# PG4200: Algorithms And Data Structures

# Lesson 01: Introduction

Dr. Andrea Arcuri

# Contact

- “Discussion Forum”
- For announcements, and questions of general interest for the whole class
- Use the discussion forum **instead of** sending me emails
  - If you send me a private email/message, I will tell you to post it on the Discussion Forum. However, if I am busy (as most of the time…), *I might just ignore your message…*

# Course Info

- 12 lessons, once a weak
- Class 1-9: *Foundation*, algorithms and data structures that all of you will need to know if you are going to work as a developer/programmer/engineer/etc.
- Class 10-12: *Advanced*, interesting and important topics, but that not all of you will need in your daily jobs
- Check each week to see if changes in schedule (time and room)

# Foundation

1. Intro
2. Stack/Queue
3. Runtime analysis and Sorting
4. Advanced Sorting
5. Tree Maps
6. Hash Maps
7. Streams
8. Graphs
9. Regular Expressions

# Advanced

10. Optimization Algorithms
11. Evolutionary Algorithms
12. Data Compression

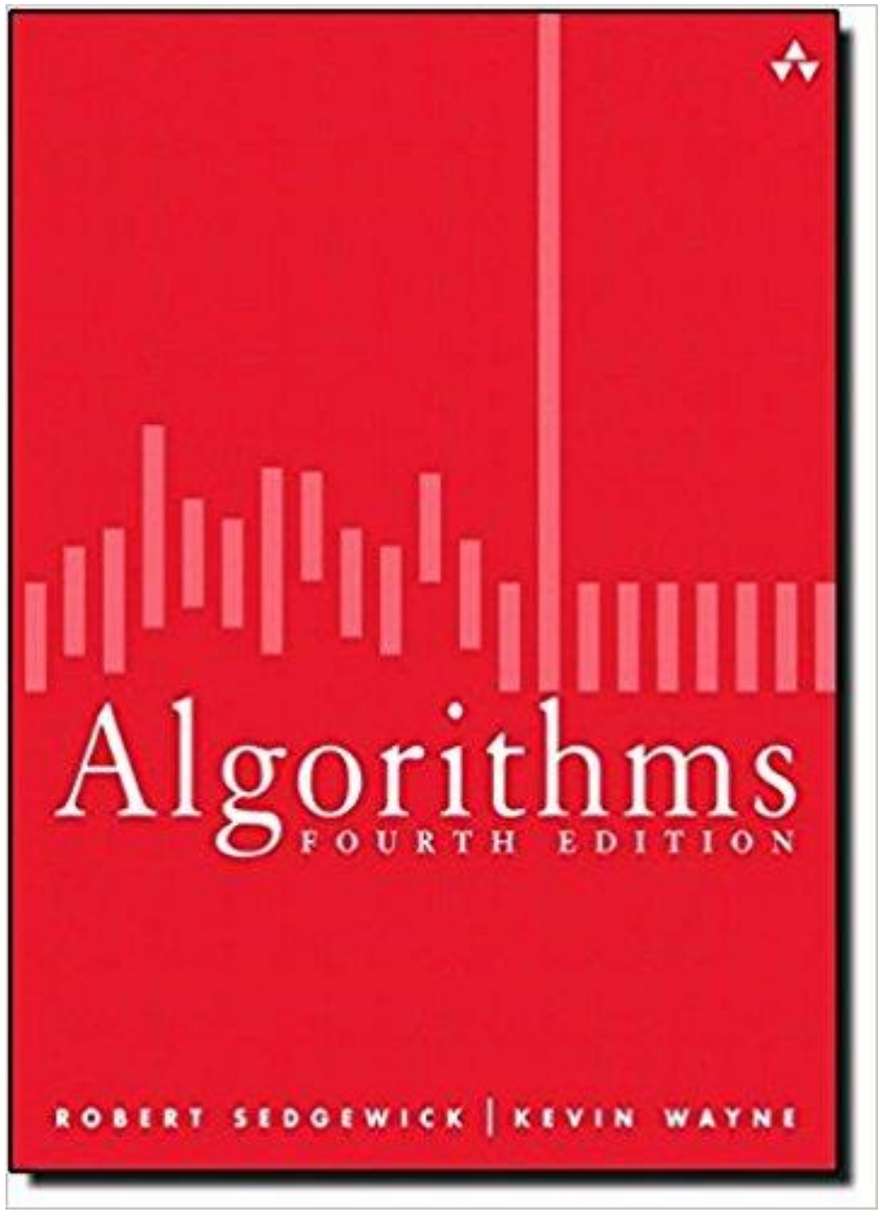
# Class Structure

- “Usually”  2+2
  - 2-3 hours of lecture: code and slides
  - 1-2 hours in which you should do exercises and get help

- **IMPORTANT**: the 1-2 hours after lecture is not only for exercises. If you are falling behind, or you need some more revision, you can ask for my help on anything related to coding

# Course Material

- Algorithms (4th Edition)
- We **actually use it** in the course, so *you should really try to get a copy*
  - and likely you ll need it also outside of this course
- Note, however, that there are plenty of resources on internet to learn Algorithms



# Git Repository

- https://github.com/arcuri82/pg4200
- Note: pull often, as new material and corrections can be added during the course
- If you add code (eg working on exercises), recall to do a Git “rebase” or “stash” before pulling (which otherwise might fail)

# Why Studying Algorithms?

- Algorithms and data structures are the foundation of programming
  - ie, the base building blocks
- Impact on all fields of engineering and science
  - internet, computer graphics, social networks, biology, physics, etc.
- In this course, we will not build whole applications (eg, web or mobile), but rather concentrate on the building blocks to enable it in the next courses

# Like it or not…

- … used in practically most programs you will write
- … algorithms and data structures are very common exercises in job interviews
  - Especially for juniors straight out of university
  - Don’t be surprised to be asked to write a stack or a queue class on a whiteboard…
  - … or other advanced algorithms

**Max Howell**
@mxcl

Follow

Google: 90% of our engineers use the software you wrote (Homebrew), but you can’t invert a binary tree on a whiteboard so fuck off.

10:07 AM - 10 Jun 2015

**7,108** Retweets **8,637** Likes

# Math

- (un)fortunately there is **math** involved…
- Math: can tell you **WHY** a particular algorithm or data structure performs in a certain way
  - As an engineer, you need to make conscious decisions about what you use
- I like math, but, in contrast to lecturers of previous years or in other universities, I put more emphasis on the programming side…

# Coding

- This course is heavily based on coding
- There are going to be slides, but in class we will spend most of the time going through source code
- **Slides will often just be a quick overview of what we will cover in the code**

# Necessary Tools

- Java 8 JDK
  - JDK 11 will come out during the course, so will not use it this year
  - JDK 9 and 10 should be avoided, as non-LTS (Long-Term-Support), and having lifespan of just 6 months
- Git
- IntelliJ Ultimate Edition
  - you might want to install JetBrains Toolbox first

# If You Skip Class…

- Usually acceptable that a student skips 1-2 classes
- You are supposed to attend, although no strict checks
- If you skip too many classes, it is **YOUR** responsibility to catch up and find out what done in class
  - you are adults, after all…

# Exams

- 3 hour written exam
- Expect around 10 questions/exercises
  - Based on slides and all code in the repository
  - Note: this might change… but if so, you will be told before the exam
- Expect theoretical questions and also the writing of code on paper (at least 1, but no more than 50%)
- The exam is based on what covered in the Git repository
  - the book just gives you extra info and clarifications
  - exam is based on code in the Git repository, NOT the one of the book

# Difficulty


- This is a **difficult** course, more difficult than what you might be used to and expect
- Not uncommon that *many* students **fail** the exam
- You need to study **each week**, and do **all** the exercises
- Forget trying to learn it in just a couple of weeks/days before the exam…
  - I keep saying it every time, but students do not believe me, and then are surprised when they get an F…

# Arrays and Lists

# Containers

- When writing programs, need to manipulate data
  - e.g., adding songs to a playlist
  - e.g., adding an item on a shopping cart like on Amazon
- *Adding*, *removing* and *searching* for elements
- Different *data structures* with different properties

# Arrays

```
int[] array = new int[3];
array[0] = 5;
array[1] = 3;
```

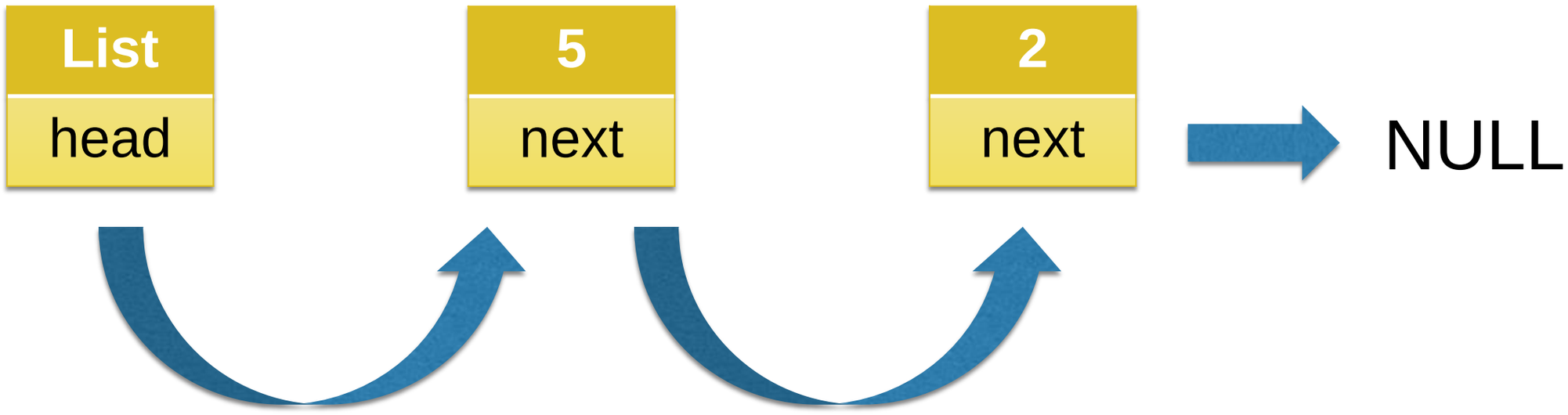
| [0] | [1] | [2] |
|---|---|---|
| 5 | 3 | 0 |

- Easy, direct access to all elements
- Possible issues when deleting elements (ie holes)
- Fixed size, decided at creation
  - If you create it for 10 elements, but then you need 11, you would need to create a new array
- Arrays are low-level constructs of Java language

# Lists

- Conceptually like arrays, but no fixed size
  - ie, you can add as many elements as you want, as long as you have enough memory
- Lists (and all data structures will see in this course) are Java objects, and not treated specially like arrays
- 2 main ways to “implement” them
- *Array-backed*: internally storing an array. Need to create new one and move over old data when full.
- *Linked-nodes*: each element has its own node object, and nodes are connected with object pointers/links (see next slide)

# Linked Lists



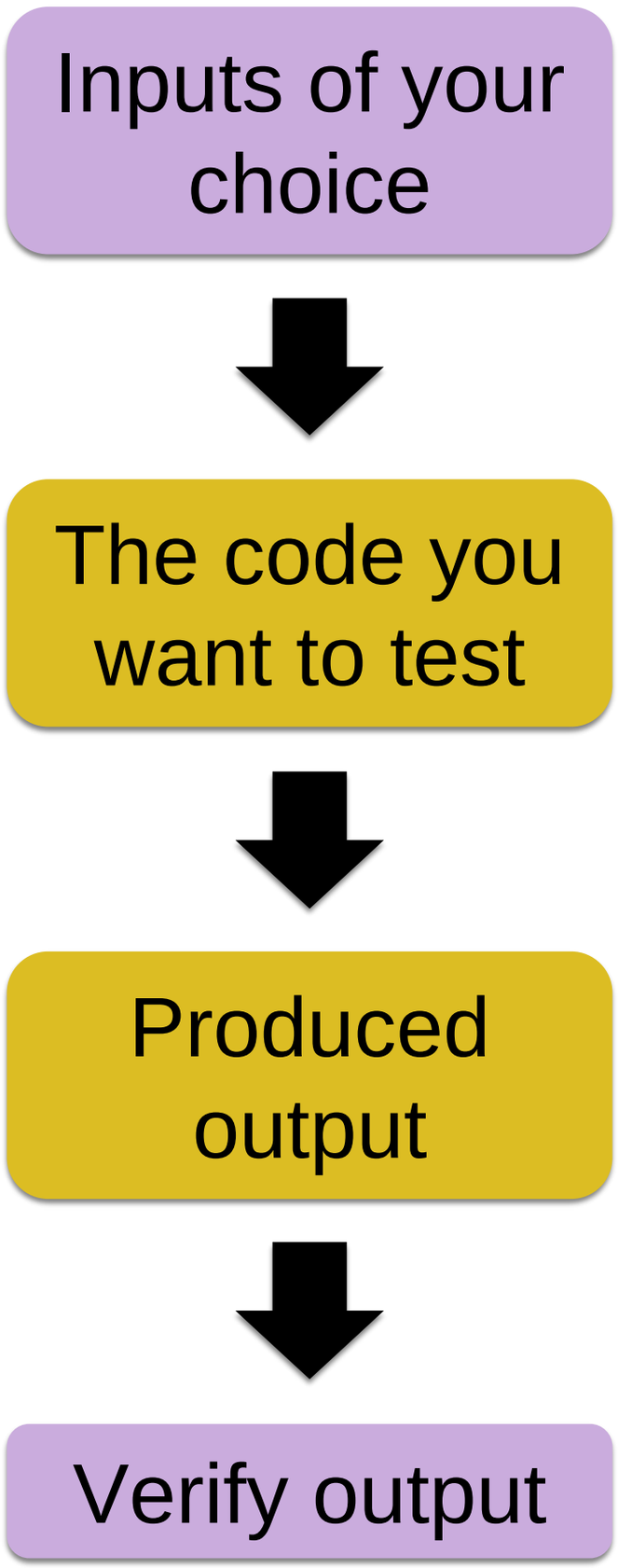

- A *node* for each element (they are objects)
- *Links* from node to node (eg, in a *next* field)
- Dynamic, not fixed size
- Accessing elements more difficult, as need to traverse the links
- Going into more details on *links* and *memory* in the *next class*
  - reason is that you first need to learn about Stacks

# Unit Testing

# Bugs

- Software has bugs, ie, errors/mistakes
- Not just students, but also professional engineers with decades of experiences make mistakes, quite often…
  - not necessarily because they are bad, but just that code nowadays can become very, very complex
- You want to check if the code you write is actually doing what it is supposed to do

# Testing




- Cannot guarantee the code is correct, but can increase your confidence in it
- You want the checking of your code to be automated
- In each test case, you **verify** the **output** generated when you run the **code** with the **inputs** of your choice

# Writing Unit Tests

- Using a library called **JUnit**
  - Note: how to configure Maven to import third-party libraries is not part of this course (and so not on the exam), but you can ask me in the breaks if you are curious (for some of you, we will dig into its low level details in Enterprise Programming 1 next semester)

- Regular code in "*src/main/java*" folder
- Test code in "*src/test/java*" folder
- A test class is just a Java class with @ *annotations*
- A test class for a class called *Foo.java* will be called *FooTest.java*, in the same package

# Main @ Annotations

- @**Test**: mark a method as a test
- @**BeforeEach**: execute method before each test
- @**BeforeAll**: execute method once before any of the tests is started
- @**AfterEach**, @**AfterAll**: same, but after the tests
- @**Disable**: temporarily disable a test, which is not going to be run

# Assertions

- When you have an output, you need to *verify* if correct
- Extra code (assertion methods) that throws an error if the output is not equal to the expected one
- *assertEquals(expected, output)*
  - throw error if *output* variable is not equal to the *expected* one
- *assertTrue(condition)*
  - throw error if *condition* is false
- *assertNotNull(output)*
  - throw error if *output* is null

# Test Example

Mark method as a test

```
@Test
public void testBase(){

        int[] array = {1, 2, 3};

        int res = ArrayExample.sum(array);

        assertEquals(6, res);
    }
```

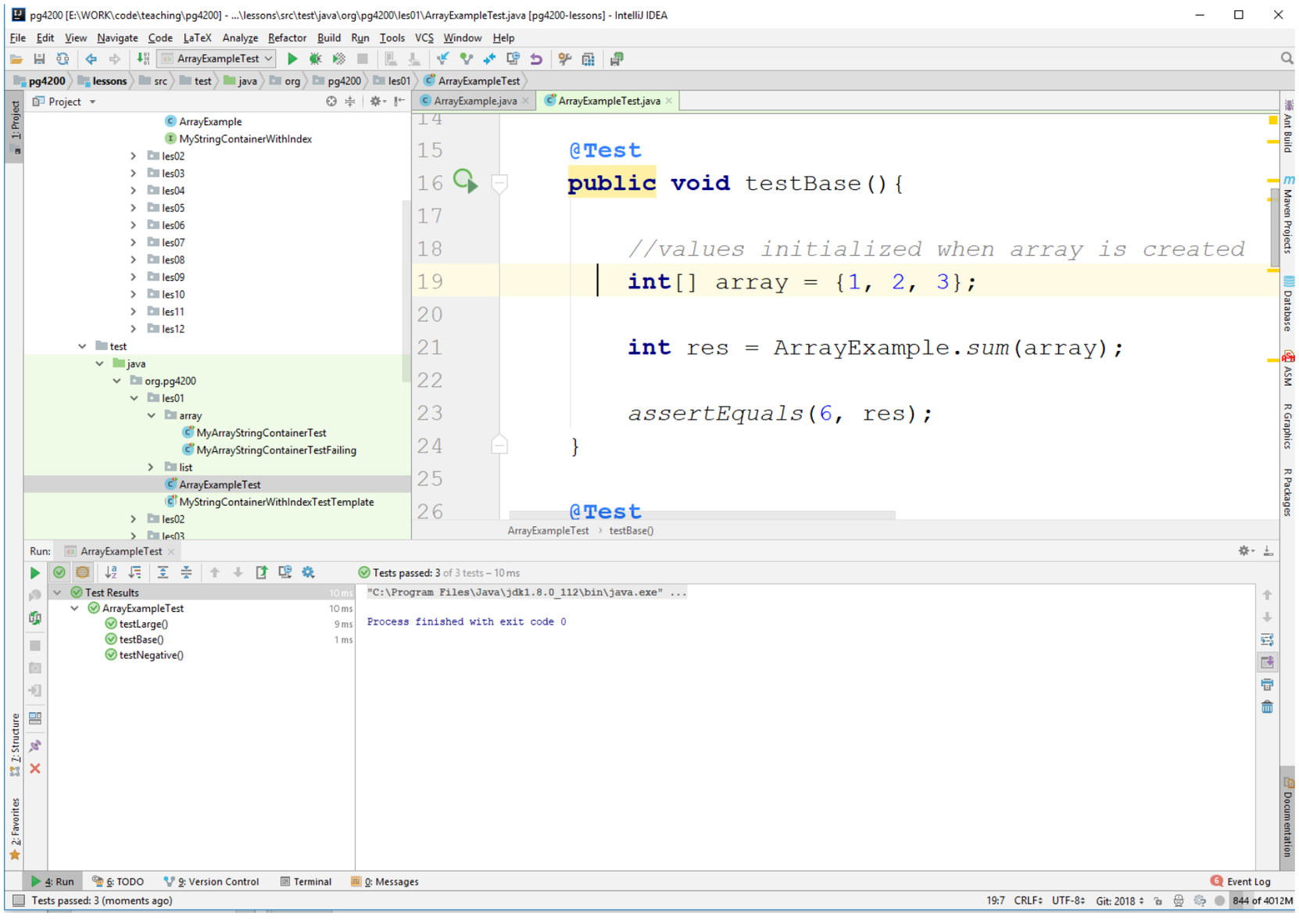
Input data

Code execution

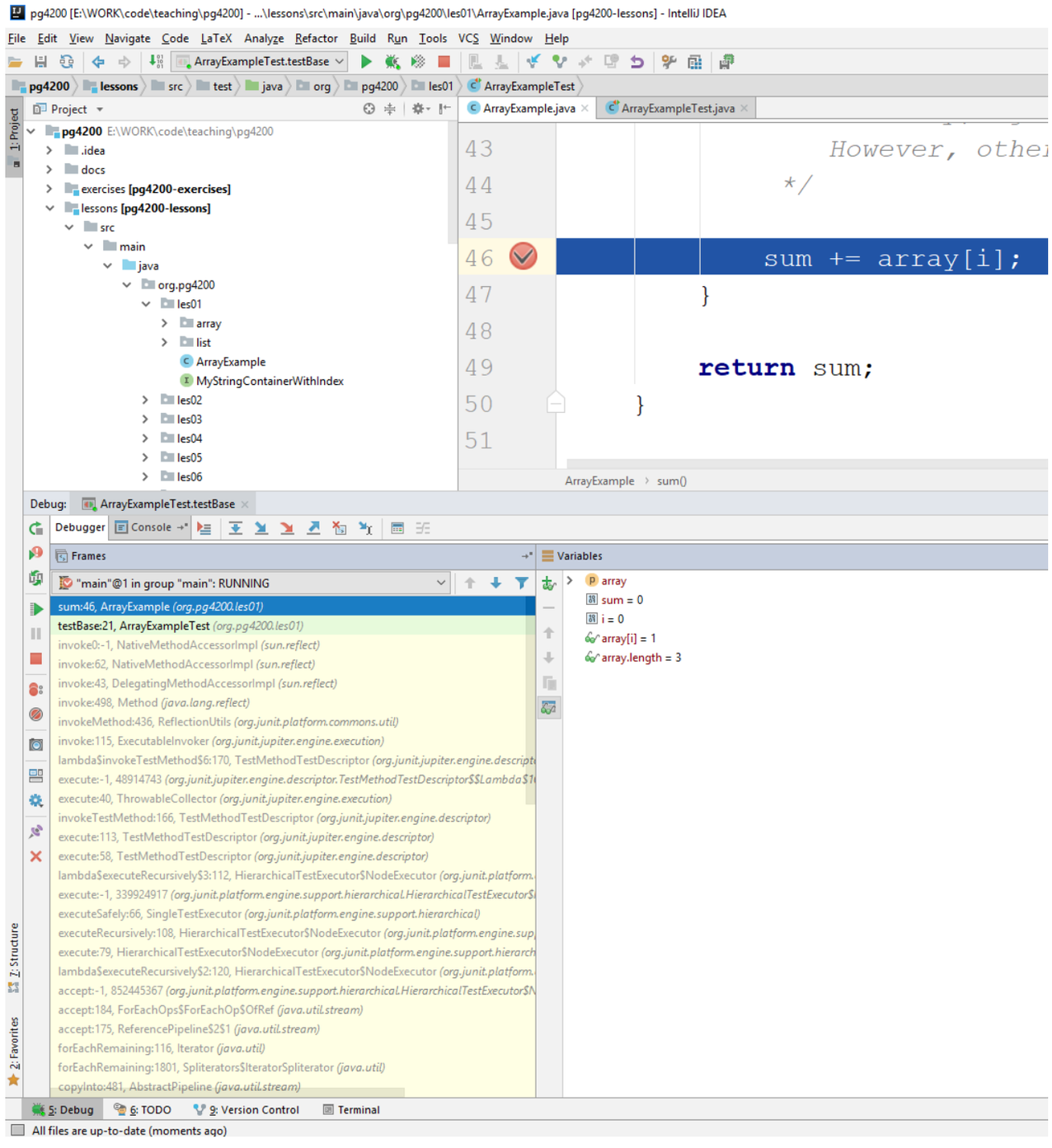
Verify output

# Running a Test

- Right-click, and choose "*Run <ClassName>*"
- Can also use "*Debug*" and "*Run With Coverage*"

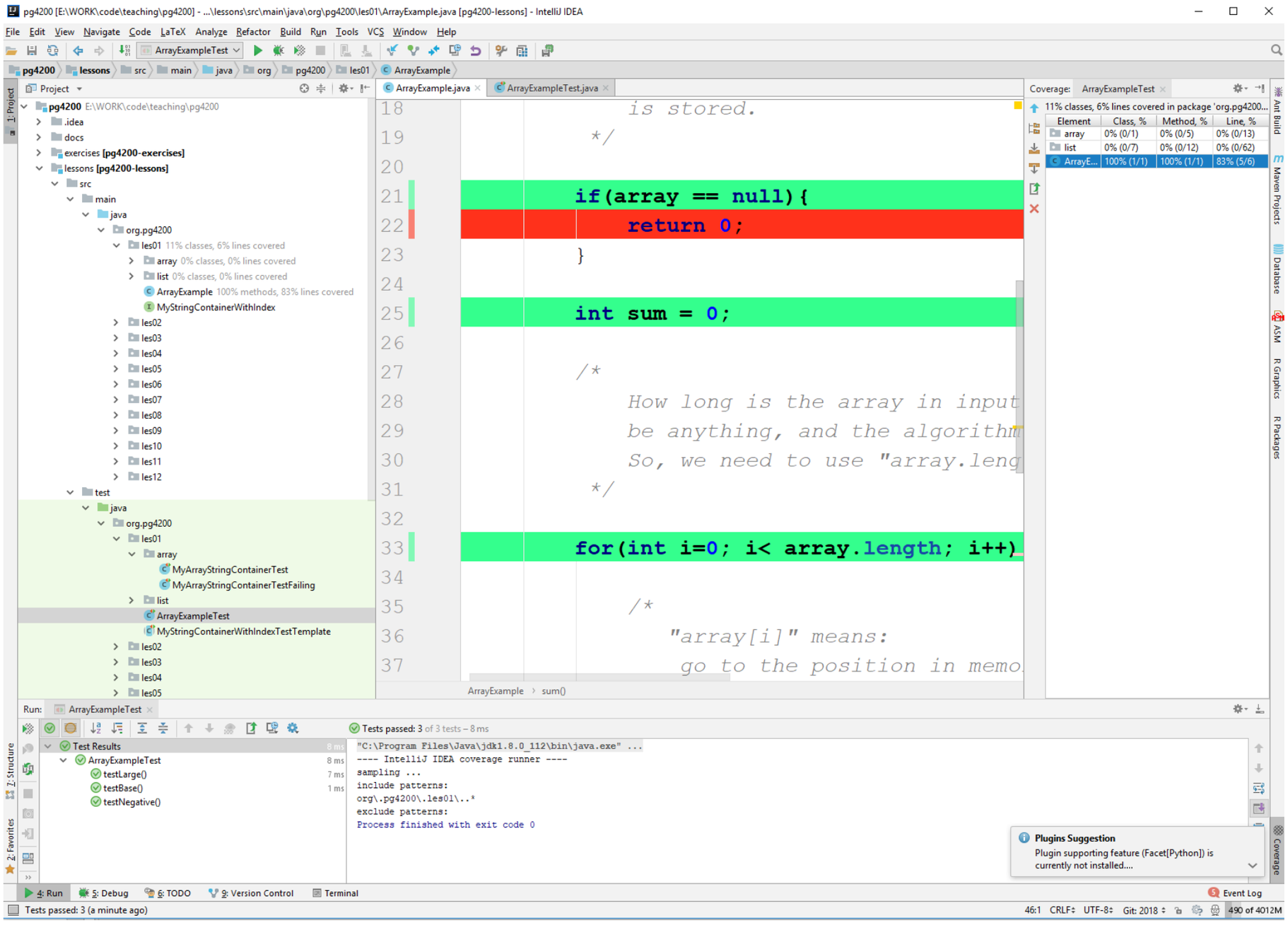


# Debugging




- **VERY IMPORTANT**
- Can put “*break points*”
- Execute one step at a time
- Inspect status of all variables, at each step
- Easier to understand with live demo

# Run With Coverage




- Can tell you how much of the code is executed
- Eg, 83% in this case
- Code that is never executed by a test, might have bugs

# Homework

- Study Book Chapter 1.1 and 1.2
- Study code in the *org.pg4200.les01* package
- Do exercises in *exercises/ex01*
- Extra: do exercises in the book