# CSE 331
# Software Design & Implementation

Winter 2023

Section 5 – HW5 implementation, Equals, Hashcode

# Administrivia

- HW5 part 1 due **tonight**! ***11pm***
  - **hw5-part1-final** tag
  - Do not include any ADT implementation in this commit/tag
  - Do include specifications, tests (Script tests + JUnit)
    - Everything must compile and javadoc generations must work

- HW5 part 2 (ADT implementation) due next week
  - Reminder: please commit and push work regularly

- Midterm on Tuesday (2/7), 5 – 6 PM
  - Review Session on Sunday (2/5) afternoon in G20
  - More times/details tba via email

# HW5 Part 2

- Now that your design is complete, you will need to implement your graph to complete HW5. This includes:
  - Anything private: private fields, private methods (such as **`checkRep`**), and private classes (optionally)
  - Abstraction Function and Representation Invariant
  - Method implementations
  - **`GraphTestDriver`** to run your script tests from HW5 Part 1

- Make sure to incorporate feedback from HW5 Part 1!

# Refresher: Format of script tests

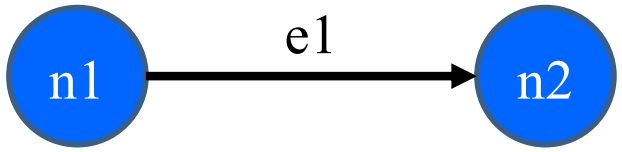
Each script test expressed as text-based script **`foo.test`**

- 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!

| 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 …* |

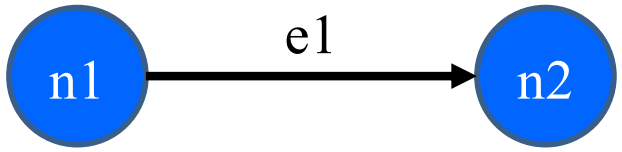
# Refresher: `example.test`

```
# 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 part 1, 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 does

# 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

```
private void executeCommand(String command, List<String> arguments) {
  ...
  switch(command) {
    case "CreateGraph":
      createGraph(arguments);
      break;
    case "AddNode":
      addNode(arguments);
      break;
  ...
}
```

The **`GraphTestDriver`** reads each line of your **`.test`** file, parses its arguments, and calls the **`GraphTestDriver`** method that corresponds to that command.

- Note: these are methods in **`GraphTestDriver`**, not your graph ADT!

# Graph Test Driver Example

## `.test` file

```
AddNode myGraph node
```

- In this example, the graph ADT class name is **`ABC`** and the name of the method to add a node is **`xyz`**
- The test driver should retrieve the graph with the corresponding name from the **`graphs`** map
  - This allows one test to have multiple graphs!
- Then, it should call methods on your graph instance and print some output (to be compared against the **`.expected`** file)

## `GraphTestDriver`

```
private final Map<String, ABC> graphs;
...
private void addNode(String graphName,
                     String nodeName){
  ABC graph = graphs.get(graphName);
  graph.xyz(...)
  output.println("added node " +
  nodeName + " to graph " +   graphName);
}
```

## `Your Graph Class`

```
class ABC {
  ...
  public ... xyz(...) {
    // this method adds a node
    ...
  }
}
```

# Graph Test Driver Output

- The Graph Test Driver is a client of our graph…
  - …but not the only client.
    - Not much different from your Junit tests!
  - Your graph should **not** be designed to be used exclusively by 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 in your graph ADT 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`**!

# Graph Implementation

- It's up to you how to implement your graph!
  - This includes the fields within your Java class

- Frequent, simple operations on your graph should have *reasonably quick* runtime
  - E.g. adding a node should not require a traversal of every node in the graph (i.e. not be `O(n)`)
  - This should affect your choice of representation

- Our graphs can get very big (see HW6…)!
  - If your graph is too slow, it might take a very long time…

# Expensive `checkReps`

- 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)` ⇔ `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 (details in lecture tomorrow):
  - *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
    - This includes your HW5!

- 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 method name but distinguished by different argument types
- Keep these details in mind if you override `equals` and `hashCode`.

# Overriding `equals`

- To override, you must have the same name and argument types
- For `equals`, you must have one argument with type `Object`
  - The argument name can be anything!
- Otherwise, it's overloading!

```
public bool equals(Object o){…}
```

## Overriding

```
public bool equals(Object o){…}
```

```
public bool equals(Object d){…}
```

## Overloading

```
public bool equals(Cat o){…}
```

# Compile-Time vs. Runtime Types

- `MusicPlayer p = new iPhone();`
- `p.playSong("Call Me Maybe");`
- The left-hand side type, a.k.a. the declared or static type, is what is checked at compile time.
  - At this point, a method signature that most closely matches the declared types is chosen from the *declared type* class of the variable or its superclasses.
- The right-hand side type is the actual type of the object.
  - At runtime, the dynamic method dispatch process looks for most specific method that exactly matches the signature found in compile time.
  - It looks in the class of the *actual type* and then its superclasses. This is the code that actually runs.
- See Kevin Lin's Autumn 2020 CSE 143 website for an extended example: https://courses.cs.washington.edu/courses/cse143/20au/election-simulator/notional-machine/

# 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`**, …

# Some Slides on Interfaces

We won't go over these, and the information will not be tested, but they are worth checking out on your own.

# Classes and interfaces

- In Java, the fundamental unit of programming is a class
  - Everything is an instance of a class (besides primitive values)

- A class can extend another class and implement interfaces
  - **`extends`** exactly one class (**`java.lang.Object`** by default)
  - **`implements`** arbitrarily many interfaces

- An interface can extend other interfaces
  - For example, **`List<E>`** is a subinterface of **`Collection<E>`**

# Classes and objects (review)

- Every object is an instance of a class
  - Class defines data content (fields) and operations (methods)
- A class inherits the fields and methods of its superclass
- Defining class also defines a type for its object instances
  - For example, a class **`Foo`** defines a type **`Foo`**
- Subtyping mirrors class inheritance
  - **`Subclass`** is a subtype of **`Superclass`**
  - **`Superclass`** is a supertype of **`Subclass`**
- We'll get into the details much more carefully next week

# Interfaces

- An interface is a bundle of method specifications
  - Method signatures automatically **`public abstract`**
  - Constants also allowed (implicitly **`public final static`**)

- An interface does *not* contain an implementation
  - Java 8 relaxed this to allow default implementations in interfaces to add lambdas to collection classes without breaking (too much) old code, but we'll ignore that in CSE 331 and only use interfaces for pure specifications.

- Defining an interface also defines a type
  - Cannot create an instance of an interface

- Subtyping mirrors interface implementation
  - Implementing class is a subtype of the interface
  - The interface is a supertype of any implementing class

# Example of an interface

```
/**
  * This interface imposes a total ordering on the objects of
  * each class that implements it. ...
  */
public interface Comparable<T> {
 /**
  * Compares this object with the specified object for order. ...
  * @param o the object to be compared.
  * @return a negative integer, zero, or a positive integer as
  *         `this` is less than, equal to, or greater than `o`.
  *
  * @throws NullPointerException if `o` is null
  * @throws ClassCastException if the type of `o` ...
  */
  public int compareTo(T o);
}
```

# Implementing an interface

- A class can implement one or more interfaces
  ```
  class Kitten implements Pettable, Huggable { … }
  ```

- Instances of an implementing class have interface type(s) as well
  - Due to subtyping
  - This is why an **`ArrayList<E>`** instance also has type **`List<E>`**

- The class must define (or inherit) all methods declared by interface
  - Except for abstract classes, which don't have to implement everything
  - Same caveat about Java 8 allowing default interface implementations

# Using interfaces

- Generally good to define interfaces for significant abstractions
  - Particularly so for general ADTs like **`List<E>`** or **`Map<K, V>`**

- Write code against interface types where applicable
  - From 14X:
    "Declare variables as **`List<E>`** instead of **`ArrayList<E>`**."
  - Best not to depend on specific implementation if interface is enough
  - Lets your code work with different implementations later on

- Interfaces and classes are appropriate in various circumstances

- In HW5, you probably don't need to define any interfaces
  - But ok if you want to and use them correctly