Recitation 2

Testing

JavaHyperText:

whitebox JUnit

Formatting preferences installed?

Try this in any Eclipse project:

1. Double-click on a class so that it opens in the editing pane.
2. Put parentheses around some expression. For example, change an assignment `x= 3;` to `x= ((3));`
3. Save the file. If the parentheses are NOT removed, then you have work to do to install your Eclipse formatting preferences.
   In JavaHyperText, link Eclipse -> 2. Import preferences ... and follow the instructions there. **This is important!**

Unit Testing

Break your program up into the smallest testable parts or units. Units should be independent.

Units are often one method or a few methods.

Test units as you go! Fix bugs before implementing the next unit.

We deal in this recitation with units that are methods or a group of methods

Writing Correct Code

Code should implement a *specification*.
If you don't have a good specification —if you don't know precisely what it's supposed to do — how can you test it?
Define specification in **Javadoc** comment

```
/** return true iff this time comes before t. */
boolean before(Time t) {…}
```

```
class Time {
    int hr;
    int min;

   before(Time t) {…}
}
```

Javadoc (java documentation) comment:
Multiline comment that starts with /** ends with */
Place it BEFORE the method

Black-box testing


```
/** return true iff this time
                comes before t. */
boolean before(Time t) {…}
```

Develop test cases based only on the unit specification, not looking at the implementation. Example:

Based on the spec, what test cases?

In how many ways

1. Can one time come before another?
2. Can time NOT come before another?

```
class Time {
    int hr, min;
    …
}
```

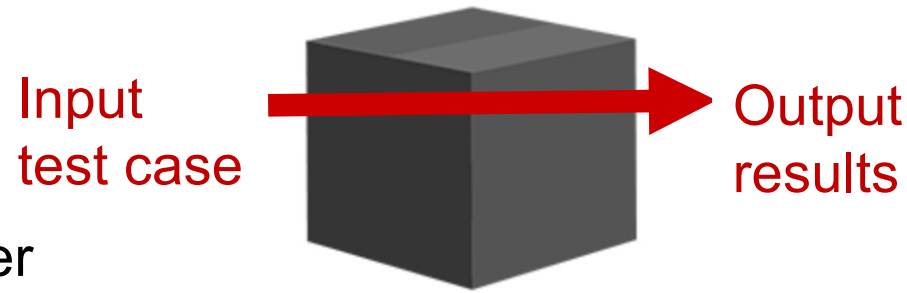
Black-box testing

Test extreme/corner cases.

Example:

For a String, test string with 0 or 1 characters.
For an array or list (as in Python), if it can be empty, test that.

Black-box testing



Critical black-box testing can uncover ambiguities in a specification.

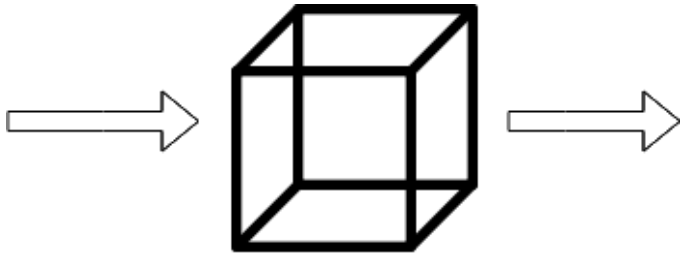
What should be returned for the call num(“member”) ?

```
/** return the number vowels in s. */
int num(String s) {…}
```

The professional programmer looks critically at test cases. If an ambiguity is uncovered, they ask the client what was meant.

White-box testing

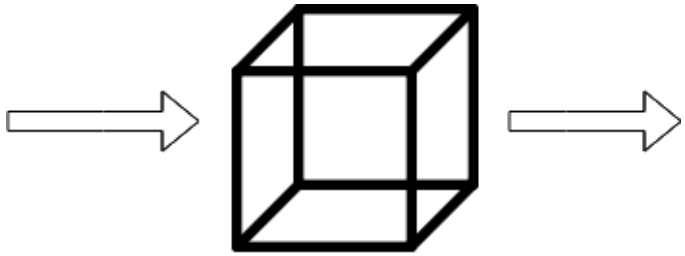
Develop test cases as you look at the implementation –in the box. Name is stupid. You can't see through a white box any better than you can see through a black box.



Better terminology: opaque and transparent boxes. Or, use Glass-box testing instead of White-box testing.

Glass-box testing: code coverage

Code coverage: Develop test cases to ensure that every piece of the program is exercised ---executed or evaluated--- in at least one test case. Common sense! If no test case exercises a piece of code, how can you it is correct?



Test each statement of a unit
Test each branch of an if-statement
Test each expression thoroughly
Test a loop for 0 iterations as well as one or more

Glass-box testing: code coverage

Consider this if-statement: `if (b || (c && d) || e) { … }`

How many test cases are needed to test it thoroughly?

JUnit Testing

- Develop and save a suite of test cases to thoroughly test a unit.
- Be able to run them all easily, whenever changes are made in the unit.
- Should not have to spend time eyeballing a lot of output output to see whether everything is correct. That's prone to error.
- System should alert you to test cases that fail.

Create JUnit testing class

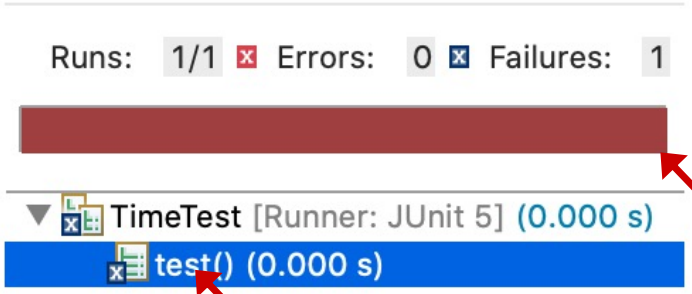
If you have trouble creating a JUnit testing class later, look it up in JavaHyperText. There's a one-html-page discussion.

(1) In Package Explorer (PE) pane, sect directory src.
(2) menu item File -> New -> JUnit Test Case.
(3) Note that it put in name …Test for the class to be created. If the Name is not there, put in a Name.
(4) Press Finish. If it asks to put JUnit 5 on the build path, DO IT!
(5) You see class …Test, with one method.

Run all methods in testing class

Select TimeTest.java in PE pane.
Use menu item Run -> Run

```
class TimeTest {
    @Test
    void test() {
        fail("Not implemented");
    }
}
```



Means an error occurred

Double-click on this to see where error occurred.

Run all methods in testing class

Select TimeTest.java in PE pane.
Use menu item Run -> Run

```
class TimeTest {
    @Test
    void test() {
        //no statement in body
    }
}
```

No error

Call in test() ran without error

assertEquals(expected val, computed val);

```
Time t1= new Time(5, 20);
assertEquals(true, t1.equals(t1));
```

If it's undefined, import it:

```
5⊖    @Test
6     void testEquals() {
7         Time t1= new Time(5, 20);
8 assertEquals(true, t1.equals(t1));
9     Add static import for 'Assert.assertEquals'
10    Add static import for 'Assertions.assertEquals'
11    Create method 'assertEquals(boolean, boolean)'
12    Rename in file (⌘2 R)
```

Only methods with @Test before them are called

assertEquals in test2 should fail.,Run program!

Why did it work?

```
class TimeTest {
    @Test
    void testEquals() {
        Time t1= new Time(5, 20);
        assertEquals(true, t1.equals(t1));
    }

    void test2() {
    Time t1= new Time(5, 20);
        assertEquals(false, t1.equals(t1));
    }
```

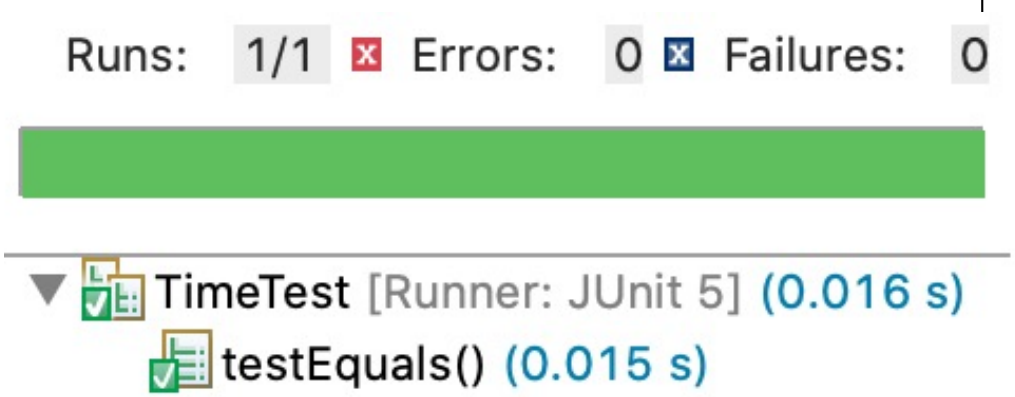
Finished after 0.096 seconds

Runs: 1/1 Errors: 0 Failures: 0

TimeTest [Runner: JUnit 5] (0.016 s)

Only methods with @Test before them are called

It didn't work! Method test2() was not call because there is not annotation @Test before it! Click the horizontal arrow before TimeTest to see this:

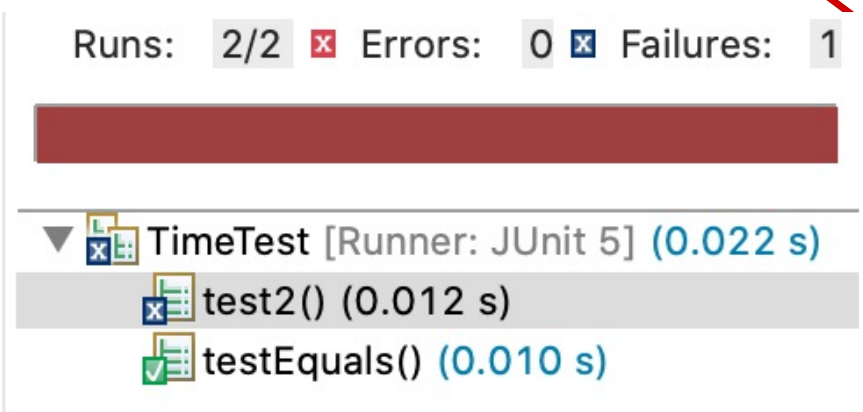



```
class TimeTest {
    @Test
    void testEquals() {
        Time t1= new Time(5, 20);
        assertEquals(true, t1.equals(t1));
    }

    void test2() {
    Time t1= new Time(5, 20);
        assertEquals(false, t1.equals(t1));
    }
```

Only methods with @Test before them are called

Put in @Test and run again:




```
class TimeTest {
    @Test
    void testEquals() {
        Time t1= new Time(5, 20);
        assertEquals(true, t1.equals(t1));
    }
    @Test
    void test2() {
    Time t1= new Time(5, 20);
        assertEquals(false, t1.equals(t1));
    }
}
```

Testing an assert statement

We want to test that the following code snippet throws an exception —stops program with an error

```
Time t= new Time(3, 30);
t.equals(null);
```

```
/** = this time is same as t;
  * Precondition: t not null. */
boolean equals(Time t) {
   assert t != null;
   return …;
}
```

Testing an assert statement

```
boolean equals(Time t) {
    assert t != null;
    return …;
}
```

You might think you could do this:

```
Time t= new Time(3, 30);
assertThrows(AssertionError.class, t.equals(null));
```

Call assertThrows, not assertEquals, to check that an assert statement causes the program to abort with an error message because its boolean expression is false.

The first argument, AssertionError.class, is used to say that the error was that an assert-statement expression was false.

Testing an assert statement

```
boolean equals(Time t) {
    assert t != null;
    return …;
}
```

You might think you could do this:

```
Time t= new Time(3, 30);
assertThrows(AssertionError.class, t.equals(null));
```

Parameters are *call-by value*. So

```
t.equals(null)
```

is evaluated to yield either true or false and
that value is given an the argument to method assertThrows.
But evaluation of t.equals(null) aborts execution!
Method assertThrows doesn't have a chance to test anything!

Testing an assert statement

To make this call-by-value idea clear:

```
boolean isZero(int k) {
    return k == 0;
}
```

The call isZero(5 + 6*20/4) is done in (at least) three steps:

1. Evaluate argument 5 + 6*20/4 to yield 35.
2. Store 35 in parameter k.
3. Execute the body of method isZero.

Testing an assert statement

```
boolean equals(Time t) {
    assert t != null;
    return …;
}
```

So in executing this:

```
Time t= new Time(3, 30);
assertThrows(AssertionError.class, t.equals(null));
```

the body of assertThrows doesn't have a chance to test evaluation of `t.equals(null)` because its evaluation is done before the body is executed.

Testing an assert statement

```
boolean equals(Time t) {
    assert t != null;
    return …;
}
```

Instead do this:

```
Time t= new Time(3, 30);
assertThrows(AssertionError.class, …);
```

Put a method here for assertThrows to call and let assertThrows call it. That method will contain t.equal(null)

Possible since Java 8 using an anonymous function —a function without a name— called a lambda.

An anonymous function

```
boolean equals(Time t) {
    assert t != null;
    return …;
}
```

Function with no parameters and name m:

```
boolean m() {
    return t1.equals(null);
}
```

Function with no parameters and NO name:

```
() -> t1.equals(null)
```

If parameters, put them within parens

Return type not needed. Inferred from context

Separate par. list from body

No return needed. Just bool exp

Testing an assert statement

```
boolean equals(Time t) {
    assert t != null;
    return …;
}
```

```
Time t= new Time(3, 30);
assertThrows(AssertionError.class, () -> t1.equals(null));
```

Use this to test that an AssertionError is thrown (by an assert statement)

The body of assertThrows will call this function.

What to study in JavaHyperText

Testing:
Testing (whitebox, blackbox, structural)

Anonymous functions:
Anonymous function
  Items 1 and 2