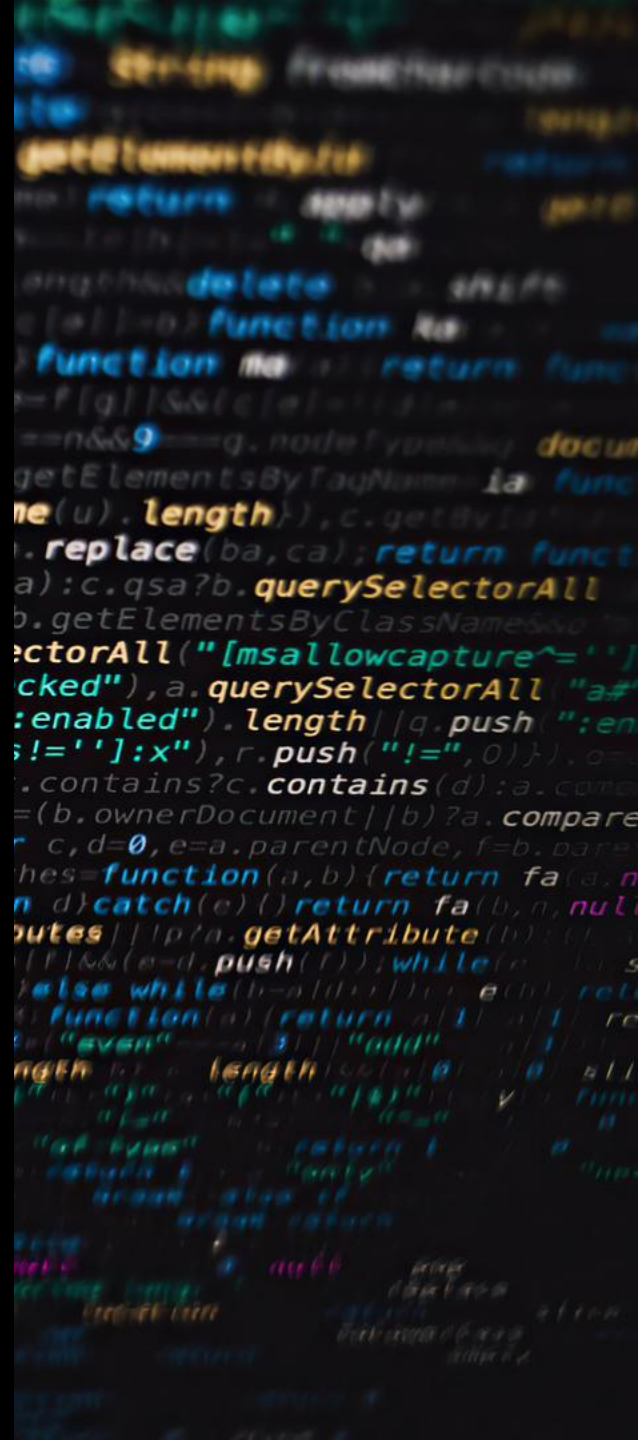
Unit Test in Java



Advanced Programming

Dr. Mojtaba Vahidi Asl

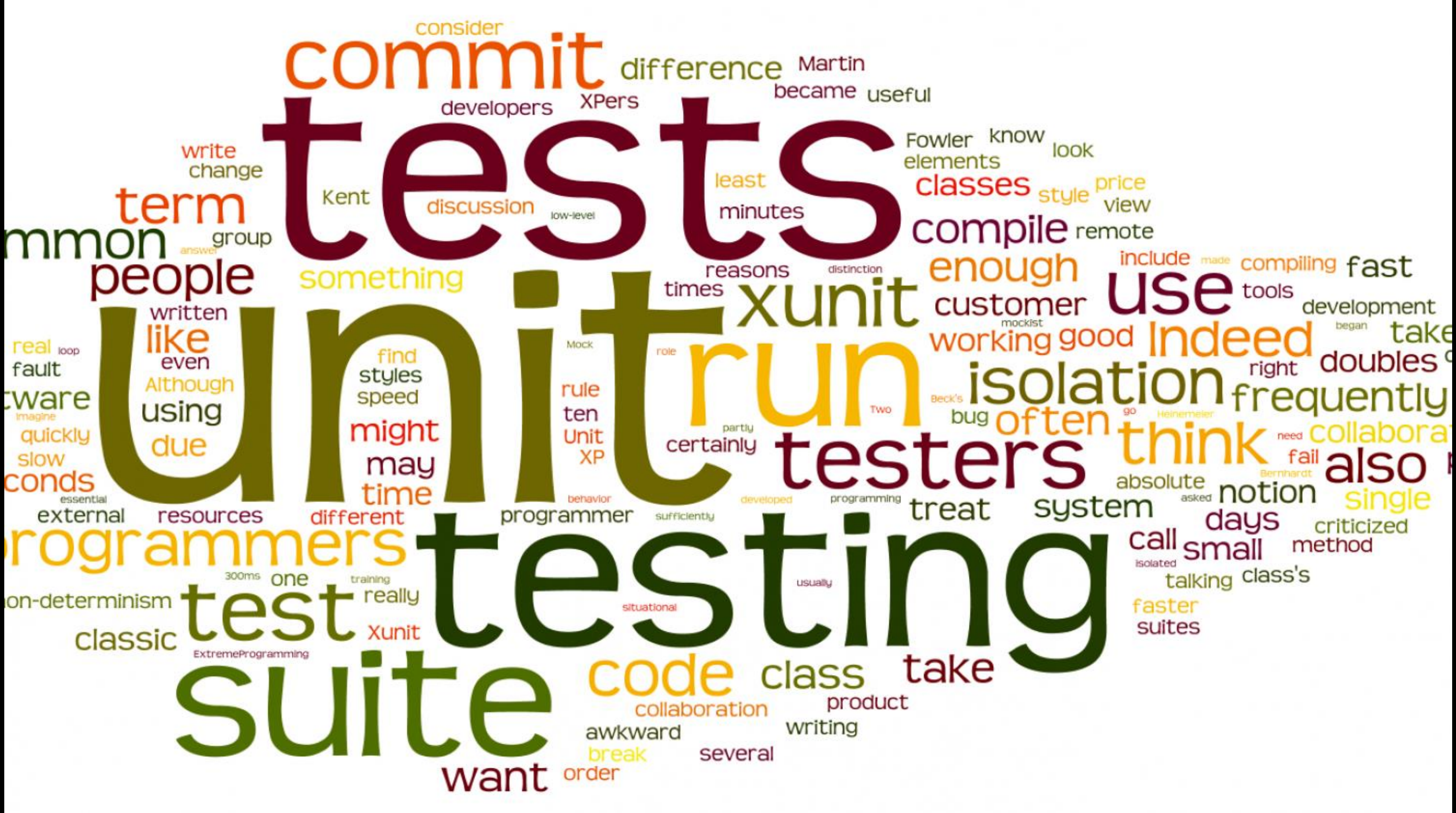
Sara Shiri , Fall 1403

What is Unit Testing?

- Unit tests are automated tests that verify the smallest parts (units) of code.
- Helps to ensure that individual functions work as expected.
- Unit tests are typically written by developers during development.

History of Unit Testing?

- Started with the concept of 'test-first' development in the 1990s.
- Part of Extreme Programming (XP) methodologies.
- JUnit, one of the most popular Java testing frameworks, was introduced by Kent Beck and Erich Gamma.


unit
tests
testing
suite
commit
run
xunit
testers
test
code
isolation
programmers
people
term
use
Indeed
think
class
take
want
compile
customer
enough
frequently
also
classes
difference
consider
Martin
became
useful
developers
XPers
write
change
Kent
discussion
Fowler
know
look
elements
least
minutes
price
style
view
remote
group
something
reasons
times
include
compiling
fast
tools
development
working
good
take
doubles
right
like
written
even
fault
real
find
styles
speed
using
Although
quickly
slow
due
might
may
time
rule
ten
Unit
XP
certainly
often
bug
collaborat
absolute
notion
single
fail
system
treat
days
small
call
criticized
method
talking
class's
faster
suites
external
resources
different
programmer
one
really
Xunit
classic
collaboration
product
awkward
writing
break
several
order

How Unit Testing Works?

- **Isolate the unit:** The smallest functional part of the code is isolated.
- **Prepare input:** Set up the necessary input for testing the method or function.
- **Call the method:** The method or function is called with the test inputs.
- **Check output:** Compare the actual output to the expected result to validate the unit’s behavior.
- **Test results:** A passed test means the unit works as expected; a failed test indicates a bug that needs fixing.

How Unit Testing Works?

- **Test Specific Code Parts:**
  Write tests for individual methods or functions.
  Each test checks if the method works correctly with specific inputs.

- **Compare Expected and Actual Output:**
  Run the method and compare the actual result with what you expect.
  If they match, the test passes.

- **Use Tools like JUnit:**
  Use testing frameworks (e.g., JUnit) to automate the test execution process
  Provides quick feedback on which methods are correct and which need fixes.

Unit Testing

Advantages:

- **Fast feedback:** Unit tests can be executed quickly, giving developers immediate feedback on whether their code is working.
- **Cost-effective:** Catching bugs early is far cheaper than fixing them later in the development cycle.
- **Documentation:** Unit tests serve as a form of documentation, showcasing how the code is supposed to behave.
- **Ease of debugging:** If a unit test fails, it often directly points to the broken code, making debugging easier.

Disadvantages:

- **Time-consuming:** Writing unit tests takes time, especially for complex systems.
- **Maintenance overhead:** As the codebase grows, maintaining a large number of tests can become burdensome.
- **Doesn’t catch all errors:** Unit tests are limited in scope and can't catch system-wide issues or performance bottlenecks.
- **Challenging for legacy code:** Adding unit tests to older, monolithic code can be difficult without heavy refactoring.

How to Use Unit Testing in Java?

```java
public class Calculator {
    public int add(int a, int b) {  no usages
        return a + b;
    }

    public int subtract(int a, int b) {  no usages
        return a - b;
    }

    public int multiply(int a, int b) {  no usages
        return a * b;
    }

    public int divide(int a, int b) {  no usages
        if (b == 0) throw new ArithmeticException("Division by zero");
        return a / b;
    }
}
```

How to Use Unit Testing in Java?

- **Frameworks:**
  using frameworks like JUnit and TestNG.

- **Setup:**
  Import JUnit (import org.junit.*).
  Define test methods using the @Test annotation.

- **Arrange-Act-Assert (AAA) Pattern:**
  Arrange: Set up the test inputs and conditions.
  Act: Call the method or function to test.
  Assert: Verify the output using assertion methods like assertEquals(expected, actual).

How to Use Unit Testing in Java?

1. Set Up JUnit Framework

2. Write Test Cases

3. Assert Results

4. Run Tests

5. Analyze Test Results

6. Handle Exceptions in Tests

How to Use Unit Testing in Java?

1. Set Up JUnit Framework:

JUnit is a popular testing framework for Java.
Make sure you have JUnit added to your project as a dependency.

2. Write Test Cases:

Use the @Test annotation to define a test method.
Test a single method or functionality per test case.

```
@Test
void testAddition() {
    Calculator calc = new Calculator();
    assertEquals( expected: 5, calc.add( a: 2,  b: 3));
}
```

How to Use Unit Testing in Java?

3. Assert Results:

Use assertions like assertEquals() to compare the expected result with the actual result. If the result matches, the test passes. if not, it fails.

4. Run Tests:

You can run tests directly from the IDE (e.g., IntelliJ or Eclipse) or through command-line tools like Maven or Gradle.

How to Use Unit Testing in Java?

5. Analyze Test Results:

A green bar means all tests passed.
A red bar indicates failed tests, signaling issues that need to be fixed.

6. Handle Exceptions in Tests:

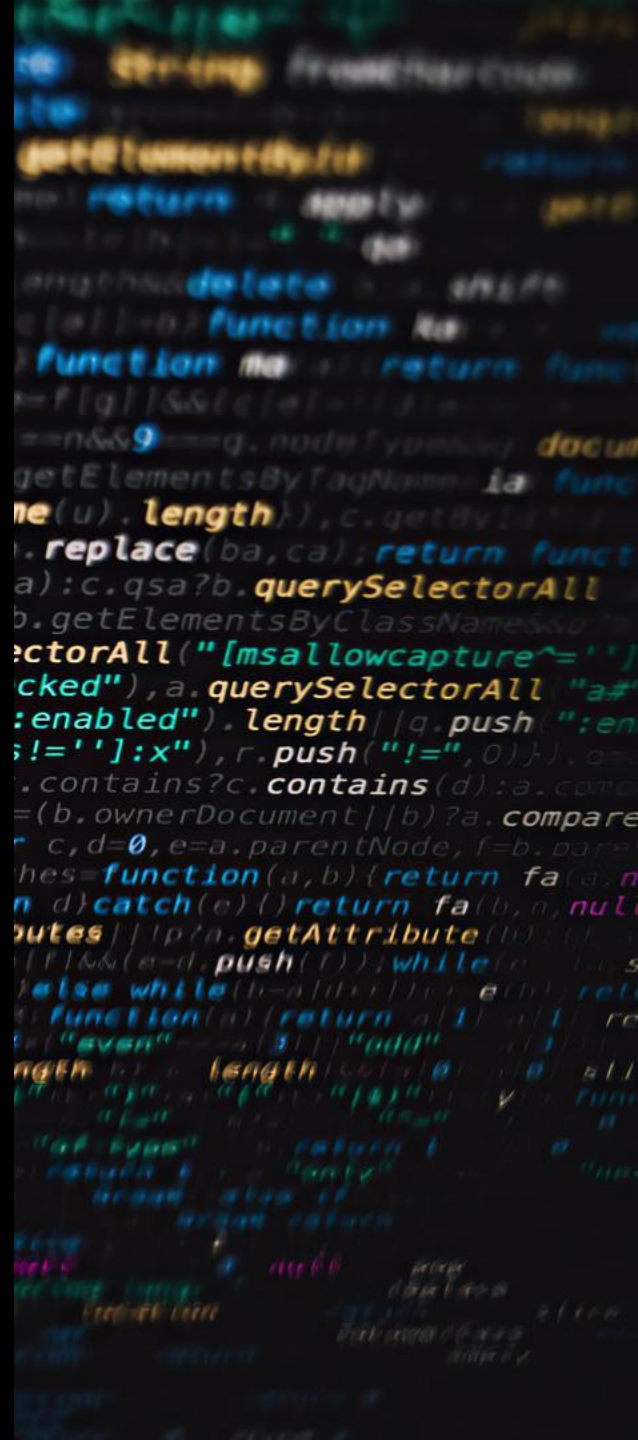
Use assertThrows() to test methods that are expected to throw exceptions.

```
    @Test
    void testDivisionByZero() {
        Calculator calc = new Calculator();
        assertThrows(ArithmeticException.class, () -> calc.divide( a: 10,  b: 0));
    }
}
```

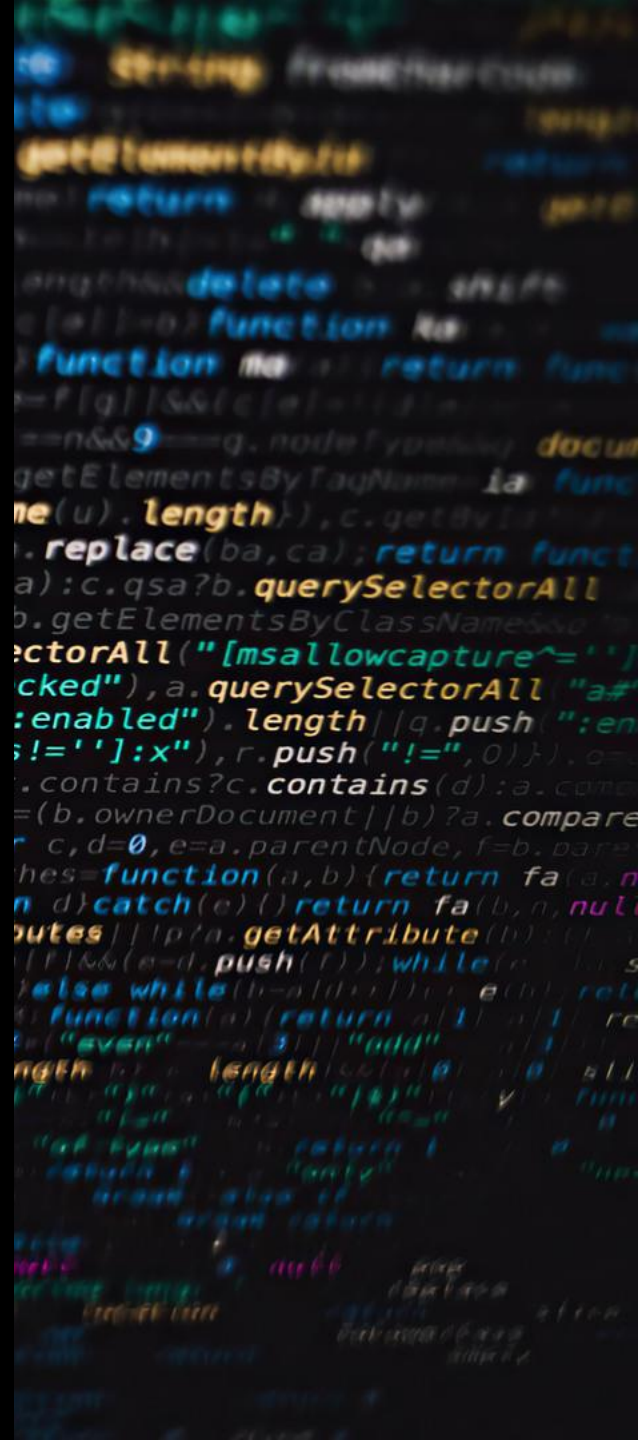
```java
public class CalculatorTest {
    @Test
    void testAddition() {
        Calculator calc = new Calculator();
        assertEquals( expected: 5, calc.add( a: 2,  b: 3));
    }
    @Test
    void testSubtraction() {
        Calculator calc = new Calculator();
        assertEquals( expected: 3, calc.subtract( a: 5,  b: 2));
    }
    @Test
    void testMultiplication() {
        Calculator calc = new Calculator();
        assertEquals( expected: 15, calc.multiply( a: 3,  b: 5));
    }
    @Test
    void testDivision() {
        Calculator calc = new Calculator();
        assertEquals( expected: 2, calc.divide( a: 10,  b: 5));
    }
    @Test
    void testDivisionByZero() {
        Calculator calc = new Calculator();
        assertThrows(ArithmeticException.class, () -> calc.divide( a: 10,  b: 0));
    }
}
```

Resources:

- https://www.javatpoint.com/unit-testing
- https://www.freecodecamp.org/news/java-unit-testing/
- https://www.code-intelligence.com/blog/how-to-do-unit-testing-in-java
- https://www.browserstack.com/guide/unit-testing-java



Any questions?



Thanks for your Attention.