CSE 331

Software Design & Implementation

Section: HW6; Equality

Reminders

- Even though no code, still need to pass pipeline!

Upcoming Deadlines

• HW5	due 11pm tonight (7/21)
• Prep. Quiz: HW6	due 11pm Monday (7/25)

Last Time…

- Modular Design
- Equals and Hashcode
- Exceptions
- Subtyping

Today's Agenda

- HW6 Overview
- Review: Equals + Hashcode

Refresher: Format of 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 …*

Refresher: `example.test`

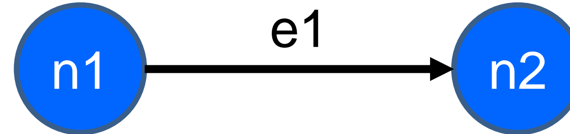
```
# Create a graph
CreateGraph graph1

# 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
```



Refresher: `example.expected`

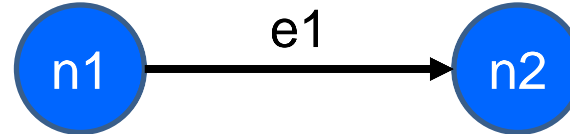
```
# Create a graph
created graph graph1

# 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 script tests work

- In HW5, you wrote script tests in the form of **`.test`** files
  - As well as an **`.expected`** file for each test's expected outcome
- The JUnit class **`ScriptFileTests`** runs all these tests
  - Looks for all the **`.test`** files in the **`src/test/resources/testScripts`** folder
  - Compares test output against corresponding **`.expected`** file
- **`ScriptFileTests`** needs a bridge to your graph implementation
  - That's exactly what the **`GraphTestDriver`** class is for

Graph Test Driver

- **`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 for **`ListNodes`** and **`ListChildren`**
  - Just to ensure predictable, deterministic output
  - Your graph implementation itself should not worry about sorting

Graph Test Driver Output

- The Graph Test Driver is a client of our graph…
  - …but not the only client.
  - Your graph should not be designed to be exclusively used for the test driver.

- ListChildren in the test driver should print out: "**the children of** *parent* **in** *graph* **are:** *child***(***label*$_{edge}$**)** …"

- This does **not** mean that you should have a method on your graph called ListChildren that returns this String
  - Because that isn't useful for other clients

Sorting with the driver

- **Use the test driver appropriately!**
  - From before: “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 or even comparable(!).

(of course they can be tested for equality with equals())

Demo

Here's a quick tour of the **`GraphTestDriver`**!

Expensive `checkRep`s

- 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/debugging, 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 `checkRep`

- Repeatedly (un)commenting sections of code is a poor solution
- Instead, use a class-level constant as a toggle
  - Ex.: `private static final boolean DEBUG = …;`
    - `false` for only the fast, essential checks
    - `true` for all the slow, thorough checks
  - Real-world code often has several such “debug levels”

```
private void checkRep() {
    assert fast_checks();
    if (DEBUG)
        assert slow_checks();
}
```

Equals and Hashcode

The **`equals`** method (review)

- Specification mandates several properties:
  - *Reflexive*: **`x.equals(x)`** is **`true`**
  - *Symmetric*: **`x.equals(y)`** $\Leftrightarrow$ **`y.equals(x)`**
  - *Transitive*: **`x.equals(y)`** $\wedge$ **`y.equals(z)`** $\Rightarrow$ **`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 (excluding **==**)
  - *Observational*: no sequence of observer operations could tell apart (excluding **==**)

The **`hashCode`** method (review)

- Specification mandates several properties:
  - *Self-consistent*: **`x.hashCode()`** shouldn't change, unless **`x`** did
  - *Equality-consistent*: **`x.equals(y)`** $\Rightarrow$ **`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

- Ideally a good **`hashCode`** method returns different values for unequal objects, but the contract does not require this.

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**.

`equals` and `hashCode` worksheet

- Let's practice…

Before next lecture...

1. Do HW5 by tonight!
   - Written portion (submit PDF on Gradescope)
   - Coding portion (push and tag on GitLab)

2. Review JUnit testing slides discussed in the last section.