# OOP with Java

## 27. Testing with JUnit

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

## Outline

website

## Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly

## Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly, including
    - debugging in Lesson 13: *Debugging*

## Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code into different methods which are smaller and thus easier to understand

## Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code into different methods which are smaller and thus easier to understand
  - dividing code into classes which bundle functionality and encapsulate (`private`) variables making sure that data can only be changed in a valid way

## Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code into different methods which are smaller and thus easier to understand
  - dividing code into classes which bundle functionality and encapsulate (`private`) variables making sure that data can only be changed in a valid way
  - dividing code focusing on different concerns into packages to make the whole project easier to understand

## Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly, including
    - debugging in Lesson 13: *Debugging*
    - dividing code into different methods which are smaller and thus easier to understand
    - dividing code into classes which bundle functionality and encapsulate (`private`) variables making sure that data can only be changed in a valid way
    - dividing code focusing on different concerns into packages to make the whole project easier to understand
    - using Java's utility classes as heavily as possible to avoid writing own code and making errors

## Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code into different methods which are smaller and thus easier to understand
  - dividing code into classes which bundle functionality and encapsulate (`private`) variables making sure that data can only be changed in a valid way
  - dividing code focusing on different concerns into packages to make the whole project easier to understand
  - using Java's utility classes as heavily as possible to avoid writing own code and making errors
  - using well-tested (open source) libraries for general tasks to reduce development time and chances to make errors

## Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code into different methods which are smaller and thus easier to understand
  - dividing code into classes which bundle functionality and encapsulate (`private`) variables making sure that data can only be changed in a valid way
  - dividing code focusing on different concerns into packages to make the whole project easier to understand
  - using Java's utility classes as heavily as possible to avoid writing own code and making errors
  - using well-tested (open source) libraries for general tasks to reduce development time and chances to make errors
- But we can still expect our code to contain errors

## Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code into different methods which are smaller and thus easier to understand
  - dividing code into classes which bundle functionality and encapsulate (`private`) variables making sure that data can only be changed in a valid way
  - dividing code focusing on different concerns into packages to make the whole project easier to understand
  - using Java's utility classes as heavily as possible to avoid writing own code and making errors
  - using well-tested (open source) libraries for general tasks to reduce development time and chances to make errors
- But we can still expect our code to contain errors
- We should test our code before shipping/using it

## Introduction

- We have already learned quite a few things that we can do to make sure that our code works correctly, including
  - debugging in Lesson 13: *Debugging*
  - dividing code into different methods which are smaller and thus easier to understand
  - dividing code into classes which bundle functionality and encapsulate (`private`) variables making sure that data can only be changed in a valid way
  - dividing code focusing on different concerns into packages to make the whole project easier to understand
  - using Java's utility classes as heavily as possible to avoid writing own code and making errors
  - using well-tested (open source) libraries for general tasks to reduce development time and chances to make errors
- But we can still expect our code to contain errors
- We should test our code before shipping/using it
- How can we do that in a structured, automated way?

## Unit Testing with JUnit

- Unit testing is a software testing method in which individual units of code are tested whether they meet the specification

## Unit Testing with JUnit

- Unit testing is a software testing method in which individual units of code are tested whether they meet the specification
- JUnit is a software framework for unit testing in Java

## Unit Testing with JUnit

- Unit testing is a software testing method in which individual units of code are tested whether they meet the specification
- JUnit is a software framework for unit testing in Java
- An application normally consists of a main classes to execute and several utility classes

## Unit Testing with JUnit

- Unit testing is a software testing method in which individual units of code are tested whether they meet the specification
- JUnit is a software framework for unit testing in Java
- An application normally consists of a main classes to execute and several utility classes
- If we use JUnit, we add a new form of classes: Tests

## Unit Testing with JUnit

- Unit testing is a software testing method in which individual units of code are tested whether they meet the specification
- JUnit is a software framework for unit testing in Java
- An application normally consists of a main classes to execute and several utility classes
- If we use JUnit, we add a new form of classes: Tests
- Usually, we make (at least one) test class for each class of the “real” code

## Unit Testing with JUnit

- Unit testing is a software testing method in which individual units of code are tested whether they meet the specification
- JUnit is a software framework for unit testing in Java
- An application normally consists of a main classes to execute and several utility classes
- If we use JUnit, we add a new form of classes: Tests
- Usually, we make (at least one) test class for each class of the “real” code
- The methods of this test class are “test cases”, each checking one aspect of the “real code”

## Unit Testing with JUnit

- Unit testing is a software testing method in which individual units of code are tested whether they meet the specification
- JUnit is a software framework for unit testing in Java
- An application normally consists of a main classes to execute and several utility classes
- If we use JUnit, we add a new form of classes: Tests
- Usually, we make (at least one) test class for each class of the “real” code
- The methods of this test class are “test cases”, each checking one aspect of the “real code”
- JUnit comes as two libraries (can be downloaded from `http://junit.org`) and with Eclipse integration

## Unit Testing with JUnit

- Unit testing is a software testing method in which individual units of code are tested whether they meet the specification
- JUnit is a software framework for unit testing in Java
- An application normally consists of a main classes to execute and several utility classes
- If we use JUnit, we add a new form of classes: Tests
- Usually, we make (at least one) test class for each class of the "real" code
- The methods of this test class are "test cases", each checking one aspect of the "real code"
- JUnit comes as two libraries (can be downloaded from `http://junit.org`) and with Eclipse integration
- We will step-by-step explore its use in Eclipse

## Enabling JUnit Support

- Eclipse provides direct, first-class JUnit support

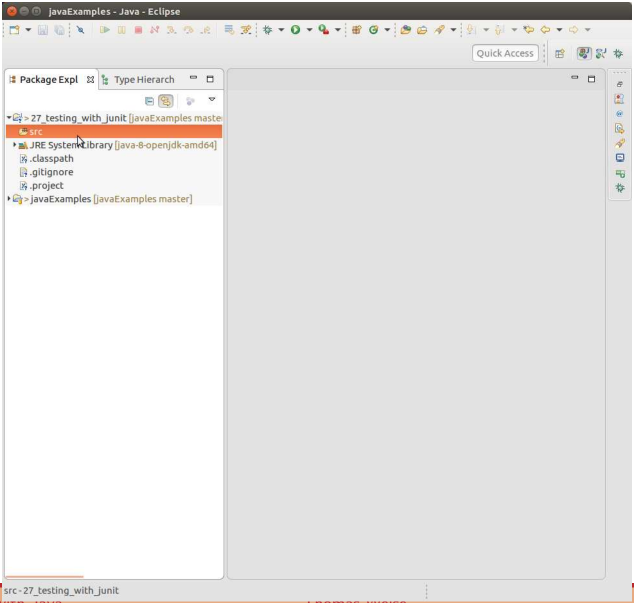
## Enabling JUnit Support

- Eclipse provides direct, first-class JUnit support
- To enable this support by adding JUnit to the build path, we do *not* need to download the JUnit `jar`s or anything. . .

## Enabling JUnit Support

- Eclipse provides direct, first-class JUnit support
- To enable this support by adding JUnit to the build path, we do *not* need to download the JUnit `jar`s or anything. . .
- First, we create a new empty Java project, let's call it `27_testing_with_junit`

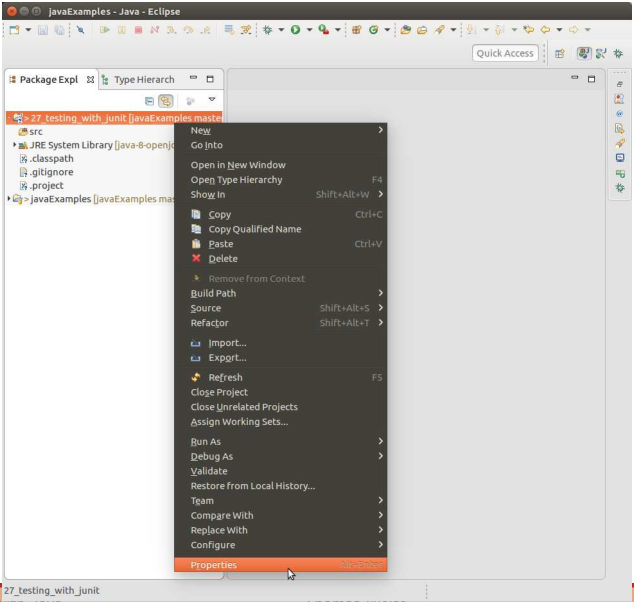
## Enabling JUnit Support


javaExamples - Java - Eclipse
Quick Access
Package Expl
Type Hierarch
27_testing_with_junit [javaExamples master]
src
JRE System Library [java-8-openjdk-amd64]
.classpath
.gitignore
.project
javaExamples [javaExamples master]
src - 27_testing_with_junit

## Enabling JUnit Support

- Eclipse provides direct, first-class JUnit support
- To enable this support by adding JUnit to the build path, we do *not* need to download the JUnit `jar`s or anything...
- First, we create a new empty Java project, let's call it `27_testing_with_junit`
- Then we right-click the project and click `Properties`

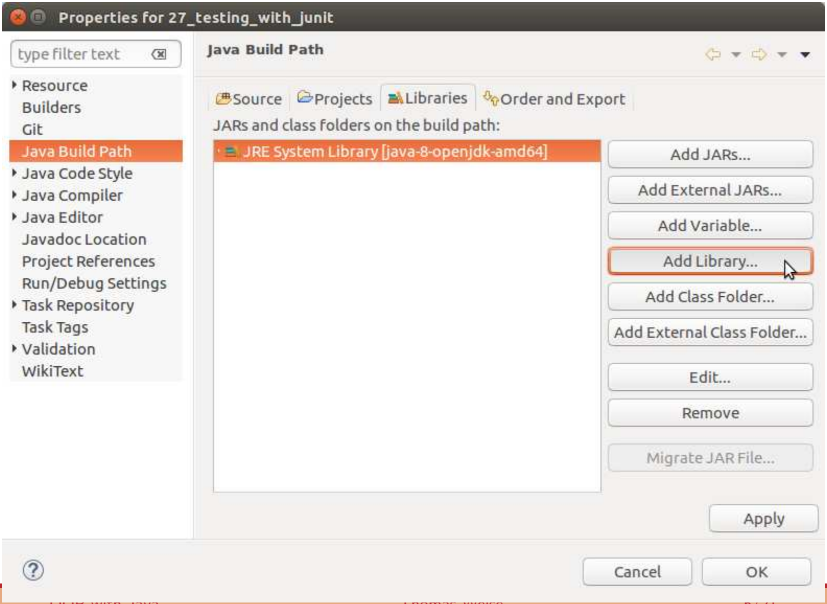
## Enabling JUnit Support



## Enabling JUnit Support

- Eclipse provides direct, first-class JUnit support
- To enable this support by adding JUnit to the build path, we do *not* need to download the JUnit `jar`s or anything...
- First, we create a new empty Java project, let's call it `27_testing_with_junit`
- Then we right-click the project and click `Properties`
- Under `Java Build Path` we select `Libraries` and click `Add Library...`

## Enabling JUnit Support



Properties for 27_testing_with_junit

type filter text

- Resource
- Builders
- Git
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Javadoc Location
- Project References
- Run/Debug Settings
- Task Repository
- Task Tags
- Validation
- WikiText

Java Build Path

Source | Projects | Libraries | Order and Export

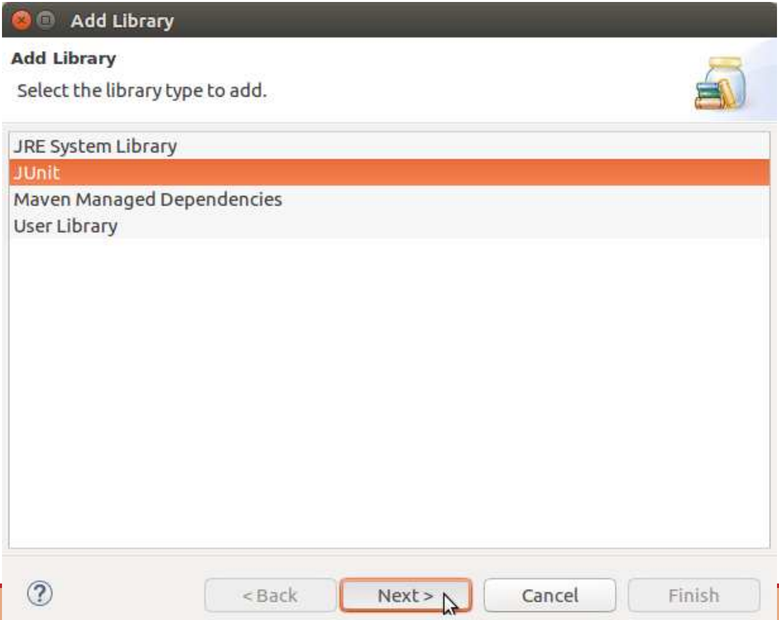
JARs and class folders on the build path:

JRE System Library [java-8-openjdk-amd64]

## Enabling JUnit Support

- Eclipse provides direct, first-class JUnit support
- To enable this support by adding JUnit to the build path, we do *not* need to download the JUnit `jar`s or anything...
- First, we create a new empty Java project, let's call it `27_testing_with_junit`
- Then we right-click the project and click `Properties`
- Under `Java Build Path` we select `Libraries` and click `Add Library...`
- We choose `JUnit` and click `Next`

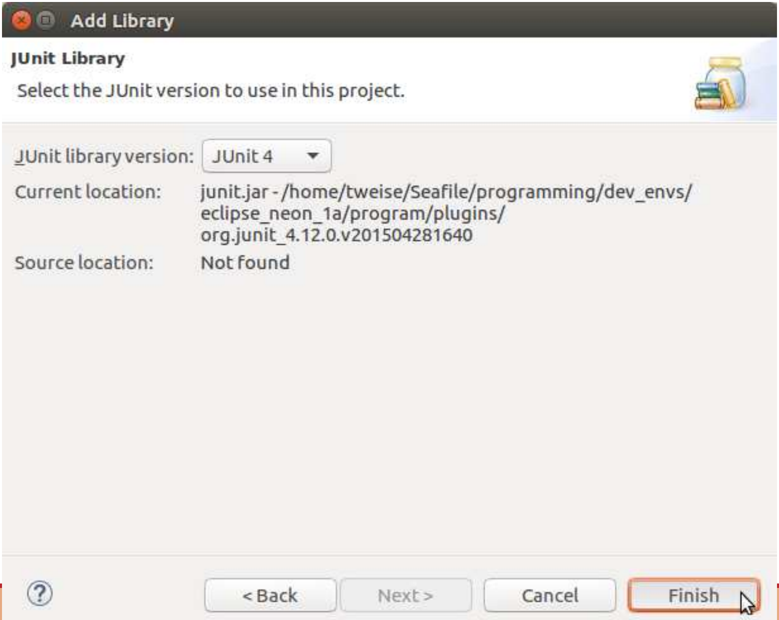
## Enabling JUnit Support



## Enabling JUnit Support

- Eclipse provides direct, first-class JUnit support
- To enable this support by adding JUnit to the build path, we do *not* need to download the JUnit `jar`s or anything...
- First, we create a new empty Java project, let's call it `27_testing_with_junit`
- Then we right-click the project and click `Properties`
- Under `Java Build Path` we select `Libraries` and click `Add Library...`
- We choose `JUnit` and click `Next`
- We choose `JUnit 4` and click `Finish`

## Enabling JUnit Support



JUnit Library

Select the JUnit version to use in this project.

JUnit library version: JUnit 4

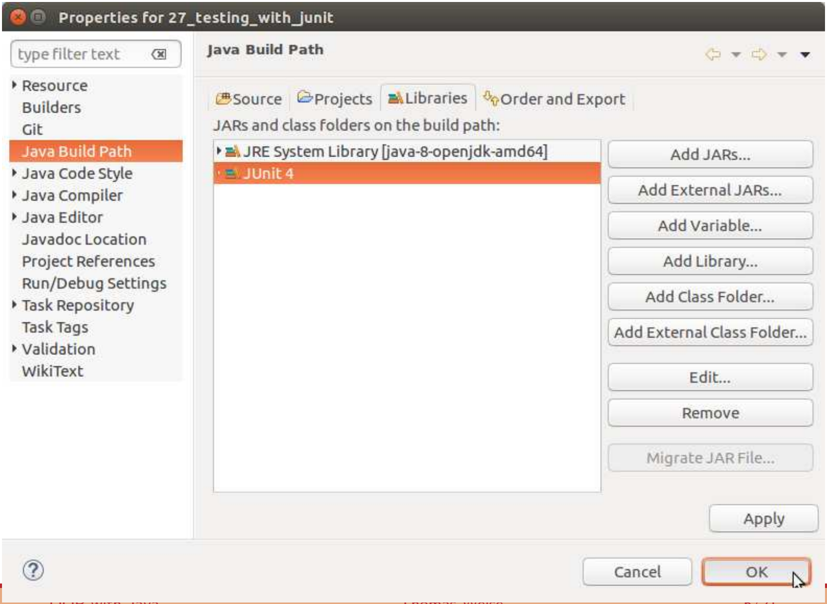
Current location: junit.jar - /home/tweise/Seafile/programming/dev_envs/eclipse_neon_1a/program/plugins/org.junit_4.12.0.v201504281640

Source location: Not found

## Enabling JUnit Support

- Eclipse provides direct, first-class JUnit support
- To enable this support by adding JUnit to the build path, we do *not* need to download the JUnit `jar`s or anything...
- First, we create a new empty Java project, let's call it `27_testing_with_junit`
- Then we right-click the project and click `Properties`
- Under `Java Build Path` we select `Libraries` and click `Add Library...`
- We choose `JUnit` and click `Next`
- We choose `JUnit 4` and click `Finish`
- We click `OK`

## Enabling JUnit Support



Properties for 27_testing_with_junit

Java Build Path

Source | Projects | Libraries | Order and Export

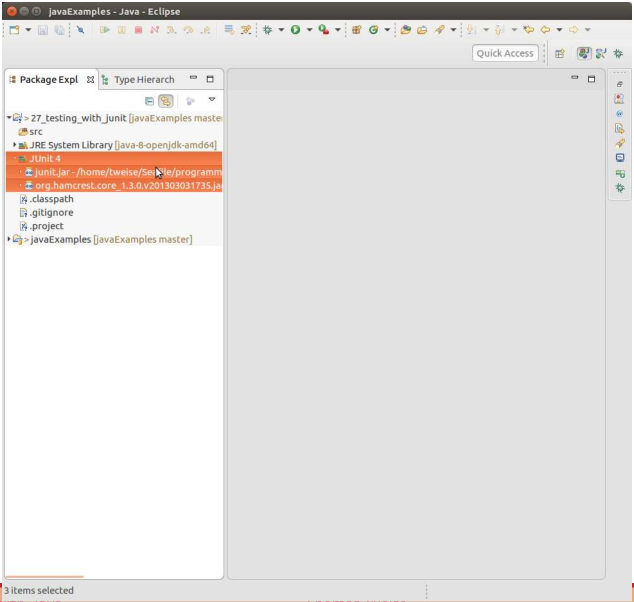
JARs and class folders on the build path:

- JRE System Library [java-8-openjdk-amd64]
- JUnit 4

## Enabling JUnit Support

- Eclipse provides direct, first-class JUnit support
- To enable this support by adding JUnit to the build path, we do *not* need to download the JUnit `jar` s or anything...
- First, we create a new empty Java project, let's call it `27_testing_with_junit`
- Then we right-click the project and click `Properties`
- Under `Java Build Path` we select `Libraries` and click `Add Library...`
- We choose `JUnit` and click `Next`
- We choose `JUnit 4` and click `Finish`
- We click `OK`
- The JUnit library has now appeared in the build path

## Enabling JUnit Support



```
▾ > 27_testing_with_junit [javaExamples master
    src
  ▸ JRE System Library [java-8-openjdk-amd64]
  ▾ JUnit 4
      junit.jar - /home/tweise/Se..le/programm
      org.hamcrest.core_1.3.0.v201303031735.ja
    .classpath
    .gitignore
    .project
▸ > javaExamples [javaExamples master]
```

3 items selected

## Project Structure

- We usually want to separate test classes from “real code” classes

## Project Structure

- We usually want to separate test classes from "real code" classes: When we create our `jar`s, we usually do not want the tests to be included
- However, we also want that tests can potentially access package private methods, because the more we can test, the better

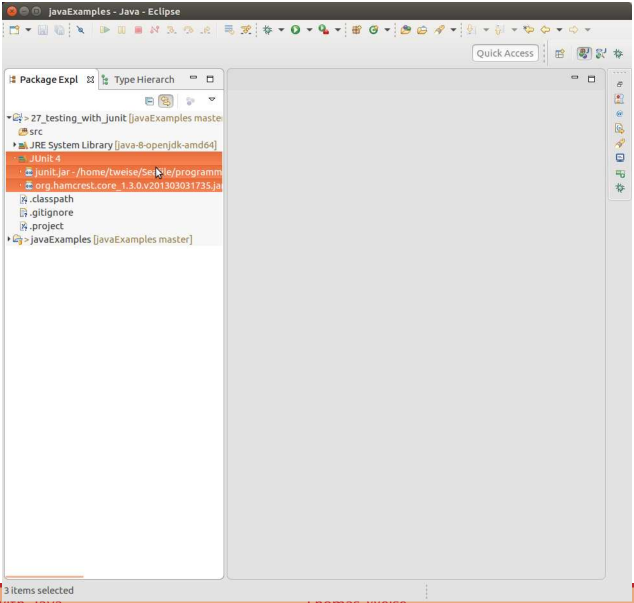
## Project Structure

- We usually want to separate test classes from "real code" classes: When we create our `jar`s, we usually do not want the tests to be included
- However, we also want that tests can potentially access package private methods, because the more we can test, the better
- Solution: Our project gets two root source folders, one for code, one for tests, with the same package hierarchy

## Project Structure

- We usually want to separate test classes from “real code” classes: When we create our `jar`s, we usually do not want the tests to be included
- However, we also want that tests can potentially access package private methods, because the more we can test, the better
- Solution: Our project gets two root source folders, one for code, one for tests, with the same package hierarchy
- Let’s continue with the empty project from before: There already is the one default source folder (`src`)

## Project Structure


javaExamples - Java - Eclipse

Quick Access

Package Expl | Type Hierarch

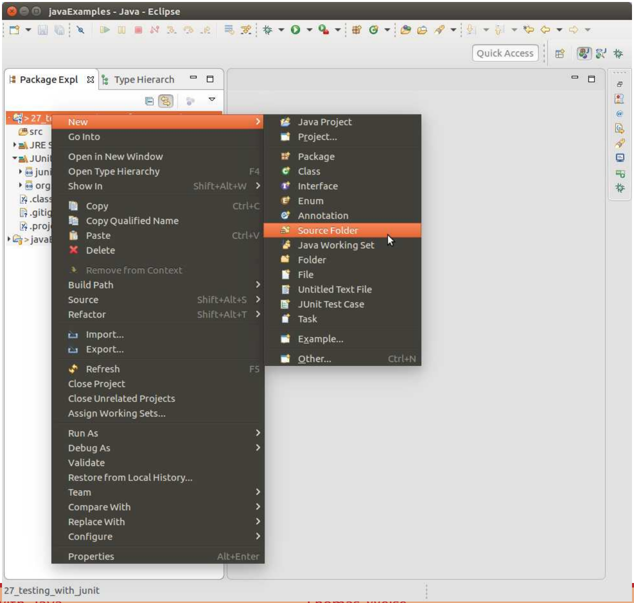
- > 27_testing_with_junit [javaExamples maste
  - src
  - JRE System Library [java-8-openjdk-amd64]
  - JUnit 4
    - junit.jar - /home/tweise/Seafile/programm
    - org.hamcrest.core_1.3.0.v201303031735.ja
  - .classpath
  - .gitignore
  - .project
- > javaExamples [javaExamples master]

3 items selected

## Project Structure

- We usually want to separate test classes from “real code” classes: When we create our `jar`s, we usually do not want the tests to be included
- However, we also want that tests can potentially access package private methods, because the more we can test, the better
- Solution: Our project gets two root source folders, one for code, one for tests, with the same package hierarchy
- Let’s continue with the empty project from before: There already is the one default source folder (`src`)
- Right-click the project, choose `New` then `Source Folder`

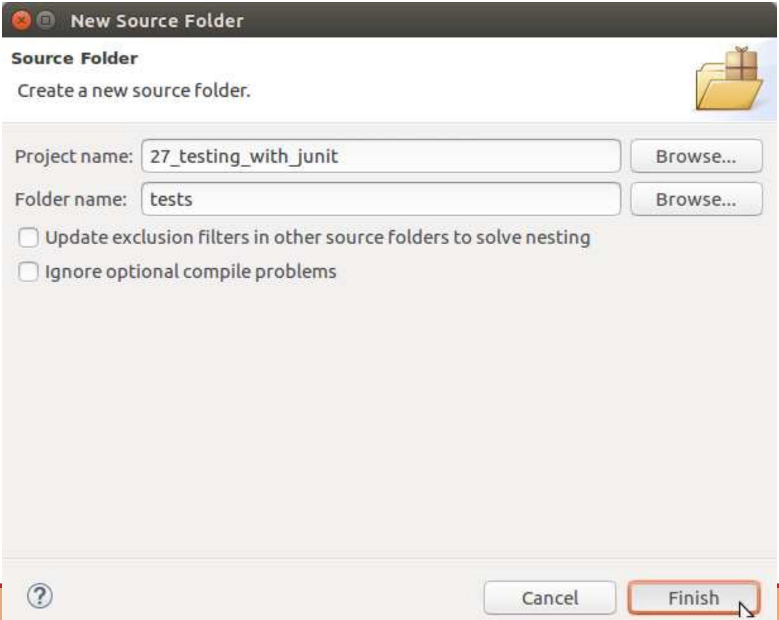
## Project Structure


javaExamples - Java - Eclipse

## Project Structure

- We usually want to separate test classes from "real code" classes: When we create our `jar`s, we usually do not want the tests to be included
- However, we also want that tests can potentially access package private methods, because the more we can test, the better
- Solution: Our project gets two root source folders, one for code, one for tests, with the same package hierarchy
- Let's continue with the empty project from before: There already is the one default source folder (`src`)
- Right-click the project, choose `New` then `Source Folder`
- Choose a good name for the test classes folder, how about `tests`, click `Finish`

## Project Structure


New Source Folder

Source Folder

Create a new source folder.

Project name: 27_testing_with_junit

Folder name: tests

☐ Update exclusion filters in other source folders to solve nesting

☐ Ignore optional compile problems

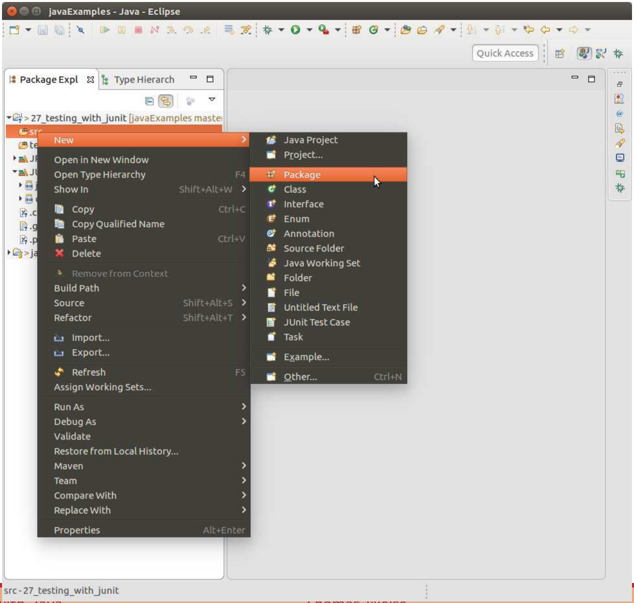
## Project Structure

- We usually want to separate test classes from "real code" classes: When we create our `jar`s, we usually do not want the tests to be included
- However, we also want that tests can potentially access package private methods, because the more we can test, the better
- Solution: Our project gets two root source folders, one for code, one for tests, with the same package hierarchy
- Let's continue with the empty project from before: There already is the one default source folder (`src`)
- Right-click the project, choose `New` then `Source Folder`
- Choose a good name for the test classes folder, how about `tests`, click `Finish`
- We now want to create the same package structure, say with root package `cn.edu.hfuu.iao`, in both folders

## Project Structure

- We usually want to separate test classes from "real code" classes: When we create our `jar`s, we usually do not want the tests to be included
- However, we also want that tests can potentially access package private methods, because the more we can test, the better
- Solution: Our project gets two root source folders, one for code, one for tests, with the same package hierarchy
- Let's continue with the empty project from before: There already is the one default source folder (`src`)
- Right-click the project, choose `New` then `Source Folder`
- Choose a good name for the test classes folder, how about `tests`, click `Finish`
- We now want to create the same package structure, say with root package `cn.edu.hfuu.iao`, in both folders
- We right-click the folder `src`, choose `New` then `Package`

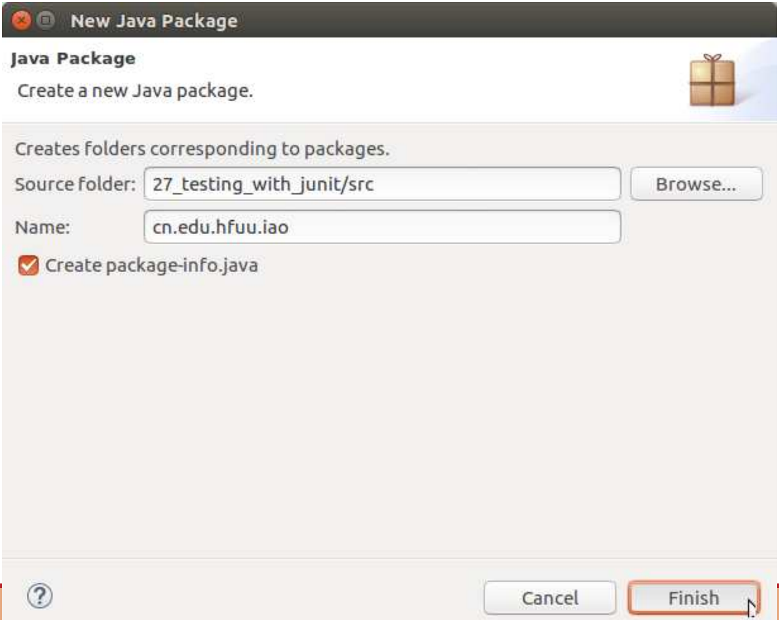
## Project Structure



## Project Structure

- However, we also want that tests can potentially access package private methods, because the more we can test, the better
- Solution: Our project gets two root source folders, one for code, one for tests, with the same package hierarchy
- Let's continue with the empty project from before: There already is the one default source folder (`src`)
- Right-click the project, choose `New` then `Source Folder`
- Choose a good name for the test classes folder, how about `tests`, click `Finish`
- We now want to create the same package structure, say with root package `cn.edu.hfuu.iao`, in both folders
- We right-click the folder `src`, choose `New` then `Package`
- We type in `cn.edu.hfuu.iao` and click `Finish`

## Project Structure


New Java Package

Java Package

Create a new Java package.

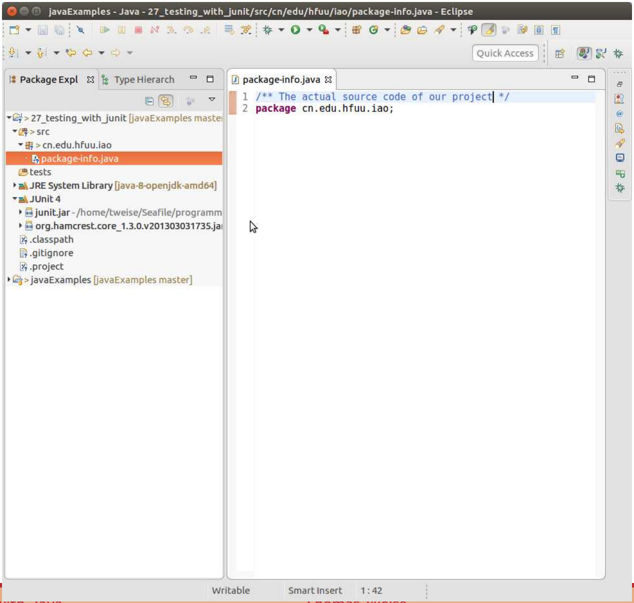
Creates folders corresponding to packages.

Source folder: 27_testing_with_junit/src

Name: cn.edu.hfuu.iao

[✓] Create package-info.java

## Project Structure


javaExamples - Java - 27_testing_with_junit/src/cn/edu/hfuu/iao/package-info.java - Eclipse

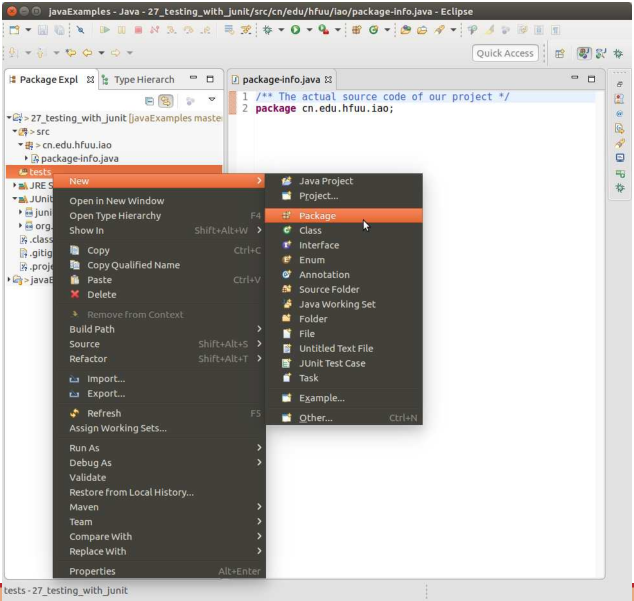
Quick Access
Package Expl
Type Hierarch
package-info.java
> 27_testing_with_junit [javaExamples maste
> src
> cn.edu.hfuu.iao
package-info.java
tests
JRE System Library [java-8-openjdk-amd64]
JUnit 4
junit.jar - /home/tweise/Seafile/programm
org.hamcrest.core_1.3.0.v201303031735.ja
.classpath
.gitignore
.project
> javaExamples [javaExamples master]
Writable
Smart Insert
1 : 42

```
/** The actual source code of our project */
package cn.edu.hfuu.iao;
```

## Project Structure

- Solution: Our project gets two root source folders, one for code, one for tests, with the same package hierarchy
- Let's continue with the empty project from before: There already is the one default source folder (`src`)
- Right-click the project, choose `New` then `Source Folder`
- Choose a good name for the test classes folder, how about `tests`, click `Finish`
- We now want to create the same package structure, say with root package `cn.edu.hfuu.iao`, in both folders
- We right-click the folder `src`, choose `New` then `Package`
- We type in `cn.edu.hfuu.iao` and click `Finish`
- We repeat the procedure for folder `tests`: right-click the folder `tests`, choose `New` then `Package`

## Project Structure

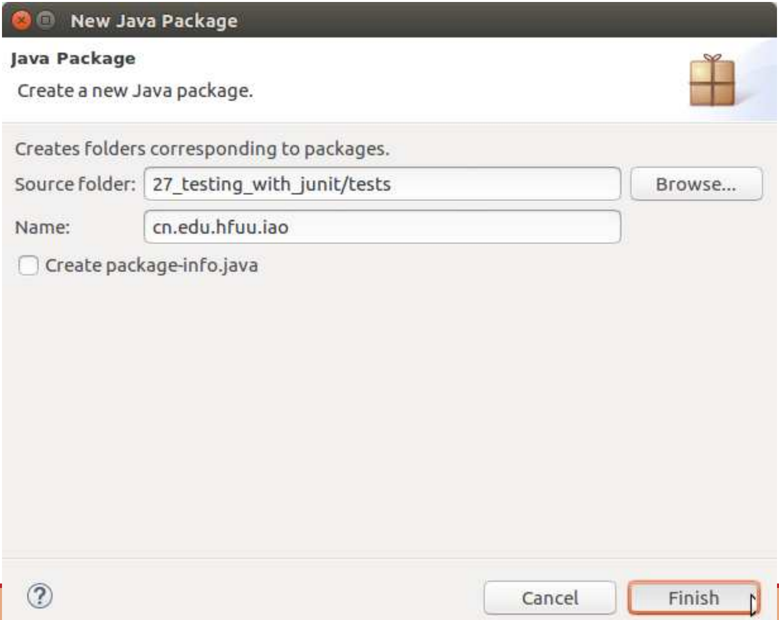


```java
/** The actual source code of our project */
package cn.edu.hfuu.iao;
```

## Project Structure

- Let's continue with the empty project from before: There already is the one default source folder (`src`)
- Right-click the project, choose `New` then `Source Folder`
- Choose a good name for the test classes folder, how about `tests`, click `Finish`
- We now want to create the same package structure, say with root package `cn.edu.hfuu.iao`, in both folders
- We right-click the folder `src`, choose `New` then `Package`
- We type in `cn.edu.hfuu.iao` and click `Finish`
- We repeat the procedure for folder `tests`: right-click the folder `tests`, choose `New` then `Package`
- We type in `cn.edu.hfuu.iao`, make sure that `create package-info.jar` is *not* selected, and click `Finish`

## Project Structure


New Java Package

Java Package

Create a new Java package.

Creates folders corresponding to packages.

Source folder: 27_testing_with_junit/tests

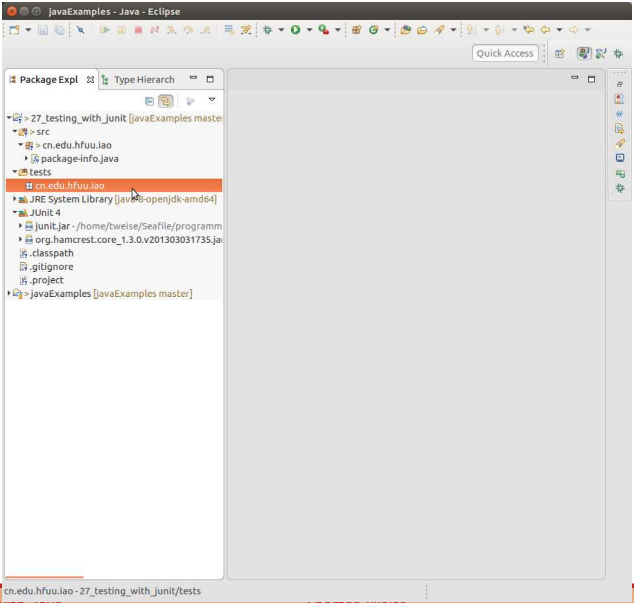
Name: cn.edu.hfuu.iao

☐ Create package-info.java

## Project Structure

- Right-click the project, choose `New` then `Source Folder`
- Choose a good name for the test classes folder, how about `tests`, click `Finish`
- We now want to create the same package structure, say with root package `cn.edu.hfuu.iao`, in both folders
- We right-click the folder `src`, choose `New` then `Package`
- We type in `cn.edu.hfuu.iao` and click `Finish`
- We repeat the procedure for folder `tests`: right-click the folder `tests`, choose `New` then `Package`
- We type in `cn.edu.hfuu.iao`, make sure that `create package-info.jar` is *not* selected, and click `Finish`
- The new package has appeared (empty) in the package explorer

# Project Structure



```
▾ > 27_testing_with_junit [javaExamples maste
  ▾ > src
    ▾ > cn.edu.hfuu.iao
      ▸ package-info.java
  ▾ tests
      cn.edu.hfuu.iao
  ▸ JRE System Library [java-8-openjdk-amd64]
  ▾ JUnit 4
    ▸ junit.jar - /home/tweise/Seafile/programm
    ▸ org.hamcrest.core_1.3.0.v201303031735.ja
    .classpath
    .gitignore
    .project
▸ > javaExamples [javaExamples master]
```

cn.edu.hfuu.iao - 27_testing_with_junit/tests

## Creating JUnit Tests

- Let us revisit (again) our Vertical Ball Throw example

## Creating JUnit Tests

- Let us revisit (again) our Vertical Ball Throw example
- And this time test its correctness

## Vertical Ball Throw with Console I/O

Listing: Vertical Ball Throw with Console I/O

```
package cn.edu.hfuu.iao;

import java.util.Scanner;

/**
 * A ball is thrown vertically upwards into the air by a $x_0 m$ tall person
 * with velocity $v_0 m/s$. Where is it after $t$ seconds?<br/>
 * $x(t) = x_0 + v_0 * t - 0.5 * g * t^2$
 */
public class VerticalBallThrow {

  /** Compute the position of a ball
   * @param x0 the height of the thrower, i.e., the initial vertical position
   * @param v0 the vertical upward velocity with which the ball is thrown
   * @param t the time at which we want to get the position $x(t)$
   * @return the position $x(t)$ of the ball at time step t
   */
  static double position(double x0, double v0, double t) {
    return x0 + (v0 * t) - 0.5d * 9.80665d * t * t;
  }

  /** The main routine
   * @param args
   *           we ignore this parameter for now */
  public static final void main(String[] args) {
    try(Scanner scanner = new Scanner(System.in)) { // initiate reading from System.in, ignore for now
      System.err.println("Enter␣size␣x0␣of␣personin␣m:"); //$NON-NLS-1$
      double x0 = scanner.nextDouble(); // read initial vertical position $x_0$
      System.err.println("Enter␣initial␣upward␣velocity␣v0␣of␣ball␣in␣m/s:"); //$NON-NLS-1$
      double v0 = scanner.nextDouble(); // read initial velocity upwards $v_0$
      System.err.println("Enter␣time␣t␣in␣s:"); //$NON-NLS-1$
      double t = scanner.nextDouble();  // read the time $t$
      System.out.println(position(x0, v0, t)); // compute and print position
    }
  }
}
```

## Creating JUnit Tests

- Let us revisit (again) our Vertical Ball Throw example
- And this time test its correctness

## Creating JUnit Tests

- Let us revisit (again) our Vertical Ball Throw example
- And this time test its correctness
- The thing to test here clearly is method `position`

## Creating JUnit Tests

- Let us revisit (again) our Vertical Ball Throw example
- And this time test its correctness
- The thing to test here clearly is method `position`
- We need to investigate whether

## Creating JUnit Tests

- Let us revisit (again) our Vertical Ball Throw example
- And this time test its correctness
- The thing to test here clearly is method `position`
- We need to investigate whether
  - it returns correct results

## Creating JUnit Tests

- Let us revisit (again) our Vertical Ball Throw example
- And this time test its correctness
- The thing to test here clearly is method `position`
- We need to investigate whether
    - it returns correct results
    - it deals with border cases correctly

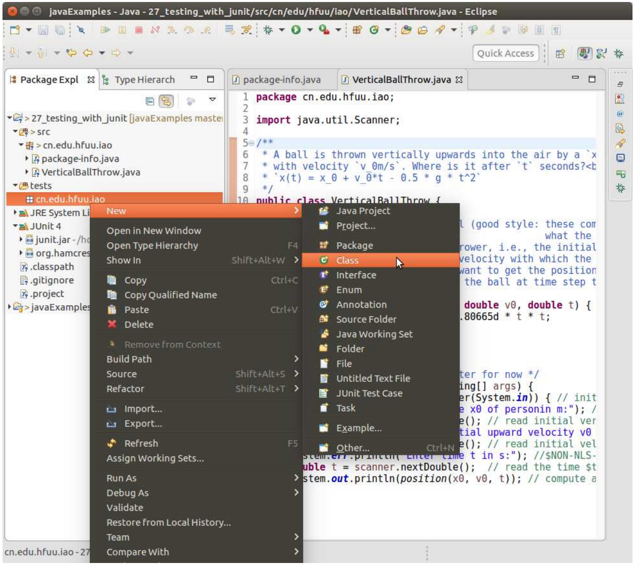
## Creating JUnit Tests

- Let us revisit (again) our Vertical Ball Throw example
- And this time test its correctness
- The thing to test here clearly is method `position`
- We need to investigate whether
    - it returns correct results
    - it deals with border cases correctly
    - it deals with invalid arguments correctly

## Creating JUnit Tests

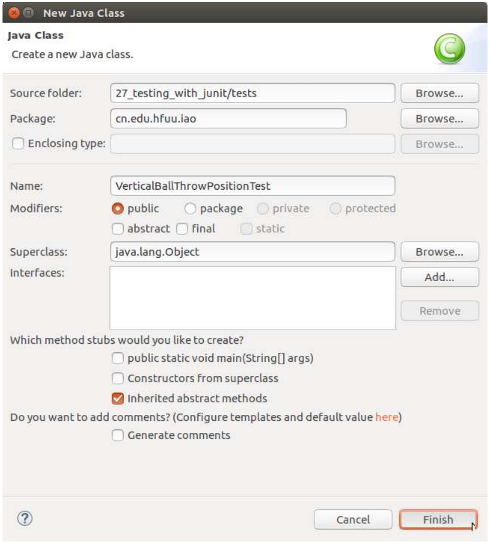
- We first create a new (test) class in package the `cn.edu.hfuu.iao` package of the `tests` folder and call it `VerticalBallThrowPositionTest`

## Creating JUnit Tests



```
package cn.edu.hfuu.iao;

import java.util.Scanner;

/**
 * A ball is thrown vertically upwards into the air by a `x
 * with velocity `v_0m/s`. Where is it after `t` seconds?<b
 * `x(t) = x_0 + v_0*t - 0.5 * g * t^2`
 */
public class VerticalBallThrow {
```

## Creating JUnit Tests

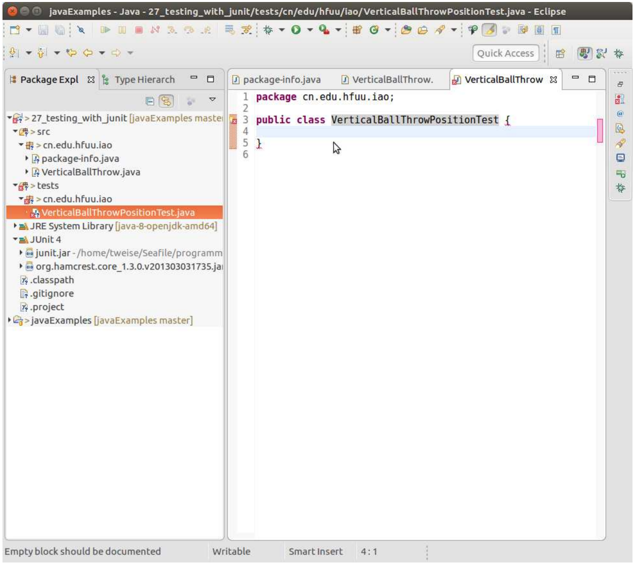


Source folder: 27_testing_with_junit/tests
Package: cn.edu.hfuu.iao
☐ Enclosing type:
Name: VerticalBallThrowPositionTest
Modifiers: [✓ public] package private protected
☐ abstract ☐ final ☐ static
Superclass: java.lang.Object
Interfaces:

Which method stubs would you like to create?
☐ public static void main(String[] args)
☐ Constructors from superclass
☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)
☐ Generate comments

## Creating JUnit Tests


javaExamples - Java - 27_testing_with_junit/tests/cn/edu/hfuu/iao/VerticalBallThrowPositionTest.java - Eclipse
Quick Access
Package Expl
Type Hierarch
package-info.java
VerticalBallThrow.
VerticalBallThrow
> 27_testing_with_junit [javaExamples master
> src
> cn.edu.hfuu.iao
package-info.java
VerticalBallThrow.java
> tests
> cn.edu.hfuu.iao
VerticalBallThrowPositionTest.java
JRE System Library [java-8-openjdk-amd64]
JUnit 4
junit.jar - /home/tweise/Seafile/programm
org.hamcrest.core_1.3.0.v201303031735.ja
.classpath
.gitignore
.project
> javaExamples [javaExamples master]
Writable
Smart Insert
4 : 1

```java
package cn.edu.hfuu.iao;

public class VerticalBallThrowPositionTest {

}
```

Empty block should be documented

## Creating JUnit Tests

- We first create a new (test) class in package the `cn.edu.hfuu.iao` package of the `tests` folder and call it `VerticalBallThrowPositionTest`
- Ok, but what should we test first?
- Let us first test some very common cases, e.g.,
    - if `position` $= 4.903325$ if $x_0 = 0$, $v_0 = 9.80665 m/s$, and $t = 1s$
    - if `position` $= 45.3867$ if $x_0 = 1$, $v_0 = 32 m/s$, and $t = 2s$

## Creating JUnit Tests

- We first create a new (test) class in package the `cn.edu.hfuu.iao` package of the `tests` folder and call it `VerticalBallThrowPositionTest`
- Ok, but what should we test first?
- Let us first test some very common cases, e.g.,
  - if `position` $= 4.903325$ if $x_0 = 0$, $v_0 = 9.80665m/s$, and $t = 1s$
  - if `position` $= 45.3867$ if $x_0 = 1$, $v_0 = 32m/s$, and $t = 2s$
  - if `position` $= 2.870075$ if $x_0 = 2$, $v_0 = 15m/s$, and $t = 3s$

## Creating JUnit Tests

- We first create a new (test) class in package the `cn.edu.hfuu.iao` package of the `tests` folder and call it `VerticalBallThrowPositionTest`
- Ok, but what should we test first?
- Let us first test some very common cases, e.g.,
  - if `position` $= 4.903325$ if $x_0 = 0$, $v_0 = 9.80665m/s$, and $t = 1s$
  - if `position` $= 45.3867$ if $x_0 = 1$, $v_0 = 32m/s$, and $t = 2s$
  - if `position` $= 2.870075$ if $x_0 = 2$, $v_0 = 15m/s$, and $t = 3s$
- We have computed these values by hand and expect that our function should return results reasonably close to them

## Creating JUnit Tests

- We first create a new (test) class in package the `cn.edu.hfuu.iao` package of the `tests` folder and call it `VerticalBallThrowPositionTest`
- Ok, but what should we test first?
- Let us first test some very common cases, e.g.,
  - if `position` $= 4.903325$ if $x_0 = 0$, $v_0 = 9.80665m/s$, and $t = 1s$
  - if `position` $= 45.3867$ if $x_0 = 1$, $v_0 = 32m/s$, and $t = 2s$
  - if `position` $= 2.870075$ if $x_0 = 2$, $v_0 = 15m/s$, and $t = 3s$
- We have computed these values by hand and expect that our function should return results reasonably close to them
- As a JUnit test, this looks as follows

## Common Test Cases for Vertical Ball Throw

### Listing: Common Test Cases for Vertical Ball Throw

```
package cn.edu.hfuu.iao;

import org.junit.Assert;
import org.junit.Test;

/** Our first test class */
public class VerticalBallThrowPositionTest {

  /** test the position for $x_0=0m$, $v_0=g=0.90665m/s$, $t?=1s$ */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x00_v0g_t1() {
    Assert.assertEquals(4.903325d,                              // the expected value
        VerticalBallThrow.position(0d, 9.80665d, 1d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }

  /** test the position for $x_0=1m$, $v_0=32m/s$, $t?=2s$ */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x01_v032_t2() {
    Assert.assertEquals(45.3867d,                       // the expected value
        VerticalBallThrow.position(1d, 32d, 2d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }

  /** test the position for $x_0=2m$, $v_0=15m/s$, $t?=3s$ */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x02_v015_t3() {
    Assert.assertEquals(2.870075d,                      // the expected value
        VerticalBallThrow.position(2d, 15d, 3d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }
}
```

## Creating JUnit Tests: Common Cases

- Each test case is placed into one method with a descriptive name

## Creating JUnit Tests: Common Cases

- Each test case is placed into one method with a descriptive name
  - Each such method is annotated with a `@Test` (between the javadoc and the method signature), telling JUnit that it is a test method

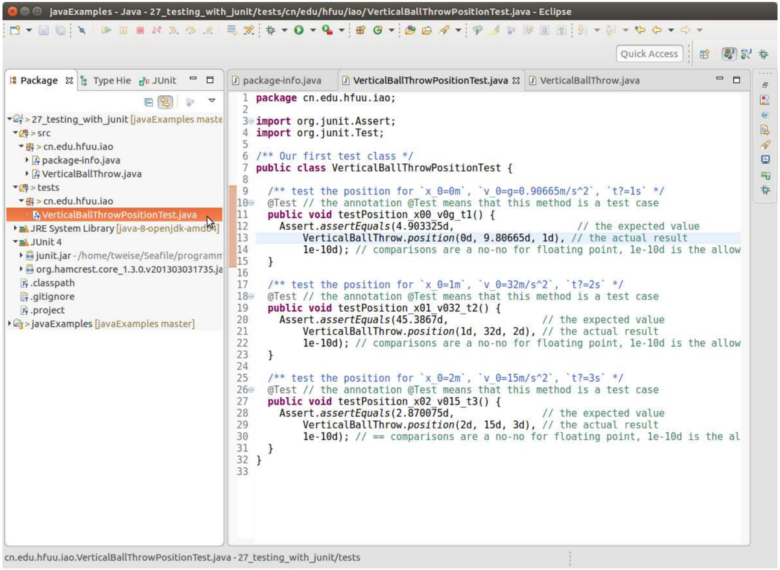
## Creating JUnit Tests: Common Cases

- Each test case is placed into one method with a descriptive name
  - Each such method is annotated with a `@Test` (between the javadoc and the method signature), telling JUnit that it is a test method
  - Inside the test method, we compare the expected result of `position` with its actual result by using one of the many `assertEquals` methods from class `org.junit.Assert`

## Creating JUnit Tests: Common Cases

- Each test case is placed into one method with a descriptive name
  - Each such method is annotated with a `@Test` (between the javadoc and the method signature), telling JUnit that it is a test method
  - Inside the test method, we compare the expected result of `position` with its actual result by using one of the many `assertEquals` methods from class `org.junit.Assert`
  - `assertEquals` would throw an exception if the expected and actual value are different (or, in case of floating point numbers, differ by more than a given maximum deviation)

## Creating JUnit Tests: Common Cases


javaExamples - Java - 27_testing_with_junit/tests/cn/edu/hfuu/iao/VerticalBallThrowPositionTest.java - Eclipse

Quick Access
Package
Type Hie
JUnit
package-info.java
VerticalBallThrowPositionTest.java
VerticalBallThrow.java
> 27_testing_with_junit [javaExamples maste
> src
> cn.edu.hfuu.iao
package-info.java
VerticalBallThrow.java
> tests
> cn.edu.hfuu.iao
VerticalBallThrowPositionTest.java
JRE System Library [java-8-openjdk-amd64]
JUnit 4
junit.jar - /home/tweise/Seafile/programm
org.hamcrest.core_1.3.0.v201303031735.ja
.classpath
.gitignore
.project
> javaExamples [javaExamples master]

```java
package cn.edu.hfuu.iao;

import org.junit.Assert;
import org.junit.Test;

/** Our first test class */
public class VerticalBallThrowPositionTest {

  /** test the position for `x_0=0m`, `v_0=g=0.90665m/s^2`, `t?=1s` */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x00_v0g_t1() {
    Assert.assertEquals(4.903325d,                        // the expected value
        VerticalBallThrow.position(0d, 9.80665d, 1d), // the actual result
        1e-10d); // comparisons are a no-no for floating point, 1e-10d is the allow
  }

  /** test the position for `x_0=1m`, `v_0=32m/s^2`, `t?=2s` */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x01_v032_t2() {
    Assert.assertEquals(45.3867d,                 // the expected value
        VerticalBallThrow.position(1d, 32d, 2d), // the actual result
        1e-10d); // comparisons are a no-no for floating point, 1e-10d is the allow
  }

  /** test the position for `x_0=2m`, `v_0=15m/s^2`, `t?=3s` */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x02_v015_t3() {
    Assert.assertEquals(2.870075d,                // the expected value
        VerticalBallThrow.position(2d, 15d, 3d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the al
  }
}
```

cn.edu.hfuu.iao.VerticalBallThrowPositionTest.java - 27_testing_with_junit/tests

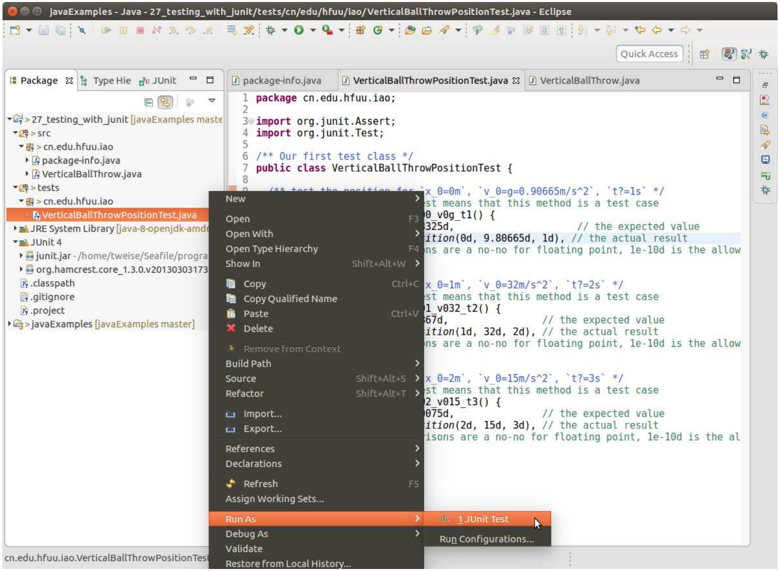
## Creating JUnit Tests: Common Cases

- Each test case is placed into one method with a descriptive name
  - Each such method is annotated with a `@Test` (between the javadoc and the method signature), telling JUnit that it is a test method
  - Inside the test method, we compare the expected result of `position` with its actual result by using one of the many `assertEquals` methods from class `org.junit.Assert`
  - `assertEquals` would throw an exception if the expected and actual value are different (or, in case of floating point numbers, differ by more than a given maximum deviation)
- Let us now run these tests

## Creating JUnit Tests: Common Cases

- Each test case is placed into one method with a descriptive name
    - Each such method is annotated with a `@Test` (between the javadoc and the method signature), telling JUnit that it is a test method
    - Inside the test method, we compare the expected result of `position` with its actual result by using one of the many `assertEquals` methods from class `org.junit.Assert`
    - `assertEquals` would throw an exception if the expected and actual value are different (or, in case of floating point numbers, differ by more than a given maximum deviation)
- Let us now run these tests
- Right-click on class `VerticalBallThrowPositionTest`, choose `Run As` and then `JUnit Test`

# Creating JUnit Tests: Common Cases

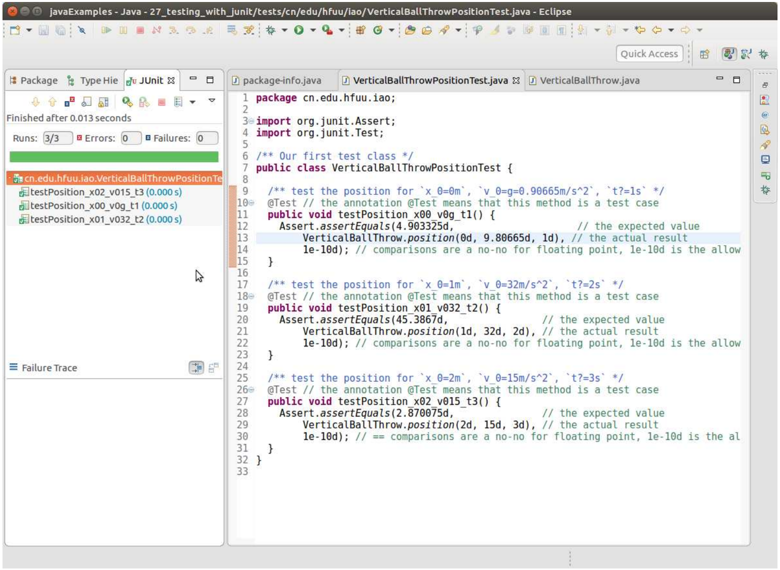


```
package cn.edu.hfuu.iao;

import org.junit.Assert;
import org.junit.Test;

/** Our first test class */
public class VerticalBallThrowPositionTest {

  /** test the position for `x_0=0m`, `v_0=g=0.90665m/s^2`, `t?=1s` */
st means that this method is a test case
0_v0g_t1() {
325d,                     // the expected value
ition(0d, 9.80665d, 1d), // the actual result
ns are a no-no for floating point, 1e-10d is the allow

x_0=1m`, `v_0=32m/s^2`, `t?=2s` */
st means that this method is a test case
1_v032_t2() {
67d,                   // the expected value
ition(1d, 32d, 2d), // the actual result
ns are a no-no for floating point, 1e-10d is the allow

x_0=2m`, `v_0=15m/s^2`, `t?=3s` */
st means that this method is a test case
2_v015_t3() {
075d,                  // the expected value
ition(2d, 15d, 3d), // the actual result
isons are a no-no for floating point, 1e-10d is the al
```

## Creating JUnit Tests: Common Cases

- Each test case is placed into one method with a descriptive name
    - Each such method is annotated with a `@Test` (between the javadoc and the method signature), telling JUnit that it is a test method
    - Inside the test method, we compare the expected result of `position` with its actual result by using one of the many `assertEquals` methods from class `org.junit.Assert`
    - `assertEquals` would throw an exception if the expected and actual value are different (or, in case of floating point numbers, differ by more than a given maximum deviation)
- Let us now run these tests
- Right-click on class `VerticalBallThrowPositionTest`, choose `Run As` and then `JUnit Test`
- All tests succeed, we get all green bars

## Creating JUnit Tests: Common Cases


javaExamples - Java - 27_testing_with_junit/tests/cn/edu/hfuu/iao/VerticalBallThrowPositionTest.java - Eclipse
Quick Access
Package
Type Hie
JUnit
package-info.java
VerticalBallThrowPositionTest.java
VerticalBallThrow.java
Failure Trace

```
Finished after 0.013 seconds
Runs: 3/3    Errors: 0    Failures: 0

cn.edu.hfuu.iao.VerticalBallThrowPositionTe
  testPosition_x02_v015_t3 (0.000 s)
  testPosition_x00_v0g_t1 (0.000 s)
  testPosition_x01_v032_t2 (0.000 s)
```

```java
package cn.edu.hfuu.iao;

import org.junit.Assert;
import org.junit.Test;

/** Our first test class */
public class VerticalBallThrowPositionTest {

  /** test the position for `x_0=0m`, `v_0=g=0.90665m/s^2`, `t?=1s` */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x00_v0g_t1() {
    Assert.assertEquals(4.903325d,                          // the expected value
        VerticalBallThrow.position(0d, 9.80665d, 1d), // the actual result
        1e-10d); // comparisons are a no-no for floating point, 1e-10d is the allow
  }

  /** test the position for `x_0=1m`, `v_0=32m/s^2`, `t?=2s` */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x01_v032_t2() {
    Assert.assertEquals(45.3867d,                     // the expected value
        VerticalBallThrow.position(1d, 32d, 2d), // the actual result
        1e-10d); // comparisons are a no-no for floating point, 1e-10d is the allow
  }

  /** test the position for `x_0=2m`, `v_0=15m/s^2`, `t?=3s` */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x02_v015_t3() {
    Assert.assertEquals(2.870075d,                    // the expected value
        VerticalBallThrow.position(2d, 15d, 3d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the al
  }
}
```

## Creating JUnit Tests: Common Cases

- Each test case is placed into one method with a descriptive name
  - Each such method is annotated with a `@Test` (between the javadoc and the method signature), telling JUnit that it is a test method
  - Inside the test method, we compare the expected result of `position` with its actual result by using one of the many `assertEquals` methods from class `org.junit.Assert`
  - `assertEquals` would throw an exception if the expected and actual value are different (or, in case of floating point numbers, differ by more than a given maximum deviation)
- Let us now run these tests
- Right-click on class `VerticalBallThrowPositionTest`, choose `Run As` and then `JUnit Test`
- All tests succeed, we get all green bars
- This does not proof anything, but so far our `position` method looks OK

## Creating JUnit Tests: Border Cases

- Now we should look at border cases, i.e., whether the method is still correct when the inputs take on extreme values

## Creating JUnit Tests: Border Cases

- Now we should look at border cases, i.e., whether the method is still correct when the inputs take on extreme values
- One extreme case here would clearly be what happens if enough time has passed so that the ball has fallen back to the ground

## Creating JUnit Tests: Border Cases

- Now we should look at border cases, i.e., whether the method is still correct when the inputs take on extreme values
- One extreme case here would clearly be what happens if enough time has passed so that the ball has fallen back to the ground
- Obviously, it can never fall below 0m. . .

## Creating JUnit Tests: Border Cases

- Now we should look at border cases, i.e., whether the method is still correct when the inputs take on extreme values
- One extreme case here would clearly be what happens if enough time has passed so that the ball has fallen back to the ground
- Obviously, it can never fall below 0m...
- The result of `position` for if $x_0 = 1$, $v_0 = 10m/s^2$, and $t = 1000s$ should be $0m$, not $-4893324m$

## Creating JUnit Tests: Border Cases

- Now we should look at border cases, i.e., whether the method is still correct when the inputs take on extreme values
- One extreme case here would clearly be what happens if enough time has passed so that the ball has fallen back to the ground
- Obviously, it can never fall below 0m...
- The result of `position` for if $x_0 = 1$, $v_0 = 10m/s^2$, and $t = 1000s$ should be $0m$, not $-4893324m$
- Let's build a test case for this

## Creating JUnit Tests: Border Cases

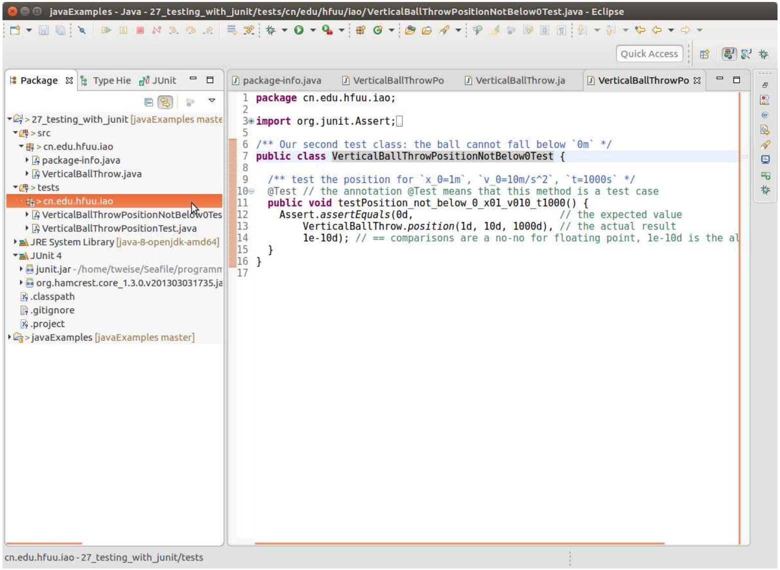
Listing: The new border case test `VerticalBallThrowPositionNotBelow0Test`

```
package cn.edu.hfuu.iao;

import org.junit.Assert;
import org.junit.Test;

/** Our second test class: the ball cannot fall below 0m */
public class VerticalBallThrowPositionNotBelow0Test {

  /** test the position for x_0 = 1m, v_0 = 10m/s, t = 1000s */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_not_below_0_x01_v010_t1000() {
    Assert.assertEquals(0d,                              // the expected value
        VerticalBallThrow.position(1d, 10d, 1000d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }
}
```

## Creating JUnit Tests: Border Cases

- Now we should look at border cases, i.e., whether the method is still correct when the inputs take on extreme values
- One extreme case here would clearly be what happens if enough time has passed so that the ball has fallen back to the ground
- Obviously, it can never fall below 0m...
- The result of `position` for if $x_0 = 1$, $v_0 = 10m/s^2$, and $t = 1000s$ should be $0m$, not $-4893324m$
- Let's build a test case for this
- We can now run this test in the same way as before, or run all tests in the tests package at once

## Creating JUnit Tests: Border Cases

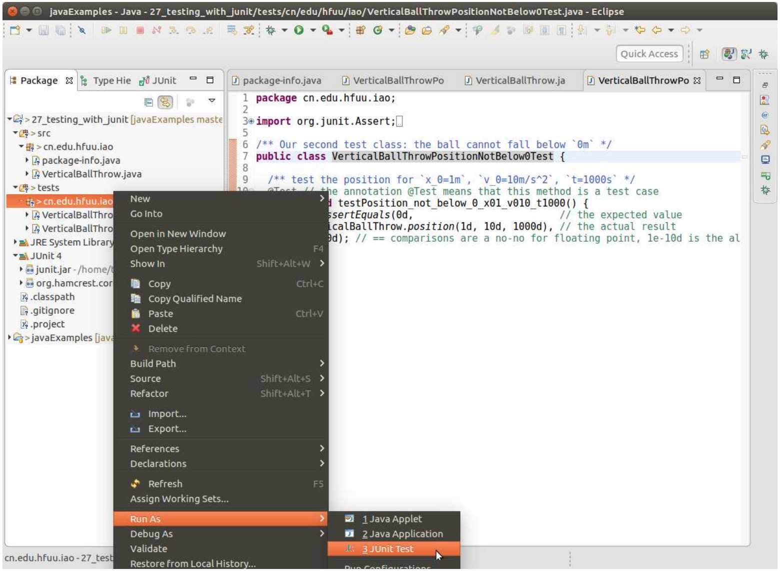

javaExamples - Java - 27_testing_with_junit/tests/cn/edu/hfuu/iao/VerticalBallThrowPositionNotBelow0Test.java - Eclipse
Quick Access
Package
Type Hie
JUnit
package-info.java
VerticalBallThrowPo
VerticalBallThrow.ja
VerticalBallThrowPo
> 27_testing_with_junit [javaExamples maste
> src
> cn.edu.hfuu.iao
package-info.java
VerticalBallThrow.java
> tests
> cn.edu.hfuu.iao
VerticalBallThrowPositionNotBelow0Tes
VerticalBallThrowPositionTest.java
JRE System Library [java-8-openjdk-amd64]
JUnit 4
junit.jar - /home/tweise/Seafile/programm
org.hamcrest.core_1.3.0.v201303031735.ja
.classpath
.gitignore
.project
> javaExamples [javaExamples master]
cn.edu.hfuu.iao - 27_testing_with_junit/tests

```java
package cn.edu.hfuu.iao;

import org.junit.Assert;

/** Our second test class: the ball cannot fall below `0m` */
public class VerticalBallThrowPositionNotBelow0Test {

  /** test the position for `x_0=1m`, `v_0=10m/s^2`, `t=1000s` */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_not_below_0_x01_v010_t1000() {
    Assert.assertEquals(0d,                          // the expected value
        VerticalBallThrow.position(1d, 10d, 1000d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the al
  }
}
```

## Creating JUnit Tests: Border Cases

- Now we should look at border cases, i.e., whether the method is still correct when the inputs take on extreme values
- One extreme case here would clearly be what happens if enough time has passed so that the ball has fallen back to the ground
- Obviously, it can never fall below 0m…
- The result of `position` for if $x_0 = 1$, $v_0 = 10m/s^2$, and $t = 1000s$ should be $0m$, not $-4893324m$
- Let's build a test case for this
- We can now run this test in the same way as before, or run all tests in the tests package at once
- We right-click the test *package*, choose `Run As`, and `JUnit Test`

## Creating JUnit Tests: Border Cases



```
package cn.edu.hfuu.iao;

import org.junit.Assert;

/** Our second test class: the ball cannot fall below `0m` */
public class VerticalBallThrowPositionNotBelow0Test {

  /** test the position for `x_0=1m`, `v_0=10m/s^2`, `t=1000s` */
  @Test // the annotation @Test means that this method is a test case
        d testPosition_not_below_0_x01_v010_t1000() {
        ssertEquals(0d,                          // the expected value
        icalBallThrow.position(1d, 10d, 1000d), // the actual result
        0d); // == comparisons are a no-no for floating point, 1e-10d is the al
```

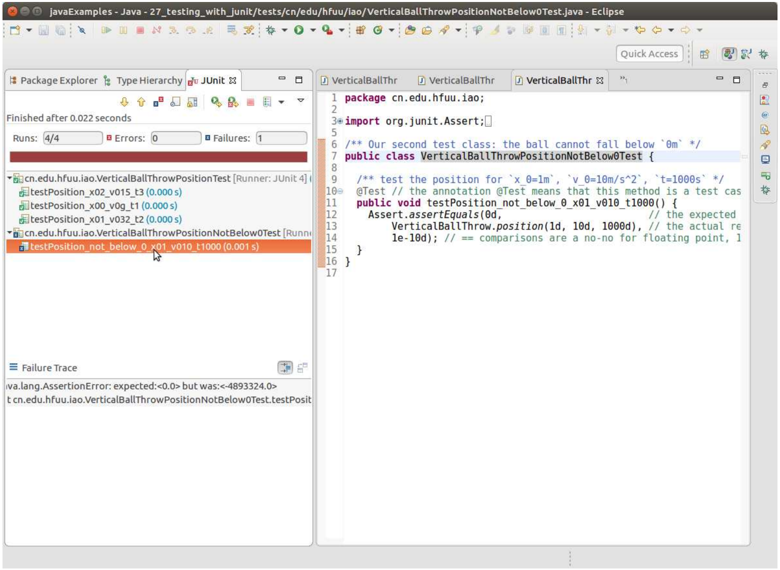
## Creating JUnit Tests: Border Cases

- One extreme case here would clearly be what happens if enough time has passed so that the ball has fallen back to the ground
- Obviously, it can never fall below 0m…
- The result of `position` for if $x_0 = 1$, $v_0 = 10m/s^2$, and $t = 1000s$ should be $0m$, not $-4893324m$
- Let's build a test case for this
- We can now run this test in the same way as before, or run all tests in the tests package at once
- We right-click the test *package*, choose `Run As`, and `JUnit Test`
- All 4 test cases in the package are executed, including the three previous tests

## Creating JUnit Tests: Border Cases

- One extreme case here would clearly be what happens if enough time has passed so that the ball has fallen back to the ground
- Obviously, it can never fall below 0m…
- The result of `position` for if $x_0 = 1$, $v_0 = 10m/s^2$, and $t = 1000s$ should be $0m$, not $-4893324m$
- Let's build a test case for this
- We can now run this test in the same way as before, or run all tests in the tests package at once
- We right-click the test *package*, choose `Run As`, and `JUnit Test`
- All 4 test cases in the package are executed, including the three previous tests
- The new test fails and becomes red, claiming `java.lang.AssertionError: expected:<0.0> but was:<-4893324.0>`, meaning that our `position` method does not guard the ball against falling through earth

## Creating JUnit Tests: Border Cases


javaExamples - Java - 27_testing_with_junit/tests/cn/edu/hfuu/iao/VerticalBallThrowPositionNotBelow0Test.java - Eclipse
Quick Access
Package Explorer
Type Hierarchy
JUnit
VerticalBallThr
VerticalBallThr
VerticalBallThr
Failure Trace

```
Finished after 0.022 seconds
Runs: 4/4    Errors: 0    Failures: 1

cn.edu.hfuu.iao.VerticalBallThrowPositionTest [Runner: JUnit 4]
  testPosition_x02_v015_t3 (0.000 s)
  testPosition_x00_v0g_t1 (0.000 s)
  testPosition_x01_v032_t2 (0.000 s)
cn.edu.hfuu.iao.VerticalBallThrowPositionNotBelow0Test [Runn
  testPosition_not_below_0_x01_v010_t1000 (0.001 s)
```

```
Failure Trace
va.lang.AssertionError: expected:<0.0> but was:<-4893324.0>
t cn.edu.hfuu.iao.VerticalBallThrowPositionNotBelow0Test.testPosit
```

```java
package cn.edu.hfuu.iao;

import org.junit.Assert;

/** Our second test class: the ball cannot fall below `0m` */
public class VerticalBallThrowPositionNotBelow0Test {

  /** test the position for `x_0=1m`, `v_0=10m/s^2`, `t=1000s` */
  @Test // the annotation @Test means that this method is a test cas
  public void testPosition_not_below_0_x01_v010_t1000() {
    Assert.assertEquals(0d,                             // the expected
        VerticalBallThrow.position(1d, 10d, 1000d), // the actual re
        1e-10d); // == comparisons are a no-no for floating point, 1
  }
}
```

## Fixed Problem: Ball Cannot Fall through Earth Anymore

- We now fix this problem by modifying `position` to first check whether the result is positive and return 0 otherwise

## Fixed Problem: Ball Cannot Fall through Earth Anymore

- We now fix this problem by modifying `position` to first check whether the result is positive and return 0 otherwise
- (For demonstration purposes, I therefore create a copy of both the source and the test package named `cn.edu.hfuu.iao_fix1`)

## Fixed Problem: Ball Cannot Fall through Earth Anymore

- We now fix this problem by modifying `position` to first check whether the result is positive and return 0 otherwise
- (For demonstration purposes, I therefore create a copy of both the source and the test package named `cn.edu.hfuu.iao_fix1`)
- The new code looks like this

# Fixed Problem: Ball Cannot Fall through Earth Anymore

## Listing: Vertical Ball Throw Positive Position Fix

```java
package cn.edu.hfuu.iao_fix1; // <-- package name changed for demo purposes

import java.util.Scanner;

/**
 * A ball is thrown vertically upwards into the air by a $x_0 m$ tall person
 * with velocity $v_0 m/s$. Where is it after $t$ seconds?<br/>
 * $x(t) = x_0 + v_0 * t - 0.5 * g * t^2$
 */
public class VerticalBallThrow {

  /** Compute the position of a ball, preventing it from falling through earth
   * @param x0 the height of the thrower, i.e., the initial vertical position
   * @param v0 the vertical upward velocity with which the ball is thrown
   * @param t the time at which we want to get the position $x(t)$
   * @return the position $x(t)$ of the ball at time step t
   */
  static double position(double x0, double v0, double t) {
    final double result = x0 + (v0 * t) - 0.5d * 9.80665d * t * t;
    return (result > 0d) ? result : 0d;
  }

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    try(Scanner scanner = new Scanner(System.in)) { // initiate reading from System.in, ignore for now
      System.err.println("Enter␣size␣x0␣of␣personin␣m:"); //$NON-NLS-1$
      double x0 = scanner.nextDouble(); // read initial vertical position $x_0$
      System.err.println("Enter␣initial␣upward␣velocity␣v0␣of␣ball␣in␣m/s:"); //$NON-NLS-1$
      double v0 = scanner.nextDouble(); // read initial velocity upwards $v_0$
      System.err.println("Enter␣time␣t␣in␣s:"); //$NON-NLS-1$
      double t = scanner.nextDouble();  // read the time $t$
      System.out.println(position(x0, v0, t)); // compute and print position
    }
  }
}
```

## Fixed Problem: Ball Cannot Fall through Earth Anymore

- We now fix this problem by modifying `position` to first check whether the result is positive and return 0 otherwise
- (For demonstration purposes, I therefore create a copy of both the source and the test package named `cn.edu.hfuu.iao_fix1`)
- The new code looks like this
- And the test code stays the same (only the package name changed)

# Fixed Problem: Ball Cannot Fall through Earth Anymore

## Listing: Vertical Ball Throw, Common Case Test

```
package cn.edu.hfuu.iao_fix1; // <-- package name changed for demo purposes

import org.junit.Assert;
import org.junit.Test;

/** Our first test class */
public class VerticalBallThrowPositionTest {

  /** test the position for $x_0=0m$, $v_0=g=0.90665m/s$, $t?=1s$ */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x00_v0g_t1() {
    Assert.assertEquals(4.903325d,                           // the expected value
        VerticalBallThrow.position(0d, 9.80665d, 1d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }

  /** test the position for $x_0=1m$, $v_0=32m/s$, $t?=2s$ */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x01_v032_t2() {
    Assert.assertEquals(45.3867d,                     // the expected value
        VerticalBallThrow.position(1d, 32d, 2d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }

  /** test the position for $x_0=2m$, $v_0=15m/s$, $t?=3s$ */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x02_v015_t3() {
    Assert.assertEquals(2.870075d,                    // the expected value
        VerticalBallThrow.position(2d, 15d, 3d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }
}
```

## Fixed Problem: Ball Cannot Fall through Earth Anymore

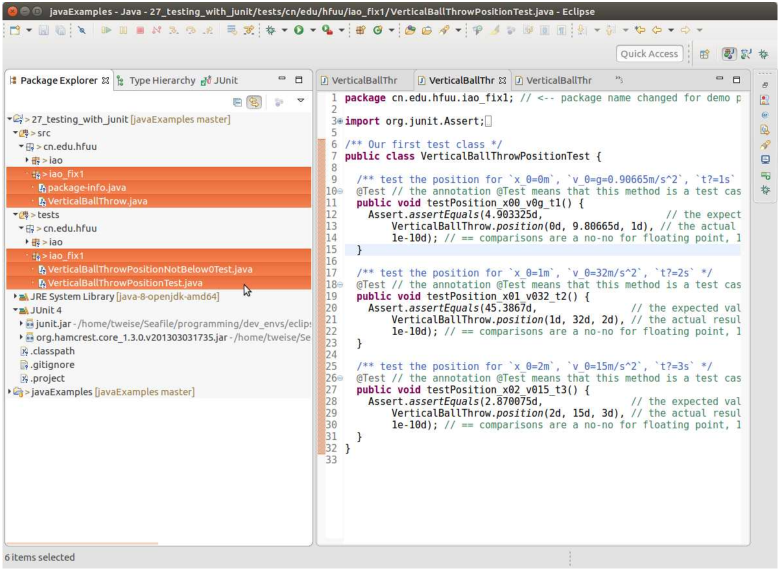
### Listing: Vertical Ball Throw, Border Case Test

```
package cn.edu.hfuu.iao_fix1; // <-- package name changed for demo purposes

import org.junit.Assert;
import org.junit.Test;

/** Our second test class: the ball cannot fall below 0m */
public class VerticalBallThrowPositionNotBelow0Test {

  /** test the position for x0 = 1m, v0 = 10m/s, t = 1000s */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_not_below_0_x01_v010_t1000() {
    Assert.assertEquals(0d,                                // the expected value
        VerticalBallThrow.position(1d, 10d, 1000d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }
}
```

## Fixed Problem: Ball Cannot Fall through Earth Anymore

- We now fix this problem by modifying `position` to first check whether the result is positive and return 0 otherwise
- (For demonstration purposes, I therefore create a copy of both the source and the test package named `cn.edu.hfuu.iao_fix1`)
- The new code looks like this
- And the test code stays the same (only the package name changed)
- We can now execute the tests again

## Fixed Problem: Ball Cannot Fall through Earth Anymore



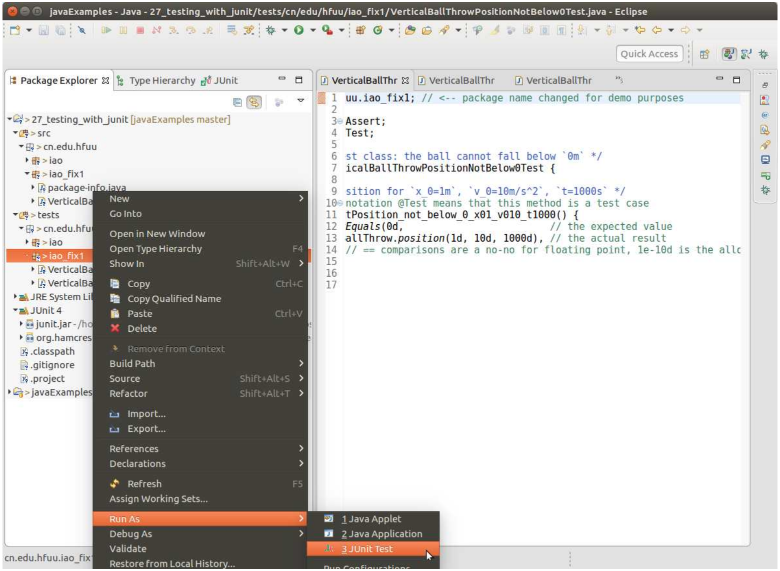
javaExamples - Java - 27_testing_with_junit/tests/cn/edu/hfuu/iao_fix1/VerticalBallThrowPositionTest.java - Eclipse

```
> 27_testing_with_junit [javaExamples master]
  > src
    > cn.edu.hfuu
      > iao
      > iao_fix1
        package-info.java
        VerticalBallThrow.java
  > tests
    > cn.edu.hfuu
      > iao
      > iao_fix1
        VerticalBallThrowPositionNotBelow0Test.java
        VerticalBallThrowPositionTest.java
  JRE System Library [java-8-openjdk-amd64]
  JUnit 4
    junit.jar - /home/tweise/Seafile/programming/dev_envs/eclip
    org.hamcrest.core_1.3.0.v201303031735.jar - /home/tweise/Se
  .classpath
  .gitignore
  .project
> javaExamples [javaExamples master]
```

```java
package cn.edu.hfuu.iao_fix1; // <-- package name changed for demo p

import org.junit.Assert;

/** Our first test class */
public class VerticalBallThrowPositionTest {

  /** test the position for `x_0=0m`, `v_0=g=0.90665m/s^2`, `t?=1s`
  @Test // the annotation @Test means that this method is a test cas
  public void testPosition_x00_v0g_t1() {
    Assert.assertEquals(4.903325d,                      // the expect
        VerticalBallThrow.position(0d, 9.80665d, 1d), // the actual
        1e-10d); // == comparisons are a no-no for floating point, 1
  }

  /** test the position for `x_0=1m`, `v_0=32m/s^2`, `t?=2s` */
  @Test // the annotation @Test means that this method is a test cas
  public void testPosition_x01_v032_t2() {
    Assert.assertEquals(45.3867d,                   // the expected val
        VerticalBallThrow.position(1d, 32d, 2d), // the actual resul
        1e-10d); // == comparisons are a no-no for floating point, 1
  }

  /** test the position for `x_0=2m`, `v_0=15m/s^2`, `t?=3s` */
  @Test // the annotation @Test means that this method is a test cas
  public void testPosition_x02_v015_t3() {
    Assert.assertEquals(2.870075d,                  // the expected val
        VerticalBallThrow.position(2d, 15d, 3d), // the actual resul
        1e-10d); // == comparisons are a no-no for floating point, 1
  }
}
```

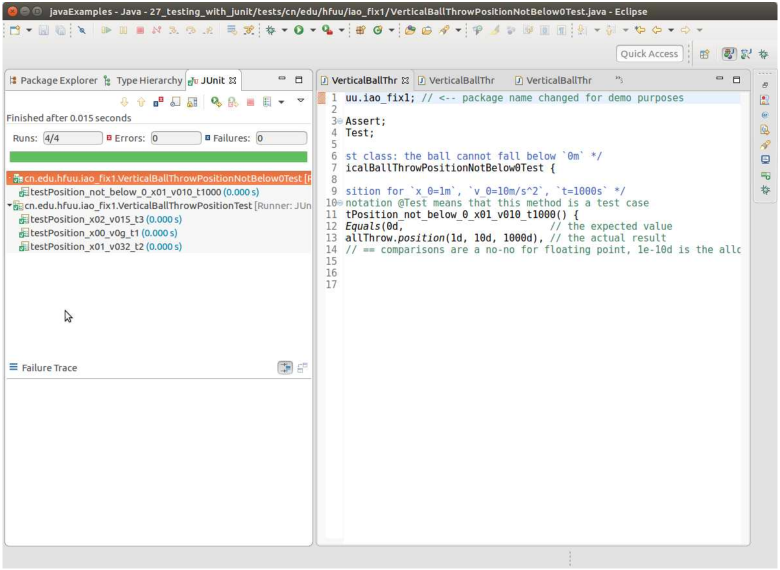
6 items selected

## Fixed Problem: Ball Cannot Fall through Earth Anymore



```
uu.iao_fix1; // <-- package name changed for demo purposes

Assert;
Test;

st class: the ball cannot fall below `0m` */
icalBallThrowPositionNotBelow0Test {

sition for `x_0=1m`, `v_0=10m/s^2`, `t=1000s` */
notation @Test means that this method is a test case
tPosition_not_below_0_x01_v010_t1000() {
Equals(0d,                          // the expected value
allThrow.position(1d, 10d, 1000d), // the actual result
// == comparisons are a no-no for floating point, 1e-10d is the allc
```

# Fixed Problem: Ball Cannot Fall through Earth Anymore



javaExamples - Java - 27_testing_with_junit/tests/cn/edu/hfuu/iao_fix1/VerticalBallThrowPositionNotBelow0Test.java - Eclipse

Finished after 0.015 seconds

Runs: 4/4 Errors: 0 Failures: 0

```
cn.edu.hfuu.iao_fix1.VerticalBallThrowPositionNotBelow0Test [R
  testPosition_not_below_0_x01_v010_t1000 (0.000 s)
cn.edu.hfuu.iao_fix1.VerticalBallThrowPositionTest [Runner: JUn
  testPosition_x02_v015_t3 (0.000 s)
  testPosition_x00_v0g_t1 (0.000 s)
  testPosition_x01_v032_t2 (0.000 s)
```

```
uu.iao_fix1; // <-- package name changed for demo purposes

Assert;
Test;

st class: the ball cannot fall below `0m` */
icalBallThrowPositionNotBelow0Test {

sition for `x_0=1m`, `v_0=10m/s^2`, `t=1000s` */
notation @Test means that this method is a test case
tPosition_not_below_0_x01_v010_t1000() {
Equals(0d,                          // the expected value
allThrow.position(1d, 10d, 1000d), // the actual result
// == comparisons are a no-no for floating point, 1e-10d is the allc
```

## Fixed Problem: Ball Cannot Fall through Earth Anymore

- We now fix this problem by modifying `position` to first check whether the result is positive and return 0 otherwise
- (For demonstration purposes, I therefore create a copy of both the source and the test package named `cn.edu.hfuu.iao_fix1`)
- The new code looks like this
- And the test code stays the same (only the package name changed)
- We can now execute the tests again
- . . . and they succeed.

## Expecting Exceptions

- We should not just test whether our code produces correct output for correct input (whether “normal” or “border” cases)

## Expecting Exceptions

- We should not just test whether our code produces correct output for correct input (whether “normal” or “border” cases)
- We should also check whether it behaves reasonable if the inputs are incorrect

## Expecting Exceptions

- We should not just test whether our code produces correct output for correct input (whether “normal” or “border” cases)
- We should also check whether it behaves reasonable if the inputs are incorrect
- In Lesson 25: *Exceptions*, we have learned that reasonable then means “throws Exceptions”

## Expecting Exceptions

- We should not just test whether our code produces correct output for correct input (whether “normal” or “border” cases)
- We should also check whether it behaves reasonable if the inputs are incorrect
- In Lesson 25: *Exceptions*, we have learned that reasonable then means “throws Exceptions”
- In our case, this would mean that

## Expecting Exceptions

- We should not just test whether our code produces correct output for correct input (whether “normal” or “border” cases)
- We should also check whether it behaves reasonable if the inputs are incorrect
- In Lesson 25: *Exceptions*, we have learned that reasonable then means “throws Exceptions”
- In our case, this would mean that `position` should probably…

## Expecting Exceptions

- We should not just test whether our code produces correct output for correct input (whether "normal" or "border" cases)
- We should also check whether it behaves reasonable if the inputs are incorrect
- In Lesson 25: *Exceptions*, we have learned that reasonable then means "throws Exceptions"
- In our case, this would mean that `position` should probably...
    - throw an `IllegalArgumentException` if $x_0 < 0$

## Expecting Exceptions

- We should not just test whether our code produces correct output for correct input (whether "normal" or "border" cases)
- We should also check whether it behaves reasonable if the inputs are incorrect
- In Lesson 25: *Exceptions*, we have learned that reasonable then means "throws Exceptions"
- In our case, this would mean that `position` should probably...
  - throw an `IllegalArgumentException` if $x_0 < 0$
  - throw an `IllegalArgumentException` if $v_0 < 0$

## Expecting Exceptions

- We should not just test whether our code produces correct output for correct input (whether "normal" or "border" cases)
- We should also check whether it behaves reasonable if the inputs are incorrect
- In Lesson 25: *Exceptions*, we have learned that reasonable then means "throws Exceptions"
- In our case, this would mean that `position` should probably. . .
    - throw an `IllegalArgumentException` if $x_0 < 0$
    - throw an `IllegalArgumentException` if $v_0 < 0$
    - throw an `IllegalArgumentException` if $t_0 < 0$

## Expecting Exceptions

- We should not just test whether our code produces correct output for correct input (whether “normal” or “border” cases)
- We should also check whether it behaves reasonable if the inputs are incorrect
- In Lesson 25: *Exceptions*, we have learned that reasonable then means “throws Exceptions”
- In our case, this would mean that `position` should probably…
    - throw an `IllegalArgumentException` if $x_0 < 0$
    - throw an `IllegalArgumentException` if $v_0 < 0$
    - throw an `IllegalArgumentException` if $t_0 < 0$
    - throw an `ArithmeticException` if its result would overflow (i.e., become infinite, or NaN)

## Expecting Exceptions

- We should not just test whether our code produces correct output for correct input (whether "normal" or "border" cases)
- We should also check whether it behaves reasonable if the inputs are incorrect
- In Lesson 25: *Exceptions*, we have learned that reasonable then means "throws Exceptions"
- In our case, this would mean that `position` should probably...
  - throw an `IllegalArgumentException` if $x_0 < 0$
  - throw an `IllegalArgumentException` if $v_0 < 0$
  - throw an `IllegalArgumentException` if $t_0 < 0$
  - throw an `ArithmeticException` if its result would overflow (i.e., become infinite, or NaN)
- We can test this with JUnit tests which will fail if a specified exception is *not* thrown

## Expecting Exceptions

- We should not just test whether our code produces correct output for correct input (whether "normal" or "border" cases)
- We should also check whether it behaves reasonable if the inputs are incorrect
- In Lesson 25: *Exceptions*, we have learned that reasonable then means "throws Exceptions"
- In our case, this would mean that `position` should probably...
  - throw an `IllegalArgumentException` if $x_0 < 0$
  - throw an `IllegalArgumentException` if $v_0 < 0$
  - throw an `IllegalArgumentException` if $t_0 < 0$
  - throw an `ArithmeticException` if its result would overflow (i.e., become infinite, or NaN)
- We can test this with JUnit tests which will fail if a specified exception is *not* thrown
- Let's make such a test

## Test Cases Expecting Exceptions for Vertical Ball Throw

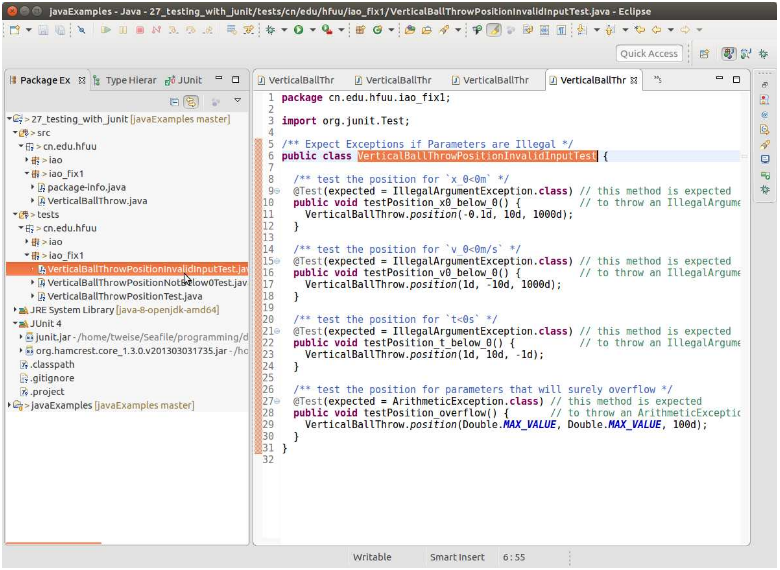
### Listing: Test Cases Expecting Exceptions for Vertical Ball Throw

```java
package cn.edu.hfuu.iao_fix1;

import org.junit.Test;

/** Expect Exceptions if Parameters are Illegal */
public class VerticalBallThrowPositionInvalidInputTest {

  /** test the position for x0 < 0m */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_x0_below_0() {          // to throw an IllegalArgumentException
    VerticalBallThrow.position(-0.1d, 10d, 1000d);
  }

  /** test the position for v0 < 0m/s */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_v0_below_0() {          // to throw an IllegalArgumentException
    VerticalBallThrow.position(1d, -10d, 1000d);
  }

  /** test the position for t < 0s */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_t_below_0() {           // to throw an IllegalArgumentException
    VerticalBallThrow.position(1d, 10d, -1d);
  }

  /** test the position for parameters that will surely overflow */
  @Test(expected = ArithmeticException.class) // this method is expected
  public void testPosition_overflow() {       // to throw an ArithmeticException
    VerticalBallThrow.position(Double.MAX_VALUE, Double.MAX_VALUE, 100d);
  }
}
```

## Running the New Tests

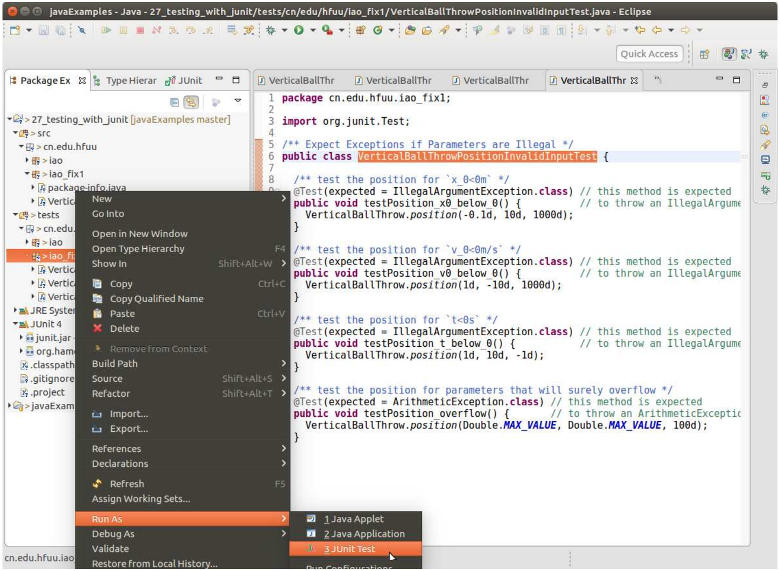
- So let us run the new tests

## Running the New Tests


javaExamples - Java - 27_testing_with_junit/tests/cn/edu/hfuu/iao_fix1/VerticalBallThrowPositionInvalidInputTest.java - Eclipse
Quick Access
Package Ex
Type Hierar
JUnit
27_testing_with_junit [javaExamples master]
src
cn.edu.hfuu
iao
iao_fix1
package-info.java
VerticalBallThrow.java
tests
cn.edu.hfuu
iao
iao_fix1
VerticalBallThrowPositionInvalidInputTest.ja
VerticalBallThrowPositionNotBelow0Test.jav
VerticalBallThrowPositionTest.java
JRE System Library [java-8-openjdk-amd64]
JUnit 4
junit.jar - /home/tweise/Seafile/programming/d
org.hamcrest.core_1.3.0.v201303031735.jar - /ho
.classpath
.gitignore
.project
javaExamples [javaExamples master]
VerticalBallThr
VerticalBallThr
VerticalBallThr
VerticalBallThr
Writable
Smart Insert
6 : 55

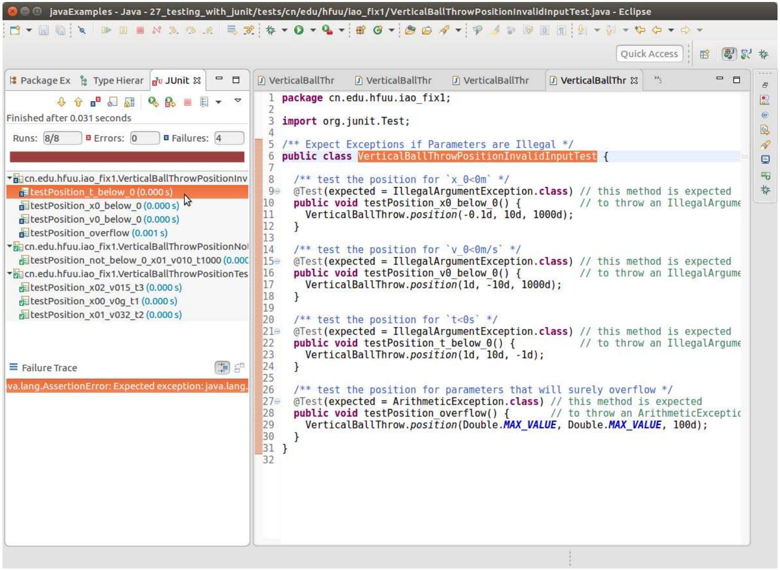
```java
package cn.edu.hfuu.iao_fix1;

import org.junit.Test;

/** Expect Exceptions if Parameters are Illegal */
public class VerticalBallThrowPositionInvalidInputTest {

  /** test the position for `x_0<0m` */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_x0_below_0() {          // to throw an IllegalArgume
    VerticalBallThrow.position(-0.1d, 10d, 1000d);
  }

  /** test the position for `v_0<0m/s` */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_v0_below_0() {          // to throw an IllegalArgume
    VerticalBallThrow.position(1d, -10d, 1000d);
  }

  /** test the position for `t<0s` */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_t_below_0() {           // to throw an IllegalArgume
    VerticalBallThrow.position(1d, 10d, -1d);
  }

  /** test the position for parameters that will surely overflow */
  @Test(expected = ArithmeticException.class) // this method is expected
  public void testPosition_overflow() {       // to throw an ArithmeticExceptic
    VerticalBallThrow.position(Double.MAX_VALUE, Double.MAX_VALUE, 100d);
  }
}
```

## Running the New Tests



```
package cn.edu.hfuu.iao_fix1;

import org.junit.Test;

/** Expect Exceptions if Parameters are Illegal */
public class VerticalBallThrowPositionInvalidInputTest {

  /** test the position for `x_0<0m` */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_x0_below_0() {         // to throw an IllegalArgume
    VerticalBallThrow.position(-0.1d, 10d, 1000d);
  }

  /** test the position for `v_0<0m/s` */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_v0_below_0() {         // to throw an IllegalArgume
    VerticalBallThrow.position(1d, -10d, 1000d);
  }

  /** test the position for `t<0s` */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_t_below_0() {          // to throw an IllegalArgume
    VerticalBallThrow.position(1d, 10d, -1d);
  }

  /** test the position for parameters that will surely overflow */
  @Test(expected = ArithmeticException.class) // this method is expected
  public void testPosition_overflow() {        // to throw an ArithmeticExceptio
    VerticalBallThrow.position(Double.MAX_VALUE, Double.MAX_VALUE, 100d);
  }
```

## Running the New Tests



javaExamples - Java - 27_testing_with_junit/tests/cn/edu/hfuu/iao_fix1/VerticalBallThrowPositionInvalidInputTest.java - Eclipse

Finished after 0.031 seconds

Runs: 8/8 Errors: 0 Failures: 4

```
cn.edu.hfuu.iao_fix1.VerticalBallThrowPositionInv
  testPosition_t_below_0 (0.000 s)
  testPosition_x0_below_0 (0.000 s)
  testPosition_v0_below_0 (0.000 s)
  testPosition_overflow (0.001 s)
cn.edu.hfuu.iao_fix1.VerticalBallThrowPositionNo
  testPosition_not_below_0_x01_v010_t1000 (0.000
cn.edu.hfuu.iao_fix1.VerticalBallThrowPositionTes
  testPosition_x02_v015_t3 (0.000 s)
  testPosition_x00_v0g_t1 (0.000 s)
  testPosition_x01_v032_t2 (0.000 s)
```

Failure Trace

```
va.lang.AssertionError: Expected exception: java.lang.
```

```java
package cn.edu.hfuu.iao_fix1;

import org.junit.Test;

/** Expect Exceptions if Parameters are Illegal */
public class VerticalBallThrowPositionInvalidInputTest {

  /** test the position for `x_0<0m` */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_x0_below_0() {          // to throw an IllegalArgume
    VerticalBallThrow.position(-0.1d, 10d, 1000d);
  }

  /** test the position for `v_0<0m/s` */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_v0_below_0() {          // to throw an IllegalArgume
    VerticalBallThrow.position(1d, -10d, 1000d);
  }

  /** test the position for `t<0s` */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_t_below_0() {           // to throw an IllegalArgume
    VerticalBallThrow.position(1d, 10d, -1d);
  }

  /** test the position for parameters that will surely overflow */
  @Test(expected = ArithmeticException.class) // this method is expected
  public void testPosition_overflow() {       // to throw an ArithmeticExceptic
    VerticalBallThrow.position(Double.MAX_VALUE, Double.MAX_VALUE, 100d);
  }
}
```

## Running the New Tests

- So let us run the new tests
- Obviously, the new tests fail, since we do not throw any exception in our code

## Fixed Problem: Code is now throwing Exceptions

- Let's now fix our code to throw appropriate exceptions

## Fixed Problem: Code is now throwing Exceptions

- Let's now fix our code to throw appropriate exceptions
- (For demonstration purposes, I therefore create a copy of both the source and the test package named `cn.edu.hfuu.iao_fix2`)

## Fixed Problem: Code is now throwing Exceptions

- Let's now fix our code to throw appropriate exceptions
- (For demonstration purposes, I therefore create a copy of both the source and the test package named `cn.edu.hfuu.iao_fix2`)
- The new code looks like this

## Fixed Problem: Code is now throwing Exceptions

### Listing: Vertical Ball Throw, Exception Fix

```
package cn.edu.hfuu.iao_fix2; // <-- package name changed again for demo purposes

import java.util.Scanner;

/**
 * A ball is thrown vertically upwards into the air by a $x_0m$ tall person
 * with velocity $v_0m/s$. Where is it after t seconds?<br/>
 * $x(t) = x_0 + v_0 * t - 0.5 * g * t^2$
 */
public class VerticalBallThrow {

  /** Compute the position of a ball (preventing it from falling through earth
   * and checking its arguments and results.
   * @param x0 the height of the thrower, i.e., the initial vertical position
   * @param v0 the vertical upward velocity with which the ball is thrown
   * @param t the time at which we want to get the position $x(t)$
   * @return the position $x(t)$ of the ball at time step t
   */
  static double position(double x0, double v0, double t) {
    if ((x0 < 0d) || (v0 < 0d) || (t < 0d)) { // check invalid arguments
      throw new IllegalArgumentException("Invalid␣arguments␣x0="  //$NON-NLS-1$
                + x0 + ",␣v0="+v0 + "t=" + t);//$NON-NLS-1$//$NON-NLS-2$
    }
    final double result = x0 + (v0 * t) - 0.5d * 9.80665d * t * t;
    if(!(Double.isFinite(result))) { // if result is infinite or NaN
      throw new ArithmeticException("Arguments␣x0="  //$NON-NLS-1$
          + x0 + ",␣v0="+v0 + "t=" + t + //$NON-NLS-1$//$NON-NLS-2$
          "␣lead␣to␣non-finite␣result␣" + result); //$NON-NLS-1$
    }
    return (result > 0d) ? result : 0d;
  }

  /** The main routine
   * @param args
   *           we ignore this parameter for now */
  public static final void main(String[] args) {
    try(Scanner scanner = new Scanner(System.in)) { // initiate reading from System.in, ignore for now
      System.err.println("Enter␣size␣x0␣of␣personin␣m:"); //$NON-NLS-1$
      double x0 = scanner.nextDouble(); // read initial vertical position $x_0$
      System.err.println("Enter␣initial␣upward␣velocity␣v0␣of␣ball␣in␣m/s:"); //$NON-NLS-1$
      double v0 = scanner.nextDouble(); // read initial velocity upwards $v_0$
      System.err.println("Enter␣time␣t␣in␣s:"); //$NON-NLS-1$
      double t = scanner.nextDouble();  // read the time $t$
      System.out.println(position(x0, v0, t)); // compute and print position
    }
  }
}
```

## Fixed Problem: Code is now throwing Exceptions

- Let's now fix our code to throw appropriate exceptions
- (For demonstration purposes, I therefore create a copy of both the source and the test package named `cn.edu.hfuu.iao_fix2`)
- The new code looks like this
- And the test code stays the same (only the package name changed)

## Fixed Problem: Code is now throwing Exceptions

Listing: Vertical Ball Throw, Common Case Test

```
package cn.edu.hfuu.iao_fix2; // <-- package name changed again for demo purposes

import org.junit.Assert;
import org.junit.Test;

/** Our first test class */
public class VerticalBallThrowPositionTest {

  /** test the position for x_0 = 0m, v_0 = g = 0.90665m/s, t? = 1s */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x00_v0g_t1() {
    Assert.assertEquals(4.903325d,                          // the expected value
        VerticalBallThrow.position(0d, 9.80665d, 1d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }

  /** test the position for x_0 = 1m, v_0 = 32m/s, t? = 2s */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x01_v032_t2() {
    Assert.assertEquals(45.3867d,                    // the expected value
        VerticalBallThrow.position(1d, 32d, 2d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }

  /** test the position for x_0 = 2m, v_0 = 15m/s, t? = 3s */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_x02_v015_t3() {
    Assert.assertEquals(2.870075d,                   // the expected value
        VerticalBallThrow.position(2d, 15d, 3d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }
}
```

## Fixed Problem: Code is now throwing Exceptions

Listing: Vertical Ball Throw, Border Case Test

```
package cn.edu.hfuu.iao_fix2; // <-- package name changed again for demo purposes

import org.junit.Assert;
import org.junit.Test;

/** Our second test class: the ball cannot fall below 0m */
public class VerticalBallThrowPositionNotBelow0Test {

  /** test the position for x0=1m, v0=10m/s, t=1000s */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_not_below_0_x01_v010_t1000() {
    Assert.assertEquals(0d,                                  // the expected value
        VerticalBallThrow.position(1d, 10d, 1000d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is the allowed deviation
  }
}
```

## Fixed Problem: Code is now throwing Exceptions

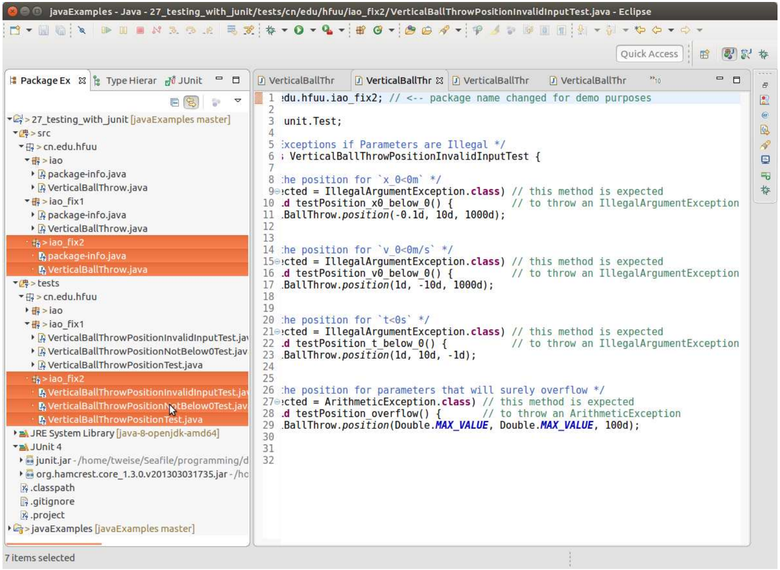
### Listing: Vertical Ball Throw, Invalid Argument Test

```
package cn.edu.hfuu.iao_fix2; // <-- package name changed for demo purposes

import org.junit.Test;

/** Expect Exceptions if Parameters are Illegal */
public class VerticalBallThrowPositionInvalidInputTest {

  /** test the position for x0 < 0m */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_x0_below_0() {         // to throw an IllegalArgumentException
    VerticalBallThrow.position(-0.1d, 10d, 1000d);
  }

  /** test the position for v0 < 0m/s */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_v0_below_0() {         // to throw an IllegalArgumentException
    VerticalBallThrow.position(1d, -10d, 1000d);
  }

  /** test the position for t < 0s */
  @Test(expected = IllegalArgumentException.class) // this method is expected
  public void testPosition_t_below_0() {          // to throw an IllegalArgumentException
    VerticalBallThrow.position(1d, 10d, -1d);
  }

  /** test the position for parameters that will surely overflow */
  @Test(expected = ArithmeticException.class) // this method is expected
  public void testPosition_overflow() {       // to throw an ArithmeticException
    VerticalBallThrow.position(Double.MAX_VALUE, Double.MAX_VALUE, 100d);
  }
}
```

## Fixed Problem: Code is now throwing Exceptions

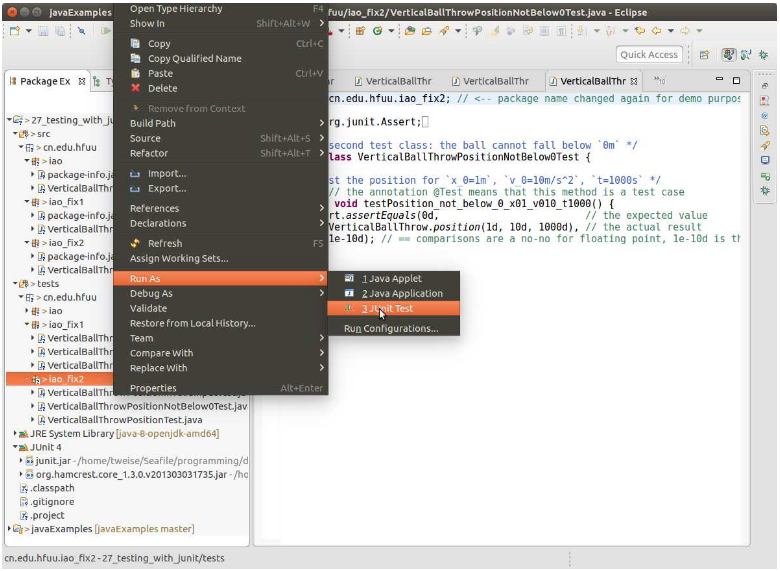
- Let's now fix our code to throw appropriate exceptions
- (For demonstration purposes, I therefore create a copy of both the source and the test package named `cn.edu.hfuu.iao_fix2`)
- The new code looks like this
- And the test code stays the same (only the package name changed)
- We can now execute the tests again

## Fixed Problem: Code is now throwing Exceptions


javaExamples - Java - 27_testing_with_junit/tests/cn/edu/hfuu/iao_fix2/VerticalBallThrowPositionInvalidInputTest.java - Eclipse
Quick Access
Package Ex
Type Hierar
JUnit
VerticalBallThr
VerticalBallThr
VerticalBallThr
VerticalBallThr
27_testing_with_junit [javaExamples master]
src
cn.edu.hfuu
iao
package-info.java
VerticalBallThrow.java
iao_fix1
package-info.java
VerticalBallThrow.java
iao_fix2
package-info.java
VerticalBallThrow.java
tests
cn.edu.hfuu
iao
iao_fix1
VerticalBallThrowPositionInvalidInputTest.ja
VerticalBallThrowPositionNotBelow0Test.jav
VerticalBallThrowPositionTest.java
iao_fix2
VerticalBallThrowPositionInvalidInputTest.ja
VerticalBallThrowPositionNotBelow0Test.jav
VerticalBallThrowPositionTest.java
JRE System Library [java-8-openjdk-amd64]
JUnit 4
junit.jar - /home/tweise/Seafile/programming/d
org.hamcrest.core_1.3.0.v201303031735.jar - /hc
.classpath
.gitignore
.project
javaExamples [javaExamples master]
7 items selected

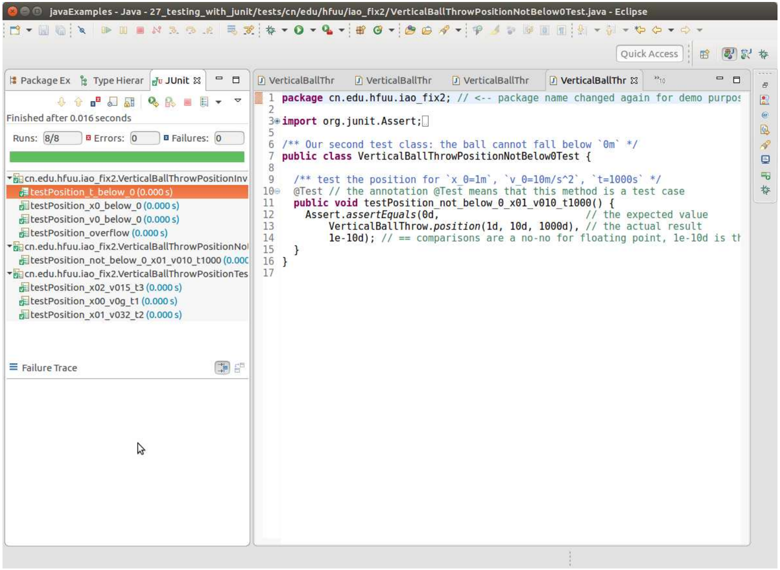
```
:du.hfuu.iao_fix2; // <-- package name changed for demo purposes

 unit.Test;

:xceptions if Parameters are Illegal */
; VerticalBallThrowPositionInvalidInputTest {

:he position for `x_0<0m` */
:cted = IllegalArgumentException.class) // this method is expected
.d testPosition_x0_below_0() {           // to throw an IllegalArgumentException
.BallThrow.position(-0.1d, 10d, 1000d);


:he position for `v_0<0m/s` */
:cted = IllegalArgumentException.class) // this method is expected
.d testPosition_v0_below_0() {           // to throw an IllegalArgumentException
.BallThrow.position(1d, -10d, 1000d);


:he position for `t<0s` */
:cted = IllegalArgumentException.class) // this method is expected
.d testPosition_t_below_0() {            // to throw an IllegalArgumentException
.BallThrow.position(1d, 10d, -1d);


:he position for parameters that will surely overflow */
:cted = ArithmeticException.class) // this method is expected
.d testPosition_overflow() {        // to throw an ArithmeticException
.BallThrow.position(Double.MAX_VALUE, Double.MAX_VALUE, 100d);
```

## Fixed Problem: Code is now throwing Exceptions



```
cn.edu.hfuu.iao_fix2; // <-- package name changed again for demo purpos

rg.junit.Assert;

second test class: the ball cannot fall below `0m` */
lass VerticalBallThrowPositionNotBelow0Test {

st the position for `x_0=1m`, `v_0=10m/s^2`, `t=1000s` */
// the annotation @Test means that this method is a test case
 void testPosition_not_below_0_x01_v010_t1000() {
rt.assertEquals(0d,                        // the expected value
VerticalBallThrow.position(1d, 10d, 1000d), // the actual result
1e-10d); // == comparisons are a no-no for floating point, 1e-10d is th
```

## Fixed Problem: Code is now throwing Exceptions


javaExamples - Java - 27_testing_with_junit/tests/cn/edu/hfuu/iao_fix2/VerticalBallThrowPositionNotBelow0Test.java - Eclipse
Quick Access
Package Ex
Type Hierar
JUnit
VerticalBallThr
VerticalBallThr
VerticalBallThr
VerticalBallThr
Failure Trace

```
Finished after 0.016 seconds
Runs: 8/8   Errors: 0   Failures: 0

cn.edu.hfuu.iao_fix2.VerticalBallThrowPositionInv
  testPosition_t_below_0 (0.000 s)
  testPosition_x0_below_0 (0.000 s)
  testPosition_v0_below_0 (0.000 s)
  testPosition_overflow (0.000 s)
cn.edu.hfuu.iao_fix2.VerticalBallThrowPositionNo
  testPosition_not_below_0_x01_v010_t1000 (0.000
cn.edu.hfuu.iao_fix2.VerticalBallThrowPositionTes
  testPosition_x02_v015_t3 (0.000 s)
  testPosition_x00_v0g_t1 (0.000 s)
  testPosition_x01_v032_t2 (0.000 s)
```

```java
package cn.edu.hfuu.iao_fix2; // <-- package name changed again for demo purpos

import org.junit.Assert;

/** Our second test class: the ball cannot fall below `0m` */
public class VerticalBallThrowPositionNotBelow0Test {

  /** test the position for `x_0=1m`, `v_0=10m/s^2`, `t=1000s` */
  @Test // the annotation @Test means that this method is a test case
  public void testPosition_not_below_0_x01_v010_t1000() {
    Assert.assertEquals(0d,                           // the expected value
        VerticalBallThrow.position(1d, 10d, 1000d), // the actual result
        1e-10d); // == comparisons are a no-no for floating point, 1e-10d is th
  }
}
```

## Fixed Problem: Code is now throwing Exceptions

- Let's now fix our code to throw appropriate exceptions
- (For demonstration purposes, I therefore create a copy of both the source and the test package named `cn.edu.hfuu.iao_fix2`)
- The new code looks like this
- And the test code stays the same (only the package name changed)
- We can now execute the tests again
- . . . and they succeed.

## Test-Driven Development

- We have now hardened our `position` method against the most common problems that could occur

## Test-Driven Development

- We have now hardened our `position` method against the most common problems that could occur
- Every piece of code that we ship should be covered with such tests

## Test-Driven Development

- We have now hardened our `position` method against the most common problems that could occur
- Every piece of code that we ship should be covered with such tests
- Matter of fact: The tests should be created before the code!

## Test-Driven Development

- We have now hardened our `position` method against the most common problems that could occur
- Every piece of code that we ship should be covered with such tests
- Matter of fact: The tests should be created before the code!
- In the so-called *Test-Driven Development*

## Test-Driven Development

- We have now hardened our `position` method against the most common problems that could occur
- Every piece of code that we ship should be covered with such tests
- Matter of fact: The tests should be created before the code!
- In the so-called *Test-Driven Development*,
    - the specification of the software is first turned into interfaces and tests for these interfaces

## Test-Driven Development

- We have now hardened our `position` method against the most common problems that could occur
- Every piece of code that we ship should be covered with such tests
- Matter of fact: The tests should be created before the code!
- In the so-called *Test-Driven Development*,
  - the specification of the software is first turned into interfaces and tests for these interfaces
  - *Afterwards*, the interfaces are implemented

## Test-Driven Development

- We have now hardened our `position` method against the most common problems that could occur
- Every piece of code that we ship should be covered with such tests
- Matter of fact: The tests should be created before the code!
- In the so-called *Test-Driven Development*,
  - the specification of the software is first turned into interfaces and tests for these interfaces
  - *Afterwards*, the interfaces are implemented
- By working in such a way, we can prevent the programmer from lazily making tests which fit to her/his code

## Test-Driven Development

- We have now hardened our `position` method against the most common problems that could occur
- Every piece of code that we ship should be covered with such tests
- Matter of fact: The tests should be created before the code!
- In the so-called *Test-Driven Development*,
  - the specification of the software is first turned into interfaces and tests for these interfaces
  - *Afterwards*, the interfaces are implemented
- By working in such a way, we can prevent the programmer from lazily making tests which fit to her/his code
- And we know, at any stage of development, that we are working with correct code

## Test-Driven Development

- We have now hardened our `position` method against the most common problems that could occur
- Every piece of code that we ship should be covered with such tests
- Matter of fact: The tests should be created before the code!
- In the so-called *Test-Driven Development*,
    - the specification of the software is first turned into interfaces and tests for these interfaces
    - *Afterwards*, the interfaces are implemented
- By working in such a way, we can prevent the programmer from lazily making tests which fit to her/his code
- And we know, at any stage of development, that we are working with correct code
- Regardless whether or not this method is used, it is clear that testing is absolutely important

## Summary

- We have learned about unit testing using JUnit
- When testing our code, we should always cover
  - the common use case
  - the border cases which are unlikely to happen but still valid use cases
  - the case of invalid input (also to ensure that our code properly and early throws exceptions)
- Ideally, all produced code should be covered by tests
- Tests cannot proof that there are no errors, they can just reduce their likelihood
- Tests allow us to ensure that different versions of our software stay compatible (if the new version passes old tests)
- Tests are used in conjunction with debugging

# 谢谢

# Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog