

Test-driven development

Creating the program in small steps

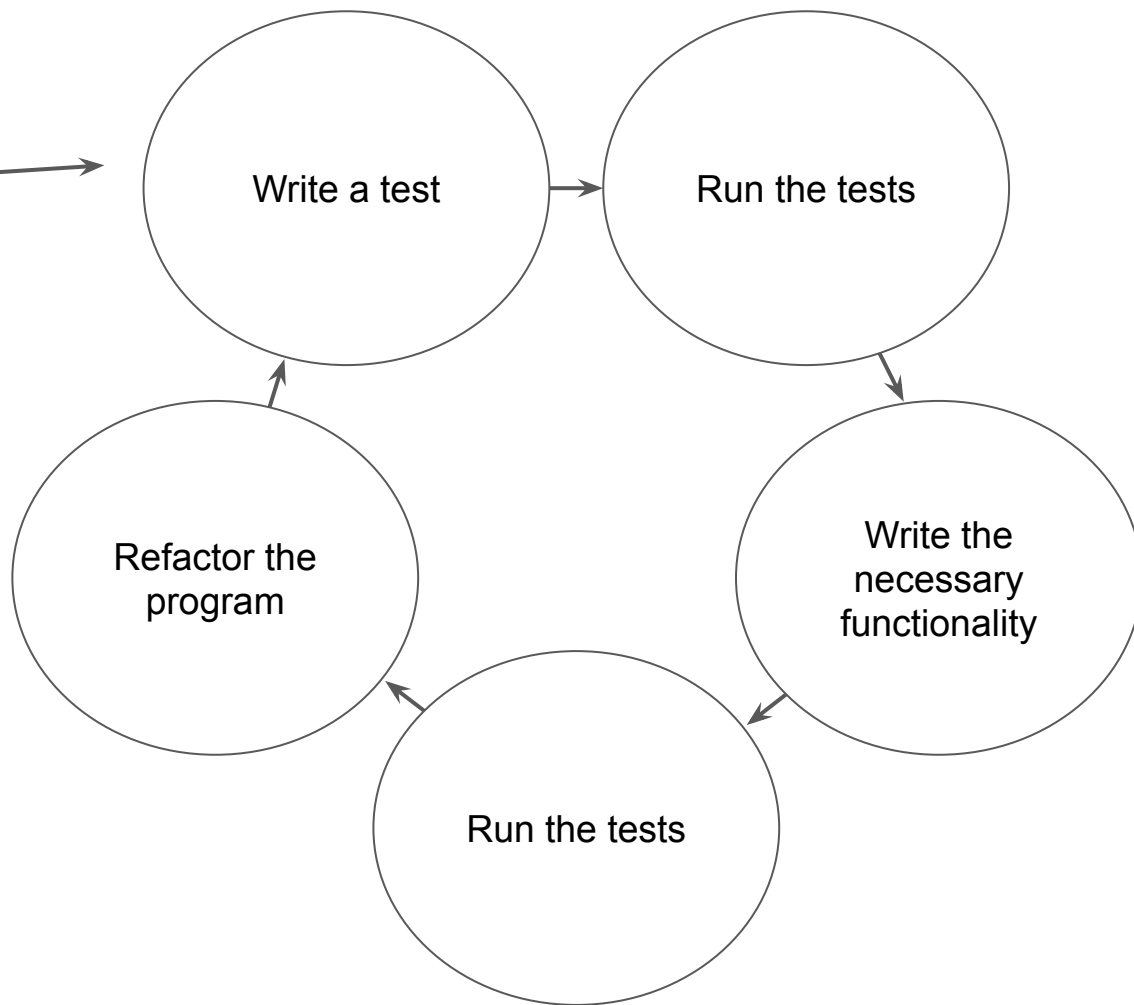
1. Create a test that tests some feature that will be added to the program.
2. Run the test. It should not pass.
 - If the test passes, move to step 1.
3. Develop the program so that it has the functionality required to pass the test.
4. Run the tests.
 - If the tests don't pass, move to step 3 and further develop the functionality.
5. Refactor
 - If the program is ready, stop..
 - Otherwise, go to step 1.

Creating the program in small steps

1. Create a test that tests some feature that will be added to the program.
2. Run the test. It should not pass.
 - If the test passes, move to step 1.
3. Develop the program so that it has the functionality required to pass the test.
4. Run the tests.
 - If the tests don't pass, move to step 3 and further develop the functionality.
5. Refactor
 - If the program is ready, stop..
 - Otherwise, go to step 1.

Refactoring means cleaning the code while maintaining the functionality of the program. Cleaning includes tasks such as improving the readability and dividing the program into smaller methods and classes.

start



Advantages

- Forces the programmer to think of the functionality before writing the code
- Results in maintainable structure, since the program is built in small parts, refactoring steadily.
- The end product contains tests, which makes further development easier: when the code is changed, it's easy to check if the existing functionality still works.
- Fewer bugs in production.

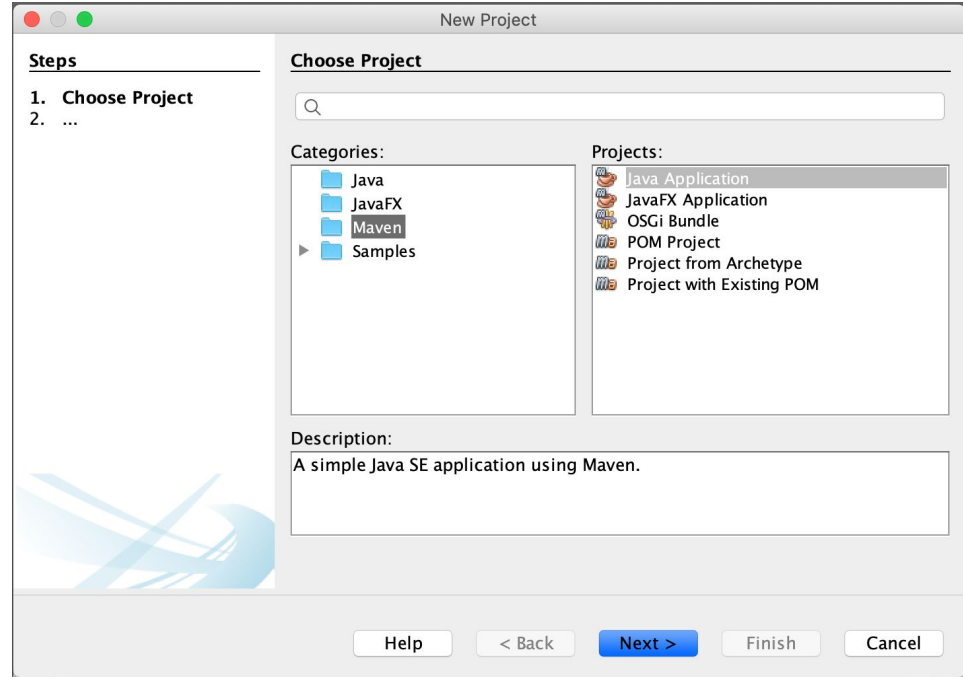
Example: exercise management

Part 1. Creating the project and including the JUnit library

Let's examine what test-driven development might look like for a program that is meant to manage exercises.

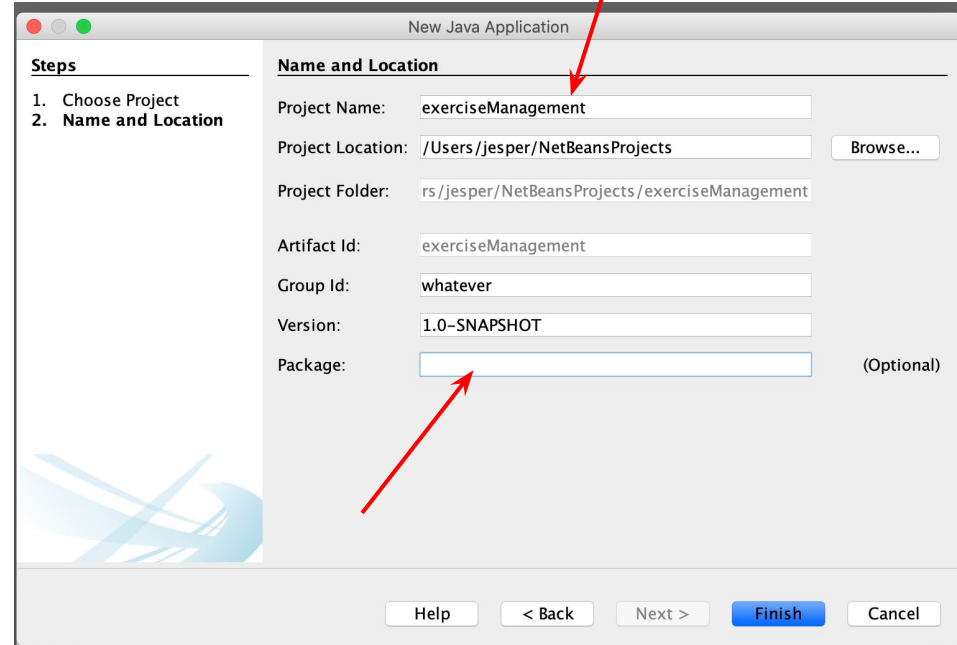
We want the exercise management software to include the possibilities to list, add, and remove exercises, as well as the ability mark an exercise as completed.

Let's start developing by creating an empty project. Do this by selecting in NetBeans File -> New Project. Choose the project category "Maven" and the project type as "Java Application".



Now we get to fill in the information for our new project. Set the project name as “exerciseManagement”. The project path tells where in the storage system the project files are located.

Keep the package field empty.



New Java Application

Steps

1. Choose Project
2. **Name and Location**

Name and Location

Project Name:

Project Location:

Project Folder:

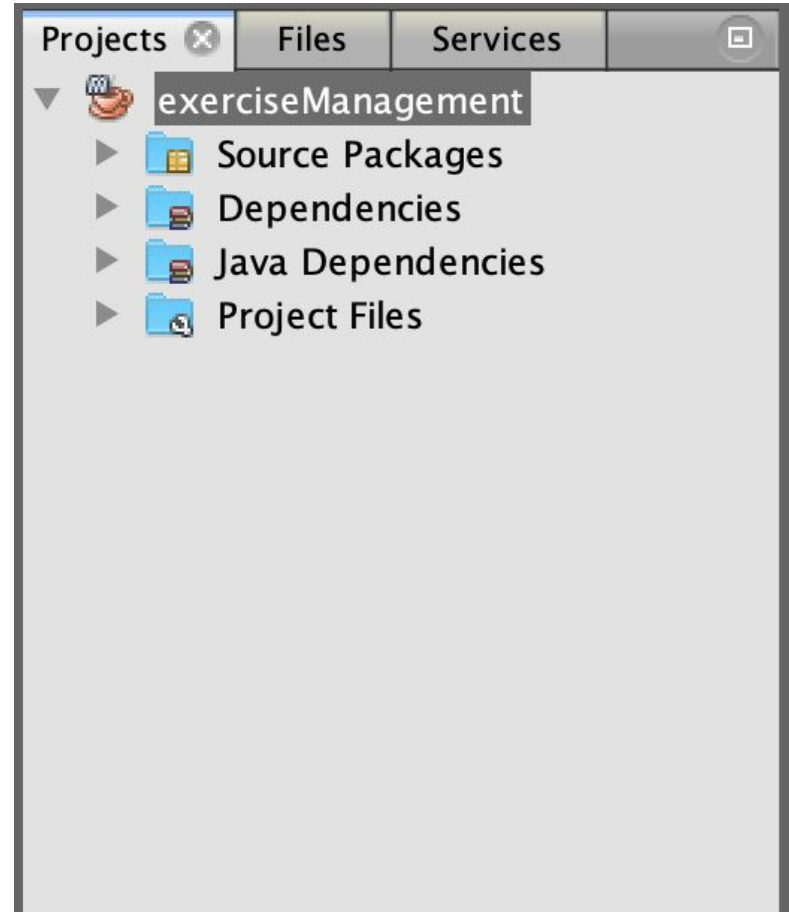
Artifact Id:

Group Id:

Version:

Package: (Optional)

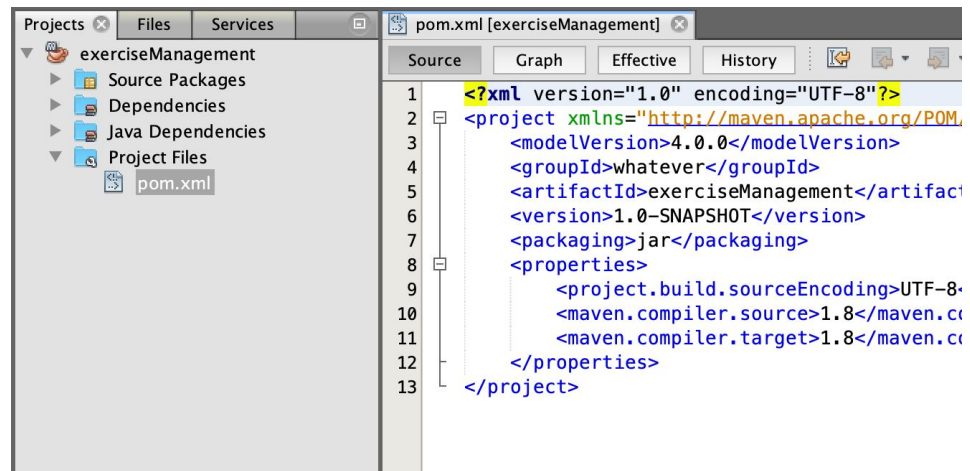
When we press Finish, the new project is created. You can view it on the left side of NetBeans.



When we press Finish, the new project is created. You can view it on the left side of NetBeans.

Next we'll add the JUnit library. It's meant for writing unit tests. (JUnit is included in the exercise templates downloaded from TMC.)

Click open the folder Project Files, and double click the 'pom.xml' file. The file details the structure of our project and the libraries it uses -- we'll take a closer look on the Advanced programming course.

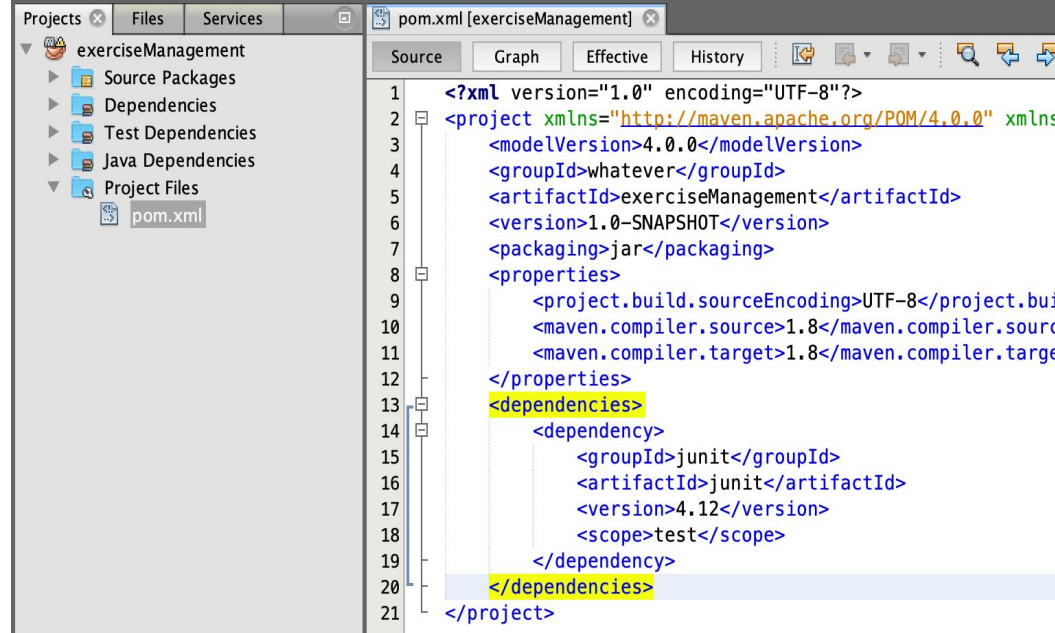


Copy the JUnit library before the line</project>:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Once the contents above are copied into the pom.xml file (before the line </project>), save the file.

This gives the project access to the JUnit library and enables us to write unit tests.

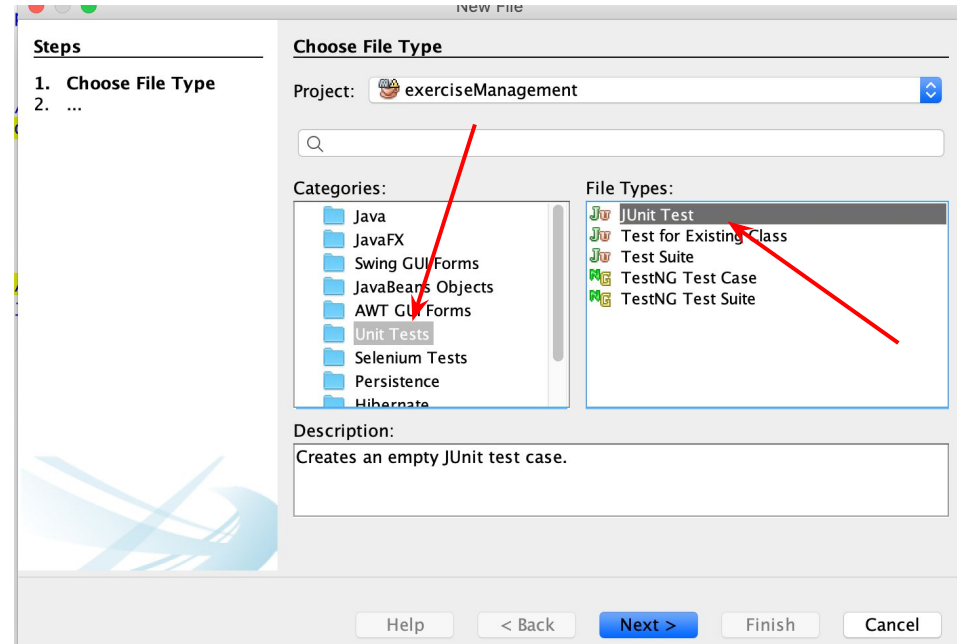


Part 2. Creating the class for unit tests

Let's create the first unit test. You can create unit tests by right-clicking the project and choosing new -> Other.

This opens a view for creating a new file. Choose the category as "Unit Tests" and the type of the created file as "JUnit Test".

Then press "Next".



You will then encounter an assistant for creating the unit test file..

Set the class name as 'ExerciseManagementTest' and choose not to generate code in the class.

NB! Ensure that the class name ends with "Test".

Press Finish when ready.

New JUnit Test

Steps

1. Choose File Type
2. **Name and Location**

Name and Location

Class Name:

Project:

Location:

Package:

Created File:

Generated Code

- ☒ Test Initializer
- ☐ Test Finalizer
- ☐ Test Class Initializer
- ☐ Test Class Finalizer

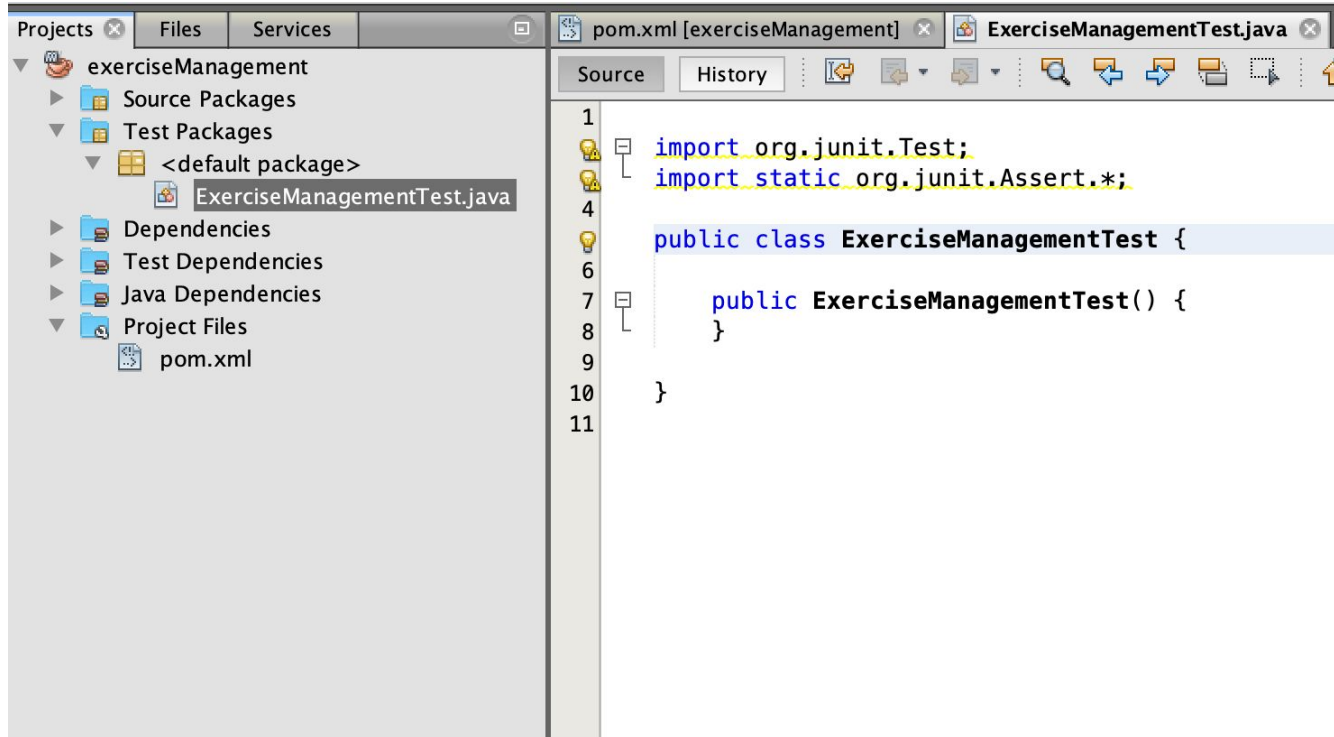
Generated Comments

- ☒ Source Code Hint

Warning: It is highly recommended that you do not place Java classes in the c

Buttons: Help, < Back, Next >, **Finish**, Cancel

Now the project folder “Test Packages” contains the class “ExerciseManagementTest”



Part 3. First unit test

Let's create the first test. The test uses a class called ExerciseManagement, and expects it to have a method called exerciseList that returns the list of exercises.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
```

```
public class ExerciseManagementTest {
```

The test method assertEquals receives two values as parameters -- the first is the expected value, and the second the value returned by the program.

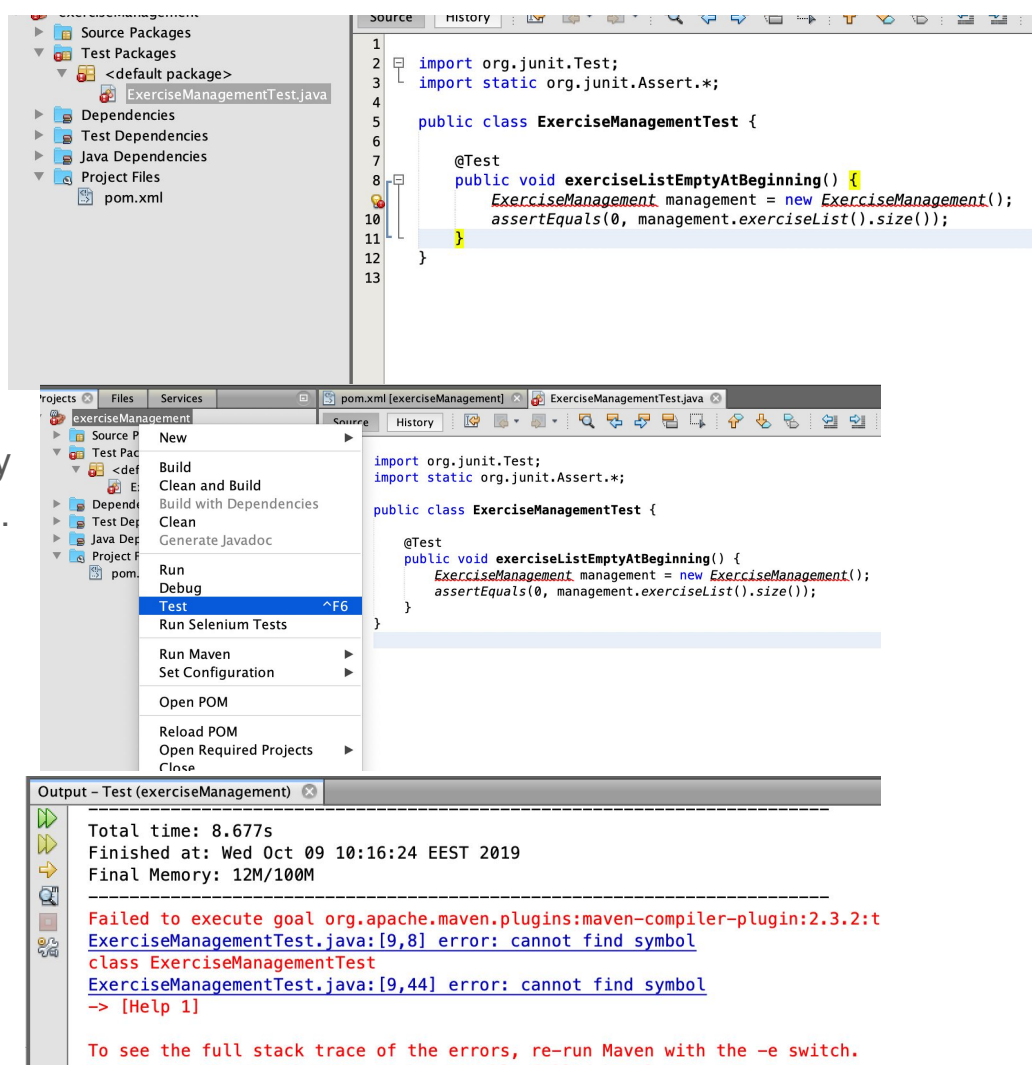
```
    @Test
    public void exerciseListEmptyAtBeginning() {
        ExerciseManagement management = new ExerciseManagement();
        assertEquals(0, management.exerciseList().size());
    }
}
```

Here the method is used for checking the size of a new exercise list: a new list should be empty.

The test is in the class
ExerciseManagementTest. Once it has been
created, it's clear that it won't pass: the class
that the test uses, ExerciseManagement, is
missing.

Nevertheless, let's run the tests. This is done by
right-clicking on the project and choosing "Test".

We notice the error "Failed to execute..."



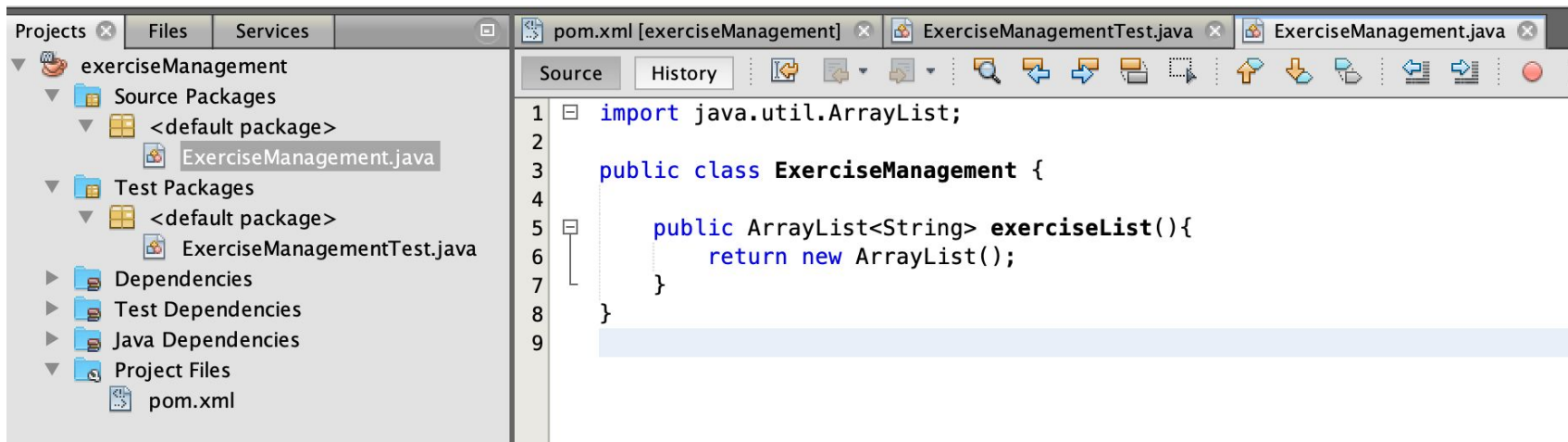
Part 4. Implementing the functionality that is required by the unit tests

Let's create the class `ExerciseManagement` (the class is added to the folder *Source Packages*). Then let's give it a method called `exerciseList` that returns a list.

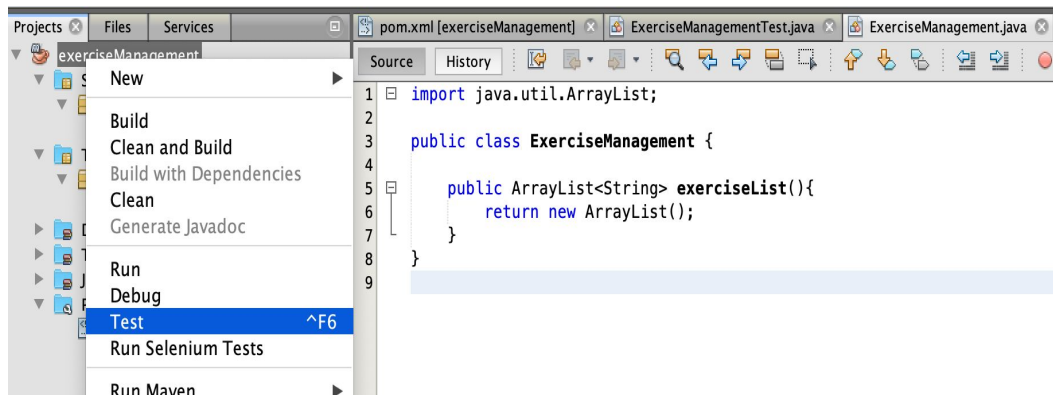
```
import java.util.ArrayList;

public class ExerciseManagement {

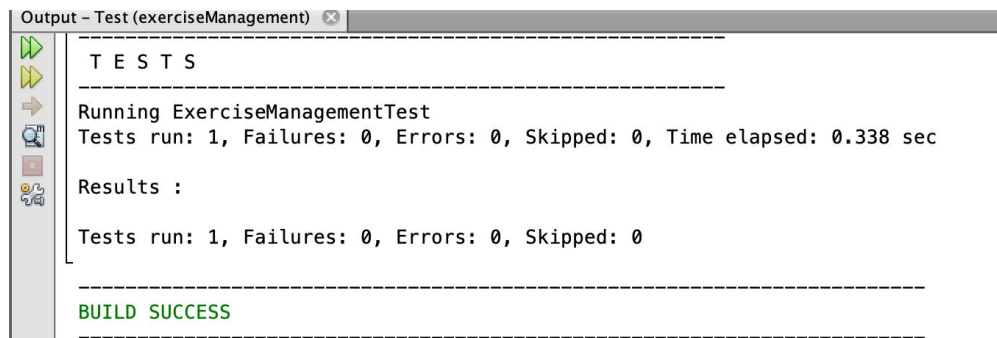
    public ArrayList<String> exerciseList() {
        return new ArrayList<>();
    }
}
```



Run the tests by right-clicking on the project and choosing “Test”.



The tests pass. There is no refactoring, so we'll continue and write the next test.



Part 5. Second unit test and implementing the related functionality

Next we'll create a new test. It examines the functionality that relates to adding new exercises.

The test uses the add method of the class ExerciseManagement, using it to add a new exercise to the exercise list. If the addition was successful, the size of the exercise list should have grown by one.

The test does function at all, since the ExerciseManagement class lacks a method called 'add'.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class ExerciseManagementTest {

    @Test
    public void exerciseListEmptyAtBeginning() {
        ExerciseManagement management = new ExerciseManagement();
        assertEquals(0, management.exerciseList().size());
    }

    @Test
    public void addingExerciseGrowsListByOne() {
        ExerciseManagement management = new ExerciseManagement();
        management.add("Write a test");
        assertEquals(1, management.exerciseList().size());
    }
}
```


We will create a method called 'add' to the ExerciseManagement class and rerun the tests. The method does nothing at the moment.

The first test we created passes, but the one we just defined does not.

The error message is "Failed: expected: <1> but was: <0>" so the test expected the value of 1 but received 0.

```
import java.util.ArrayList;

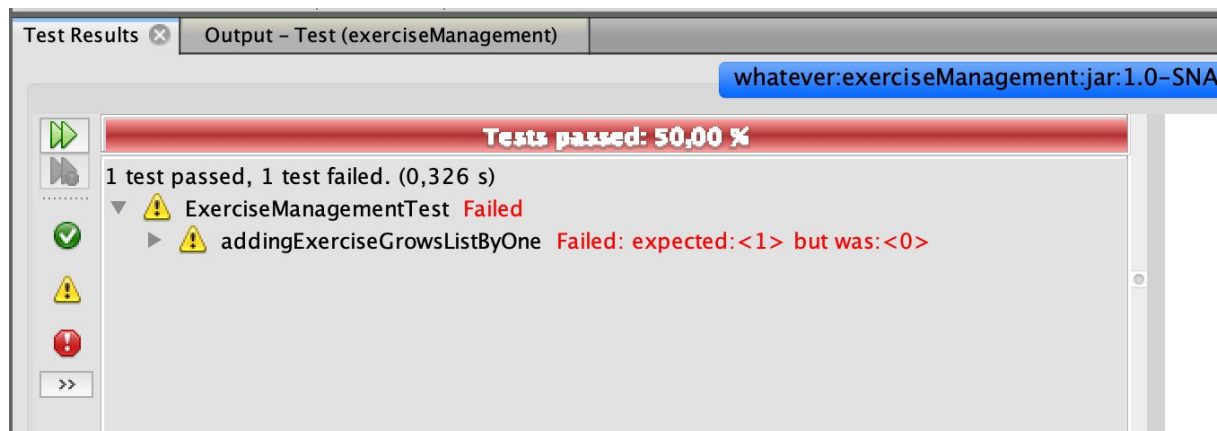
public class ExerciseManagement {

    public ArrayList<String> exerciseList() {
        return new ArrayList<>();
    }

    public void add(String exercise) {

    }

}
```



Let's modify the functionality of the ExerciseManagement class. We'll add an object variable: alist that contains the exercises. We'll only modify the add method so that it passes the test, but will not in fact do the thing we want it to.

Run the tests -- they should pass, and we'll move on to writing the next test.

```
import java.util.ArrayList;

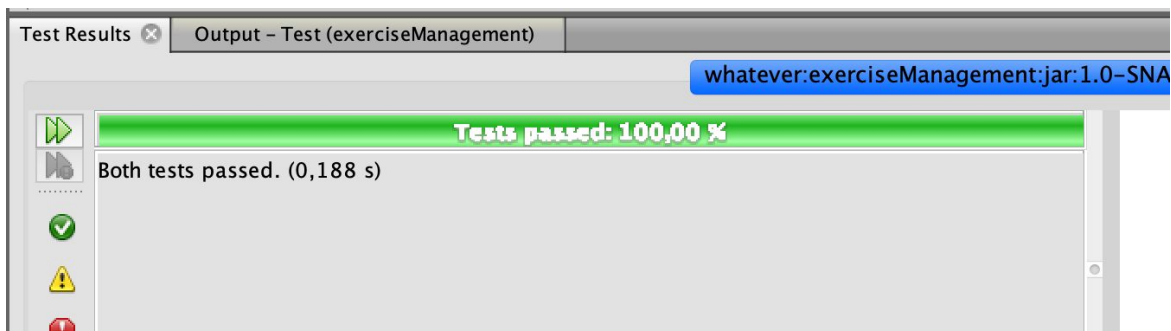
public class ExerciseManagement {

    private ArrayList<String> exercises;

    public ExerciseManagement() {
        this.exercises = new ArrayList<>();
    }

    public ArrayList<String> exerciseList() {
        return this.exercises;
    }

    public void add(String exercise) {
        this.exercises.add("New");
    }
}
```



Part 6. Third unit test and the related functionality

Let's enhance our tests: the added exercise should be included in the list of exercises.

The JUnit library offers a method called `assertTrue`, which demands that the parameter it receives is finally evaluates as true.

Once we have added the new test to the program, not all tests pass, again.

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import org.junit.Test;

public class TehtavienhallintaTest {

    @Test
    public void exerciseListEmptyAtBeginning() {
        ExerciseManagement management = new ExerciseManagement();
        assertEquals(0, management.exerciseList().size());
    }

    @Test
    public void addingExerciseGrowsListByOne() {
        ExerciseManagement management = new ExerciseManagement();
        management.add("Write a test");
        assertEquals(1, management.exerciseList().size());
    }

    @Test
    public void addedExerciseIsInList() {
        ExerciseManagement management = new ExerciseManagement();
        management.add("Write a test");
        assertTrue(management.exerciseList().contains("Write a test"));
    }
}
```

A crafty programmer modified the ExerciseManagement so that the 'add' method always adds the string 'Write a test' to the exercise list.

This would result in a situation where the tests would pass but the program would still not offer a working solution for adding exercises.

Let's rather modify the ExerciseManagement class to add any exercise given to the 'add' method to its list of exercises.

```
import java.util.ArrayList;

public class ExerciseManagement {

    private ArrayList<String> exercises;

    public ExerciseManagement() {
        this.exercises = new ArrayList<>();
    }

    public ArrayList<String> exerciseList() {
        return this.exercises;
    }

    public void add(String exercise) {
        this.exercises.add(exercise);
    }
}
```

We notice some repetition in the test class
-- a suitable moment for refactoring!

Let's make an ExerciseManagement
object variable and initialize it at the
beginning of each test.

Initializing a variable can be done by
adding to the test class a method
'initialize'.

We annotate the 'initialize' method with
@Before, which guides the program to
execute this method before each test..

Running the tests after refactoring shows
that they all still pass.

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import org.junit.Before;
import org.junit.Test;

public class ExerciseManagementTest {

    private ExerciseManagement management;

    @Before
    public void initialize() {
        management = new ExerciseManagement();
    }

    @Test
    public void exerciseListEmptyAtBeginning() {
        ExerciseManagement management = new ExerciseManagement();
        assertEquals(0, management.exerciseList().size());
    }

    @Test
    public void addingExerciseGrowsListByOne() {
        ExerciseManagement management = new ExerciseManagement();
        management.add("Write a test");
        assertEquals(1, management.exerciseList().size());
    }

    @Test
    public void addedExerciseInList() {
        ExerciseManagement management = new ExerciseManagement();
        management.add("Write a test");
        assertTrue(management.exerciseList().contains("Write a test"));
    }
}
```

Part 7. Fourth unit test and implementing the related functionality

Let's start adding the functionality to mark an exercise as completed.

Once a new test has been written, the situation is again that not all tests pass.

```
// ...  
@Test  
public void exerciseCanBeMarkedAsCompleted() {  
    management.add("New exercise");  
    management.markAsCompleted("New exercise");  
    assertTrue(manager.isCompleted("New exercise"));  
}  
// ...
```


The new functionality calls for the methods 'markAsCompleted' and 'isCompleted' in the ExerciseManagement class.

The tests don't check how the methods work in detail, so at first let's add the bare minimum of functionality.

The tests will pass after this.

```
// ...  
public void markAsCompleted(String exercise) {  
  
}  
  
public boolean isCompleted(String exercise) {  
    return true;  
}  
// ...
```

Part 8. Fifth unit test and implementing the related functionality

So far we have checked that the wanted functionality exists, but we have not really checked that unwanted behavior doesn't occur.

```
// ...  
@Test  
public void ifNotMarkedCompletedIsNotCompleted() {  
    management.add("New exercise");  
    management.markAsCompleted("New exercise");  
    assertFalse(management.isCompleted("Some exercise"));  
}  
// ...
```

If we focus on the existence of wanted functionality in writing tests, the tests might provide quite a narrow view into the functionality of the program.

Next, let's write a test to check that an exercise that has NOT been marked as completed is not completed.

The tests won't pass.

Again, let's implement enough functionality to pass the tests. We are going to have to make a fairly big change: we'll add a different list for exercises that have been marked as completed..

All the tests pass again after this.

However, some questions remain. Should the exercises returned by 'exerciseList' be marked as completed? Is it really possible to mark an exercise completed even if it has not been added to the list of exercises previously?

```
import java.util.ArrayList;

public class ExerciseManagement{

    private ArrayList<String> exercises;
    private ArrayList<String> completedExercises;

    public ExerciseManagement() {
        this.exercises = new ArrayList<>();
        this.completedExercises = new ArrayList<>();
    }

    public List<String> exerciseList() {
        return this.exercises;
    }

    public void add(String exercise) {
        this.exercises.add(exercise);
    }

    public void markAsCompleted(String exercise) {
        this.completedExercises.add(exercise);
    }

    public boolean isCompleted(String exercise) {
        return this.completedExercises.contains(exercise);
    }
}
```

We are going to do our first slightly bigger change in the inner structure of the program. An exercise is clearly a concept, so it is probably worthwhile to create a class to represent it.

Let's create the class Exercise. The class has a name and knowledge of whether it has been completed.

```
public class Exercise{  
  
    private String name;  
    private boolean completed;  
  
    public Tehtava(String name) {  
        this.name = name;  
        this.completed = false;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setCompleted(boolean completed) {  
        this.completed = completed;  
    }  
  
    public boolean isCompleted() {  
        return completed;  
    }  
}
```

Now let's change the structure of ExerciseManagement so that instead of strings the class will store the exercises as instances of the Exercise class.

Take notice that the definitions of the methods don't change, even though their internal workings do.

Even though the change had a large impact on the inner structure of the ExerciseManagement, the tests still pass.

Test-driven development would continue in the same manner until the wanted basic functionality would be in place.

```
import java.util.ArrayList;

public class ExerciseManagement{

    private ArrayList<Exercise> exercises;

    public ExerciseManagement() {
        this.exercises = new ArrayList<>();
    }

    public ArrayList<String> exerciselist() {
        ArrayList<String> list = new ArrayList<>();
        for (Exercise exercise: exercises) {
            list.add(exercise.getNamei());
        }

        return list;
    }

    public void add(String exercisea) {
        this.exercises.add(new Exercise(exercisea));
    }

    public void markAsCompleted(String exercise) {
        for (Exercise ex: exercises) {
            if (ex.getName().equals(exercise)) {
                ex.setCompleted(true);
            }
        }
    }

    public boolean isCompleted(String exercise) {
        for (Exercise ex: exercises) {
            if (ex.getName().equals(exercise)) {
                return ex.isCompleted();
            }
        }

        return false;
    }
}
```

Summary

In test-driven development the functionality of the program is constructed in small steps. The programmer first writes a test that examines the wished functionality, and then writes the program code that passes that test.