.CompoundTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.612 s -- in Z
[INFO] Running ZooSystemTest.ZooKeeperTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s -- in ZooSy
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.745 s
[INFO] Finished at: 2024-11-18T16:50:40+08:00
```

Congratulations! You've done the initial practice on Junit and Maven. We have not covered Gradle in this manual, but you can find their resources below.

6. Resources

Many good Unit Testing tutorials can be found online, e.g.,
Friendly Testing Framework for Java – https://junit.org/junit5/docs/current/user-guide/.

Gradle with IntelliJ
https://www.jetbrains.com/help/idea/gradle.html

TDD in IntelliJ
https://www.jetbrains.com/help/idea/tdd-with-intellij-idea.html

The documentation for JUnit, including all the assertions and statements.
Unit Testing with JUnit - https://www.vogella.com/tutorials/JUnit/article.html.

Good and thorough article on using testing in Eclipse.
https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2FgettingStarted%2Fqs-junit.htm.