Unit Testing

BootcampOS

- What's unit testing?
  - Automated, isolated, detects problems early, force developers to think
- Why do unit testing?
  - Edit and pray vs. cover and modify, edge cases, what unit tests do for you
- Other types of testing
  - QA, manual vs. automated testing
  - Who does what
  -
- What makes a good unit test?
  - Traits
    - Fast - time in ms
    - Reliable, repeatable - should consistently pass or fail
    - Independent - runs independently of other tests
    - Obvious - it's clear why test failed
  - Test logic - happy paths and edge cases
  - Arrange, act, and assert
- Creating unit tests
  - Conventions
  - JUnit annotations
  - Running tests
  - Debugging tests
  - Test errors
  - Example test method

Agenda

- Socrative quiz
- Review
  - Inheritance and polymorphism
- Lecture
  - Quick review: Inheritance and polymorphism
  - Software Development Lifecycle (SDLC)
  - Unit testing
- Lecture code

Review

Inheritance & Polymorphism

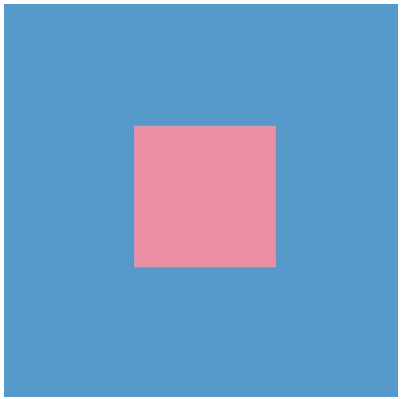
Inheritance

- When one class (a subclass) inherits properties and behavior from another class (its superclass) (aka. “child” and “parent” classes)
- Described as an *IS-A* relationship (e.g., a Car *IS-A* Vehicle)
- Achieved with the `extends` keyword
- A class can only extend one class at a time (but it can implement multiple interfaces)

Polymorphism

- Refers to code that can take on “many forms”
- Allows an object to be generically treated as its super class and still get a specific response from the subclass (e.g., `List` → `ArrayList` or `List` → `LinkedList`)
- Good examples include method overloading and overriding:

- Over**loading** - having multiple implementations of the same method within the same class
- Over**riding** - when a subclass has its own implementation of a method defined in its superclass



Lecture

The Software Development Life Cycle & Testing

Objectives

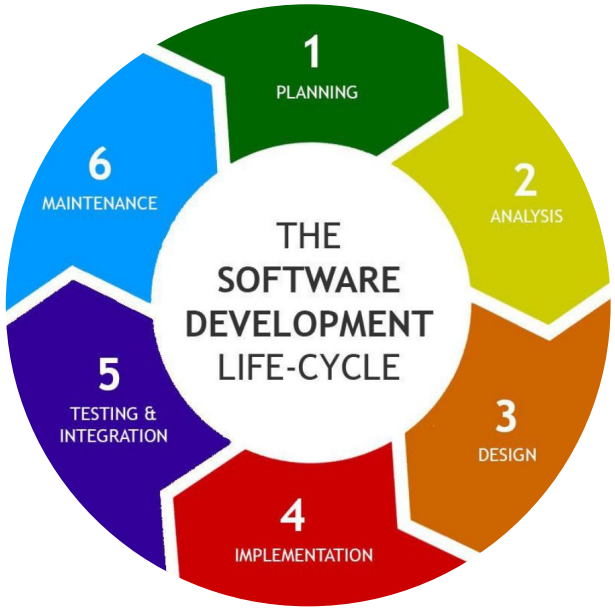
By the end of today's session, you should . . .

- Be familiar with some of the key aspects of the **SDLC**
- Know some of the differences between **Waterfall** and **Agile**
- Know the difference between **exploratory** and **regression testing**
- Know the difference between **unit**, **integration**, and **acceptance testing**
- Be able to **create and run unit tests** with the **JUnit 4** testing framework

The Software Development Life Cycle (SDLC)

The Software Development Life Cycle

- A structured process for creating and maintaining software
- There are several industry standards for the SDLC, but organizations often have their own flavors



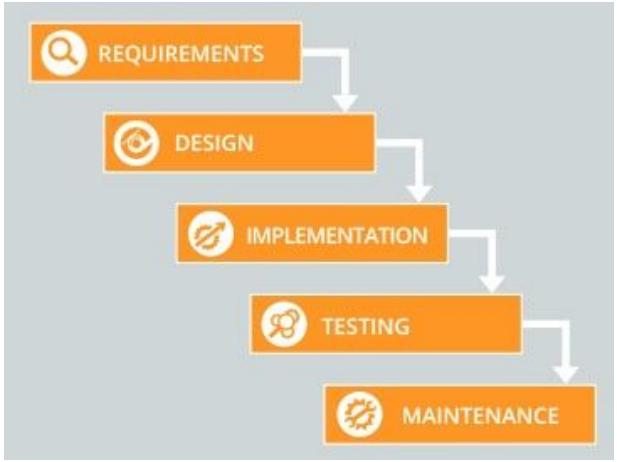
Agile

VS.

Waterfall

Waterfall vs. Agile workflow

Waterfall

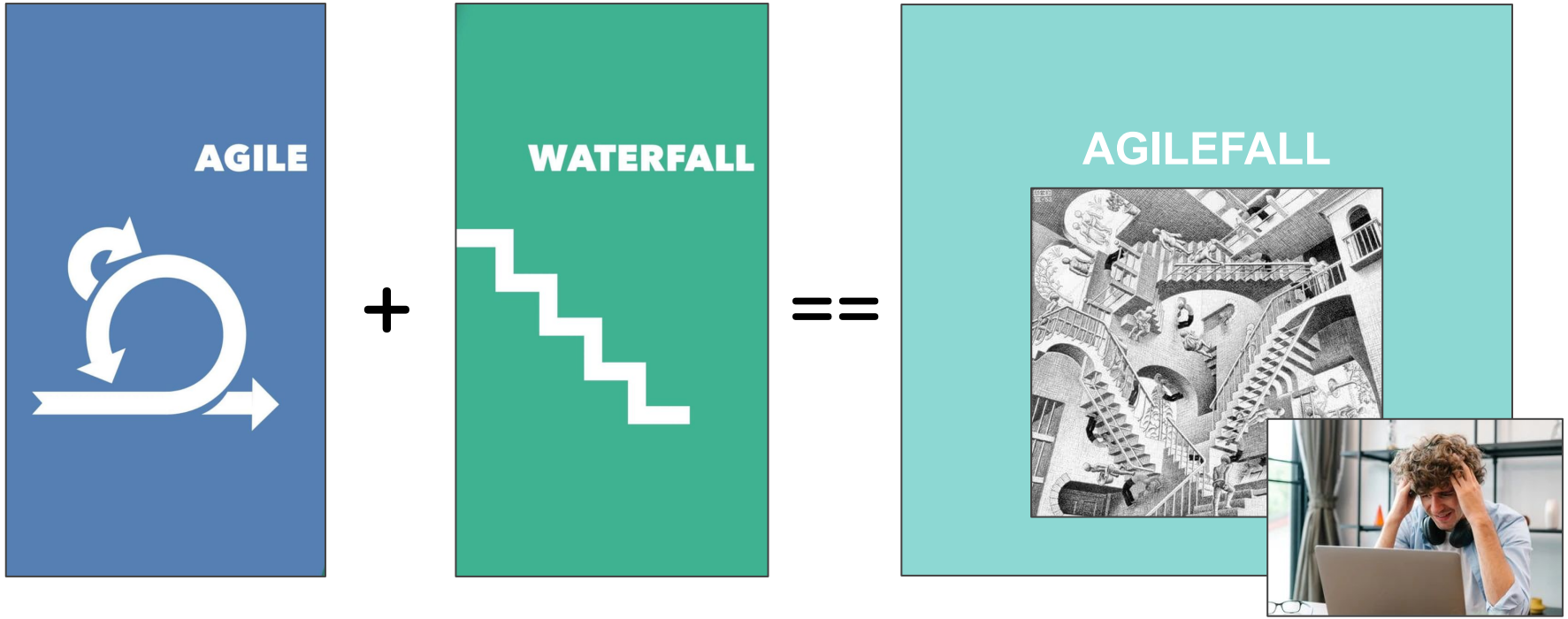


Agile

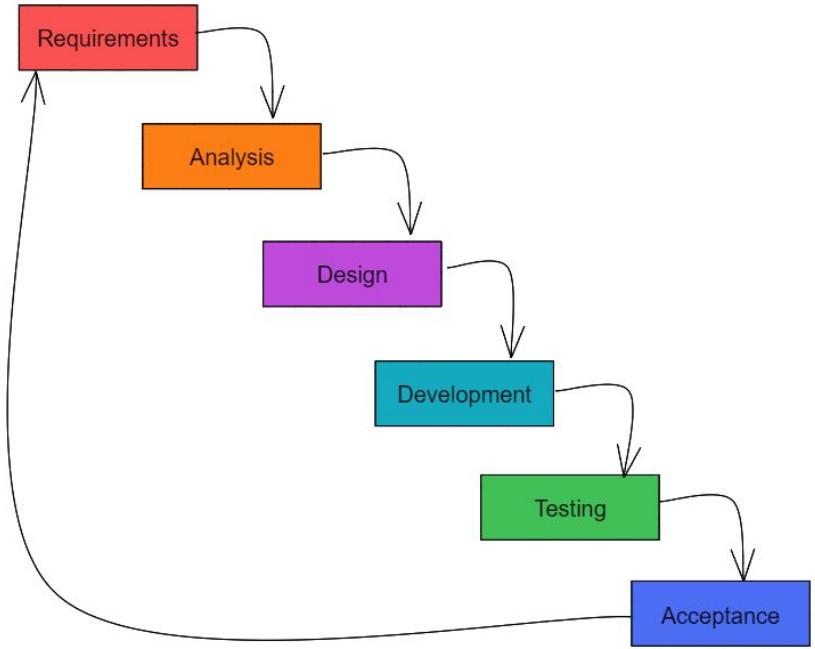
Waterfall vs. Agile comparison

Waterfall	Agile
• Linear, sequential process • Highly structured • Scope is set at the beginning of the project • Testing is performed after the Build phase	• Iterative process • Highly flexible • Requirements are allowed to change during development • Testing is performed concurrently with development

Agilefall: Developer purgatory



Waterfall methodology



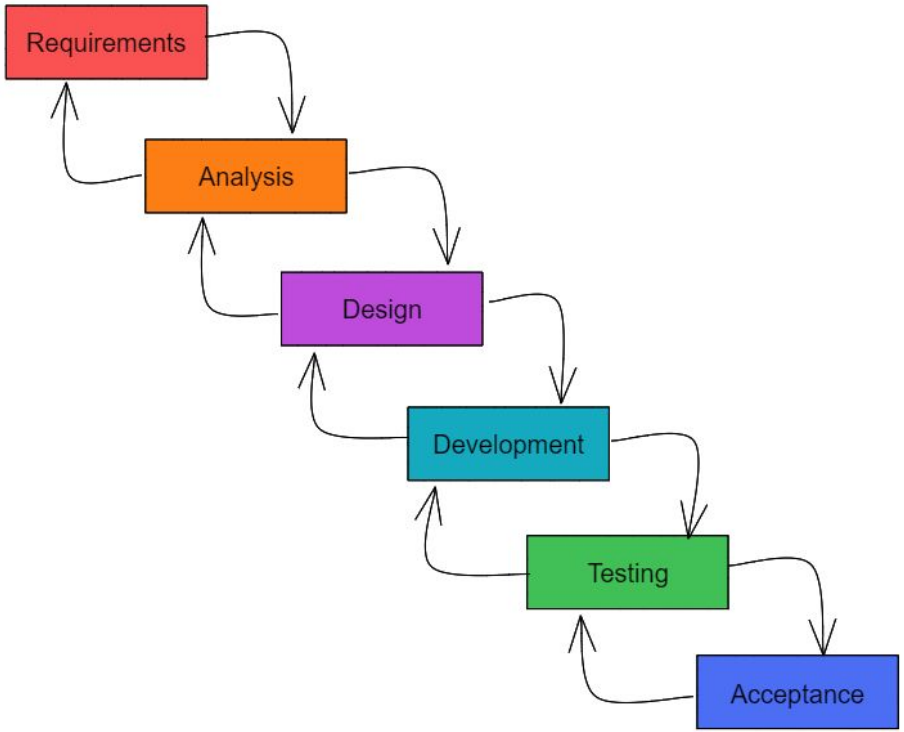
Advantages

- Easy to Manage
- Process well documented
- Easily adaptable
- Allows shifting teams

Disadvantages

- Slow
- Long lead time
- Rigid

Agile methodology

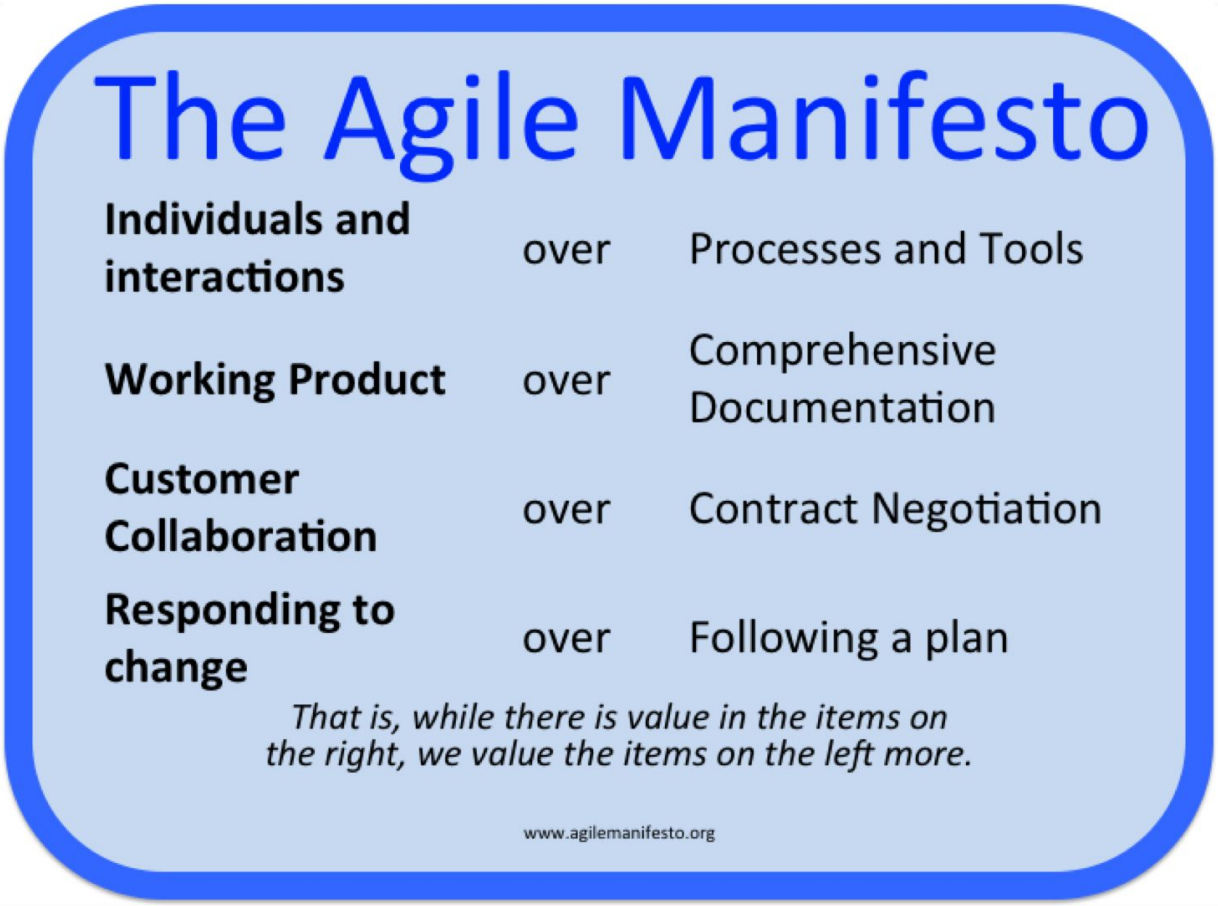


Advantages

- Continuous Delivery
- Flexible and fluid
- Short lead time (product released in parts)
- Specialized, stable teams

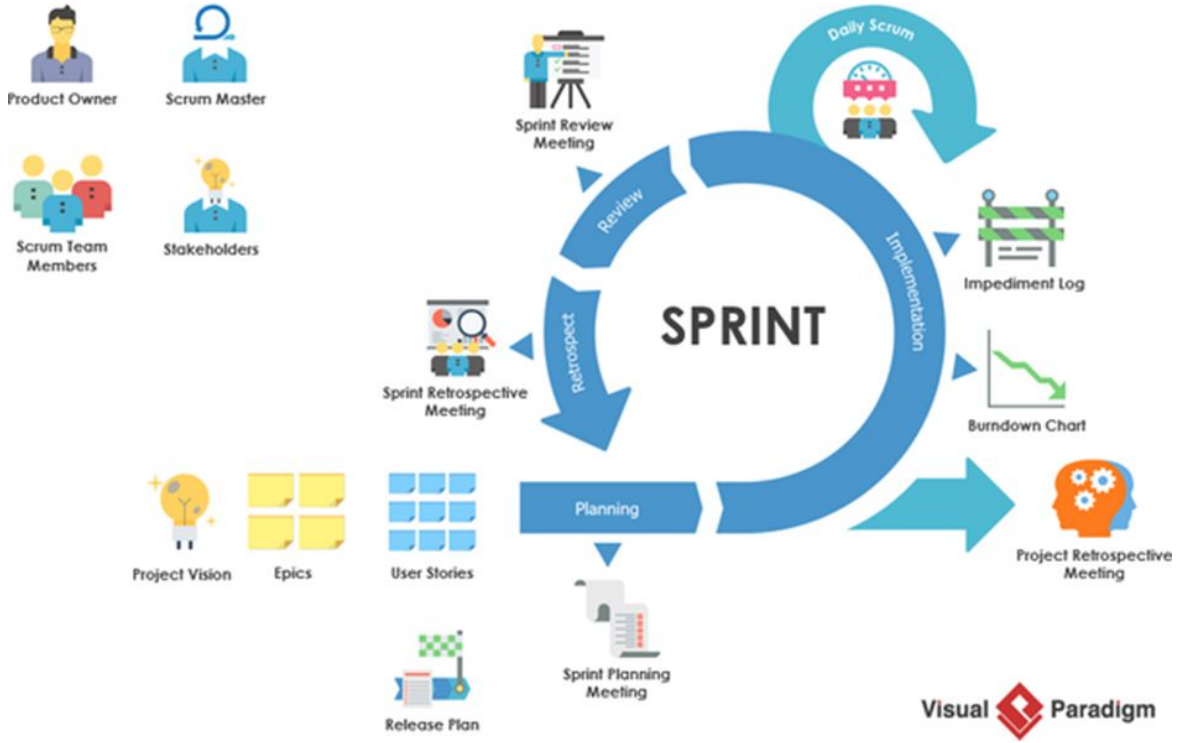
Disadvantages

- Requires a set team of 4-12 specialists
- Harder to manage
- Not suitable for all projects



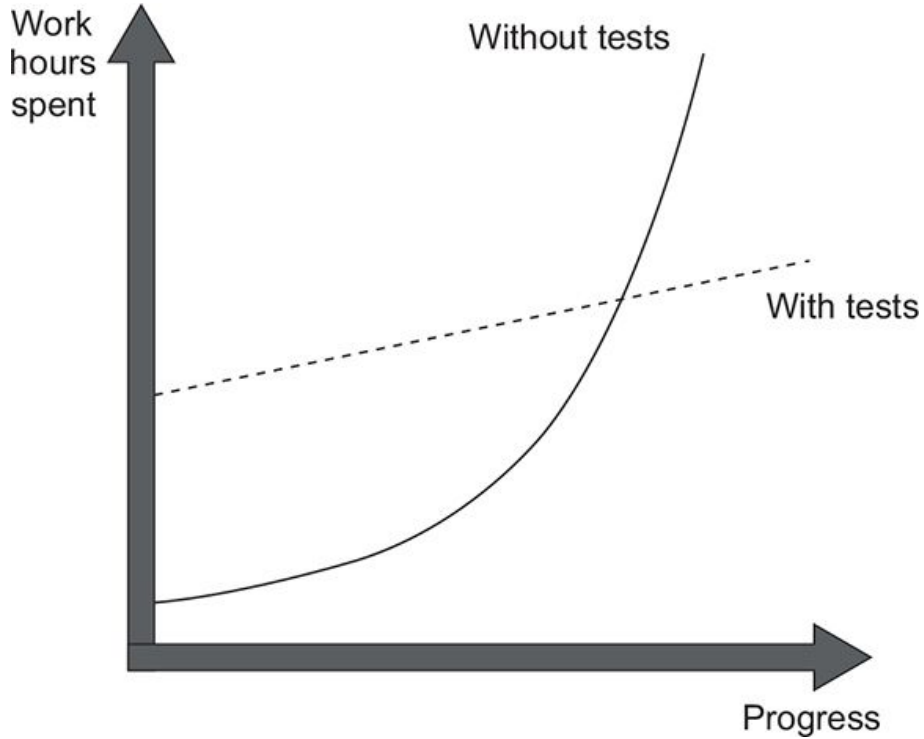
Original Agile Manifesto

The Agile-Scrum Framework



Testing Overview

Testing is an investment



Manual vs. Automated Testing

Manual	Automated
Creativity	Speed / efficiency
Subjectivity	Lower cost per execution
	Repeatability / dependability
	Accuracy

Exploratory vs. Regression Testing

- 👩‍🚀**Exploratory Testing** explores the new functionality of the system looking for defects, missing features, or other opportunities for improvement. Almost always manual.
- ⬅️**Regression Testing** validates that existing functionality continues to operate as expected. Usually automated.

Types of testing

- 🔍**Unit Testing** tests the smallest units possible (i.e., individual methods).
- 🤝**Integration Testing** tests how various units or parts of the program interact with each other.
  - Can also be used to validate some external dependencies, like database systems or API’s
- ✅**User Acceptance Testing (UAT)** tests the functionality from the end user’s perspective to ensure that requirements are satisfied. Can be automated or manual.

Some other types of testing

- 🛡️**Security testing:** Is our data safe?
- 🏋️**Performance testing:** It works with 1 user, but what about a million?
- 📲**Platform testing:** It works great on my laptop, but what if I pull it up on my phone?
- 👁️ **Accessibility testing:** Can all of our customers use and enjoy it?

What should my unit tests test for?

- 🙂 The “happy path,” which is the ideal scenario in which your code runs as expected and no errors occur
- 😱 Exceptions (which we’ll talk about next week) and errors that you anticipate may occur during execution
- 🚩Edge-cases, which occur at the extreme ends (boundaries) of your allowable data range

What shouldn’t my unit tests test for?

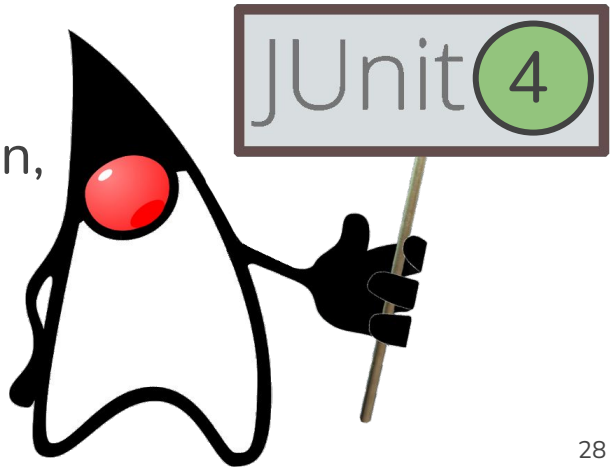
You don’t need to write unit tests for super basic methods where the outcome is highly predictable and the likelihood of an exception or error is low. For example, you don’t need to write tests for all of your **getters and setters**.

- **Setters should only be tested if they contain important or complex logic**, such as validating the data being passed into them meets certain requirements.

Unit Testing with JUnit 4

Overview

- JUnit (`org.junit`) is the most popular testing framework for Java
  - We will be using JUnit 4
- All related tests can be written in a single test class containing several methods
- Each test method should contain an assertion, which compares the results of your code against an expected value



Rules and characteristics

Characteristics:

1. Fast
2. Repeatable
3. Independent
4. Obvious

Rules:

1. No external dependencies
2. One logical assertion per test
3. Test code should be the same quality as product code
4. Test early, test often (don’t save them for the end!)

The 3 A's of unit testing

1 2 3

Arrange	Act	Assert
Setup the conditions for the test	Perform the action being tested	Validate that the expected outcome occurred

Given... **When...** **Then...**

JUnit annotations

- Annotations are metadata added to Java code to communicate the **purpose of the code** or **how/when to use it** to the JVM or a framework (like JUnit).
- Annotations have to be imported.

JUnit test life cycle

The life cycle of JUnit tests is controlled by annotations on public methods in the class.

@Before	@Test	@After
Runs before each test to perform setup	Runs the method as a test case	Runs after each test to perform cleanup

```
@Before
public void setup() { }
```

```
@Test
public void test_something() { }
```

```
@After
public void cleanup() { }
```

JUnit test lifecycle

Typical file order

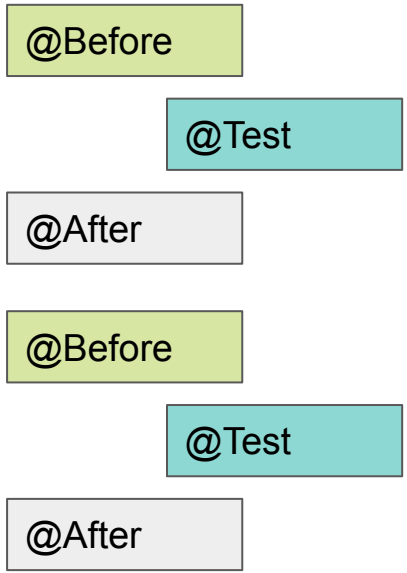
```
@Before
public void setup() { }
```

```
@After
public void cleanup() { }
```

```
@Test
public void test_1() { }
```

```
@Test
public void test_2() { }
```

Method run order



Assertions

The `Assert` class (`org.junit.Assert`) allows you to verify test results.

Some examples:

```
Assert.assertTrue( optionalMessage, booleanCondition )

Assert.assertFalse( optionalMessage, booleanCondition )

Assert.assertEquals( optionalMessage, expectedValue, actualValue )

Assert.assertEquals( optionalMessage, expectedDouble, actualDouble, precision )

Assert.fail()
```

Another resource: https://www.baeldung.com/junit-assertions

Asserting doubles

```
Assert.assertEquals(expected, actual, delta);
```

The delta is the absolute difference between the expected result and the actual result.

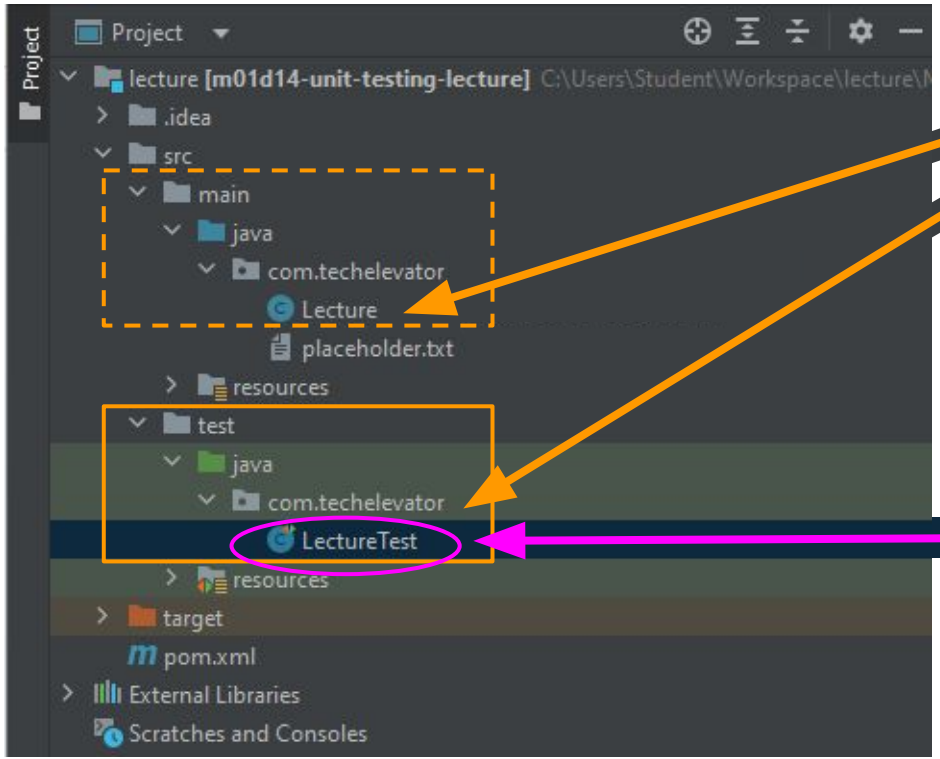
If the difference is greater than the delta, the numbers are seen as not equal in value.

If the result is less than or equal to the delta, the numbers are treated as equal.

`assertEquals` evaluates the delta using the following formula:

```
if (Math.abs(expected - actual) > delta) {
      Assert.fail();
}
```

Package and file naming conventions



The test class (LectureTest.java) is under the same package structure in the test package as the class it's testing (Lecture.java) is in the main package :

src\main\java\com.techelevator\Lecture.java
src\test\java\com.techelevator\LectureTest.java

The test class has the same name as the class it's testing + Test :

LectureTest.java tests **Lecture.java**

Test method naming conventions

- Test names should be very obvious (often quite wordy)
  - Often includes the name of the method being tested
  - Often includes input and expected output

- Using underscored_names vs. camelCasedNames

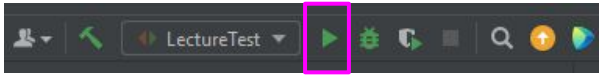
```
72      @Test
73      public void length_returns_the_number_of_characters_in_a_String() {...}
```

JUnit in IntelliJ

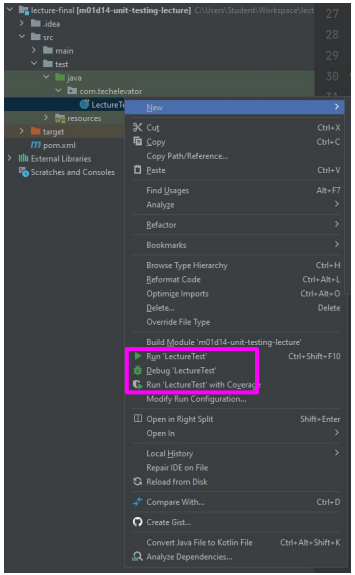
Running tests

Right-click line number next to class declaration

Use the run toolbar in the top-right corner



Right-click the file in the Project pane



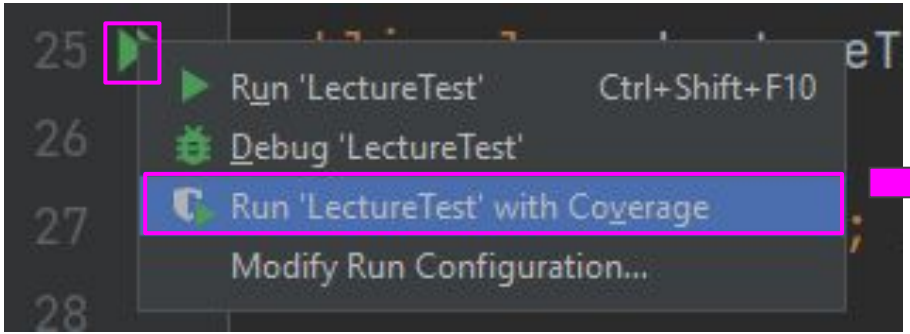
You can also right-click the line number next to (or the code inside of) an individual test to run it specifically.

```
72      @Test
73      public void length_returns_the_number_of_characters_in_a_String() {...}
```

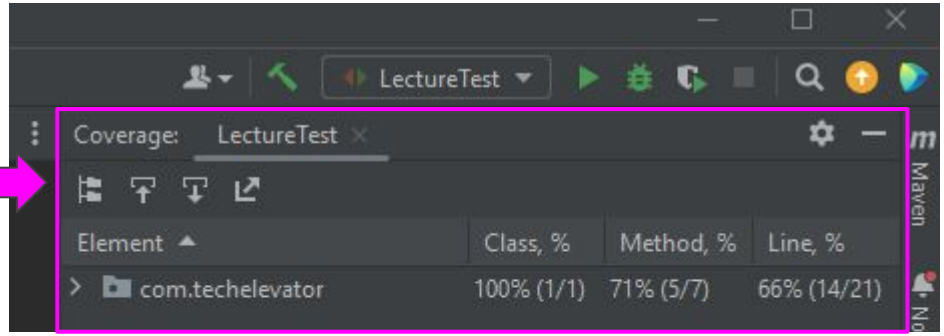
Code coverage

- Code coverage represents the percentage of your code that you've tested.
- **It can't tell you that you've tested "enough!"**

How to run with coverage

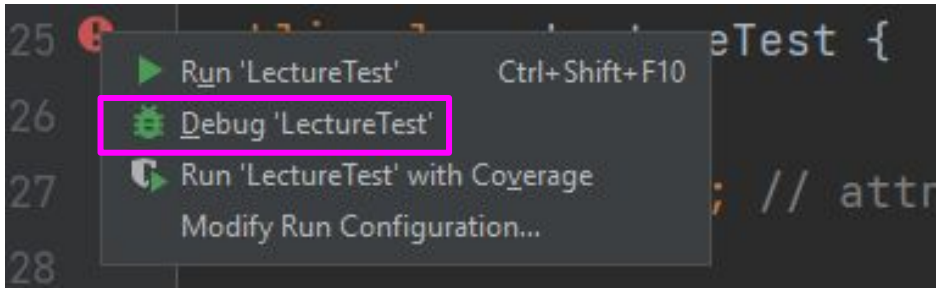


Code coverage pane



Debugging tests

- You can use the debugger on test files just like on any other Java file.
- If you missed Andy's bonus lecture on the debugger, you can find it here: [Bonus lecture: Using the debugger | Andy Chong Sam]
- Keep in mind: Sometimes the problem is in the application code, and sometimes it's in the test itself.



Let’s checkout some JUnit code!

Please do a
`git pull upstream main`
before we begin.