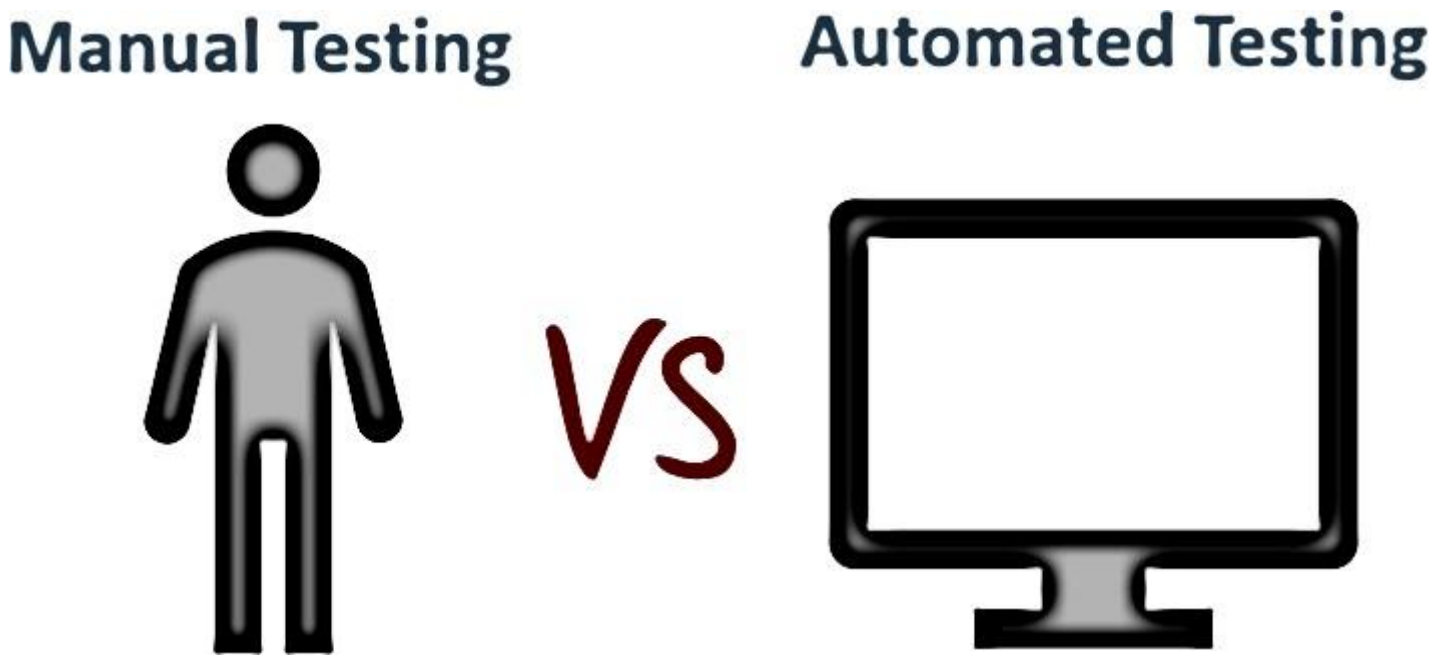
Advance Software Engineering
Unit Test

By:

Dr. Salwa Osama

Unit test with Junit



Manual Testing VS. Automation Testing

Manual Testing is done manually by QA analyst (Human) whereas Automation Testing is done with the use of script, code and automation tools (computer) by a tester.

Manual Testing process is not accurate because of the possibilities of human errors whereas the Automation process is reliable because it is code and script based.

Manual Testing is a time-consuming process whereas Automation Testing is very fast.

Manual Testing is possible without programming knowledge whereas Automation Testing is not possible without programming knowledge.

Manual Testing allows random Testing whereas Automation Testing doesn’t allow random Testing.

Unit Testing Vs. Functional Testing

Factors	Unit Testing	Functional Testing
Definition And Purpose	Testing individual modules	Testing the functionality as per user requirements
Written by	Developers	Tester
Testing Techniques	White-box testing	Black box testing
Errors	Code branches, Edge cases	Software / Application
No. Of Test Cases	Higher than another testing	Lower than unit and integration testing
Cost And Maintains	Low	High
Changes	Changes frequently	Low rates of changes

Bugs and testing

software reliability: Probability that a software system will not cause failure under specified conditions.

- Measured by uptime, MTTF (mean time till failure), crash data.

Bugs are inevitable in any complex software system.

- Industry estimates: 10-50 bugs per 1000 lines of code.
- A bug can be visible or can hide in your code until much later.

testing: A systematic attempt to reveal errors.

- Failed test: an error was demonstrated.
- Passed test: no error was found (for this particular situation).

Difficulties of testing

Perception by some developers and managers:

- Testing is seen as a junior job.
- Assigned to the least experienced team members.
- Done as an afterthought (if at all).
  - "My code is good; it won't have bugs. I don't need to test it."
  - "I'll just find the bugs by running the client program."

Limitations of what testing can show you:

- It is impossible to completely test a system.
- Testing does not always directly reveal the actual bugs in the code.
- Testing does not prove the absence of errors in software.

Unit testing

unit testing: Looking for errors in a subsystem in isolation.

- Generally a "subsystem" means a particular class or object.
- The Java library **JUnit** helps us to easily perform unit testing.

The basic idea:

- For a given class Foo, create another class FooTest to test it, containing various "test case" methods to run.
- Each method looks for particular results and passes / fails.

JUnit provides "**assert**" commands to help us write tests.

- The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail.

A JUnit test class

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

public class name {
    ...

    @Test
    public void name() {   // a test case method
        ...
    }
}
```

- A method with `@Test` is flagged as a JUnit test case.
  - All `@Test` methods run when JUnit runs your test class.

JUnit assertion methods

`assertTrue(`**`test`**`)`	fails if the boolean test is `false`
`assertFalse(`**`test`**`)`	fails if the boolean test is `true`
`assertEquals(`**`expected, actual`**`)`	fails if the values are not equal
`assertSame(`**`expected, actual`**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**`expected, actual`**`)`	fails if the values *are* the same (by ==)
`assertNull(`**`value`**`)`	fails if the given value is *not* `null`
`assertNotNull(`**`value`**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. `assertEquals(`**`"message", expected, actual`**`)`

ArrayIntList JUnit test

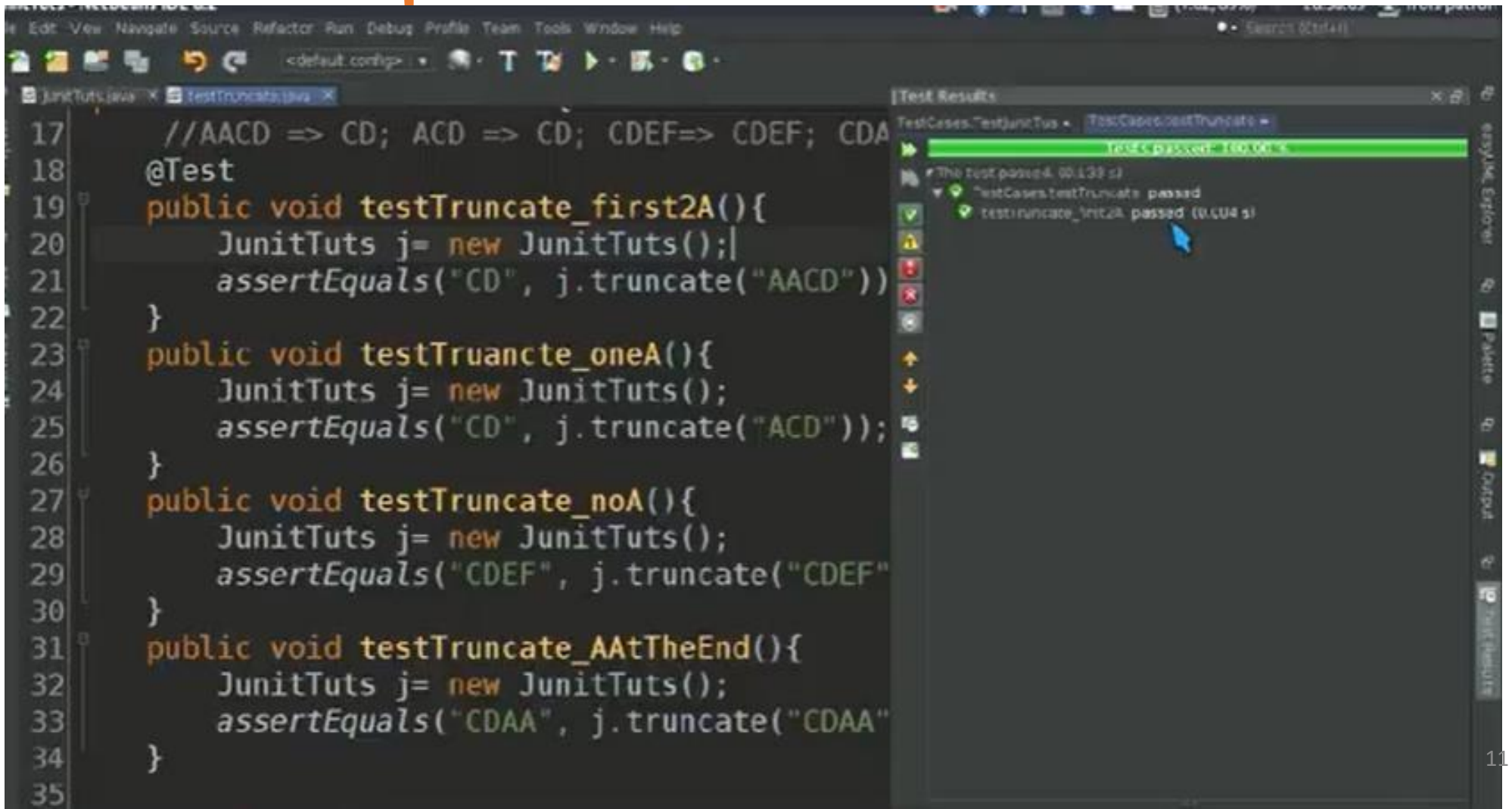
```
import org.junit.*;
import static org.junit.Assert.*;

public class TestArrayIntList {
    @Test
    public void testAddGet1() {
        ArrayIntList list = new ArrayIntList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }

    @Test
    public void testIsEmpty() {
        ArrayIntList list = new ArrayIntList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
        list.remove(0);
        assertTrue(list.isEmpty());
    }
```

Running a test

- Right click it in the netbeans **Run**
- The JUnit bar will show **green** if all tests pass, **red** if any fail.
- The Failure Trace shows which tests failed, if any, and why.



JUnit exercise

Given a `Date` class with the following methods:

```
• public Date(int year, int month, int day)
• public Date()                          // today
• public int getDay(), getMonth(), getYear()
• public void addDays(int days)    // advances by days
• public int daysInMonth()
• public String dayOfWeek()              // e.g. "Sunday"
• public boolean equals(Object o)
• public boolean isLeapYear()
• public void nextDay()                  // advances by 1 day
• public String toString()
```

- Come up with unit tests to check the following:
  - That no `Date` object can ever get into an invalid state.
  - That the `addDays` method works properly.
    - It should be efficient enough to add 1,000,000 days in a call.

What's wrong with this?

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 2);
        assertEquals(d.getDay(), 19);
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 3);
        assertEquals(d.getDay(), 1);
    }
}
```

Well-structured assertions

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(2050, d.getYear());   // expected
        assertEquals(2, d.getMonth());     // value should
        assertEquals(19, d.getDay());      // be at LEFT
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals("year after +14 days", 2050, d.getYear());
        assertEquals("month after +14 days", 3, d.getMonth());
        assertEquals("day after +14 days", 1, d.getDay());
    }   // test cases should usually have messages explaining
}       // what is being checked, for better failure output
```

Expected answer objects

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals(expected, d);   // use an expected answer
    }                                // object to minimize tests

                                     // (Date must have toString
    @Test                            //  and equals methods)
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }
}
```

Naming test cases

```
public class DateTest {
    @Test
    public void test_addDays_withinSameMonth_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, actual);
    }
    // give test case methods really long descriptive names

    @Test
    public void test_addDays_wrapToNextMonth_2() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
    // give descriptive names to expected/actual values
}
```

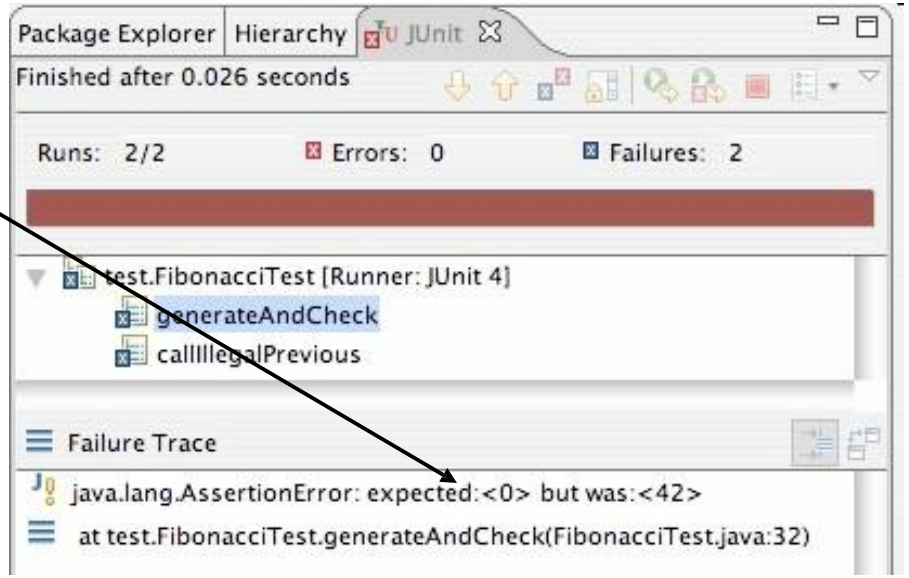
What's wrong with this?

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals(
            "should have gotten " + expected + "\n" +
            " but instead got " + actual\n",
            expected, actual);
    }
    ...
}
```

Good assertion messages

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals("adding one day to 2050/2/15",
            expected, actual);
    }
    ...
}
```

```
// JUnit will already show
// the expected and actual
// values in its output;
//
// don't need to repeat them
// in the assertion message
```



Tests with a timeout

```
@Test(timeout = 5000)
public void name() { ... }
```

- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
...

@Test(timeout = TIMEOUT)
public void name() { ... }
```

- Times out / fails after 2000 ms

Testing for exceptions

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

- Will pass if it *does* throw the given exception.
  - If the exception is *not* thrown, the test fails.
  - Use this to test for expected errors.

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayIntList list = new ArrayIntList();
    list.get(4);   // should fail
}
```

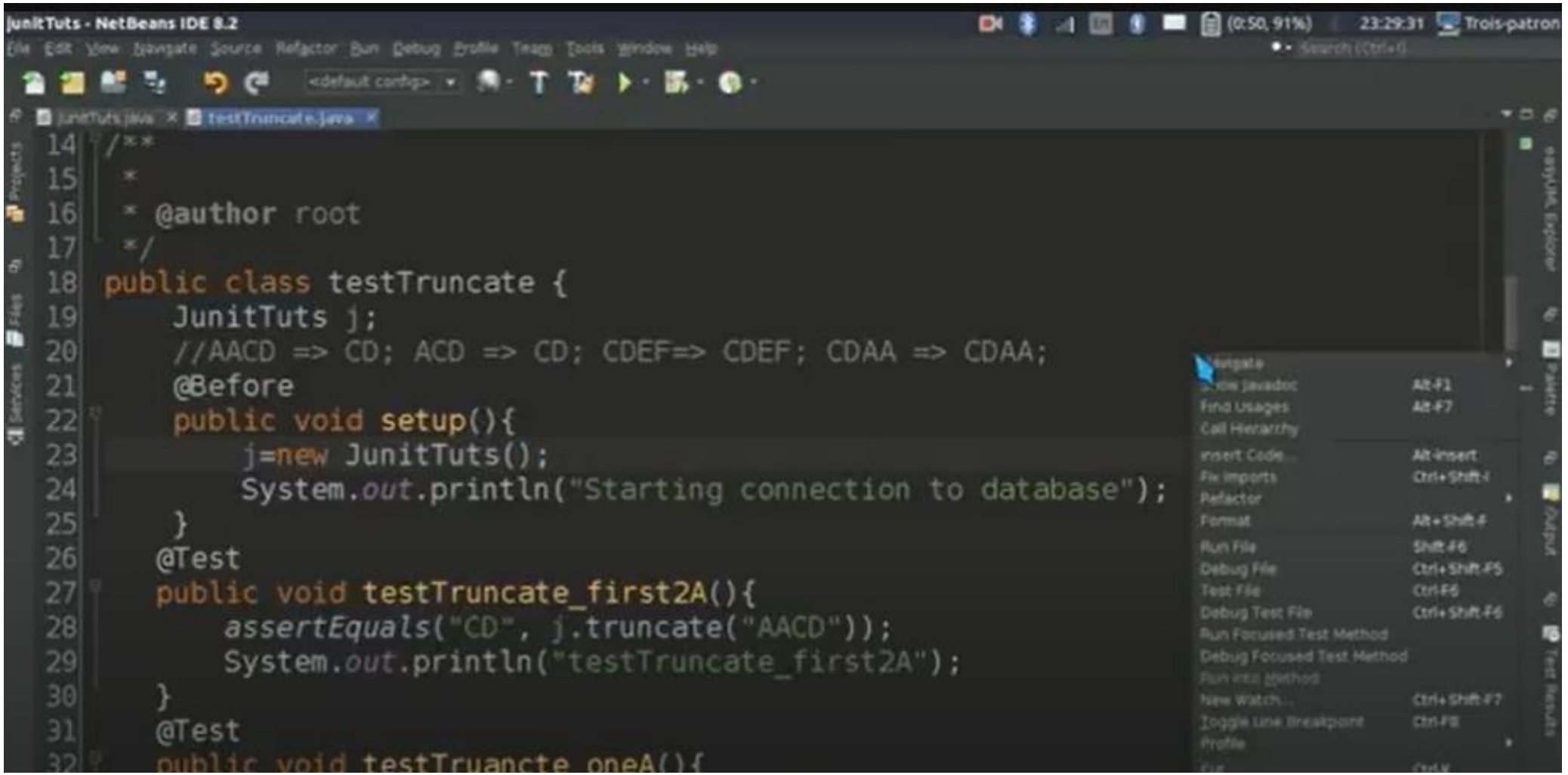
Setup and teardown

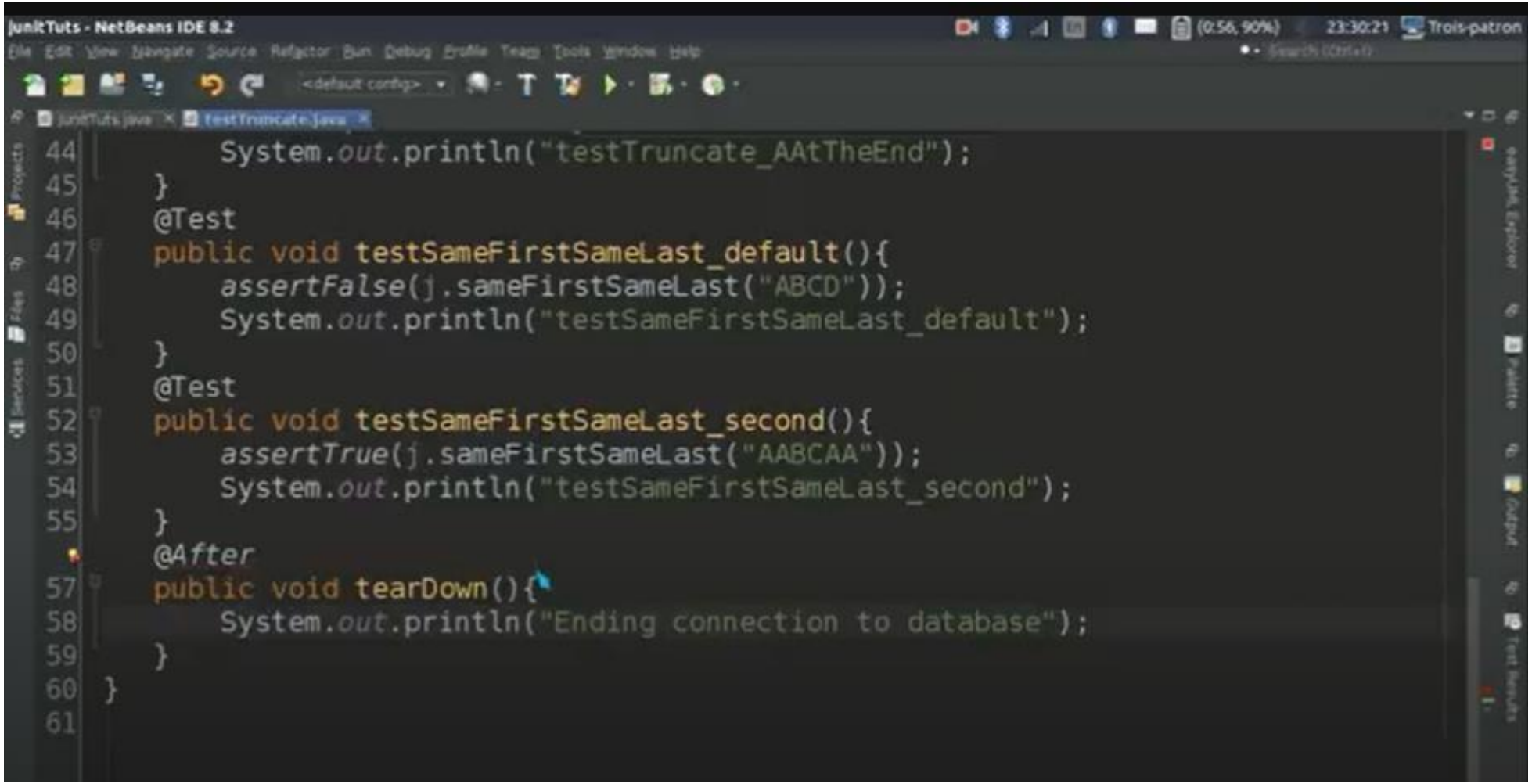
```
@Before
public void name() { ... }
@After
public void name() { ... }
```

- methods to run before/after each test case method is called

```
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```

- methods to run once before/after the entire test class runs


JunitTuts - NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
JunitTuts.java
testTruncate.java
/**
*
* @author root
*/
public class testTruncate {
JunitTuts j;
//AACD => CD; ACD => CD; CDEF=> CDEF; CDAA => CDAA;
@Before
public void setup(){
j=new JunitTuts();
System.out.println("Starting connection to database");
}
@Test
public void testTruncate_first2A(){
assertEquals("CD", j.truncate("AACD"));
System.out.println("testTruncate_first2A");
}
@Test
public void testTruancte_oneA(){


JunitTuts - NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
JunitTuts.java
testTruncate.java
System.out.println("testTruncate_AAtTheEnd");
}
@Test
public void testSameFirstSameLast_default(){
assertFalse(j.sameFirstSameLast("ABCD"));
System.out.println("testSameFirstSameLast_default");
}
@Test
public void testSameFirstSameLast_second(){
assertTrue(j.sameFirstSameLast("AABCAA"));
System.out.println("testSameFirstSameLast_second");
}
@After
public void tearDown(){
System.out.println("Ending connection to database");
}
}

Tips for testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.

- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size

- Think about empty cases and error cases
  - 0, -1, null; an empty list or array

- test behavior in combination
  - maybe `add` usually works, but fails after you call `remove`
  - make multiple calls; maybe `size` fails the second time only

Tips for testing

- Test one thing at a time per test method.
  - 10 small tests are much better than 1 test 10x as large.

- Each test method should have few (likely 1) assert statements.
  - If you assert many things, the first that fails stops the test.
  - You won't know whether a later assertion would have failed.

- Tests should avoid logic.
  - minimize `if/else`, `loops`, `switch`, etc.
  - avoid `try/catch`
    - If it's supposed to throw, use `expected=` ... if not, let JUnit catch it.

Test-driven development

Be

Unit tests can be written after, during, or even before coding.

- test-driven development: Write tests, then write code to pass them.

Imagine

Imagine that we'd like to add a method subtractWeeks to our Date class, that shifts this Date backward in time by the given number of weeks.

Write

Write code to test this method before it has been written.

- Then once we do implement the method, we'll know if it works.

JUnit summary

- Tests need *failure atomicity* (ability to know exactly what failed).
  - Each test should have a clear, long, descriptive name.
  - Assertions should always have clear messages to know what failed.
  - Write many small tests, not one big test.
    - Each test should have roughly just 1 assertion at its end.
- Always use a `timeout` parameter to every test.
- Test for expected errors / exceptions.
- Choose a descriptive assert method, not always `assertTrue`.
- Choose representative test cases from equivalent input classes.
- Avoid complex logic in test methods if possible.
- Use helpers, `@Before` to reduce redundancy between tests.


THANK YOU!
TEAM
WORK
!?!