CSE 331
Software Design & Implementation

Spring 2021
Section 5 – HW5 implementation, Review
Administrivia

• Done with HW5 part 1
  – *hw5-part1-final* tag
  – Do not include any ADT implementation in this commit/tag

• HW5 part 2 (ADT implementation) due next Wednesday.
  – Reminder (1): *No generics for now!*
  – Reminder (2): Be sure to add/commit/push new files in git
  – Reminder (3): Remember to commit and push your code often, even if your assignment isn’t finished yet!
Agenda

• Walk-through of the test-script driver (to run .test files)

• Representation Exposure

• Managing an expensive checkRep

• Review of equals and hashCode

• Brief mid-point summary/review
Rep-Exposure Exercise
Rep-Exposure Exercise (Solution)
Refresher: Format of script tests

Each script test 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 and output order!

<table>
<thead>
<tr>
<th>Command (in <code>foo.test</code>)</th>
<th>Output (in <code>foo.expected</code>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateGraph <code>name</code></td>
<td>created graph <code>name</code></td>
</tr>
<tr>
<td>AddNode <code>graph label</code></td>
<td>added node <code>label</code> to <code>graph</code></td>
</tr>
<tr>
<td>AddEdge <code>graph parent child label</code></td>
<td>added edge <code>label</code> from <code>parent</code> to <code>child</code> in <code>graph</code></td>
</tr>
<tr>
<td>ListNodes <code>graph</code></td>
<td><code>graph contains: label</code>_node` ...</td>
</tr>
<tr>
<td>ListChildren <code>graph parent</code></td>
<td><code>the children of </code>parent<code>in</code>graph<code> are: child (label</code>edge<code>)</code> ...</td>
</tr>
<tr>
<td># This is comment text ...</td>
<td># This is comment text ...</td>
</tr>
</tbody>
</table>
Refresher: `example.test`

```plaintext
# 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
```
Refresher: 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)
How the spec. tests work

- In HW5 pt 1, you wrote spec. tests in the form of `.test` scripts
  - As well as a `.expected` file for each test’s expected outcome

- The JUnit class `ScriptFileTests` runs all these tests
  - Looks for all the `.test` files in its directory
  - Compares test output against corresponding `.expected` file

- `ScriptFileTests` needs a bridge to your graph implementation
  - That’s exactly what the `GraphTestDriver` class is for
Driver for spec. test scripts

- **GraphTestDriver** knows how to read these test scripts

- **GraphTestDriver** calls a method to “do” each verb
  - `CreateGraph`, `AddNode`, `AddEdge` …
  - One method stub per script command for you to fill with calls to your graph code

- **Note**: Completed test driver should sort lists before printing
  - Just to ensure predictable, deterministic output
  - Your graph implementation itself should not worry about sorting
Demo

Here’s a quick tour of the GraphTestDriver!
Sorting with the driver

- **Use the test driver appropriately!**
  - From last slide: “Completed test driver should sort lists before printing.”

- Script test output for hw5 needs to be sorted so we can mechanically check it.

- This means sorted output for tests does **NOT** mean sorted internal storage in graph.
  - If sorting behavior is needed, Graph ADT clients (including the test driver) can sort those labels.
In other words…

The Graph ADT in general should **NOT** assume that node or edge labels are sorted.
Expensive `checkRep`

- A complicated rep. invariant can be expensive to check
  - Especially iterating over internal collection(s)
  - For example, examining every edge in a graph

- A slow `checkRep` could cause our grading scripts to time-out
  - Can be really useful during testing/deugging, but
  - Need to disable the really slow checks before submitting

- We have a tension between two goals:
  - Thorough, possibly slow checking for development
  - Essential, necessarily fast checking for production/grading

- What to do?
Use a debug flag to tune \texttt{checkRep}

• Repeatedly (un)commenting sections of code is a poor solution

• Instead, use a class-level constant as a toggle
  – \texttt{private static final boolean DEBUG = \ldots;}
    • \texttt{false} for only the fast, essential checks
    • \texttt{true} for all the slow, thorough checks
  – Real-world code often has several such “debug levels”

```java
private void checkRep() {
    assert fast_checks();
    if (DEBUG)
        assert slow_checks();
}
```
The `equals` method (review)

- Specification mandates several properties:
  - Reflexive: `x.equals(x)` is true
  - Symmetric: `x.equals(y) ⇔ y.equals(x)`
  - Transitive: `x.equals(y) ∧ y.equals(z) ⇒ x.equals(z)`
  - Consistent: `x.equals(y)` shouldn’t change, unless perhaps `x` or `y` did
  - Null uniqueness: `x.equals(null)` is false

- Several notions of equality:
  - Referential: literally the same object in memory
  - Behavioral: no sequence of operations could tell apart
  - Observational: no sequence of observer operations could tell apart
The `hashCode` method (review)

- Specification mandates several properties:
  - *Self-consistent*: `x.hashCode()` shouldn’t change, unless `x` did
  - *Equality-consistent*: `x.equals(y) ⇒ x.hashCode() == y.hashCode()`

- Equal objects *must* have the same hash code.
  - Implementations of `equals` and `hashCode` work together for this
  - If you override `equals`, you *must* override `hashCode` as well
Overriding `equals` and `hashCode`

- A subclass method overrides a superclass method, when…
  - They have the exact same name
  - They have the exact same argument types

- An overriding method should satisfy the overridden method’s spec.

- Always use `@override` tag when overriding `equals` and `hashCode` (or any other overridden method)

- Note: Method overloading is not the same as overriding
  - Same name but distinguished by different argument types

- Keep these details in mind if you override `equals` and `hashCode`.
Your turn!

Spend a few minutes on the worksheet problems, then we’ll go over answers.
Topics covered so far

- **Reasoning about code:**
  Hoare logic, forward/backward reasoning, loop invariants, …

- **Specification:**
  JavaDoc, stronger v. weaker, satisfaction, substitutability, …

- **Data abstraction:**
  ADT spec./impl., abstraction functions, rep. invariants, …
  – Including `checkRep` as covered in lecture/section

- **Testing:**
  unit v. system, black-box v. clear-box, spec. v. impl., …

- **Modularity:**
  (de)composition, cohesion, coupling, open-closed principle, …

- **Object identity:**
  equivalence relation, `equals`, `hashCode`, …