# CSE 331 Software Design and Implementation

**Homework 4:** ***Designing and Implementing an ADT***
**Due:** May 7 2013@ **11pm**

## Handout H4

Contents:

## Introduction

In this assignment you will design, implement, and test a graph ADT. Given an abstract, high-level description of the desired ADT, you will develop a working Java implementation. You get to choose both the public interface and internal representation, then decide what unit tests to write to demonstrate that your implementation works correctly.

Although this is an individual assignment, in the design stage you will learn more and produce a better design by bouncing your ideas off others and incorporating their own ideas. To encourage and facilitate discussion, we are relaxing the standard collaboration policy somewhat for Problem 2 (the interface design stage). You must first attempt to come up with an interface design on your own, but then you are strongly encouraged to discuss your design with your classmates who have also attempted a design (as well as the course staff during office hours) as much as you like, without restrictions on what materials you can bring to or from meetings. On the other parts of this assignment, including the implementation stage, the standard collaboration policy still applies.

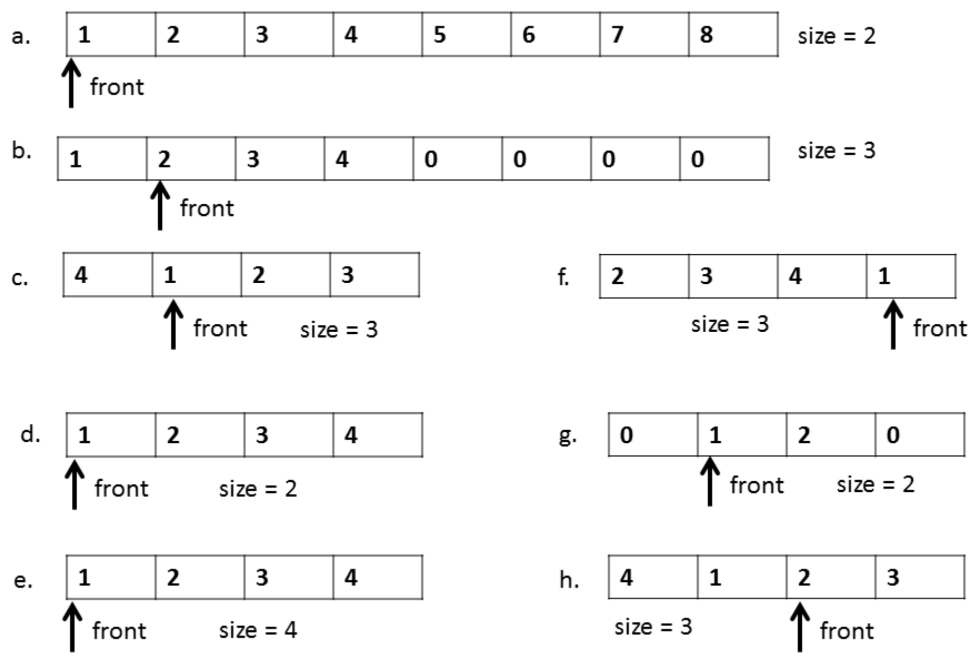
This assignment is the first of a multi-part project. Read Homework 5 to get an idea of one application for your graph ADT, but there will be more. By the end of the quarter, you will build a fully-fledged application for getting directions to travel between two buildings on the UW campus.

## Problem 1: Written Exercises [9 points]

This part is designed to test your understanding of some of the ADT concepts from lecture. To get started, update your working copy of your repository to get the files for this assignment. Place your answers to the questions below in the file `hw4/answers/problem1.txt`.

a. For each of the classes below, write the abstraction function and representation invariant, referring to the source code provided in the `hw4/problem1` package of your project.
   1. `IntQueue1`
   2. `IntQueue2`

b. Below are several snapshots of an `IntQueue2` object's internal state at different points in a program. Which snapshots are equivalent to each other at the abstract level? In other words, partition the snapshots into groups based on which are identical to each other from the client's perspective.


a. 1 2 3 4 5 6 7 8 size = 2 front
b. 1 2 3 4 0 0 0 0 size = 3 front
c. 4 1 2 3 front size = 3
d. 1 2 3 4 front size = 2
e. 1 2 3 4 front size = 4
f. 2 3 4 1 size = 3 front
g. 0 1 2 0 front size = 2
h. 4 1 2 3 size = 3 front

c. Below are signatures for various methods and constructors. For each problem, state and justify in 1-2 sentences (per problem) whether the method or constructor could possibly expose the representation, given the information available. Explain any assumptions you made.
   1. `public int solveEquations(int x, int y, int z)`
   2. `public String[] decode(boolean slowly)`
   3. `private Date myBirthday()`
   4. `public String toString()`
   5. `public Iterator<Integer> elements()`

6. `public Deck(List<Card> cards)`

## Building a Graph

### Definitions and Terminology

In the rest of this assignment you will design, implement, and test a *directed labeled multi-graph*.

A *graph* is a collection of *nodes* (also called vertices) and *edges*. Each edge connects two nodes. In a *directed graph*, edges are one-way: an edge `e = ?A,B?` indicates B that is directly reachable from A. To indicate that B is directly reachable from A *and* A from B, a directed graph would have edges `?A,B?` and `?B,A?`.

The *children* of node B are the nodes to which there is an edge *from* B. In Fig. 1, the children of B are A and C. Similarly, the parents of B are the nodes from which there is an edge *to* B. In Fig. 1, B only has one parent, A.

A *path* is a sequence of edges `?node1,node2?, ?node2,node3?, ?node3,node4?, ...`. In other words, a path is an ordered list of edges, where an edge *to* some node is immediately followed by an edge *from* that node. In Fig. 1, one possible path is `?B,A?,?A,B?,?B,C?`. This path represents traveling from B to A to B to C. A path may traverse a given edge twice.

A *reflexive edge* is an edge which starts and ends at the same node. In Fig 1. there is a reflexive edge from A to A: `?A,A?`. Your graph implementation must allow for reflexive edges.

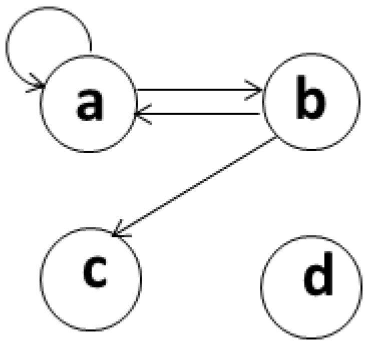


Figure 1: A simple directed graph with four nodes.

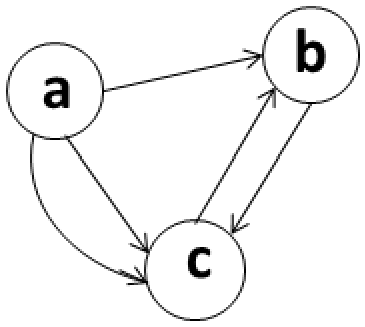


Figure 2: A directed multigraph.

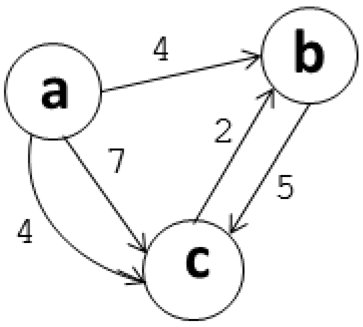


Figure 2: A directed labeled multigraph.

In a *multigraph*, there can be any number of edges (zero, one, or more) between a pair of nodes. Fig. 2 shows a multigraph with 2 edges from A to C.

In a *labeled graph* (Fig. 3), every edge has a label containing information of some sort. Labels are not unique: multiple edges may have the same label. In theory a graph could contain two "identical" edges with the same starting point, ending point, and label, but for this project you may decide whether to allow identical edges in your implementation. (Whatever you decide, make sure it is clearly documented so clients understand how they can use your ADT.)

If you want to learn more, read Wikipedia's definition of a graph. Then if you still have a question, ask the course staff.

Many interesting problems can be represented with graphs. For example:

- A graph can represent airline flights between cities, where each city is a node and an edge `?A,B?` indicates that there is a flight from A to B. The edge label might represent the cost in money (airfare), time (length of flight), or distance.
- To find walking routes across the UW campus, you can build a graph where nodes represent buildings and other locations and edges represent walking paths connecting two locations. The label/cost of an edge is the physical length of that path.
- The Web can be modeled as a graph with node for every webpage and an edge `?A,B?` if page A links to page B. The label could indicate the anchor text for a link on page A, or the number of links from page A to page B.
- Facebook is essentially a giant graph with nodes for users and edges between friends. (You can see a visualization of the Facebook graph.)

## Problem 2: Write a Specification for Graph [25 points]

To start, you will specify the API for a Java class or classes representing a directed labeled multigraph. The API, or public interface, is the collection of public classes and methods that clients of your graph ADT will use. We recommend that you work in iterative layers of detail. Start rough - preferably with pencil and paper - by brainstorming what operations the ADT needs to support, how it might be logically broken into modules (classes/interfaces), and how these modules connect to each other. Then, jot down a list of methods for each class that provide these operations. Think through some possible client applications, particularly the application in Homework 5, to get an idea for what operations are needed. Perhaps write some pseudocode (or even real code) for the application in Homework 5 and make sure your graph interface meets its needs. (Note: don't worry about implementing the breadth-first search algorithm described in Homework 5 for this assignment. We prefer that you focus on the lower-level operations needed to build the graph and to be able to perform breadth-first search.)

Initially you should keep your design rough - don't write formal class and method specifications with all the proper clauses right away. Your design will likely go through multiple iterations, and it's easier to throw away parts before you've invested too much effort in them.

Once you are satisfied with your high-level design, write a Javadoc specification for each class and method. Follow the format we have been using in CSE 331, remembering to use both standard Javadoc tags (`param`, `returns`, `throws`) and ones introduced for this course (`specfield`, `derivedfield`, `requires`, `effects`\ , `modifies`). A good approach is to create skeleton implementations of your classes containing only method "stubs", then write your specifications in place in the code. A stub is a not-yet-implemented method whose body simply throws an exception, as you saw in the Homework 3 starter code. Stubbing out a class gives you the flexibility to write client code and tests that use it before completing the implementation. The skeleton code will not behave correctly or pass tests, but the program will compile and run.

For this assignment, you may restrict your graph to store the data in nodes and edge labels as `String`s. We strongly recommend you **DO NOT** use Generics for this assignment, but we do not absolutely forbid it. If you do use generics, you MUST heed the warning about compiling your code in the hints section. In a future assignment, you will use generics to make your ADT work with other data types - text, integers, doubles, etc. You may assume nodes are uniquely identified by their data contents: that is, no two nodes store the same data.

Design problems such as this one are open-ended by nature: we are not looking for one "right" design. There are principles of good design you have learned in lecture, however. You will also find suggestions of what to consider and how to approach the problem in the hints section. Also, designs naturally evolve as requirements change, so try to make your code extensible, but don't over-generalize.

Please include in your turnin a brief description (at most one sentence for each operation) of why you included the operations you did and why you feel they are a sufficient interface to a graph. If your design includes multiple classes or interfaces, explain why you included each one; if not, explain whether you considered additional classes and why you decided not to include them. This should be placed in `hw4/answers/problem2.txt`. The only other deliverables for this problem are the Javadoc class and method specifications as they appear in the source code you submit in Problem 4; you do not need to submit your Javadoc in a separate file.

## Problem 3: Write Tests for Graph [12 points]

Write a black-box test suite for your Graph specifications. It's important to write your tests before your code, but your tests will not pass until you have completed Problem 4.

Make sure you are familiar with the difference between implementation and specification tests, described in the testing handout. Specifically, specification tests must satisfy the *staff-provided specification* (including details of format and layout); in other words, they must be valid tests for any other student's implementation for this assignment. By contrast, implementation tests are unit tests for methods from the unique specification and implementation you designed in problems 2 and 3.

a. **Implementation Tests:** Write unit tests for the methods of your classes in one or more JUnit test classes, just like you saw in Homework 3. Create one test class per public ADT class, and be sure to write tests for every public method. Add your test classes to `test/ImplementationTests.java`, mimicking the structure of the provided `SpecificationTests.java` from Homework 3 and 4.
b. **Specification Tests:** Because we didn't specify any of the class or method names you need to write for this assignment, specification tests cannot test your interface directly. Instead, you will construct specification test cases in the format specified in the Test Script File Format section. Each test case should consist of a "test" file containing a series of commands, and an "expected" file containing the output that the commands should produce. The file names should have the same base name but end with `.test` and `.expected`, respectively. For example, you may have a command file named `testAdd.test` with expected output in `testAdd.expected`. These files must be in the `test` directory alongside `ScriptFileTests.java`. They should have descriptive file names and comments that explain what case is being tested, and just like methods in unit tests, each test file should only test one distinct condition.

   When you run the provided JUnit test ScriptFileTests (which is also run by the JUnit test suite SpecificationTests), it will find all of the test files in that directory and run them, saving their output as "testAdd.actual" (for example) in the `bin/hw4` directory of your project. It then compares the actual file to the expected file and fails if they are different. (Hint for Eclipse users: the `bin` directory is not visible from inside Eclipse, so you'll need to access it directly through the file system.)

   We *may* run each student's `SpecificationTests` against each other student's assignment, and how you fare will determine a small part of your grade. (It would not make sense for us to do this cross-checking with `ImplementationTests`. If you don't know why, please review the difference between the two varieties of tests until you understand this point.)

c. **Documentation:** Include with your test cases one short paragraph of documentation explaining your testing strategy, and which (if any) testing heuristics you used. Place this writeup in `hw4/answers/problem3.txt`.

It is important to check every aspect of the output files to make sure that your tests are running correctly, **including whitespace**. We use an automated script to compare the expected and actual output of your files and report if they don't match.

For turnin, you are required to commit any new JUnit test classes you add, your updated version of `ImplementationTests.java` listing these new classes, and the `.test` and`.expected` files you write. You should not modify `SpecificationTests.java`.

## Problem 4: Implement Graph [25 points]

There are many ways to represent a graph. Here are a few:

- As a collection of edges.
- As an adjacency list, in which each node is associated with a list of its outgoing edges.
- As an adjacency matrix, which explicitly represents, for every pair ?A,B? of edges, whether there is a link from A to B, and how many.

a. In one brief paragraph, describe your representation. For at least the three representations above and your representation (if it is not one of the ones above), explain an advantage of that representation in a sentence or two. Given these advantages, briefly explain why you chose the representation you did. You should place this discussion in a file called `hw4/answers/problem4.txt`.

b. Place a proper **abstraction function and representation invariant** for your Graph data type (and any other ADTs you create) in your source code. Also implement a private `checkRep()` method, which will help in finding errors in your implementation.

c. Provide an implementation of your graph data type. We ask that you strive first for a good design before worrying about performance right now. Eventually, however, your path-finding application will create and operate on very large sized graphs, so the scalability of your Graph implementation will be important. Since the path-finding algorithm must frequently look up the children for a given node, this operation should be performed quickly even for large graph sizes (aim for constant time here). Your graph building operations should also be reasonably efficient. As your implementation will likely use classes in the Java Collections Framework, you should understand the computational complexity of classes such as **`HashMap`** and **`ArrayList`**.

d. Be sure to call your `checkRep()` where appropriate.

e. Once you've finished your implementation, you should think about whether or not new tests are needed in addition to those you wrote before you started coding. If so, you should add these to your test suite. You should **append to the end of your test strategy writeup** in Problem 3 a description of any new tests you added, or why you feel that your original tests alone are sufficient.

## Problem 5: Write a Test Driver for Graph [5 points]

The staff-supplied testing driver HW4TestDriver should read input from standard input or a specified file using the format described under the Test Script File Format section and print its results to standard output. We have provided a skeleton implementation that takes care of parsing the input file format. Complete this file by adding code where specified by comments to perform the appropriate operations on your ADT. Please be sure to use the `PrintWriter` stored in the `output` field in the tester to print the desired output.

### Reflection [1 point]

Please answer the following questions in a file named `reflection.txt` in your `answers/` directory. Answer briefly, but in enough detail to help you improve your own practice via introspection and to enable the course staff to improve CSE 331 in the future.

a. In retrospect, what could you have done better to reduce the time you spent solving this assignment?
b. What could the CSE 331 staff have done better to improve your learning experience in this assignment?
c. What do you know now that you wish you had known before beginning the assignment?

### Collaboration [1 point]

Please answer the following questions in a file named `collaboration.txt` in your `answers/` directory.

The standard collaboration policy applies to this assignment, with the exceptions described in the Introduction.

State whether or not you collaborated with other students. If you did collaborate with other students, state their names and a brief description of how you collaborated.

### Time Spent

Tell us how long you spent on this homework via this catalyst survey: https://catalyst.uw.edu/webq/survey/mernst/198074.

## Returnin [25 points]

After you receive your corrected assignment back from your TA, you will need to resubmit a corrected version of the code. You will have until Thursday, May 16 at 11pm to returnin your code. The returnin script will be enabled a week earlier, at 11pm Thursday, May 9.

You will only receive credit if you pass all the tests, and you have a fixed number of trials without penalty. See the returnin331 documentation for details.

## Test script file format

Because you and your classmates will have different specifications for the classes in this assignment, it is important that there is a standardized interface to use and test your code. To that end, we specify a text-based scripting format used to write instructions that will be executed by your graph.

The testing script is a simple text file with one command listed per line. Each line consists of words separated by white space. The first word on each line is a command name. The remaining words are arguments to that command. To simplify parsing the file, graph names and node and edge data may contain only alphanumeric ASCII characters.

There are example programs in the `hw4/test` directory.

The following is a description of the valid commands. Each command has an associated output which should be printed to standard output (typically, the console) when the command is executed. Lines that have a hash (#) as their first character are considered comment lines and are only echoed to the output when running the test script. Lines that are blank should cause a blank line to be printed to the output. These commands were chosen for ease of testing and are *not* meant to suggest what methods you should include in your graph specifications or how you should implement them. For example, it is unlikely to make sense for your graph ADT to store a name for the graph.

**CreateGraph *graphName***

Creates a new graph named *graphName*. The graph is initially empty (has no nodes and no edges). The command's output is:

```
created graph graphName
```

If the graph already exists, the output of this command is not defined.

Note that graph names are used purely in the test script; it is unlikely to make sense for your graph ADT to store a name.

**AddNode *graphName nodeData***

Adds a node represented by the string *nodeData* to the graph named *graphName*. The command's output is:

```
added node nodeData to graphName
```

If a node with this data is already in the graph, the output of this command is not defined.

**AddEdge *graphName parentNode childNode edgeLabel***

Creates an edge from *parentNode* to *childNode* with label *edgeLabel* in the graph named *graphName*. The command's output is:

```
added edge edgeLabel from parentNode to childNode in graphName
```

If either of the nodes does not exist in the graph, the output of this command is not defined. If an identical edge (same parent, child, and label) already exists, the output of this command is not defined either, as it is left to your discretion whether to allow identical edges in your implementation.

**ListNodes *graphName***

This command has no effect on the graph. Its output starts with:

```
graphName contains:
```

and is followed, on the same line, by a space-separated list of the node data contained in each node of the graph. The nodes should appear in alphabetical order. There is a single space between the colon and the first node name, but no space if there are no nodes.

**ListChildren *graphName parentNode***

This command has no effect on the graph. Its output starts with:

```
the children of parentNode in graphName are:
```

and is followed, on the same line, by a space-separated list of entries of the form *node(edgeLabel)*, where *node* is a node in *graphName* to which there is an edge from *parentNode* and *edgeLabel* is the label on that edge. If there are multiple edges between two nodes, there should be a separate *node(edgeLabel)* entry for each edge. The nodes should appear in alphabetical order by node name and secondarily by edge label, e.g. `firstNode(someEdge) secondNode(edgeA) secondNode(edgeB) secondNode(edgeC) thirdNode(anotherEdge)`. If some node N has a reflexive edge `?N,N?`, it is it's own child and should be listed if you call listChildren on the node N. There is a single space between the colon and the first node name, but no space if there are no children.

## Sample input and output

We have provided example input and output files in your `hw4/test` directory. The behavior of the test driver on malformed input files is not defined; you may assume the input files are well-formed.

In addition, HW4TestDriver has a main method that allows it to be run directly through Eclipse or from the command line. You can enter commands at the console, and it will echo the results.

To run it from the command line execute the appropriate version below, adjusting pathnames as necessary.

### Linux or Mac:

```
# Compile your program
cd ~/workspace331/cse331/src/hw4
ant build
# Run HW4TestDriver
cd ~/workspace331/cse331/bin
java hw4.test.HW4TestDriver
```

### Windows:

```
# Compile your program
Z:
cd Z:\workspace331\cse331\src\hw4
ant build
# Run HW4TestDriver
cd Z:\workspace331\cse331\bin
java hw4.test.HW4TestDriver
```

To stop, press the key combination control-d if you are running it from the command-line, or press the red square button above the console if you are running it from Eclipse.

## Hints

### Writing Specifications

To give you some sense of the kinds of issues you should be considering in your design, here are some questions you might want to consider. These don't in general have simple answers. You'll need to exercise judgment, and think carefully about how decisions you make interfere with each other.

- Will the graph be mutable or immutable?
- Will the graph be implemented as a single class, or will there be a Java interface for the Graph specification and a separate class for the implementation?
- Will edges be objects in their own right? Will they be visible to a client of the abstract type?
- Will nodes be objects in their own right? Will they be visible to a client of the abstract type?
- When will the user specify the nodes and/or edges in the graph? (In the constructor? With an insertion method? Both? Can the user add multiple nodes and/or edges at once?)
- What kind of iterators will the type provide?
- Will the type provide any views, like the set view returned by the `entrySet()` method of `java.util.Map`?
- Will the type implement any standard Java collection interfaces?
- Will the type use any standard Java collections in its implementation?

In choosing what operations/methods to include, strive to include enough that the ADT will be convenient and useful for a client, but avoid the temptation to write an "everything but the kitchen sink" API. Generally speaking, it is better to design a minimal than a maximal API. In the real world, you can always add methods later. However, you can never remove them from a published API, and such methods may over-constrain the implementation in the future.

Make good use of the course staff. If you have concrete questions, then take your specification to office hours to get some feedback on your design and style. This is likely to save you a lot of time!

## Working Incrementally

Although it is generally a bad idea to start coding before you have thought deeply, it often makes sense to work incrementally, interleaving design and coding. Once you have a sketch of your specification, you may want to write some experimental code. This should give you some concrete feedback on how easy it is to implement the methods you've specified. You may even want to start at the end, and write the code that uses your type, so that you can be confident that the methods you provide will be sufficient.

This strategy can backfire and degenerate into mindless hacking, leaving you with a pile of low-quality code and an incoherent specification. To avoid that, bear three things in mind:

a. You must be willing to start again: experimental code isn't experimental if you're not prepared to throw it away.
b. Whenever you start coding, you must have a firm idea of what you're trying to implement. There's no point starting to code to a specification that is vague and missing crucial details. That doesn't mean that your specification must be complete and polished, but it does mean that you shouldn't start coding a method until at least you have its own specification written.
c. You *must* write down the specification of a method and not just imagine it; it's too easy to delude yourself. Try to write it on paper and mull it over before you start any coding. It's tempting to sit in front of an editor, write some specification as comments, and then start coding around them, but this tends not to be nearly so effective.

## Designing Tests

It can be difficult to come up with a good test suite. You would like to test a variety of "interesting" graphs, but what are interesting graphs? One possible approach is a "0, 1, 2" case analysis: test scripts with 0, 1, and 2 graphs are interesting; graphs with 0, 1, and 2 nodes and 0, 1, and 2 edges are interesting. For each method, 0, 1, and 2 parameters and 0, 1, and 2 results are interesting; for example: AddEdge on nodes that currently have 0, 1, and 2 children; ListChildren on nodes with 0, 1, and 2 children; etc. This approach, while certainly not required, can give a good way to structure your tests to cover many important cases without having too much redundancy.

## Abstraction function, representation invariant, and checkRep

Include an abstraction function, representation invariant, and private `checkRep()` method in all new classes you create that represent an ADT. If a class does not represent an ADT, place a comment that explicitly says so where the AF and RI would normally go. (For example, classes that contain only static methods and are never constructed usually do not represent an ADT - though you are unlikely to write any such classes for HW4.) Please come to office hours if you feel unsure about what counts as an ADT and what doesn't.

Be conscious of how certain operations in `checkRep()`, particularly iterating over a large dataset, may affect the "big-O" runtime of your methods. If your program suffers performance problems in Homework 4 or 5, `checkRep()` is a good place to start looking for problems.

It is hard to balance the utility of the `checkRep()` method with how expensive it may be to run. A good approach is to call `checkRep()` as much as possible (generally at the beginning and end of every method), but to **disable the `checkRep()` when you turn in your code so that our tests don't timeout.** A good way to do this is to have a static final constant variable that is checked in your `checkRep()` such that it only runs when the constant variable is set.

## A warning about equals and hashCode

You may find it useful to define a class or classes that implement method equals. If you do that, *be sure* to also provide a consistent definition of method hashCode, otherwise your objects may behave strangely if used in containers like HashMaps. There is a good discussion of the issues involved in *Effective Java* (item 9 in the 2nd edition).

## A warning about using generics

You are permitted **but discouraged** from implementing generic classes on this assignment (that will come later). A generic class is one defined as something like

```
public class Graph<N,E> {
   ...
}
```

where a client could then construct a `Graph<String,String>`, `Graph<City,Road>`, etc.

If you choose to write a generic class, be aware that Eclipse's built-in compiler sometimes handles generics differently from `javac`, the standard command-line compiler. This means code that compiles in Eclipse may *not* compile when we run our grading scripts, leaving us unable to test your code and significantly impacting your grade. **You MUST periodically make sure your code compiles (builds) with `javac`** by running `ant build` from the `hw4` directory. You can do this in one of two ways:

**Through Eclipse:** Right-click `build.xml` under `hw4` and click `Run As >> Ant Build...` (note the ellipsis). In the window that appears, select `build [from import ../common.xml]` and hit `Run`.
- **At the command line:** Run `ant build` from the `hw4` directory.

If Ant reports that the build succeeded, all is well: your code compiles with `javac`.

Hint: after encountering build errors in Ant, you may find that classes which previously compiled are now reporting "[some class] cannot be resolved" errors in Eclipse. You can fix these errors by doing a clean build: go to `Project >> Properties >> Clean...`, select your `cse331` project, and hit `OK`.

This warning is only directed at students writing their own generic classes. Simply using, say, a `List<String>` or `Comparator<Foo>` (as you have been doing all along) should be fine.

## What to Turn In

You should add and commit the following files to SVN:

- `hw4/answers/problem1.txt`
- `hw4/answers/problem2.txt`
- `hw4/answers/problem3.txt`
- `hw4/answers/problem4.txt`
- `hw4/answers/reflection.txt`
- `hw4/answers/collaboration.txt`
- `hw4/*.java` *[Java classes for your graph implementation]*
- `hw4/test/*.test`
- `hw4/test/*.expected`
- `hw4/test/*Test.java` *[Other JUnit test classes you create]*

Additionally, make sure to commit any updates to `hw4/test/ImplementationTests` and `hw4/test/HW4TestDriver.java`.

Remember to run the validate tool to verify that you have submitted all files and that your code compiles and runs correctly when compiled with `javac` on attu (which sometimes behaves differently from the Eclipse compiler).

## Errata

None yet.

## Q & A

This section will list clarifications and answers to common questions about assignments. We'll try to keep it as up-to-date as possible, so this should be the first place to look (after carefully rereading the assignment handout and the specifications) when you have a problem.

For problems or questions regarding this page, contact: cse331-staff@cs.washington.edu.