CSE 331
Software Design & Implementation

Autumn 2021

Section 4 – Rep Exposure, HW5, Testing

Administrivia

- HW4 due tonight (at 11PM)!
- HW5-1 and HW5-2 Spec out on the website
  - Always plan for work taking 3x longer than expected, so start early!
- Any questions?

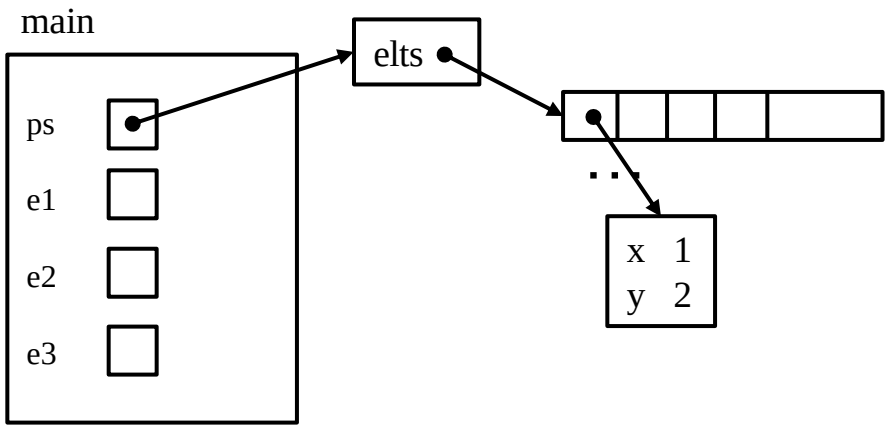
Agenda

- Rep exposure worksheet
- Testing in practice
  - Script Testing
  - JUnit Testing
- Testing exercise

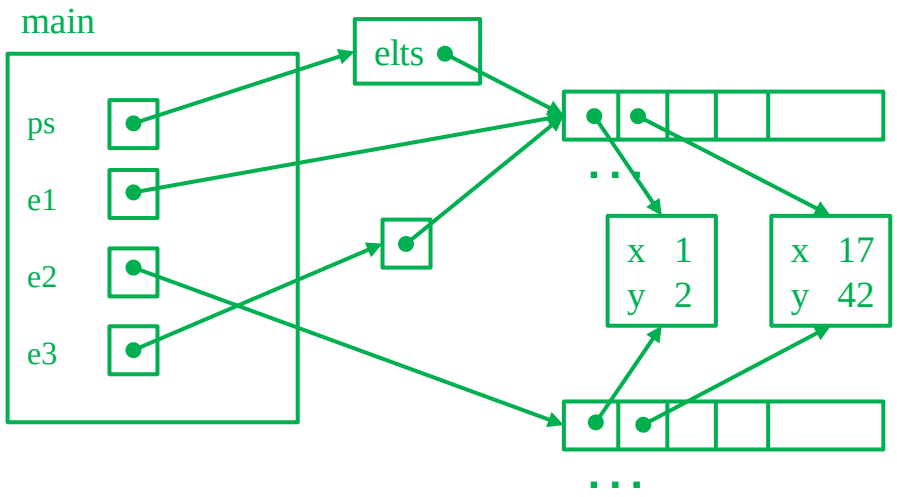
Rep Exposure Worksheet

- Rep Exposure: external (write) access to the private representation of a class
- Let's take a look at a class and look for any rep exposure
- Work in pairs

Rep-Exposure Exercise



Rep-Exposure Exercise (Solution)



HW5: Design before implementation

- HW5: Building an ADT for labeled, directed graphs
  - Labeled: Nodes and edges have label values (**`String`**s)
  - Directed: Edges have direction
  - Edges with same source and destination will have unique labels

HW5: Design before implementation

- HW5: Building an ADT for labeled, directed graphs
  - Labeled: Nodes and edges have label values (**`String`**s)
  - Directed: Edges have direction
  - Edges with same source and destination will have unique labels

- The exact interface of your **`Graph`** class is up to you
  - So, no given JUnit tests bundled with the starter code
  - Advice: Look ahead at HW6 and consider its likely needs
  - Reminder: *Not a generic class.*

HW5: Design before implementation

- HW5: Building an ADT for labeled, directed graphs
  - Labeled: Nodes and edges have label values (**`String`**s)
  - Directed: Edges have direction
  - Edges with same source and destination will have unique labels

- The exact interface of your **`Graph`** class is up to you
  - So, no given JUnit tests bundled with the starter code
  - Advice: Look ahead at HW6 and consider its likely needs
  - Reminder: *Not a generic class.*

- HW5 split into 2 parts
  1. **Design and specify a graph ADT**
  2. Implement that ADT specification

HW5: Testing

- The design process includes crafting a good test suite
  - Script tests and JUnit tests

- **Script Tests** (**`src/test/resources/testScripts/`**)
  - Test script files ***`name`*`.test`** with corresponding ***`name`*`.expected`**
  - Validate behavior intrinsic to high-level concept (abstract meaning)
  - Tested properties should be expected of any solution to HW5

- **JUnit Tests** (**`src/test/java/graph/junitTests/`**)
  - JUnit test classes
  - Validate behavior that can't be tested with script tests.

- If you can validate a behavior using either test type, use a script test!

HW5: Why Script Tests?

- Everyone's implementation could (will!) be different, so we (staff) cannot write JUnit tests for everyone to use or to use for checking everyone's code.

- We still need a way to test that you specify and implement the proper behavior, so we use script tests that work regardless of the implementation.

- They test what the methods are doing, they don't care how the methods are doing it.

HW5: Script Tests

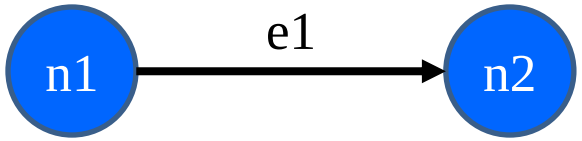
Each script test is expressed as text-based script ***`foo.test`***

- One command per line, of the form: **`Command`** arg_1 arg_2 …
- Script's output compared against ***`foo.expected`***
- Precise details specified in the homework
- Match format ***exactly***, including whitespace!

Command (in *`foo.test`*)	Output (in *`foo.expected`*)
`CreateGraph` *name*	**`created graph`** *name*
`AddNode` *graph label*	**`added node`** *label* **`to`** *graph*
`AddEdge` *graph parent child label*	**`added edge`** *label* **`from`** *parent* **`to`** *child* **`in`** *graph*
`ListNodes` *graph*	*graph* **`contains:`** $label_{node}$ …
`ListChildren` *graph parent*	**`the children of`** *parent* **`in`** *graph* **`are:`** *child* **`(`**$label_{edge}$**`)`** …
`#` *This is comment text …*	**`#`** *This is comment text …*

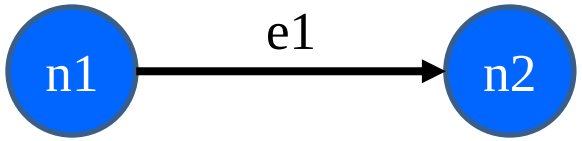
HW5: `example.test`

```
# Create a graph
CreateGraph graph1

# Add a pair of nodes
AddNode graph1 n1
AddNode graph1 n2

# Add an edge
AddEdge graph1 n1 n2 e1

# Print all nodes in the graph
ListNodes graph1

# Print all child nodes of n1 with outgoing edge
ListChildren graph1 n1
```



HW5: `example.expected`

```
# Create a graph
created graph graph1

# Add a pair of nodes
added node n1 to graph1
added node n2 to graph1
```




```
# Add an edge
added edge e1 from n1 to n2 in graph1

# Print all nodes in the graph
graph1 contains: n1 n2

# Print all child nodes of n1 with outgoing edge
the children of n1 in graph1 are: n2(e1)
```

HW5: Creating a script test

1. Write test steps as script commands in a file **`foo.test`**
2. Write expected ("correct") output in a file **`foo.expected`**
   - …taking care to match the output format *exactly*
3. Place both files under **`src/test/resources/testScripts/`**
4. Run all such tests via the Gradle task **`scriptTests`**
   - After class implemented and **`GraphTestDriver`** stubs filled

- Let's try writing one…

HW5: Script Test Driver

- Script tests do not magically call your Graph methods
- We need someway to map script test commands (`AddNode graph1 n1`) to some Java code that uses the methods of your graph class
- You have to write this translation
  - Again, we don't know "how" to add a node to your graph, e.g.
- Let's take a look at the starter code…

HW5: Script tests vs. JUnit Tests

- Script tests will not cover every case for your graph:
  - What if you have additional methods beyond those script commands?
  - What about “bad” input for your graph?
  - What happens when you try to add the same node twice?
  - …
- We need some way to test graph operations that cannot be tested by our script tests
- For this, we use JUnit (like in HW3 and HW4)

HW5: Creating JUnit tests

1. Create JUnit test class in **`src/test/java/graph/junitTests/`**
2. Write a test method for each unit test
3. Run all such tests via the Gradle task **`junitTests`**

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

/** Document class... */
public class FooTests {
  /** Document method... */
  @Test
  public void testBar() { ... /* JUnit assertions */ }
}
```

HW5: Creating JUnit tests

1. Note: Your JUnit tests will fail in hw5 part 1, because you have not implemented the actual methods yet
   - The same goes for your script tests

2. You will do that in part 2

JUnit for test authors

The following slides are included for reference and add additional material that you'll need to write tests for HW 5.

Writing tests with JUnit

Annotate a method with **`@Test`** to flag it as a JUnit test

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

/** Unit tests for my Foo ADT implementation */
public class FooTests {
  @Test
  public void testBar() {
    ... /* use JUnit assertions in here */
  }
}
```

Common JUnit assertions

JUnit's documentation has a full list, but these are the most common assertions.

Assertion	Failure condition
`assertTrue(`***`test`***`)`	***`test`*** `== false`
`assertFalse(`***`test`***`)`	***`test`*** `== true`
`assertEquals(`***`expected, actual`***`)`	***`expected`*** and ***`actual`*** are not equal
`assertSame(`***`expected, actual`***`)`	***`expected`*** `!=` ***`actual`***
`assertNotSame(`***`expected, actual`***`)`	***`expected`*** `==` ***`actual`***
`assertNull(`***`value`***`)`	***`value`*** `!= null`
`assertNotNull(`***`value`***`)`	***`value`*** `== null`

Any JUnit assertion can also take a string to show in case of failure, *e.g.*, **`assertEquals("helpful message", expected, actual)`**.

Always* use >= 1 JUnit Assertion

- If you don't use any JUnit assertions, you are only checking that no exception/error occurs

- That's a pretty weak notion of passing a test; rarely the best test you could write

- Having more than one JUnit assertion in a test may make sense, but one is the most common scenario
  - "Each test should test one (new) thing" (most of the time)

* = Special-case coming in a couple slides 

JUnit assertions vs Java's assert

- Use JUnit assertions **only in JUnit test code**
  - JUnit assertions have names like **`assertEquals`**, **`assertNotNull`**, **`assertTrue`**
  - Part of JUnit framework used to report test results
    - Accessed via **`import org.junit`**….
  - **Don't** use in ordinary Java code (***never*** **`import org.junit`**…. in non-JUnit code)

- Use Java's **`assert`** statement in ordinary Java code
  - Use liberally to annotate/check "must be true" / "must not happen" / etc. conditions
  - Use in **`checkRep()`** to detect failure if problem(s) found
  - **Do not** use in JUnit tests to check test result – does not interact properly with JUnit framework to report results

Checking for a thrown exception

- Need to test that your code throws exceptions as specified

- This kind of test method fails if its body does *not* throw an exception of the named class
  - May not need any JUnit assertions inside the test method

```
@Test(expected=IndexOutOfBoundsException.class)
public void testGetEmptyList() {
  List<String> list = new ArrayList<String>();
  list.get(0);
}
```

Test ordering, setup, clean-up

JUnit does not promise to run tests in any particular order.

However, JUnit can run helper methods for common setup/cleanup

- Run before/after *each* test method in the class:

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

- Run before/after *all* test methods in the class:

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

Tips for effective testing

- Use constants instead of hard-coded values
  - Makes change easier later on
- Take advantage of assertion messages
- Give a descriptive name to each unit test (method)
  - Verbose but clear is better than short and inscrutable
  - Don't go overboard, though :-)
- Write tests with a simple structure
  - Isolate bugs one at a time with successive assertions
  - Helps avoid bugs in your tests too!
- Aim for thorough test coverage
  - Big/small inputs, common/edge cases, exceptions, ...

Test Design Worksheet

- Work in pairs
- Give logic of the tests, not actual code
- Only test operations provided on the worksheet
- More details in lecture if additional information/review needed