Exam 1

- Out at end of class today, **due Tuesday at beginning of class**
  - I will answer questions about PS3 (including anything in PS3 comments) while the exam is out (including Monday’s office hours)

- **Work alone**
- **Open resources**: use any resources you want, but cite anything that is not part of the course materials

PS3: Available for pick-up by Monday.

---

**Reality Check**

- Writing abstraction functions, rep invariants, testing code thoroughly, reasoning about correctness, etc. for a big program is a ridiculous amount of work!
- Does anyone really do this?
  - Yes (and a lot more), but usually only when its really important to get things right:
  - Cost per line of code:
    - Small, unimportant projects: $1-5/line
    - WindowsNT: about $100/line

---

**StringGraph, IntegerGraph, etc.**

**StringGraph**

```java
public class StringGraph

OVERVIEW: A StringGraph is a directed graph where V is a set of Strings, and E is a set of edges. Each edge is a pair (v1, v2), representing an edge from v1 to v2 in G. A typical StringGraph is < {v1, v2, ..., vn} , { (v_a1, v_b1), (v_a2, v_b2), ... } > where each ai and bi is in [1, n].

public void addNode(String s) throws DuplicateException;
public void addEdge(String s, String t) throws NoNodeException, DuplicateException;
... public Set<String> getAdjacent();
```

**IntegerGraph**

```java
public class IntegerGraph

OVERVIEW: A IntegerGraph is a directed graph where V is a set of ints, and E is a set of edges. Each edge is a pair (v1, v2), representing an edge from v1 to v2 in G. A typical IntegerGraph is < {v1, v2, ..., vn} , { (v_a1, v_b1), (v_a2, v_b2), ... } > where each ai and bi is in [1, n].

public void addNode(int s) throws DuplicateException;
public void addEdge(int s, int t) throws NoNodeException, DuplicateException;
... public Set<Integer> getAdjacent();
```

**CityGraph**

```java
public class CityGraph

OVERVIEW: A CityGraph is a directed graph where V is a set of City objects, and E is a set of edges. Each edge is a pair (v1, v2), representing an edge from v1 to v2 in G. A typical CityGraph is < {v1, v2, ..., vn} , { (v_a1, v_b1), (v_a2, v_b2), ... } > where each ai and bi is in [1, n].

public void addNode(City s) throws DuplicateException;
public void addEdge(City s, City t) throws NoNodeException, DuplicateException;
... public Set<City> getAdjacent();
```

**URLGraph**

```java
public class URLGraph

OVERVIEW: A URLGraph is a directed graph where V is a set of URL objects, and E is a set of edges. Each edge is a pair (v1, v2), representing an edge from v1 to v2 in G. A typical URLGraph is < {v1, v2, ..., vn} , { (v_a1, v_b1), (v_a2, v_b2), ... } > where each ai and bi is in [1, n].

public void addNode(URL s) throws DuplicateException;
public void addEdge(URL s, URL t) throws NoNodeException, DuplicateException;
... public Set<URL> getAdjacent();
```

---

**Type Parameters**

- We want to have Graphs with many different types of nodes
- How should we declare the **Graph** methods?

```java
public void addNode(T s)
public void addEdge(T s, T t)
public Set<T> getAdjacent()
```

We don’t want just one Graph datatype. We want different Graphs for different node types.

---

**Generic Datatype**

```java
public class Graph<T> {
... public void addNode(T s) ...
... public void addEdge(T s, T t) ...
... public Set<T> getAdjacent() ...
}
```

Note: Java did not support generic datatypes until version 1.5 (this is why the book doesn’t use them)
Specifying the Generic Graph

```java
public class Graph<T>

OVERVIEW: A Graph is a directed graph where V is a set of T objects, and E is a set of edges. Each edge is a pair (v1, v2), representing an edge from v1 to v2 in G. A typical Graph is < {v1, v2, ..., vn} , { (v_a1, v_b1), (v_a2, v_b2), ... } > where each ai and bi is in [1, n].

public void addNode(T s) throws DuplicateException
// MODIFIES: this
// EFFECTS: If s is the name of a node in this, throws DuplicateException. Otherwise, adds s to the nodes in this, with no adjacent nodes:
G_post = < V_pre U {s} , E_pre >
```

Implementing a Generic Graph

```java
class StringGraph

public void addNode(String s)
throws DuplicateException
MODIFIES: this
EFFECTS: If s is the name of a node in this, throws DuplicateException. Otherwise, adds s to the nodes in this, with no adjacent nodes:
G_post = < V_pre U {s} , E_pre >
```

NodeRecord (for StringGraph)

```java
class NodeRecord implements Comparable<NodeRecord> {
String key;
Set<String> values;

NodeRecord(String p_key) {
key = p_key;
values = new TreeSet<String>();
}

public int compareTo(NodeRecord n) {
// This assertion should be guaranteed by the rep invariant
assert !key.equals(n.key) || (n.values == values);
return key.compareTo(n.key);
}

public boolean matches(String p_key) {
return key.equals(p_key);
}

public void addEdge(String p_edge) throws DuplicateException {
if (values.contains(p_edge))
throw new DuplicateException("<" + key + ", " + p_edge + ">");
values.add(p_edge);
}
```

Generic NodeRecord

```java
class NodeRecord implements Comparable<NodeRecord> {
T key;
Set<T> values;

NodeRecord(T p_key) {
key = p_key;
values = new TreeSet<T>();
}

public int compareTo(NodeRecord n) {
// This assertion should be guaranteed by the rep invariant
assert !key.equals(n.key) || (n.values == values);
return key.compareTo(n.key);
}

public boolean matches(T p_key) {
return key.equals(p_key);
}

public void addEdge(T p_edge) throws DuplicateException {
if (values.contains(p_edge))
throw new DuplicateException("<" + key + ", " + p_edge + ">");
values.add(p_edge);
}
```

Abstraction Function:
Nodes = { el.key| el is an element in rep }
Edges =  { ( el.key, value ) | el is an element in rep and value is an element in el.values}

Rep Invariant:
rep != null
elements of rep are not null
for all elements e in rep:
for all elements f in e.values: f = el.key for some el element el in rep

Why do we need to copy values?
```java
public String toString() {
    String res = "<";
    boolean firstone = true;
    for (NodeRecord r : rep) {
        if (firstone) { firstone = false; } else { res += ","; }
        res += r.key;
    }
    res += "; { ";
    firstone = true;
    for (NodeRecord r : rep) {
        for (T e : r.values) {
            if (firstone) { firstone = false; } else { res += ","; }
            res += "(" + r.key + ", " + e + ")";
        }
    }
    res += "} >";
    return res;
}
```

Abstraction Function:
Nodes = \{ el.key | el is an element in rep \}
Edges = \{ (el.key, value) | el is an element in rep and value is an element in el.values \}

Charge

Exam 1 is due Tuesday
Next week: Subtyping
PS4: Designing with Data Abstractions and Subtyping