Test Driven Development with Greenfoot

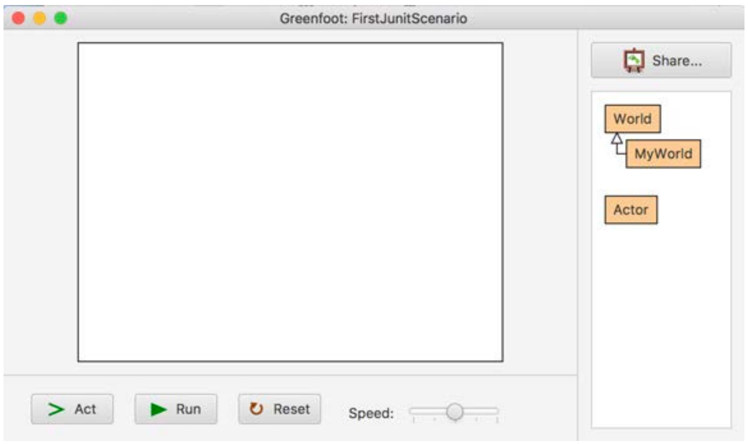
francisco.guerra@ulpgc.es

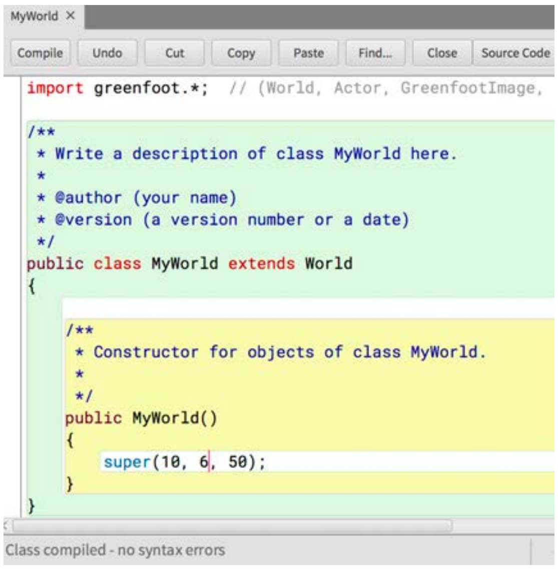
Version 1.2

This document shows how to apply Test Driven Development (TDD) while learning to program with Greenfoot.

- **Test Driven Development** is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements. This software process, that is related to the test-first programming concepts of extreme programming and that it is an efficient software production process, it is also adequate in the programming learning process. A programming teacher can use it in a theoretical explanation to show different behaviors of the software or in a laboratory practice he can set the objectives in the proposed exercises. But who else can take advantage of this way of programming is the student. When using TDD in the development of a program, the student acquires knowledge while increasing their confidence, becoming a better programmer in less time.
- **Greenfoot** is an integrated development environment using Java or Stride designed primarily for educational purposes at the high school and undergraduate level. It allows easy development of two-dimensional graphical applications, such as simulations and interactive games.

Let´s create a new Java Project into Greenfoot tool. As usual, this project has only the World class ***MyWorld***. The size of the scenario has been limited to (10, 6, 50) so that the examples fit well in this document.

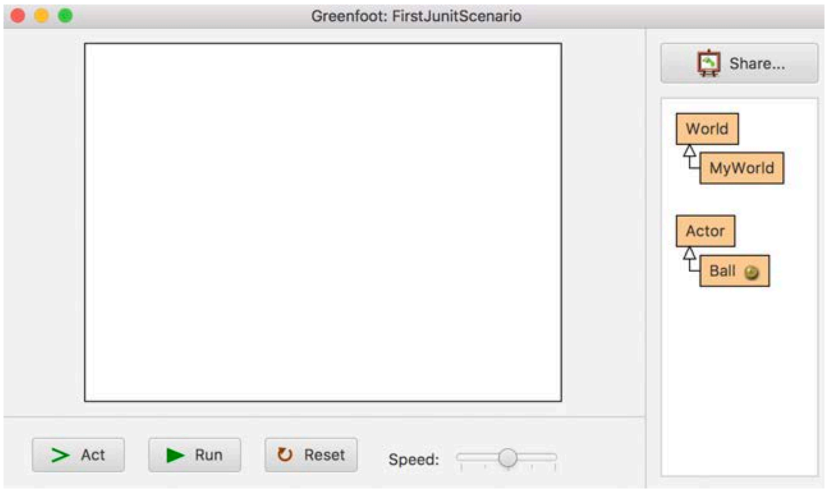


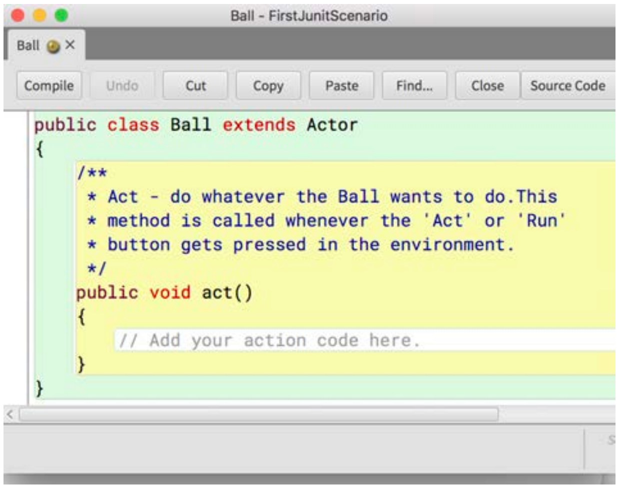


```java
import greenfoot.*;  // (World, Actor, GreenfootImage,

/**
 * Write a description of class MyWorld here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class MyWorld extends World
{

    /**
     * Constructor for objects of class MyWorld.
     *
     */
    public MyWorld()
    {
        super(10, 6, 50);
    }
}
```

Class compiled - no syntax errors

Next, let´s create one Actor class, for example a ball.

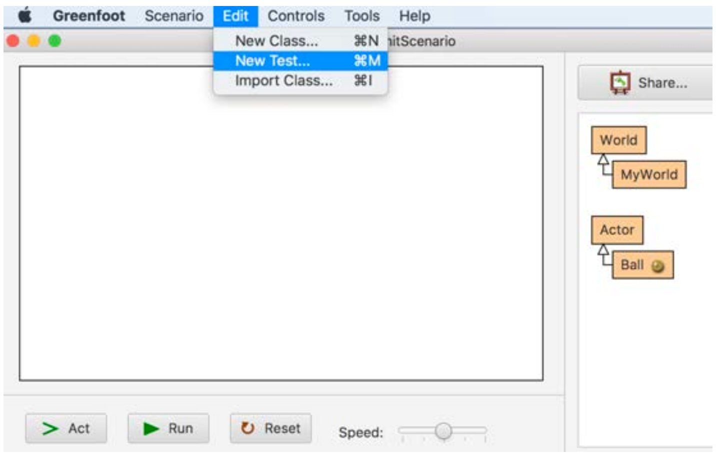


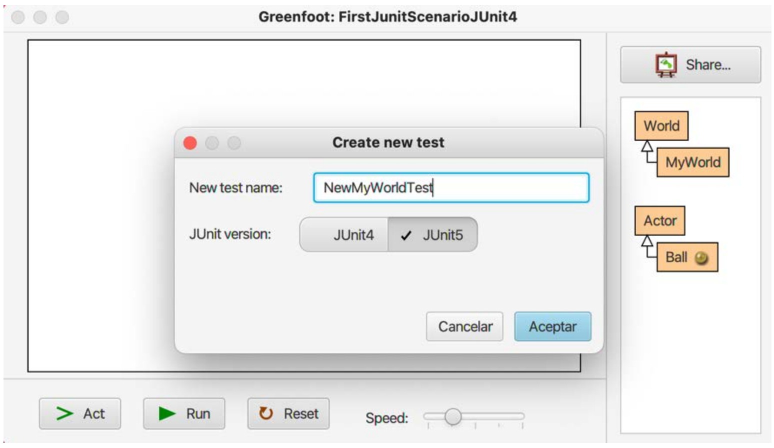


```java
public class Ball extends Actor
{
    /**
     * Act - do whatever the Ball wants to do.This
     * method is called whenever the 'Act' or 'Run'
     * button gets pressed in the environment.
     */
    public void act()
    {
        // Add your action code here.
    }
}
```

The RED and GREEN steps of TDD

Both classes, ***MyWord*** and ***Ball,*** have not any behavior yet. TDD can help us to program these classes. Let´s assume that ***MyWorld*** *always begins with a ball in the position (2, 1)*. Before program this requirement, let´s include a test that verifies it in the project. This is **RED** step of TDD. The Greenfoot Version with TDD includes a menu entry to create this test. The name of test usually is the name of class associated plus the word "*Test*", ***MyWorldTest*** will be in this case.





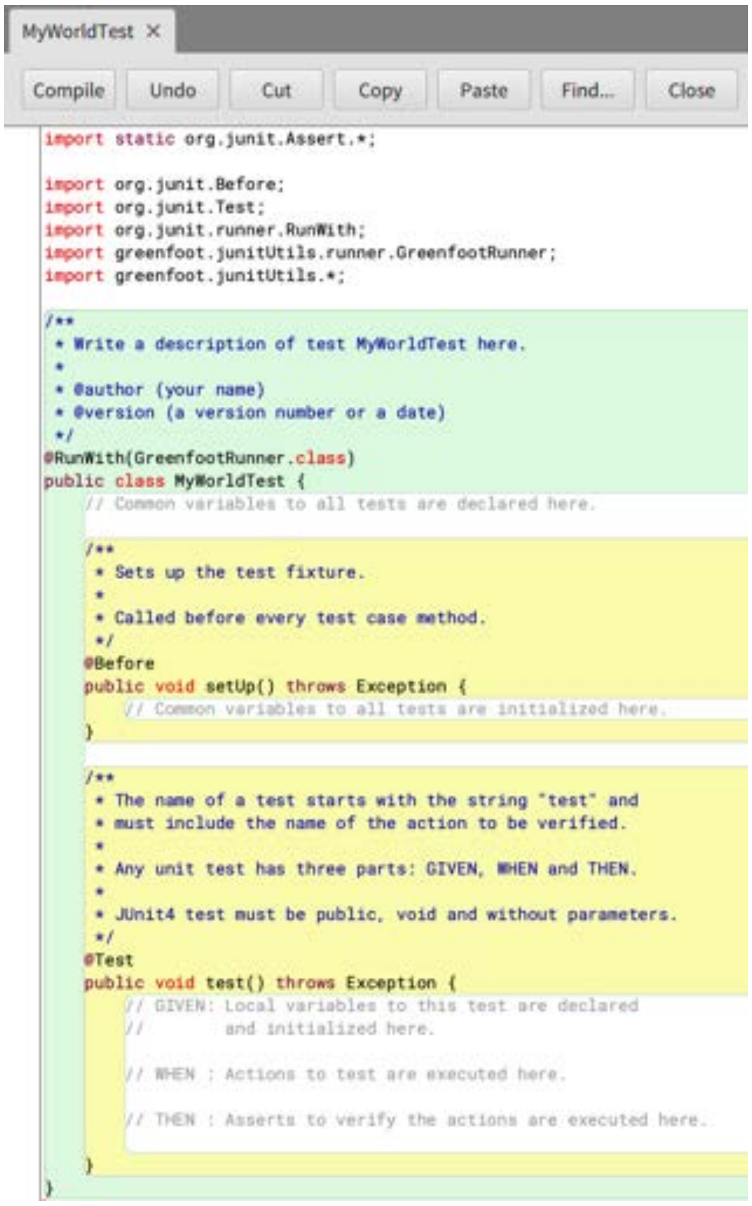
Create new test

New test name: NewMyWorldTest

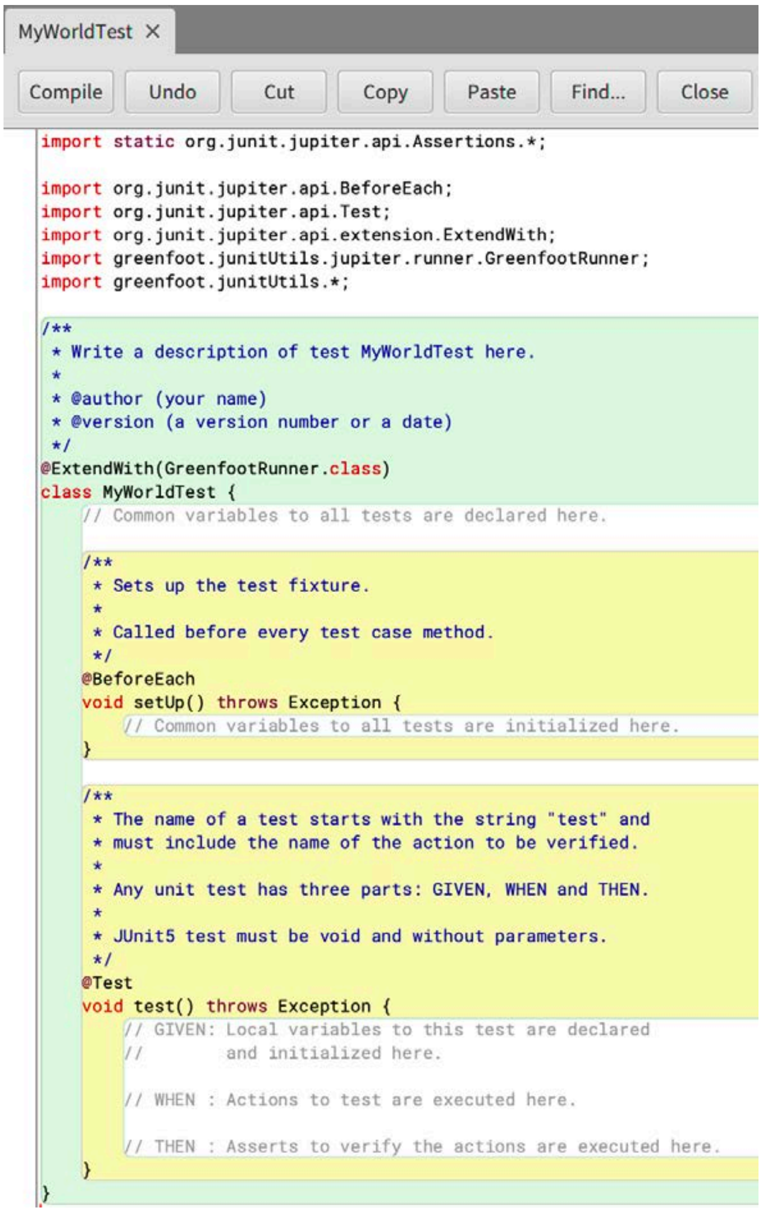
JUnit version: JUnit4 [✓ JUnit5]

Cancelar Aceptar

Greenfoot helps the students, because the test is created by a template with a commented skeleton of Junit framework test: one for Junit4 version and other Junit5 version. The two templates only differ in the imports of the chosen JUnit framework version or some words that although different, have a similar behavior, such as *RunWith* and *ExtendWith* or *Before* and *BeforeEach*. An important difference is that the JUnit5 version allows declaring local tests, that is, removing the public modifier.

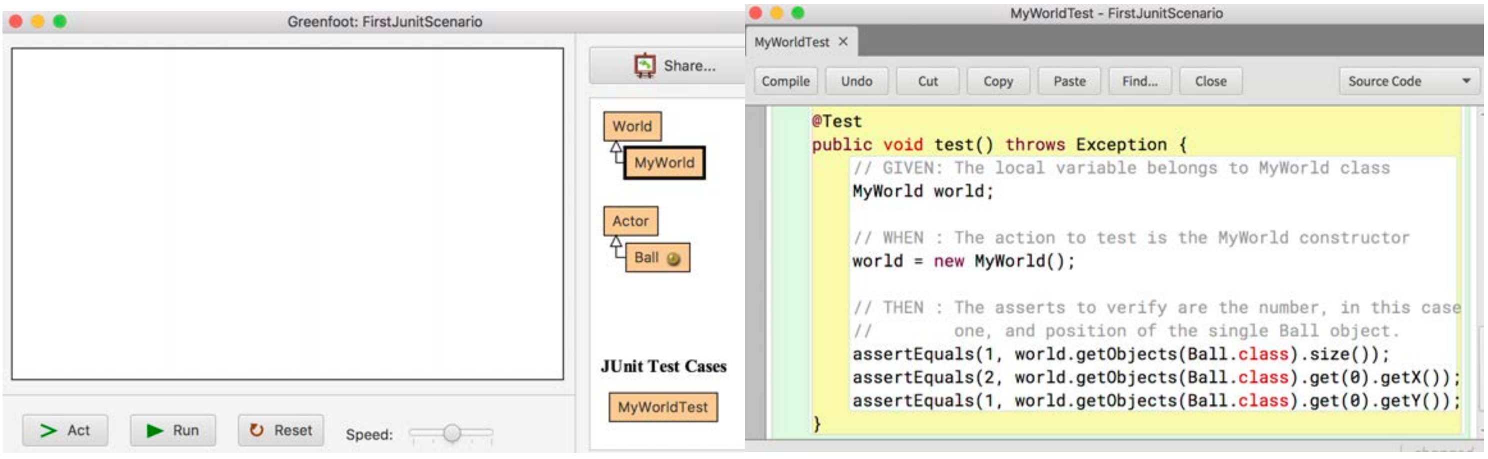


```
import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import greenfoot.junitUtils.runner.GreenfootRunner;
import greenfoot.junitUtils.*;

/**
 * Write a description of test MyWorldTest here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
@RunWith(GreenfootRunner.class)
public class MyWorldTest {
    // Common variables to all tests are declared here.

    /**
     * Sets up the test fixture.
     *
     * Called before every test case method.
     */
    @Before
    public void setUp() throws Exception {
        // Common variables to all tests are initialized here.
    }

    /**
     * The name of a test starts with the string "test" and
     * must include the name of the action to be verified.
     *
     * Any unit test has three parts: GIVEN, WHEN and THEN.
     *
     * JUnit4 test must be public, void and without parameters.
     */
    @Test
    public void test() throws Exception {
        // GIVEN: Local variables to this test are declared
        //        and initialized here.

        // WHEN : Actions to test are executed here.

        // THEN : Asserts to verify the actions are executed here.
    }
}
```



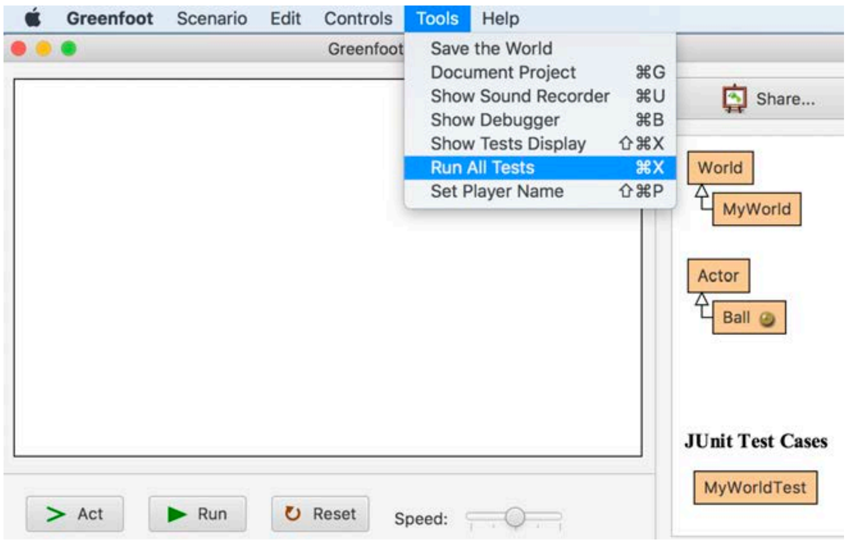
```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import greenfoot.junitUtils.jupiter.runner.GreenfootRunner;
import greenfoot.junitUtils.*;

/**
 * Write a description of test MyWorldTest here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
@ExtendWith(GreenfootRunner.class)
class MyWorldTest {
    // Common variables to all tests are declared here.

    /**
     * Sets up the test fixture.
     *
     * Called before every test case method.
     */
    @BeforeEach
    void setUp() throws Exception {
        // Common variables to all tests are initialized here.
    }

    /**
     * The name of a test starts with the string "test" and
     * must include the name of the action to be verified.
     *
     * Any unit test has three parts: GIVEN, WHEN and THEN.
     *
     * JUnit5 test must be void and without parameters.
     */
    @Test
    void test() throws Exception {
        // GIVEN: Local variables to this test are declared
        //        and initialized here.

        // WHEN : Actions to test are executed here.

        // THEN : Asserts to verify the actions are executed here.
    }
}
```

The Greenfoot IDE has a separate area to show the unit test cases of the project. The previous requirement, "***MyWorld** always begins with a ball in the position (2, 1)*", can be expressed in Java by the next ***GIVEN***/***WHEN***/***THEN*** of test included in MyWorldTest:

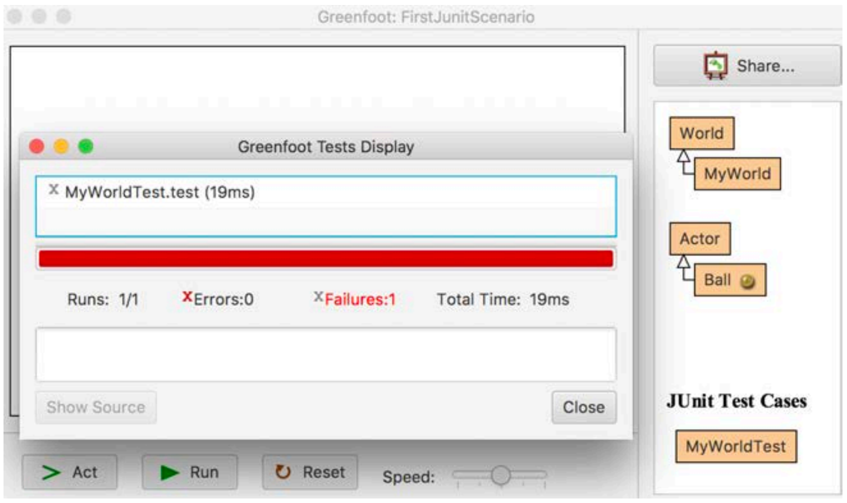


```
@Test
public void test() throws Exception {
    // GIVEN: The local variable belongs to MyWorld class
    MyWorld world;

    // WHEN : The action to test is the MyWorld constructor
    world = new MyWorld();

    // THEN : The asserts to verify are the number, in this case
    //        one, and position of the single Ball object.
    assertEquals(1, world.getObjects(Ball.class).size());
    assertEquals(2, world.getObjects(Ball.class).get(0).getX());
    assertEquals(1, world.getObjects(Ball.class).get(0).getY());
}
```

There is an entry in the menu to request that all the tests of the project be executed.



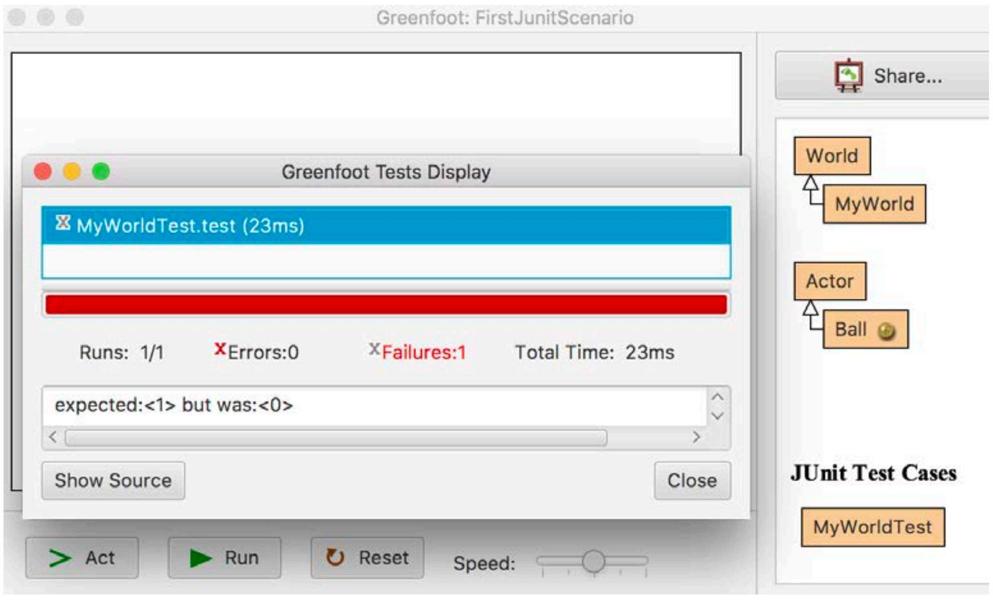
Because there is only one test in the project, the results display only shows this test. When this test is executed, the result is **RED** because ***MyWorld*** has not been programmed yet.



MyWorldTest.test (19ms)

Runs: 1/1 Errors:0 Failures:1 Total Time: 19ms

Greenfoot Test Display shows the fail information when the failed test is clicked by the mouse. The test result informs that expected <1> but was <0>, because the world has not any ball yet.

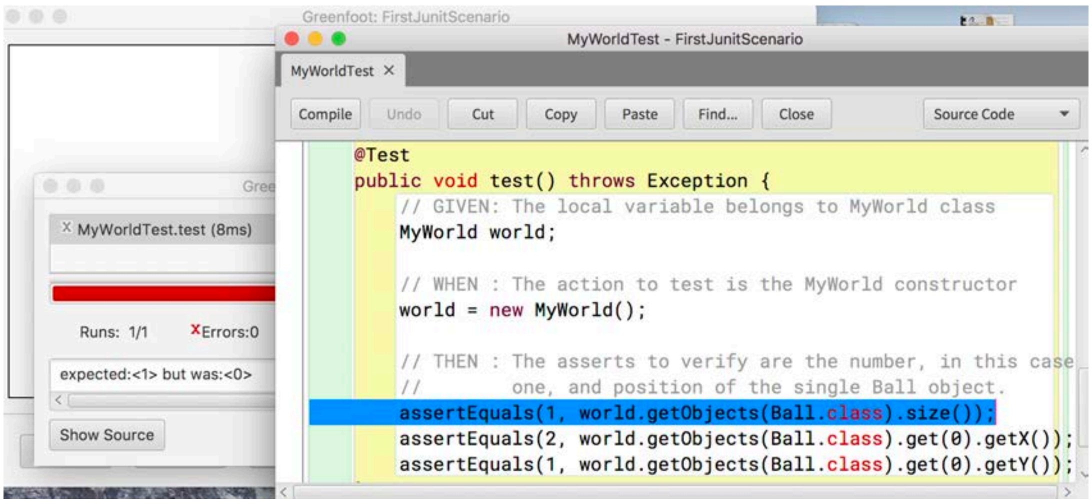


MyWorldTest.test (23ms)

Runs: 1/1 Errors:0 Failures:1 Total Time: 23ms

```
expected:<1> but was:<0>
```

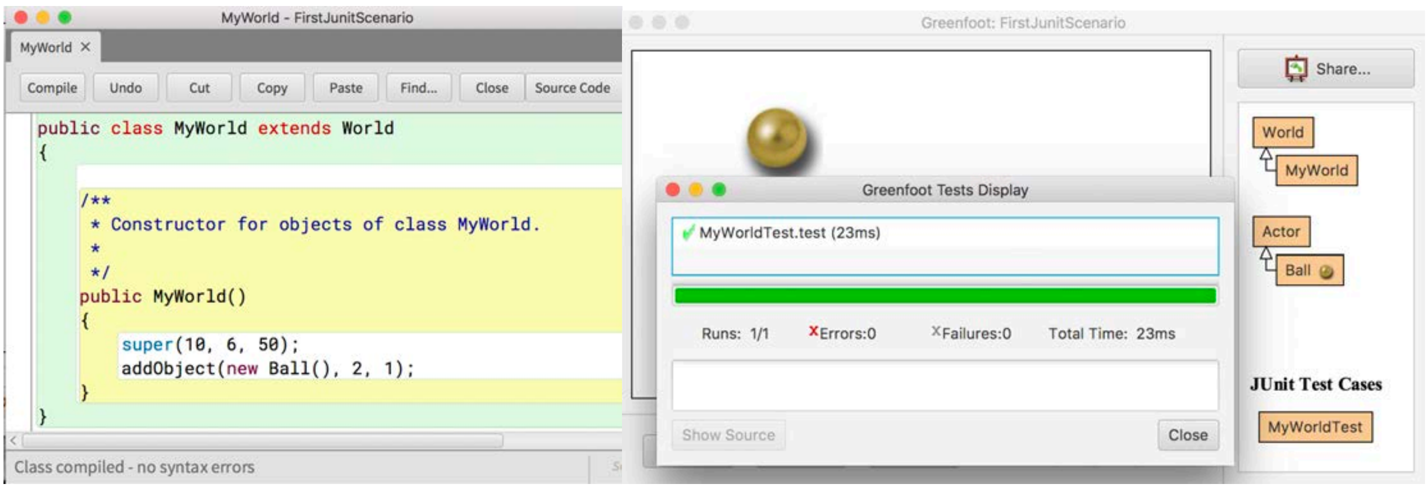
In addition, with a double click of the mouse in the failed test or by pressing the "*Show source*" button, you can see the line of the test where the statement that has detected the fault is found.



expected:<1> but was:<0>

```
@Test
public void test() throws Exception {
    // GIVEN: The local variable belongs to MyWorld class
    MyWorld world;

    // WHEN : The action to test is the MyWorld constructor
    world = new MyWorld();

    // THEN : The asserts to verify are the number, in this case
    //        one, and position of the single Ball object.
    assertEquals(1, world.getObjects(Ball.class).size());
    assertEquals(2, world.getObjects(Ball.class).get(0).getX());
    assertEquals(1, world.getObjects(Ball.class).get(0).getY());
```

Next, ***MyWorld*** can be programmed to enforce the requirement. This is **GREEN** step of TDD. After the requirement is programmed, it must be verified it executing the test. Although Greenfoot shows the ball inside the world, only with the sight could not be assured that the ball is in the correct position. However, the green test result ensures that the requirement is met.

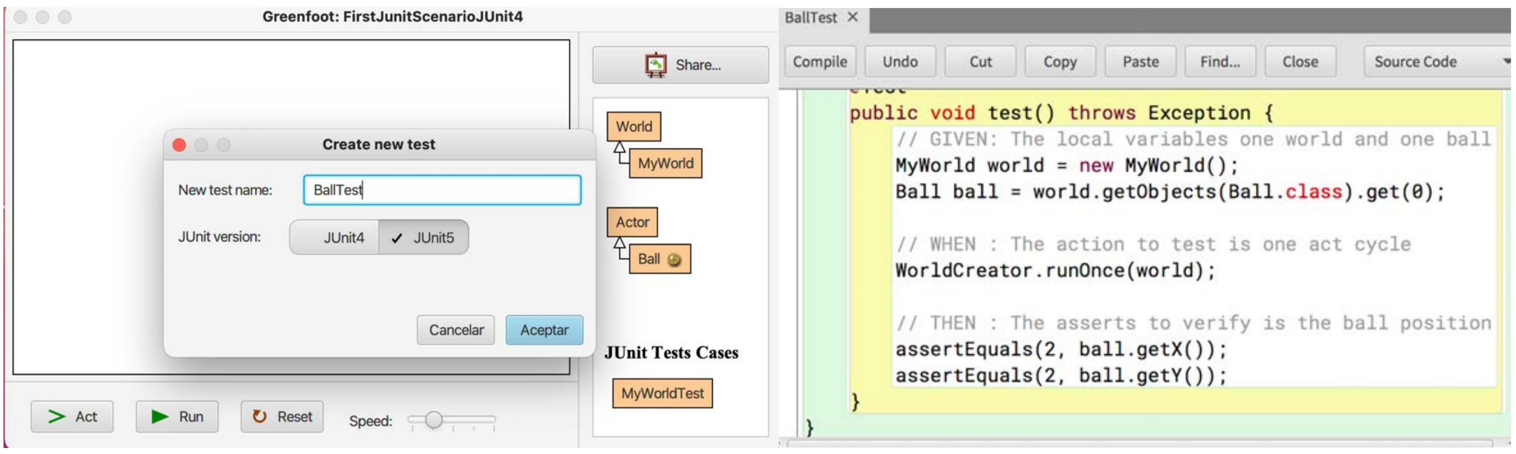


```
public class MyWorld extends World
{

    /**
     * Constructor for objects of class MyWorld.
     *
     */
    public MyWorld()
    {
        super(10, 6, 50);
        addObject(new Ball(), 2, 1);
    }
}
```

Class compiled - no syntax errors

It is usual that a requirement needs several tests

Now TDD cycle can be begun with the next requirement: "*While the ball does not arrive to the down border, it goes down a position every time*". This requirement can be expressed in Java by the next test:



Create new test

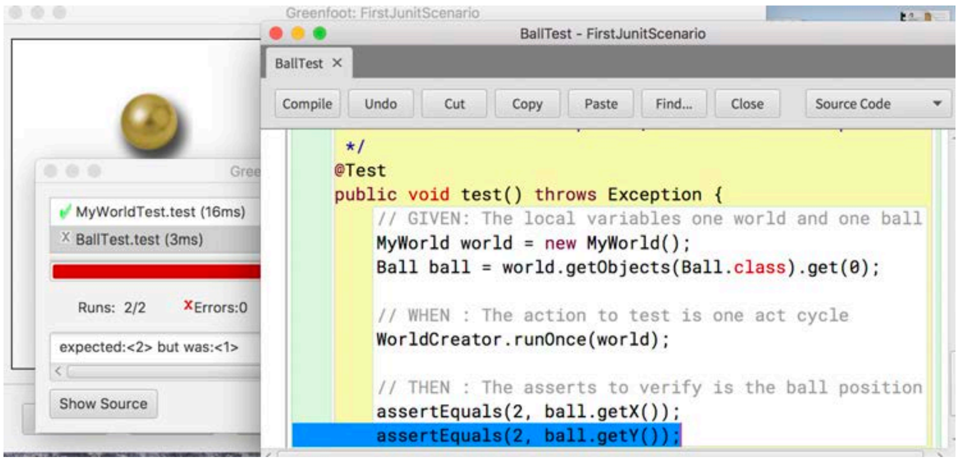
New test name: BallTest

JUnit version: JUnit4 [✓ JUnit5]

Cancelar Aceptar

```
public void test() throws Exception {
    // GIVEN: The local variables one world and one ball
    MyWorld world = new MyWorld();
    Ball ball = world.getObjects(Ball.class).get(0);

    // WHEN : The action to test is one act cycle
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify is the ball position
    assertEquals(2, ball.getX());
    assertEquals(2, ball.getY());
}
}
```

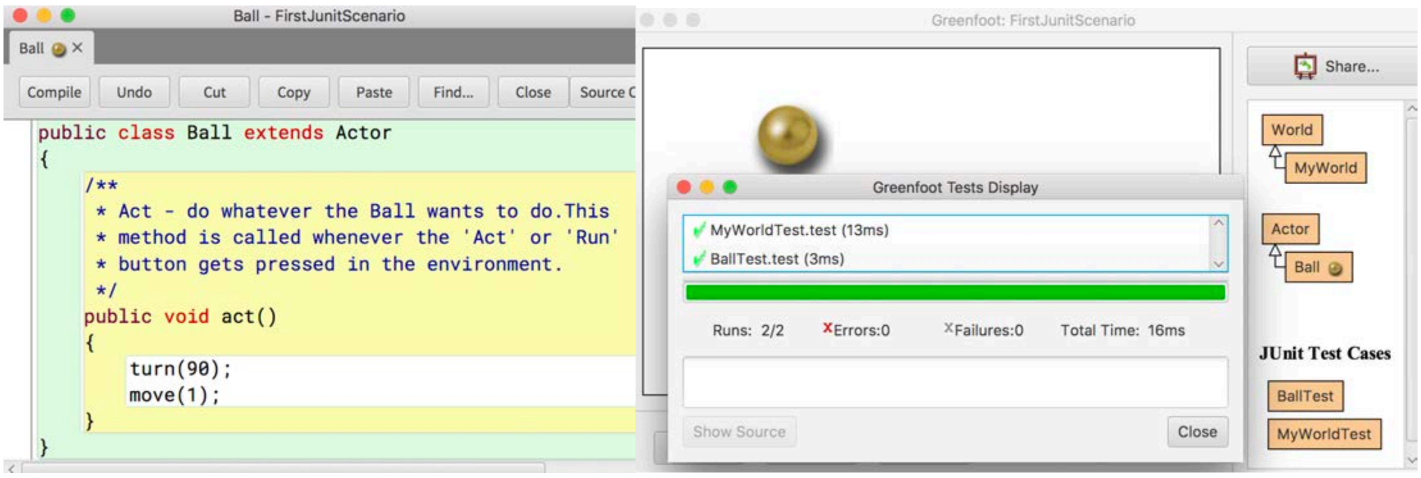
Since the Ball class has not yet been programmed, the result of the test is **RED**. It is very important that the first result of any new test is **RED**, because this is the way how we can verify that the test is doing its job well. When clicking with the mouse on the test that has failed, it can be verified that the test is correct, because it is waiting for the ball, which initially starts from the position (2,1), to increase its position on the Y axis, passing to the position (2,2).



```
MyWorldTest.test (16ms)
BallTest.test (3ms)
Runs: 2/2   Errors:0
expected:<2> but was:<1>
```

```
*/
@Test
public void test() throws Exception {
    // GIVEN: The local variables one world and one ball
    MyWorld world = new MyWorld();
    Ball ball = world.getObjects(Ball.class).get(0);

    // WHEN : The action to test is one act cycle
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify is the ball position
    assertEquals(2, ball.getX());
    assertEquals(2, ball.getY());
```

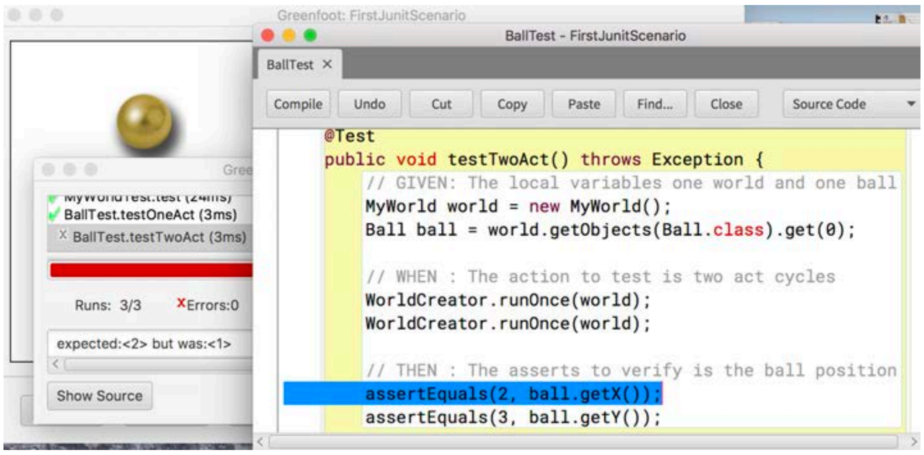
Next, the **GREEN** step must be performed, in other words, the Ball class is programmed to satisfy the requirement that has been imposed.



```
public class Ball extends Actor
{
    /**
     * Act - do whatever the Ball wants to do.This
     * method is called whenever the 'Act' or 'Run'
     * button gets pressed in the environment.
     */
    public void act()
    {
        turn(90);
        move(1);
    }
}
```

```
MyWorldTest.test (13ms)
BallTest.test (3ms)
Runs: 2/2   Errors:0   Failures:0   Total Time: 16ms
```

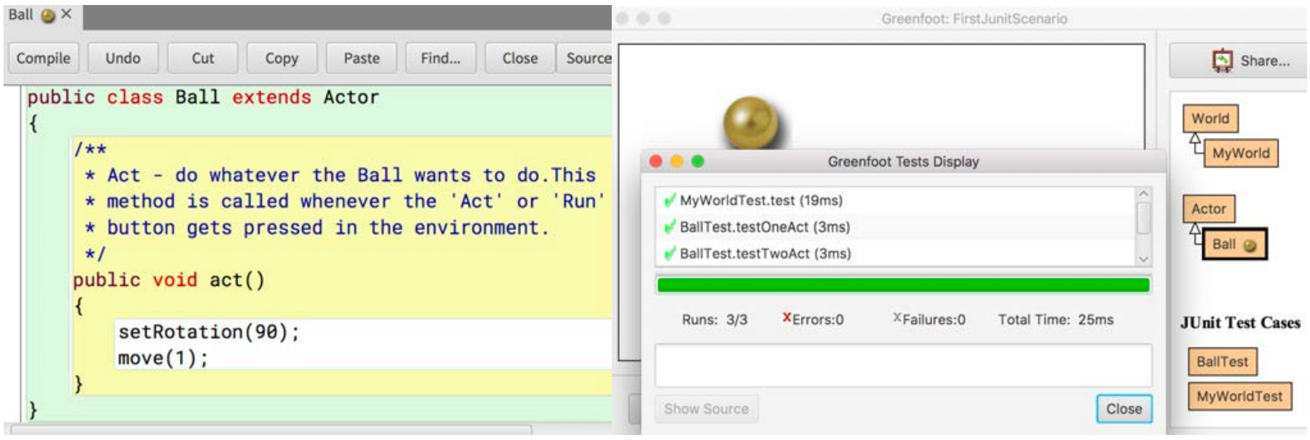
The requirement that is being programmed needs more than one test. In fact, as usual, any requirement needs several tests to be able to verify that it has been achieved completely. It will be the experience that will help you find all the tests that we must add in order to be able to verify it in a complete way. By adding a second test, that checks the position of the ball after two cycles of ***act()*** method, it can be seen that the ball has not reached the desired position, in other words, the solution programmed to enforce the first test does not It's enough.



```
MyWorldTest.test (24ms)
BallTest.testOneAct (3ms)
BallTest.testTwoAct (3ms)
Runs: 3/3   Errors:0
expected:<2> but was:<1>
```

```
@Test
public void testTwoAct() throws Exception {
    // GIVEN: The local variables one world and one ball
    MyWorld world = new MyWorld();
    Ball ball = world.getObjects(Ball.class).get(0);

    // WHEN : The action to test is two act cycles
    WorldCreator.runOnce(world);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify is the ball position
    assertEquals(2, ball.getX());
    assertEquals(3, ball.getY());
```

Since, in the first programmed solution, the ball rotates 90 degrees in each cycle of the ***act()*** method, in the second cycle it moves in the X axis and does not continue to fall towards the edge. The new solution that has been programmed to enforce both tests is always to fix the same direction, not accumulating rotations.

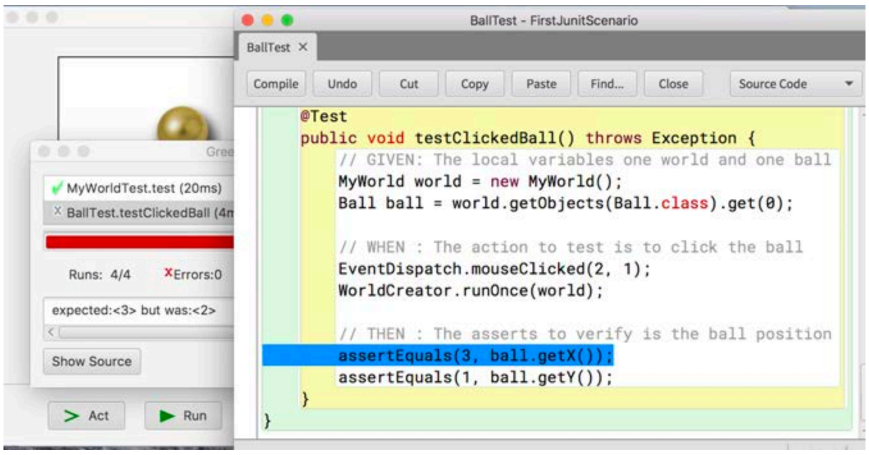


```
public class Ball extends Actor
{
    /**
     * Act - do whatever the Ball wants to do.This
     * method is called whenever the 'Act' or 'Run'
     * button gets pressed in the environment.
     */
    public void act()
    {
        setRotation(90);
        move(1);
    }
}
```

```
Greenfoot Tests Display
✓ MyWorldTest.test (19ms)
✓ BallTest.testOneAct (3ms)
✓ BallTest.testTwoAct (3ms)
Runs: 3/3   ✗Errors:0   ✗Failures:0   Total Time: 25ms
```

Sometimes, a mouse event must be simulated

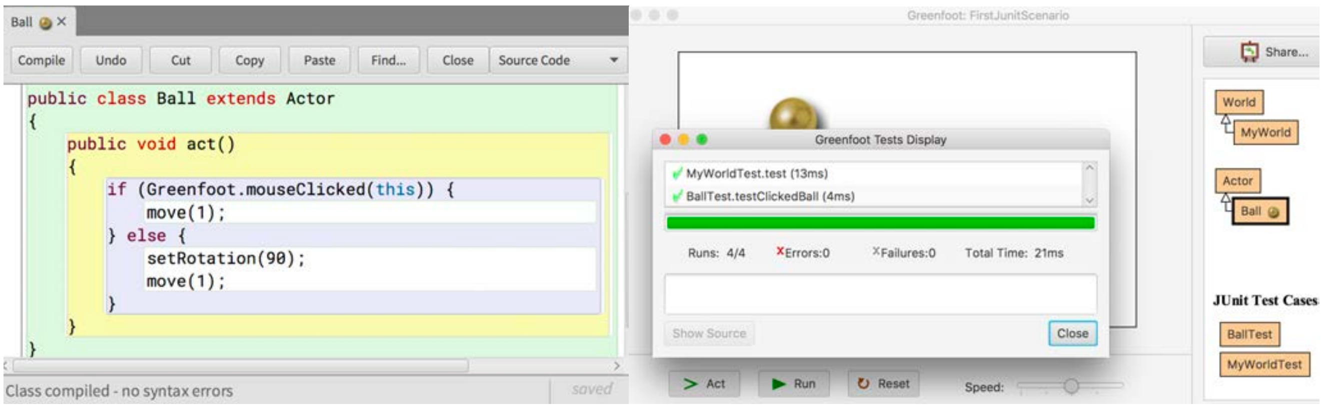
Now TDD cycle can be begun with the next requirement: "*Each time the ball is clicked with the mouse, a corrected one moves on the X axis, without falling on the Y axis*". A test that specifies this requirement must simulate the click event of the mouse. To get it in the ***greenfoot.junitUtils*** package is the ***EventDispatch*** class that simulates keyboard and mouse events. So, the **RED** step of the next TDD cycle that we started with this requirement must simulate a mouse click before indicating that the ***act()*** method of the ball is executed.



```
@Test
public void testClickedBall() throws Exception {
    // GIVEN: The local variables one world and one ball
    MyWorld world = new MyWorld();
    Ball ball = world.getObjects(Ball.class).get(0);

    // WHEN : The action to test is to click the ball
    EventDispatch.mouseClicked(2, 1);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify is the ball position
    assertEquals(3, ball.getX());
    assertEquals(1, ball.getY());
}
}
```

```
✓ MyWorldTest.test (20ms)
✗ BallTest.testClickedBall (4m
Runs: 4/4   ✗Errors:0
expected:<3> but was:<2>
```

Next, ***Ball*** class can be programmed to enforce the new requirement. The "**if**" statement allows you to differentiate several possible alternatives in a program. The requirements that we have up to now have established two possible cases for a ball: to fall if it is not clicked or to move to the right if it is clicked.

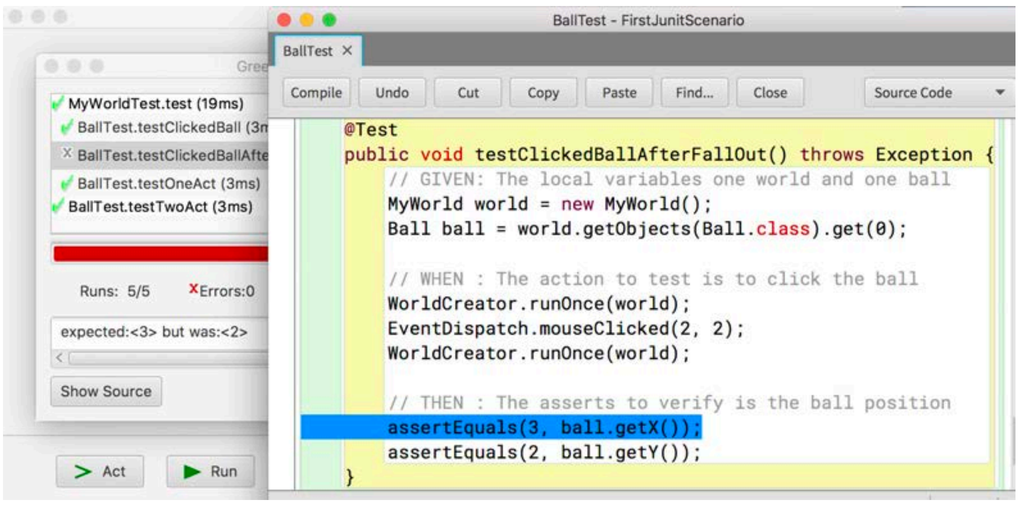


```
public class Ball extends Actor
{
    public void act()
    {
        if (Greenfoot.mouseClicked(this)) {
            move(1);
        } else {
            setRotation(90);
            move(1);
        }
    }
}
```

Class compiled - no syntax errors

```
Greenfoot Tests Display
✓ MyWorldTest.test (13ms)
✓ BallTest.testClickedBall (4ms)
Runs: 4/4   ✗Errors:0   ✗Failures:0   Total Time: 21ms
```

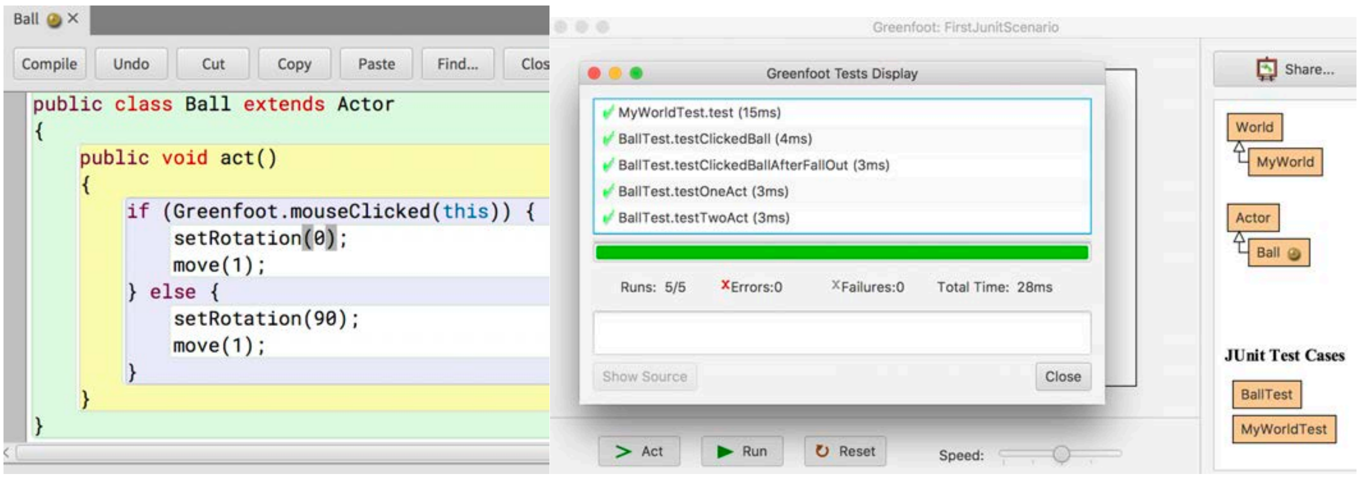
This requirement that is being programmed also needs more than one test. Let´s assume a ball that has dropped a position on the Y axis, and then this ball is clicked with the mouse. If this ball was at the initial position (2,1), the movement on the Y axis takes it to the position (2,2) and the mouse click should make it reach the position (3,2). However, the test added to the project, which contains this case, fails.



```
MyWorldTest.test (19ms)
BallTest.testClickedBall (3m
BallTest.testClickedBallAfte
BallTest.testOneAct (3ms)
BallTest.testTwoAct (3ms)

Runs: 5/5    Errors:0

expected:<3> but was:<2>
```

```java
@Test
public void testClickedBallAfterFallOut() throws Exception {
    // GIVEN: The local variables one world and one ball
    MyWorld world = new MyWorld();
    Ball ball = world.getObjects(Ball.class).get(0);

    // WHEN : The action to test is to click the ball
    WorldCreator.runOnce(world);
    EventDispatch.mouseClicked(2, 2);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify is the ball position
    assertEquals(3, ball.getX());
    assertEquals(2, ball.getY());
}
```

This happens because when the first cycle of ***act()*** method is executed, the ball is turned 90 degrees to move on the Y axis, and then when it is clicked with the mouse, it moves on the axis where it is at this moment, that is the Y axis. A simple solution is to correct the rotation before moving on the X axis.



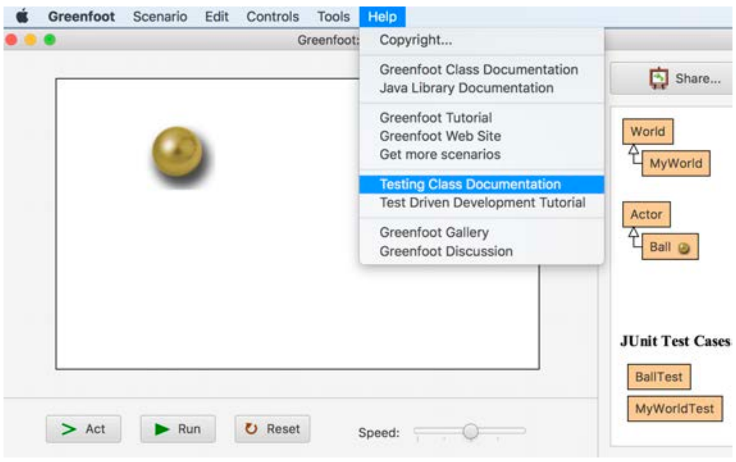
```java
public class Ball extends Actor
{
    public void act()
    {
        if (Greenfoot.mouseClicked(this)) {
            setRotation(0);
            move(1);
        } else {
            setRotation(90);
            move(1);
        }
    }
}
```

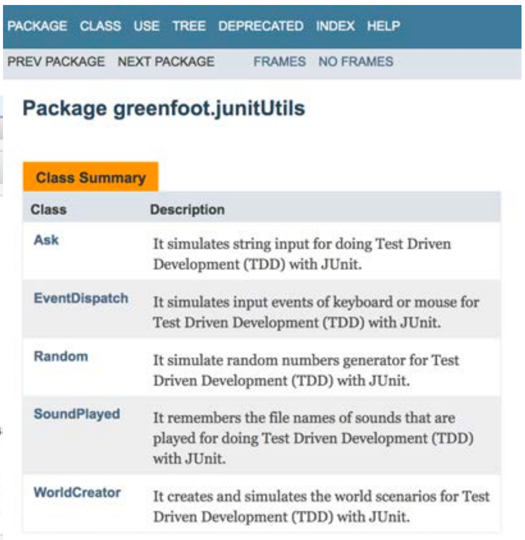
```
MyWorldTest.test (15ms)
BallTest.testClickedBall (4ms)
BallTest.testClickedBallAfterFallOut (3ms)
BallTest.testOneAct (3ms)
BallTest.testTwoAct (3ms)

Runs: 5/5    Errors:0    Failures:0    Total Time: 28ms
```

Now the new test is **GREEN**, that means, the requirement that was wanted to be included in the project, that has been programmed, is met. But the most important is that also the rest of the tests confirm that all the requirements programmed up to now are fulfilled. This characteristic of programming with the **RED** / **GREEN** cycle of TDD is what increases the confidence of the programmers who use it and improves the learning process. This confidence increases even when some of the previous tests fail and you have to rewrite the solution so that all the tests give a **GREEN** result, in other words, that all the requirements are met. This offers the programmer a broad knowledge of the state in which the solution is found at that moment and helps to take the next steps without the fear of losing what has been achieved.

Testing class documentation

The TDD cycle that has just been done uses the ***EventDispatch.mouseClicked(x,y)*** method of ***greenfoot.junitUtils*** package to simulate that the user clicks with the mouse on the ball object that is in the position (x,y). In the Greenfoot IDE menu, there is an entry to see how to use some methods that can help build the test that is needed.



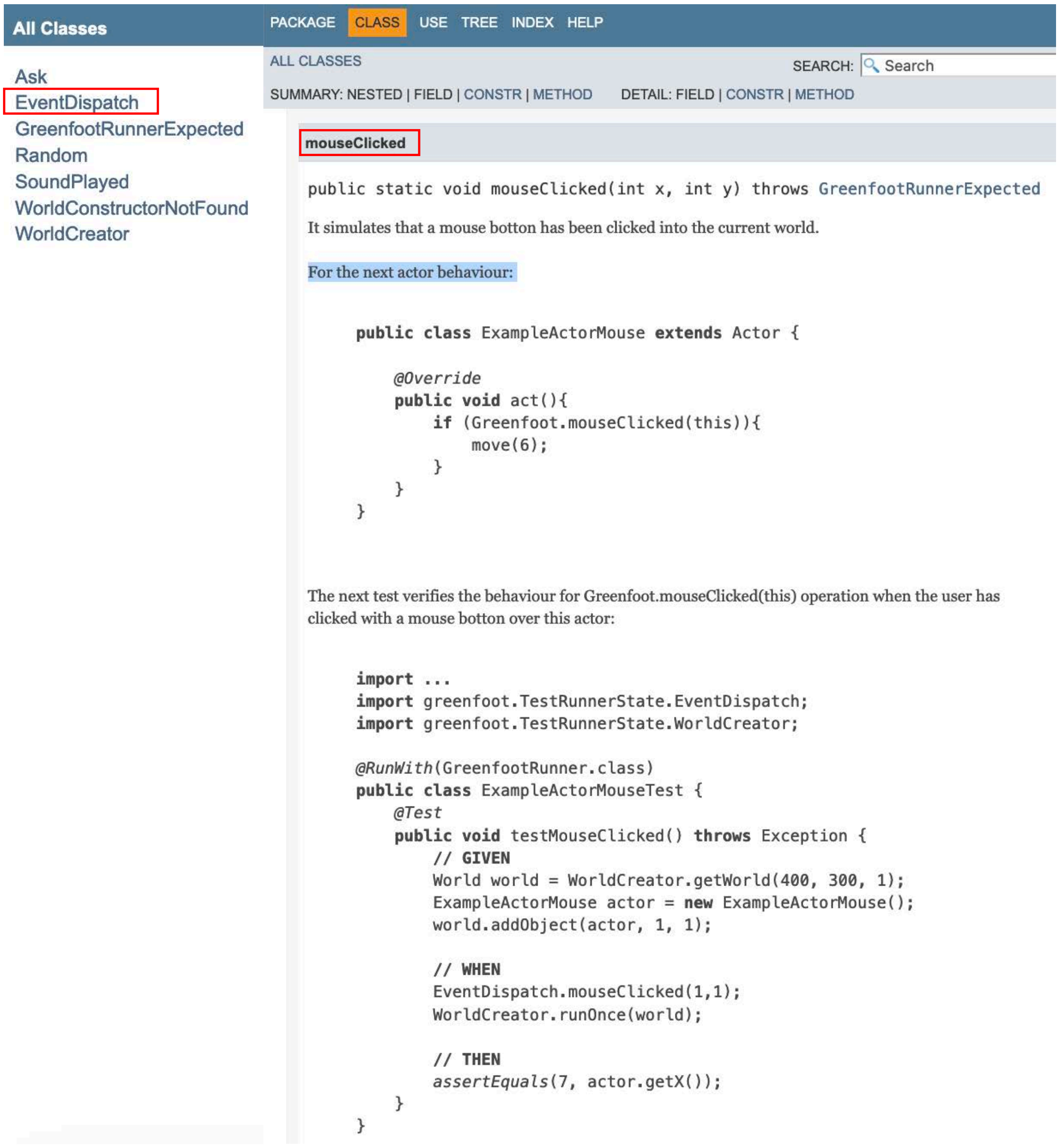

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package greenfoot.junitUtils

Class Summary

Class	Description
Ask	It simulates string input for doing Test Driven Development (TDD) with JUnit.
EventDispatch	It simulates input events of keyboard or mouse for Test Driven Development (TDD) with JUnit.
Random	It simulate random numbers generator for Test Driven Development (TDD) with JUnit.
SoundPlayed	It remembers the file names of sounds that are played for doing Test Driven Development (TDD) with JUnit.
WorldCreator	It creates and simulates the world scenarios for Test Driven Development (TDD) with JUnit.

For example, in this help web page you could have consulted an example of use for the simulation of the mouse event that was needed. Each example that contains this help consists of two parts: the test that is shown as an example template, preceded by the piece of program that is supposed to be tested. To make the user manual simple and short, all the tests that the help contains are checking a piece of the program that meets the requirements, in other words, the result of all the tests is **GREEN**.



mouseClicked

```
public static void mouseClicked(int x, int y) throws GreenfootRunnerExpected
```

It simulates that a mouse botton has been clicked into the current world.

For the next actor behaviour:

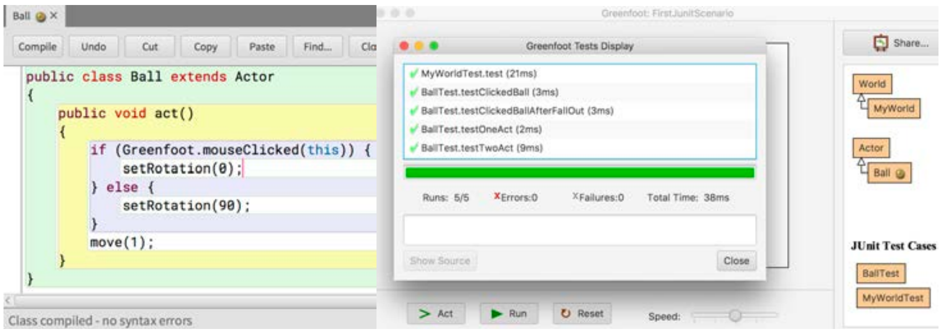
```
public class ExampleActorMouse extends Actor {

    @Override
    public void act(){
        if (Greenfoot.mouseClicked(this)){
            move(6);
        }
    }
}
```

The next test verifies the behaviour for Greenfoot.mouseClicked(this) operation when the user has clicked with a mouse botton over this actor:

```
import ...
import greenfoot.TestRunnerState.EventDispatch;
import greenfoot.TestRunnerState.WorldCreator;

@RunWith(GreenfootRunner.class)
public class ExampleActorMouseTest {
    @Test
    public void testMouseClicked() throws Exception {
        // GIVEN
        World world = WorldCreator.getWorld(400, 300, 1);
        ExampleActorMouse actor = new ExampleActorMouse();
        world.addObject(actor, 1, 1);

        // WHEN
        EventDispatch.mouseClicked(1,1);
        WorldCreator.runOnce(world);

        // THEN
        assertEquals(7, actor.getX());
    }
}
```

The REFACTORING step of TDD

It is important to keep the project clean and commented. For this reason, after the **GREEN** step, the **REFACTORING** step is usually carried out. This step of the TDD cycle consists principally in the restructuring of the project code to avoid duplication, at the same time as comments are made or better names are chosen for the identifiers. Although the project that is being used as an example is small and simple, it is always possible to improve a program. It is going to begin by taking the action of movement that exists in the two programmed cases by a common factor.



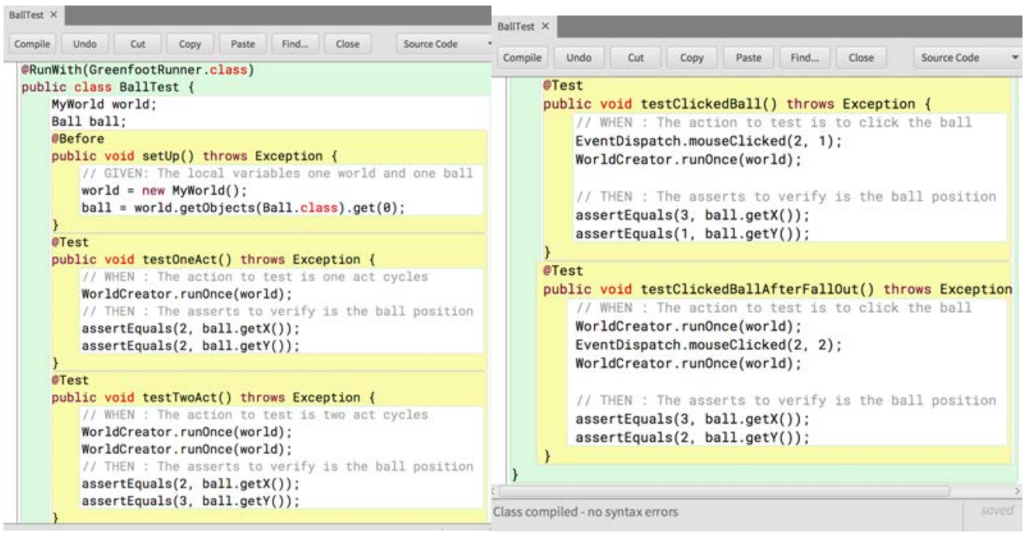
```java
public class Ball extends Actor
{
    public void act()
    {
        if (Greenfoot.mouseClicked(this)) {
            setRotation(0);
        } else {
            setRotation(90);
        }
        move(1);
    }
}
```

```
MyWorldTest.test (21ms)
BallTest.testClickedBall (3ms)
BallTest.testClickedBallAfterFallOut (3ms)
BallTest.testOneAct (2ms)
BallTest.testTwoAct (9ms)

Runs: 5/5    Errors:0    Failures:0    Total Time: 38ms
```

Class compiled - no syntax errors

It is essential to execute all tests after any change in the programmed code. This action will be performed each time a change is made, and this execution will be made even if the change is very small. If any test is **RED**, the code must be modified again until all the tests are **GREEN** again. Reaching even the **REFACTORING** if a satisfactory solution is not found. This way of working allows to detect an erroneous decision early and correct it in time.

The coding of the tests can also be improved. The action that most frequently is done is to remove the part of the ***GIVEN*** common to all the tests and put it in the operation ***setup***(), since the environment of execution of the tests always runs the ***setup***() before each test.

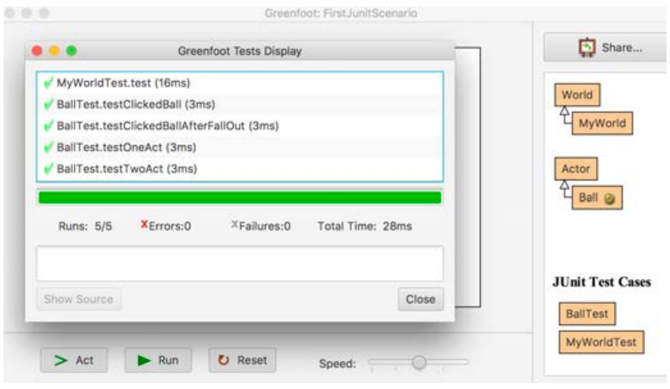


```java
@RunWith(GreenfootRunner.class)
public class BallTest {
    MyWorld world;
    Ball ball;
    @Before
    public void setUp() throws Exception {
        // GIVEN: The local variables one world and one ball
        world = new MyWorld();
        ball = world.getObjects(Ball.class).get(0);
    }
    @Test
    public void testOneAct() throws Exception {
        // WHEN : The action to test is one act cycles
        WorldCreator.runOnce(world);
        // THEN : The asserts to verify is the ball position
        assertEquals(2, ball.getX());
        assertEquals(2, ball.getY());
    }
    @Test
    public void testTwoAct() throws Exception {
        // WHEN : The action to test is two act cycles
        WorldCreator.runOnce(world);
        WorldCreator.runOnce(world);
        // THEN : The asserts to verify is the ball position
        assertEquals(2, ball.getX());
        assertEquals(3, ball.getY());
    }
```

```java
    @Test
    public void testClickedBall() throws Exception {
        // WHEN : The action to test is to click the ball
        EventDispatch.mouseClicked(2, 1);
        WorldCreator.runOnce(world);

        // THEN : The asserts to verify is the ball position
        assertEquals(3, ball.getX());
        assertEquals(1, ball.getY());
    }
    @Test
    public void testClickedBallAfterFallOut() throws Exception
        // WHEN : The action to test is to click the ball
        WorldCreator.runOnce(world);
        EventDispatch.mouseClicked(2, 2);
        WorldCreator.runOnce(world);

        // THEN : The asserts to verify is the ball position
        assertEquals(3, ball.getX());
        assertEquals(2, ball.getY());
    }
}
```

Class compiled - no syntax errors

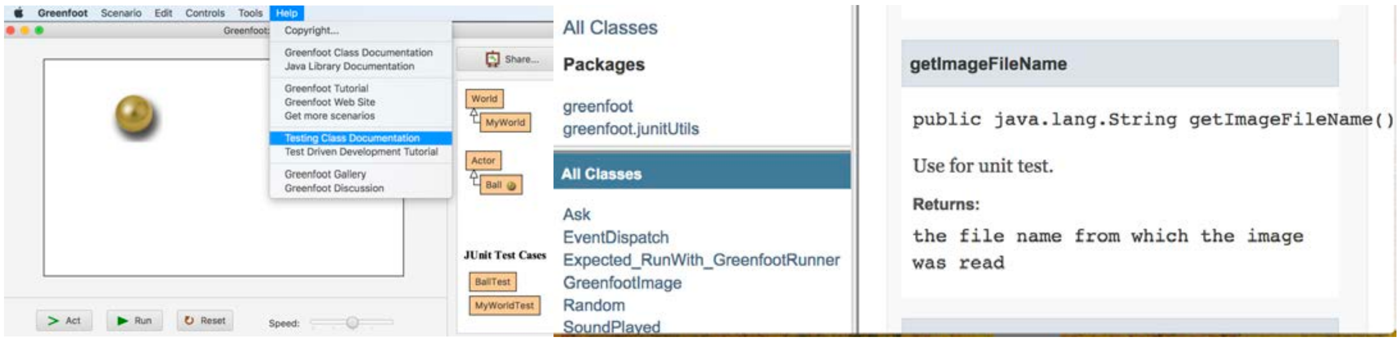
This **REFACTORING** is also valid because when executing all the tests of the project, the result of its execution is **GREEN**. Although in the previous **REFACTORING**, the **GIVEN** of all the tests is in the ***setUp()*** method, it is common that part of the **GIVEN** of each test is in the test itself, that means, many tests usually differentiate in some element of their initialization.



```
MyWorldTest.test (16ms)
BallTest.testClickedBall (3ms)
BallTest.testClickedBallAfterFallOut (3ms)
BallTest.testOneAct (3ms)
BallTest.testTwoAct (3ms)

Runs: 5/5    Errors:0    Failures:0    Total Time: 28ms
```

Images that change randomly

Now TDD cycle can be begun with the next requirement. “*There is a 20% chance that a ball will change its color before moving*”. It is going to be assumed that "gold-ball.png" is the image that the constructor associates with an object of the Ball class, and that the other possible image is "Steel-ball.png". In the Greenfoot IDE menu, a new method of ***greenfoot.GreenfootImage*** class allows to consult the name of the file from where the image was read.



All Classes

Packages

greenfoot
greenfoot.junitUtils

All Classes

Ask
EventDispatch
Expected_RunWith_GreenfootRunner
GreenfootImage
Random
SoundPlayed

getImageFileName

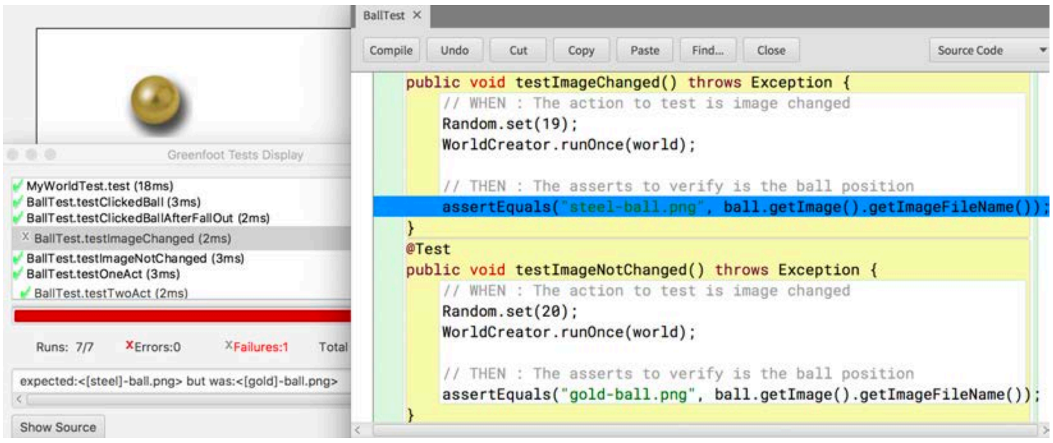
```
public java.lang.String getImageFileName()
```

Use for unit test.

Returns:

```
the file name from which the image
was read
```

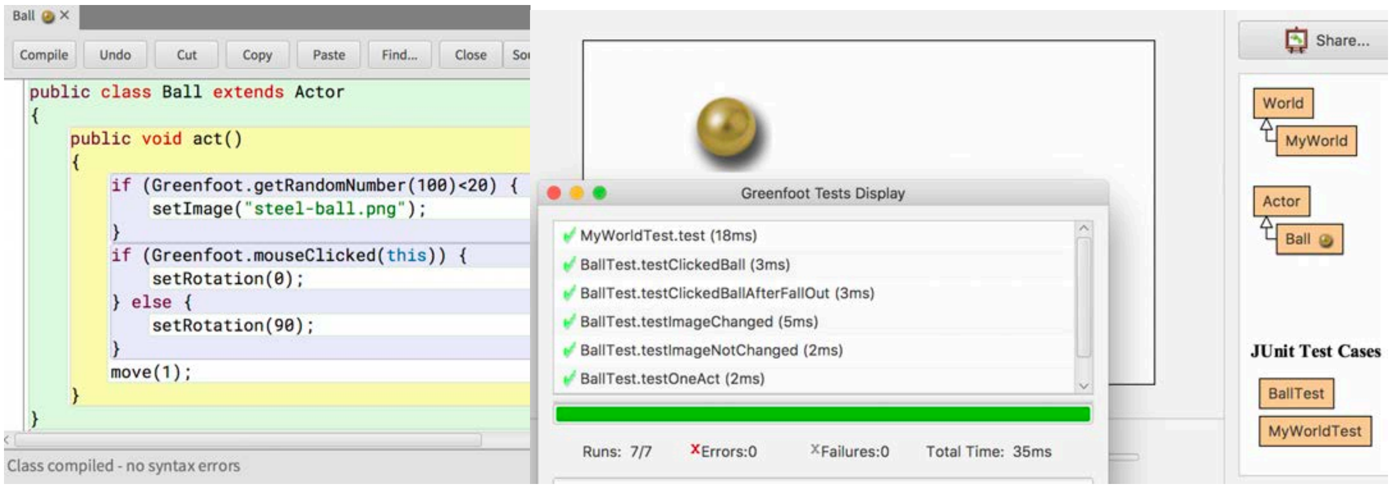
Let´s use the ***Greenfoot.getRandomNumber(100)*** method to generate a random number between 0 and 99. The values between 0 and 19 are going to be used to decide that the ball changes color. However, any test must not have any data that can change in different executions. For this reason, the **RED** step uses the ***Random*** class of the ***greenfoot.junitUtils*** package to simulate the generation of random numbers and creates the two possible cases: generate a number between 0 and 19 to check that the ball changes color 20% of the time and another number that is between 20 and 99 to check that 80% of the time does not change.



```
✓ MyWorldTest.test (18ms)
✓ BallTest.testClickedBall (3ms)
✓ BallTest.testClickedBallAfterFallOut (2ms)
× BallTest.testImageChanged (2ms)
✓ BallTest.testImageNotChanged (3ms)
✓ BallTest.testOneAct (3ms)
✓ BallTest.testTwoAct (2ms)

Runs: 7/7    ×Errors:0    ×Failures:1    Total

expected:<[steel]-ball.png> but was:<[gold]-ball.png>
```

```
public void testImageChanged() throws Exception {
    // WHEN : The action to test is image changed
    Random.set(19);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify is the ball position
    assertEquals("steel-ball.png", ball.getImage().getImageFileName());
}
@Test
public void testImageNotChanged() throws Exception {
    // WHEN : The action to test is image changed
    Random.set(20);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify is the ball position
    assertEquals("gold-ball.png", ball.getImage().getImageFileName());
}
```

Next, the **GREEN** step must be performed, in other words, the Ball class is programmed to satisfy the requirement that has been imposed.



```
public class Ball extends Actor
{
    public void act()
    {
        if (Greenfoot.getRandomNumber(100)<20) {
            setImage("steel-ball.png");
        }
        if (Greenfoot.mouseClicked(this)) {
            setRotation(0);
        } else {
            setRotation(90);
        }
        move(1);
    }
}
```

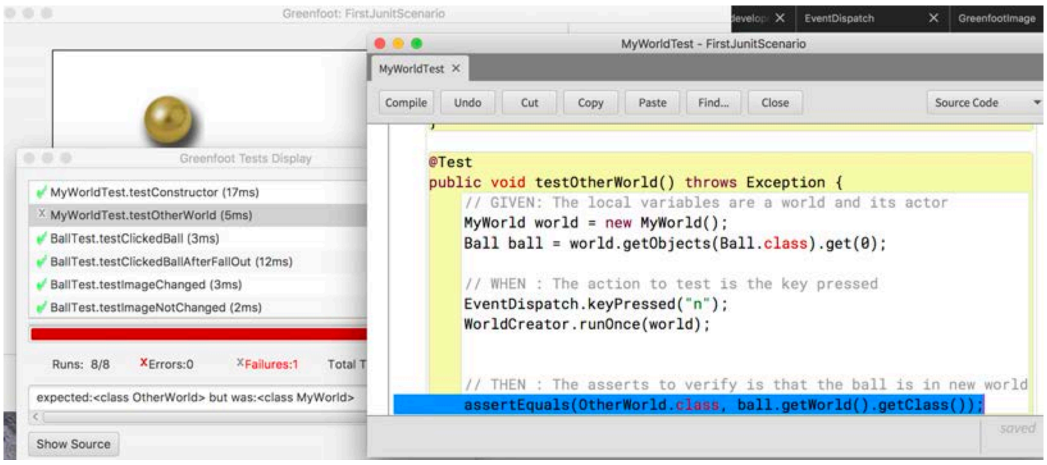
Class compiled - no syntax errors

```
✓ MyWorldTest.test (18ms)
✓ BallTest.testClickedBall (3ms)
✓ BallTest.testClickedBallAfterFallOut (3ms)
✓ BallTest.testImageChanged (5ms)
✓ BallTest.testImageNotChanged (2ms)
✓ BallTest.testOneAct (2ms)

Runs: 7/7    ×Errors:0    ×Failures:0    Total Time: 35ms
```

A **GREEN** result of the tests does not ensure that the requirement is fulfilled completely. The following exercise can be proposed here: "Design a test for the case of two consecutive act () cycles where the random numbers that are generated are between 0 and 19, check that the result is **RED**, and reprogram the Ball class so that the result of all the tests is **GREEN** ".

Another world appears when the "n" key is pressed

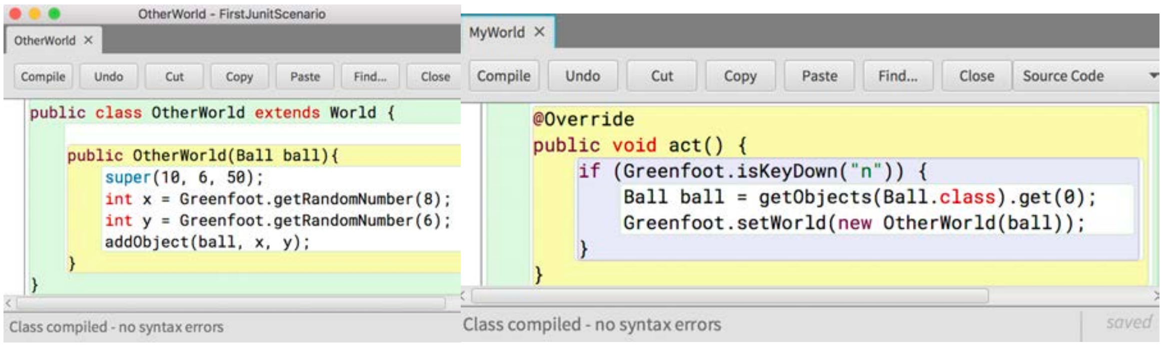
Now TDD cycle can be begun with the next requirement: "*When the active world in the scenario is an object of the MyWorld class and the "n" key is pressed, in this scenario a new world of the OtherWorld class is activated*". This requirement is possible because a scenario can have several classes that derive from the ***greenfoot.World*** class and the ***Greenfoot.setWorld***(...) method allows to activate a new world during the execution of the scenario. The **RED** step of TDD cycle can write with the next test.



```
MyWorldTest.testConstructor (17ms)
MyWorldTest.testOtherWorld (5ms)
BallTest.testClickedBall (3ms)
BallTest.testClickedBallAfterFallOut (12ms)
BallTest.testImageChanged (3ms)
BallTest.testImageNotChanged (2ms)

Runs: 8/8    Errors:0    Failures:1    Total T

expected:<class OtherWorld> but was:<class MyWorld>
```

```
@Test
public void testOtherWorld() throws Exception {
    // GIVEN: The local variables are a world and its actor
    MyWorld world = new MyWorld();
    Ball ball = world.getObjects(Ball.class).get(0);

    // WHEN : The action to test is the key pressed
    EventDispatch.keyPressed("n");
    WorldCreator.runOnce(world);


    // THEN : The asserts to verify is that the ball is in new world
    assertEquals(OtherWorld.class, ball.getWorld().getClass());
```

After checking that the designed test is correct, in other words, the result is **RED** and the message that this test gives is as expected, the **GREEN** step can be performed. In this TDD cycle, a new class must be created, the ***OtherWorld*** class, and the ***act()*** method must be added to the ***MyWorld*** class, where the object of the ***OtherWorld*** class will be created when you press the "n" key on the keyboard.



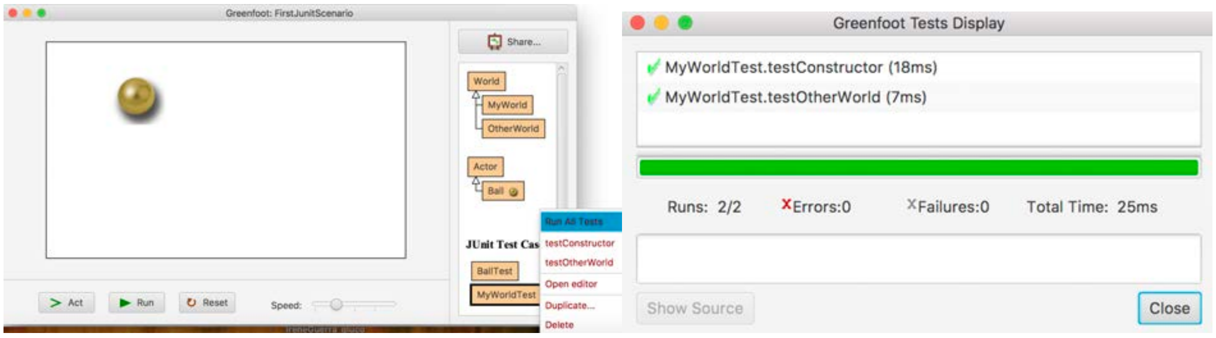
```
public class OtherWorld extends World {

    public OtherWorld(Ball ball){
        super(10, 6, 50);
        int x = Greenfoot.getRandomNumber(8);
        int y = Greenfoot.getRandomNumber(6);
        addObject(ball, x, y);
    }
}
```

Class compiled - no syntax errors

```
@Override
public void act() {
    if (Greenfoot.isKeyDown("n")) {
        Ball ball = getObjects(Ball.class).get(0);
        Greenfoot.setWorld(new OtherWorld(ball));
    }
}
```

Class compiled - no syntax errors

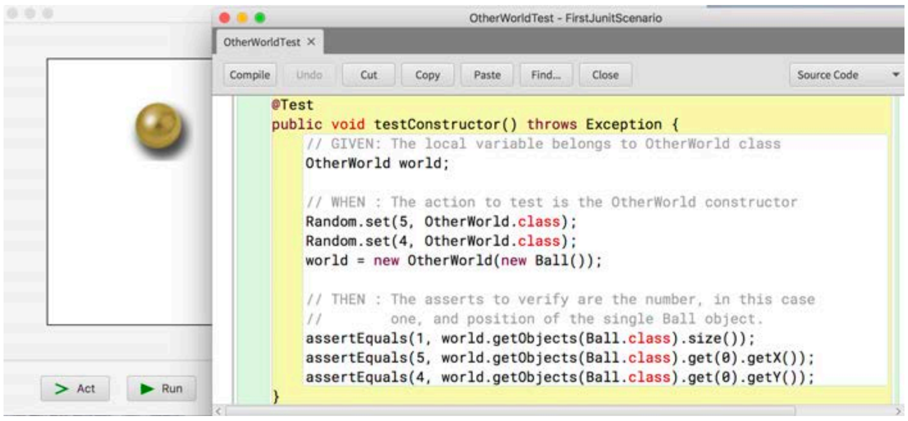
The Greenfoot project that is being programmed in this document already has many tests and, although all the tests must always be executed before moving on to the next TDD cycle, in the middle of the TDD cycle it is sometimes clearer to only execute the tests that are directly related. To achieve this, the Greenfoot IDE allows to select only one test or all the tests of the same class to be executed.



```
MyWorldTest.testConstructor (18ms)
MyWorldTest.testOtherWorld (7ms)

Runs: 2/2    Errors:0    Failures:0    Total Time: 25ms
```

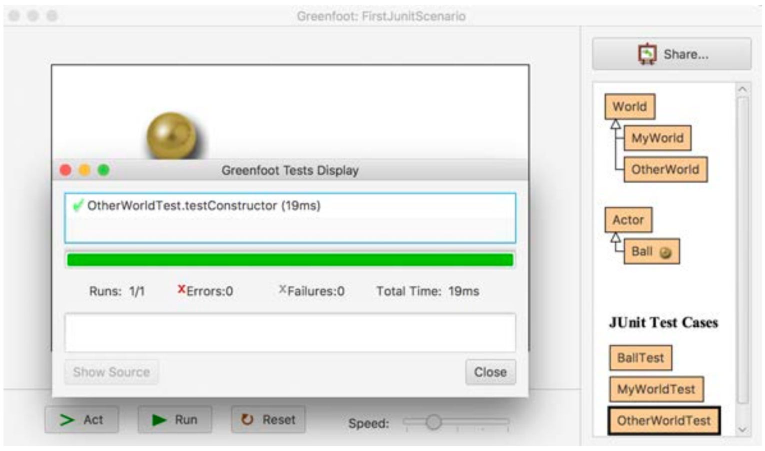
Test the constructor of a world that uses random numbers.

In the previous TDD cycle, the ***OtherWorld*** class has been added to the project and the construction of an object of the new class has been verified, fulfilling the requirement of the TDD cycle that has just been completed. However, there is not yet a test that directly verifies that the construction of an object of the ***OtherWorld*** class is correct. The test that is going to be performed should give a **GREEN** result and therefore we will not be doing a new TDD cycle, but we will be in the **REFACTORING** step of the previous TDD cycle.



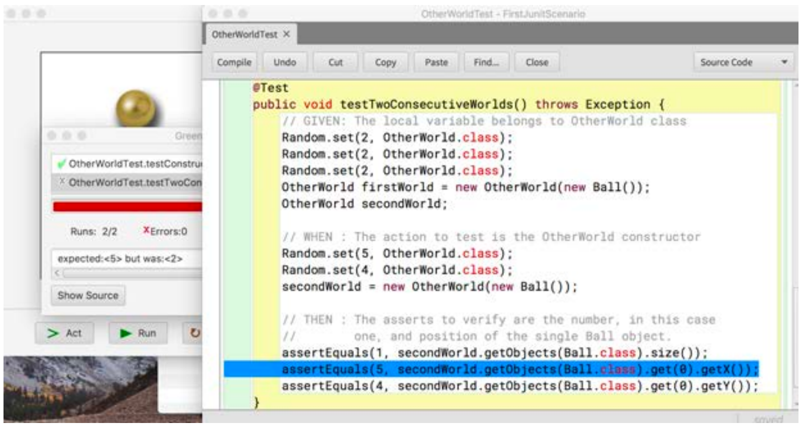
```java
@Test
public void testConstructor() throws Exception {
    // GIVEN: The local variable belongs to OtherWorld class
    OtherWorld world;

    // WHEN : The action to test is the OtherWorld constructor
    Random.set(5, OtherWorld.class);
    Random.set(4, OtherWorld.class);
    world = new OtherWorld(new Ball());

    // THEN : The asserts to verify are the number, in this case
    //        one, and position of the single Ball object.
    assertEquals(1, world.getObjects(Ball.class).size());
    assertEquals(5, world.getObjects(Ball.class).get(0).getX());
    assertEquals(4, world.getObjects(Ball.class).get(0).getY());
}
```

The ***Random*** package is used to generate the two random numbers since the constructor de ***OtherWorld*** needs these random numbers as the coordinates of the ball added to the world, so that the result of the test will be **GREEN**.



```
✓ OtherWorldTest.testConstructor (19ms)

Runs: 1/1    Errors:0    Failures:0    Total Time: 19ms
```

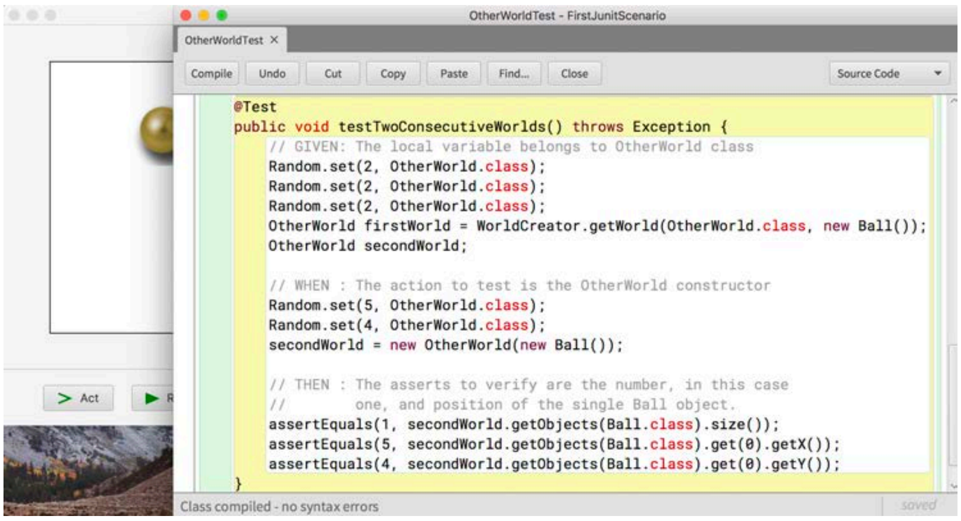
However, the construction of a world that needs random numbers does not work well when random numbers were previously generated that have not been used. This is the case of the next test where three random numbers have been generated to create the first world, that only two of them are used, and then a second world need be created with two different random numbers.



```
✓ OtherWorldTest.testConstru
× OtherWorldTest.testTwoCon

Runs: 2/2    Errors:0

expected:<5> but was:<2>
```

```java
@Test
public void testTwoConsecutiveWorlds() throws Exception {
    // GIVEN: The local variable belongs to OtherWorld class
    Random.set(2, OtherWorld.class);
    Random.set(2, OtherWorld.class);
    Random.set(2, OtherWorld.class);
    OtherWorld firstWorld = new OtherWorld(new Ball());
    OtherWorld secondWorld;

    // WHEN : The action to test is the OtherWorld constructor
    Random.set(5, OtherWorld.class);
    Random.set(4, OtherWorld.class);
    secondWorld = new OtherWorld(new Ball());

    // THEN : The asserts to verify are the number, in this case
    //        one, and position of the single Ball object.
    assertEquals(1, secondWorld.getObjects(Ball.class).size());
    assertEquals(5, secondWorld.getObjects(Ball.class).get(0).getX());
    assertEquals(4, secondWorld.getObjects(Ball.class).get(0).getY());
}
```

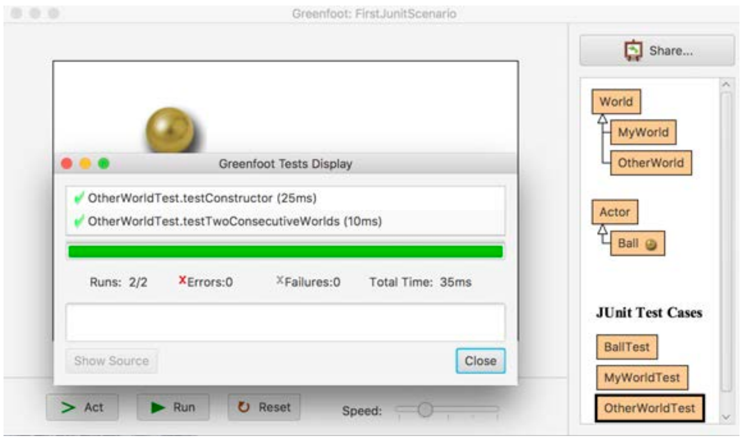
The ***WorldCreator*** package offers the ***getWorld(...)*** method to create objects that are derived from the world class and that require events in their construction. This method cleans the buffers used to simulate events, preventing the following actions from being affected by unused data. To this method, in addition to the classes of the world that is going to be built, you will be given the list of parameters that your constructor needs.



```
@Test
public void testTwoConsecutiveWorlds() throws Exception {
    // GIVEN: The local variable belongs to OtherWorld class
    Random.set(2, OtherWorld.class);
    Random.set(2, OtherWorld.class);
    Random.set(2, OtherWorld.class);
    OtherWorld firstWorld = WorldCreator.getWorld(OtherWorld.class, new Ball());
    OtherWorld secondWorld;

    // WHEN : The action to test is the OtherWorld constructor
    Random.set(5, OtherWorld.class);
    Random.set(4, OtherWorld.class);
    secondWorld = new OtherWorld(new Ball());

    // THEN : The asserts to verify are the number, in this case
    //        one, and position of the single Ball object.
    assertEquals(1, secondWorld.getObjects(Ball.class).size());
    assertEquals(5, secondWorld.getObjects(Ball.class).get(0).getX());
    assertEquals(4, secondWorld.getObjects(Ball.class).get(0).getY());
}
```

Class compiled - no syntax errors

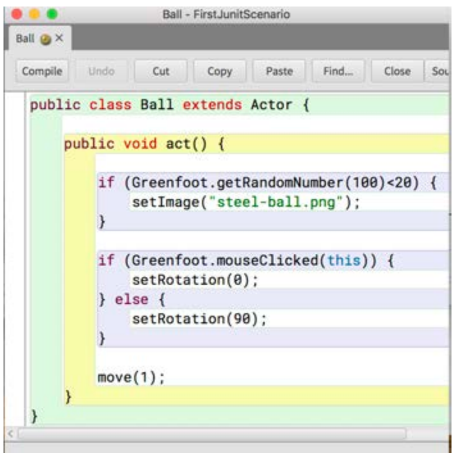
The result is that now the second world is created with the two desired random numbers, ignoring the random number that was not used in the construction of the first world.



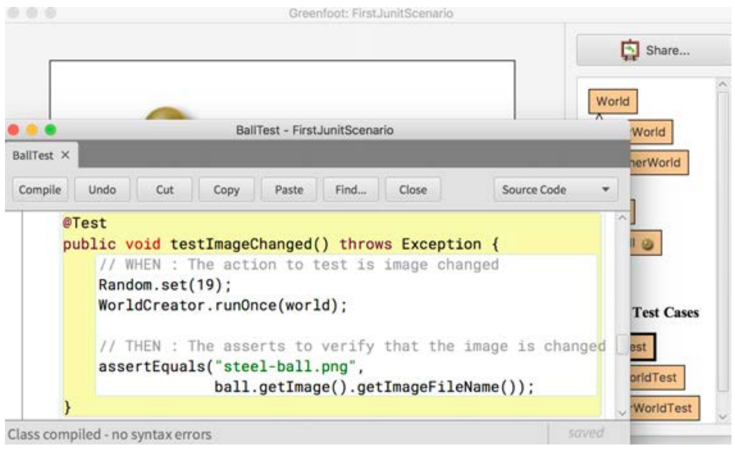
```
✓ OtherWorldTest.testConstructor (25ms)
✓ OtherWorldTest.testTwoConsecutiveWorlds (10ms)

Runs: 2/2    Errors:0    Failures:0    Total Time: 35ms
```

The *act()* method can often be called directly

Let’s remember the ball behavior associated with mouse clicked: “*There is a 20% chance that a ball will change its color before moving*”.



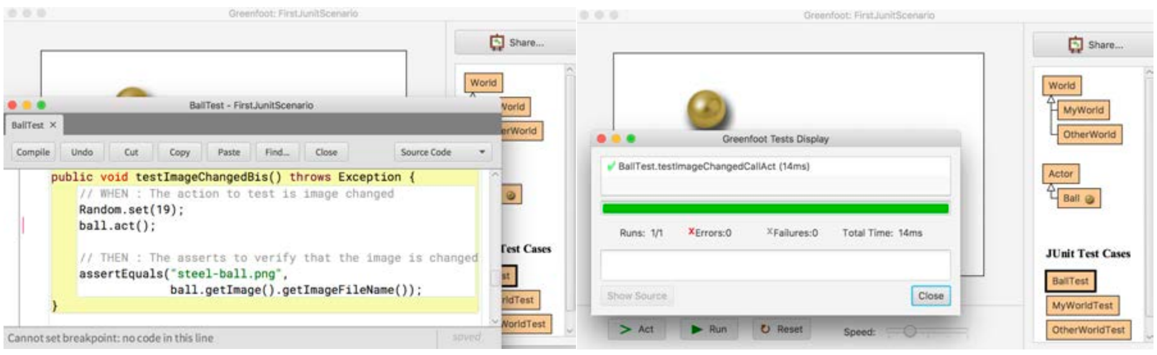
```
public class Ball extends Actor {

    public void act() {

        if (Greenfoot.getRandomNumber(100)<20) {
            setImage("steel-ball.png");
        }

        if (Greenfoot.mouseClicked(this)) {
            setRotation(0);
        } else {
            setRotation(90);
        }

        move(1);
    }
}
```



```
@Test
public void testImageChanged() throws Exception {
    // WHEN : The action to test is image changed
    Random.set(19);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify that the image is changed
    assertEquals("steel-ball.png",
                 ball.getImage().getImageFileName());
}
```

Class compiled - no syntax errors

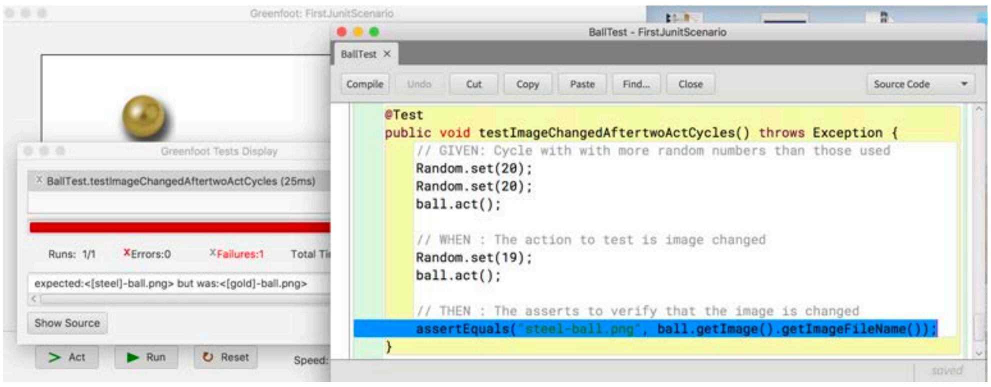
The previous test, that verified image change behavior, can be programmed calling the ***act()*** method of ball.



```
public void testImageChangedBis() throws Exception {
    // WHEN : The action to test is image changed
    Random.set(19);
    ball.act();

    // THEN : The asserts to verify that the image is changed
    assertEquals("steel-ball.png",
                 ball.getImage().getImageFileName());
}
```

```
BallTest.testImageChangedCallAct (14ms)
Runs: 1/1    Errors:0    Failures:0    Total Time: 14ms
```

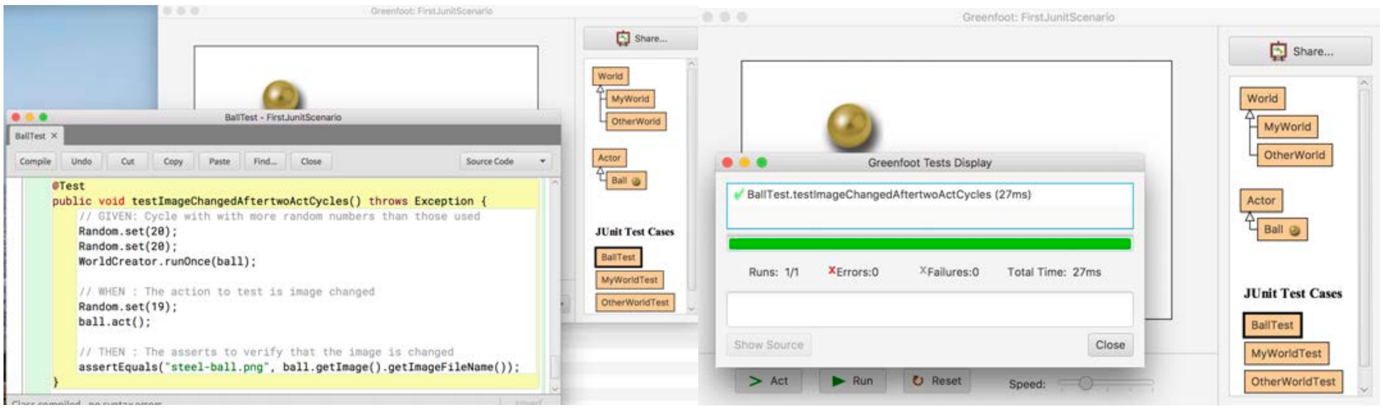
The direct call to the ***act()*** method does not always work correctly because there may be data from the simulation of events that interfere with the part of the test that remains to be executed. The following test shows the generation of two random numbers and the call to an ***act()*** that only uses one of these numbers. The unused random number remains in the pending events queue waiting for a call to request it. For this reason, the next call to ***act()*** does not work as expected by the programmer because instead of using the random number defined in the test just before, it uses that number that was not used in the previous call.



```
@Test
public void testImageChangedAftertwoActCycles() throws Exception {
    // GIVEN: Cycle with with more random numbers than those used
    Random.set(20);
    Random.set(20);
    ball.act();

    // WHEN : The action to test is image changed
    Random.set(19);
    ball.act();

    // THEN : The asserts to verify that the image is changed
    assertEquals("steel-ball.png", ball.getImage().getImageFileName());
}
```

```
BallTest.testImageChangedAftertwoActCycles (25ms)
Runs: 1/1    Errors:0    Failures:1
expected:<[steel]-ball.png> but was:<[gold]-ball.png>
```

Although this simple example is easy to arrange, defining a single random number for the first ***act()***, it is not always so obvious because the events can be many and generated in the program code. The ***runOnce***() method of the ***WorldCreator*** class cleans the event buffers. For this reason, although a test that directly calls the ***act()*** method may seem clearer, it is safer to use ***runOnce()*** to call the ***act()*** method. There are several versions of ***runOnce()***: you can execute the act of a single actor (this is the version that was used in **`testImageChangedAftertwoActCycles`**) , or you can execute the act of a list of actors, or you can execute the act of a list of actors and their world, or you can execute the act of all the actors and their world (this is the version that is most used in the tests of this tutorial).



```
@Test
public void testImageChangedAftertwoActCycles() throws Exception {
    // GIVEN: Cycle with with more random numbers than those used
    Random.set(20);
    Random.set(20);
    WorldCreator.runOnce(ball);

    // WHEN : The action to test is image changed
    Random.set(19);
    ball.act();

    // THEN : The asserts to verify that the image is changed
    assertEquals("steel-ball.png", ball.getImage().getImageFileName());
}
```

```
BallTest.testImageChangedAftertwoActCycles (27ms)
Runs: 1/1    Errors:0    Failures:0    Total Time: 27ms
```

In the source of the project that has been programmed in this tutorial you can see the two possible versions of each test: one that calls ***act()*** through the call to ***runOnce()*** and another that calls directly to ***act()***.

Project sources

MyWorld class

```java
import greenfoot.*;

public class MyWorld extends World {

    public MyWorld() {
        super(10, 6, 50);
        addObject(new Ball(), 2, 1);
    }

    @Override
    public void act() {
        if (Greenfoot.isKeyDown("n")) {
            Ball ball = getObjects(Ball.class).get(0);
            Greenfoot.setWorld(new OtherWorld(ball));
        }
    }
}
```

OtherWorld class

```java
import greenfoot.*;

public class OtherWorld extends World {

    public OtherWorld(Ball ball){
        super(10, 6, 50);
        int x = Greenfoot.getRandomNumber(8);
        int y = Greenfoot.getRandomNumber(6);
        addObject(ball, x, y);
    }
}
```

Ball class

```java
import greenfoot.*;

public class Ball extends Actor {

    public void act() {

        if (Greenfoot.getRandomNumber(100)<20) {
            setImage("steel-ball.png");
        }

        if (Greenfoot.mouseClicked(this)) {
            setRotation(0);

        } else {
            setRotation(90);
        }

        move(1);
    }
}
```

Test cases of MyWorld class for JUnit4 version

```java
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import greenfoot.junitUtils.runner.GreenfootRunner;
import greenfoot.junitUtils.*;

@RunWith(GreenfootRunner.class)
public class MyWorldTest {

    // Common variables to all tests are declared here.

    /**
     * Sets up the test fixture
     * Called before every test case method.
     */
    @Before
    public void setUp() throws Exception {
        // Common variables to all tests are initialized here.
    }

    @Test
    public void testConstructor() throws Exception {
        // GIVEN: The local variable belongs to MyWorld class
        MyWorld world;

        // WHEN : The action to test is the MyWorld constructor
        world = new MyWorld();

        // THEN : The asserts to verify are the number, in this case
        //        one, and position of the single Ball object.
        assertEquals(1, world.getObjects(Ball.class).size());
        assertEquals(2, world.getObjects(Ball.class).get(0).getX());
        assertEquals(1, world.getObjects(Ball.class).get(0).getY());
    }

    @Test
    public void testOtherWorld() throws Exception {
        // GIVEN: The local variables are a world and its actor
        MyWorld world = new MyWorld();
        Ball ball = world.getObjects(Ball.class).get(0);

        // WHEN : The action to test is the key pressed
        EventDispatch.keyPressed("n");
        WorldCreator.runOnce(world);


        // THEN : The asserts to verify is that the ball is in new world
        assertEquals(OtherWorld.class, ball.getWorld().getClass());
    }

    @Test
    public void testOtherWorldBis() throws Exception {
        // GIVEN: The local variables are a world and its actor
        MyWorld world = new MyWorld();
        Ball ball = world.getObjects(Ball.class).get(0);

        // WHEN : The action to test is the key pressed
        EventDispatch.keyPressed("n");
        world.act();


        // THEN : The asserts to verify is that the ball is in new world
        assertEquals(OtherWorld.class, ball.getWorld().getClass());
    }
}
```

Test cases of OtherWorld class for JUnit4 version

```java
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import greenfoot.junitUtils.runner.GreenfootRunner;
import greenfoot.junitUtils.*;

@RunWith(GreenfootRunner.class)
public class OtherWorldTest {

    // Common variables to all tests are declared here.

    /**
     * Sets up the test fixture
     * Called before every test case method.
     */
    @Before
    public void setUp() throws Exception {
        // Common variables to all tests are initialized here.
    }

    @Test
    public void testConstructor() throws Exception {
        // GIVEN: The local variable belongs to OtherWorld class
        OtherWorld world;

        // WHEN : The action to test is the OtherWorld constructor
        Random.set(5, OtherWorld.class);
        Random.set(4, OtherWorld.class);
        world = new OtherWorld(new Ball());

        // THEN : The asserts to verify are the number, in this case
        //        one, and position of the single Ball object.
        assertEquals(1, world.getObjects(Ball.class).size());
        assertEquals(5, world.getObjects(Ball.class).get(0).getX());
        assertEquals(4, world.getObjects(Ball.class).get(0).getY());
    }

    @Test
    public void testTwoConsecutiveWorlds() throws Exception {
        // GIVEN: The local variable belongs to OtherWorld class
        Random.set(2, OtherWorld.class);
        Random.set(2, OtherWorld.class);
        Random.set(2, OtherWorld.class);
        OtherWorld firstWorld = WorldCreator.getWorld(OtherWorld.class, new Ball());
        OtherWorld secondWorld;

        // WHEN : The action to test is the OtherWorld constructor
        Random.set(5, OtherWorld.class);
        Random.set(4, OtherWorld.class);
        secondWorld = new OtherWorld(new Ball());

        // THEN : The asserts to verify are the number, in this case
        //        one, and position of the single Ball object.
        assertEquals(1, secondWorld.getObjects(Ball.class).size());
        assertEquals(5, secondWorld.getObjects(Ball.class).get(0).getX());
        assertEquals(4, secondWorld.getObjects(Ball.class).get(0).getY());
    }
}
```

Test cases of Ball class for JUnit4 version

```java
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import greenfoot.junitUtils.runner.GreenfootRunner;
import greenfoot.junitUtils.*;

@RunWith(GreenfootRunner.class)
public class BallTest {

    MyWorld world;
    Ball ball;

    @Before
    public void setUp() throws Exception {
        // GIVEN: The local variables one world and one ball
        world = new MyWorld();
        ball = world.getObjects(Ball.class).get(0);
    }


    @Test
    public void testOneAct() throws Exception {
        // WHEN : The action to test is one act cycles
        WorldCreator.runOnce(world);

        // THEN : The asserts to verify is the ball position
        assertEquals(2, ball.getX());
        assertEquals(2, ball.getY());
    }

    @Test
    public void testOneActBis() throws Exception {
        // WHEN : The action to test is one act cycles
        ball.act();

        // THEN : The asserts to verify is the ball position
        assertEquals(2, ball.getX());
        assertEquals(2, ball.getY());
    }

    @Test
    public void testTwoAct() throws Exception {
        // WHEN : The action to test is two act cycles
        WorldCreator.runOnce(world);
        WorldCreator.runOnce(world);

        // THEN : The asserts to verify is the ball position
        assertEquals(2, ball.getX());
        assertEquals(3, ball.getY());
    }

    @Test
    public void testTwoActBis() throws Exception {
        // WHEN : The action to test is two act cycles
        ball.act();
        ball.act();

        // THEN : The asserts to verify is the ball position
        assertEquals(2, ball.getX());
        assertEquals(3, ball.getY());
    }
```

```java
@Test
public void testClickedBall() throws Exception {
    // WHEN : The action to test is to click the ball
    EventDispatch.mouseClicked(2, 1);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify is the ball position
    assertEquals(3, ball.getX());
    assertEquals(1, ball.getY());
}

@Test
public void testClickedBallBis() throws Exception {
    // WHEN : The action to test is to click the ball
    EventDispatch.mouseClicked(2, 1);
    ball.act();

    // THEN : The asserts to verify is the ball position
    assertEquals(3, ball.getX());
    assertEquals(1, ball.getY());
}

@Test
public void testClickedBallAfterFallOut() throws Exception {
    // WHEN : The action to test is to click the ball
    WorldCreator.runOnce(world);
    EventDispatch.mouseClicked(2, 2);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify is the ball position
    assertEquals(3, ball.getX());
    assertEquals(2, ball.getY());
}

@Test
public void testClickedBallAfterFallOutBis() throws Exception {
    // WHEN : The action to test is to click the ball
    ball.act();
    EventDispatch.mouseClicked(2, 2);
    ball.act();

    // THEN : The asserts to verify is the ball position
    assertEquals(3, ball.getX());
    assertEquals(2, ball.getY());
}

@Test
public void testImageChanged() throws Exception {
    // WHEN : The action to test is image changed
    Random.set(19);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify that the image is changed
    assertEquals("steel-ball.png", ball.getImage().getImageFileName());
}

@Test
public void testImageChangedBis() throws Exception {
    // WHEN : The action to test is image changed
    Random.set(19);
    ball.act();

    // THEN : The asserts to verify that the image is changed
    assertEquals("steel-ball.png", ball.getImage().getImageFileName());
}
```

```
    @Test
    public void testImageNotChanged() throws Exception {
        // WHEN : The action to test is image changed
        Random.set(20);
        WorldCreator.runOnce(world);

        // THEN : The asserts to verify is that it is the same image
        assertEquals("gold-ball.png", ball.getImage().getImageFileName());
    }

    @Test
    public void testImageNotChangedBis() throws Exception {
        // WHEN : The action to test is image changed
        Random.set(20);
        ball.act();

        // THEN : The asserts to verify is that it is the same image
        assertEquals("gold-ball.png", ball.getImage().getImageFileName());
    }

    @Test
    public void testImageChangedAftertwoActCycles() throws Exception {
        // GIVEN:
        Random.set(20);
        Random.set(20);
        WorldCreator.runOnce(world);

        // WHEN : The action to test is image changed
        Random.set(19);
        ball.act();

        // THEN : The asserts to verify that the image is changed
        assertEquals("steel-ball.png", ball.getImage().getImageFileName());
    }


}
```

Test cases of MyWorld class for JUnit5 version

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import greenfoot.junitUtils.jupiter.runner.GreenfootRunner;
import greenfoot.junitUtils.*;

@ExtendWith(GreenfootRunner.class)
class MyWorldTest {

    // Common variables to all tests are declared here.

    /**
     * Sets up the test fixture
     * Called before every test case method.
     */
    @BeforeEach
    void setUp() throws Exception {
        // Common variables to all tests are initialized here.
    }

    @Test
    void testConstructor() throws Exception {
        // GIVEN: The local variable belongs to MyWorld class
        MyWorld world;

        // WHEN : The action to test is the MyWorld constructor
        world = new MyWorld();

        // THEN : The asserts to verify are the number, in this case
        //        one, and position of the single Ball object.
        assertEquals(1, world.getObjects(Ball.class).size());
        assertEquals(2, world.getObjects(Ball.class).get(0).getX());
        assertEquals(1, world.getObjects(Ball.class).get(0).getY());
    }

    @Test
    void testOtherWorld() throws Exception {
        // GIVEN: The local variables are a world and its actor
        MyWorld world = new MyWorld();
        Ball ball = world.getObjects(Ball.class).get(0);

        // WHEN : The action to test is the key pressed
        EventDispatch.keyPressed("n");
        WorldCreator.runOnce(world);


        // THEN : The asserts to verify is that the ball is in new world
        assertEquals(OtherWorld.class, ball.getWorld().getClass());
    }

    @Test
    void testOtherWorldBis() throws Exception {
        // GIVEN: The local variables are a world and its actor
        MyWorld world = new MyWorld();
        Ball ball = world.getObjects(Ball.class).get(0);

        // WHEN : The action to test is the key pressed
        EventDispatch.keyPressed("n");
        world.act();


        // THEN : The asserts to verify is that the ball is in new world
        assertEquals(OtherWorld.class, ball.getWorld().getClass());
    }
}
```

Test cases of OtherWorld class for JUnit5 version

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import greenfoot.junitUtils.jupiter.runner.GreenfootRunner;
import greenfoot.junitUtils.*;

@ExtendWith(GreenfootRunner.class)
class OtherWorldTest {

    // Common variables to all tests are declared here.

    /**
     * Sets up the test fixture
     * Called before every test case method.
     */
    @BeforeEach
    void setUp() throws Exception {
        // Common variables to all tests are initialized here.
    }

    @Test
    void testConstructor() throws Exception {
        // GIVEN: The local variable belongs to OtherWorld class
        OtherWorld world;

        // WHEN : The action to test is the OtherWorld constructor
        Random.set(5, OtherWorld.class);
        Random.set(4, OtherWorld.class);
        world = new OtherWorld(new Ball());

        // THEN : The asserts to verify are the number, in this case
        //        one, and position of the single Ball object.
        assertEquals(1, world.getObjects(Ball.class).size());
        assertEquals(5, world.getObjects(Ball.class).get(0).getX());
        assertEquals(4, world.getObjects(Ball.class).get(0).getY());
    }

    @Test
    void testTwoConsecutiveWorlds() throws Exception {
        // GIVEN: The local variable belongs to OtherWorld class
        Random.set(2, OtherWorld.class);
        Random.set(2, OtherWorld.class);
        Random.set(2, OtherWorld.class);
        OtherWorld firstWorld = WorldCreator.getWorld(OtherWorld.class, new Ball());
        OtherWorld secondWorld;

        // WHEN : The action to test is the OtherWorld constructor
        Random.set(5, OtherWorld.class);
        Random.set(4, OtherWorld.class);
        secondWorld = new OtherWorld(new Ball());

        // THEN : The asserts to verify are the number, in this case
        //        one, and position of the single Ball object.
        assertEquals(1, secondWorld.getObjects(Ball.class).size());
        assertEquals(5, secondWorld.getObjects(Ball.class).get(0).getX());
        assertEquals(4, secondWorld.getObjects(Ball.class).get(0).getY());
    }
}
```

Test cases of Ball class for JUnit5 version

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import greenfoot.junitUtils.jupiter.runner.GreenfootRunner;
import greenfoot.junitUtils.*;

@ExtendWith(GreenfootRunner.class)
class BallTest {

    MyWorld world;
    Ball ball;

    @BeforeEach
    void setUp() throws Exception {
        // GIVEN: The local variables one world and one ball
        world = new MyWorld();
        ball = world.getObjects(Ball.class).get(0);
    }


    @Test
    void testOneAct() throws Exception {
        // WHEN : The action to test is one act cycles
        WorldCreator.runOnce(world);

        // THEN : The asserts to verify is the ball position
        assertEquals(2, ball.getX());
        assertEquals(2, ball.getY());
    }

    @Test
    void testOneActBis() throws Exception {
        // WHEN : The action to test is one act cycles
        ball.act();

        // THEN : The asserts to verify is the ball position
        assertEquals(2, ball.getX());
        assertEquals(2, ball.getY());
    }

    @Test
    void testTwoAct() throws Exception {
        // WHEN : The action to test is two act cycles
        WorldCreator.runOnce(world);
        WorldCreator.runOnce(world);

        // THEN : The asserts to verify is the ball position
        assertEquals(2, ball.getX());
        assertEquals(3, ball.getY());
    }

    @Test
    void testTwoActBis() throws Exception {
        // WHEN : The action to test is two act cycles
        ball.act();
        ball.act();

        // THEN : The asserts to verify is the ball position
        assertEquals(2, ball.getX());
        assertEquals(3, ball.getY());
    }
```

```java
@Test
void testClickedBall() throws Exception {
    // WHEN : The action to test is to click the ball
    EventDispatch.mouseClicked(2, 1);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify is the ball position
    assertEquals(3, ball.getX());
    assertEquals(1, ball.getY());
}

@Test
void testClickedBallBis() throws Exception {
    // WHEN : The action to test is to click the ball
    EventDispatch.mouseClicked(2, 1);
    ball.act();

    // THEN : The asserts to verify is the ball position
    assertEquals(3, ball.getX());
    assertEquals(1, ball.getY());
}

@Test
void testClickedBallAfterFallOut() throws Exception {
    // WHEN : The action to test is to click the ball
    WorldCreator.runOnce(world);
    EventDispatch.mouseClicked(2, 2);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify is the ball position
    assertEquals(3, ball.getX());
    assertEquals(2, ball.getY());
}

@Test
void testClickedBallAfterFallOutBis() throws Exception {
    // WHEN : The action to test is to click the ball
    ball.act();
    EventDispatch.mouseClicked(2, 2);
    ball.act();

    // THEN : The asserts to verify is the ball position
    assertEquals(3, ball.getX());
    assertEquals(2, ball.getY());
}

@Test
void testImageChanged() throws Exception {
    // WHEN : The action to test is image changed
    Random.set(19);
    WorldCreator.runOnce(world);

    // THEN : The asserts to verify that the image is changed
    assertEquals("steel-ball.png", ball.getImage().getImageFileName());
}

@Test
void testImageChangedBis() throws Exception {
    // WHEN : The action to test is image changed
    Random.set(19);
    ball.act();

    // THEN : The asserts to verify that the image is changed
    assertEquals("steel-ball.png", ball.getImage().getImageFileName());
}
```

```java
    @Test
    void testImageNotChanged() throws Exception {
        // WHEN : The action to test is image changed
        Random.set(20);
        WorldCreator.runOnce(world);

        // THEN : The asserts to verify is that it is the same image
        assertEquals("gold-ball.png", ball.getImage().getImageFileName());
    }

    @Test
    void testImageNotChangedBis() throws Exception {
        // WHEN : The action to test is image changed
        Random.set(20);
        ball.act();

        // THEN : The asserts to verify is that it is the same image
        assertEquals("gold-ball.png", ball.getImage().getImageFileName());
    }

    @Test
    void testImageChangedAftertwoActCycles() throws Exception {
        // GIVEN:
        Random.set(20);
        Random.set(20);
        WorldCreator.runOnce(world);

        // WHEN : The action to test is image changed
        Random.set(19);
        ball.act();

        // THEN : The asserts to verify that the image is changed
        assertEquals("steel-ball.png", ball.getImage().getImageFileName());
    }


}
```