CURTIN UNIVERSITY (CRICOS number: 00301J)

Department of Computing, Faculty of Engineering and Science

Data Structures and Algorithms (COMP1002)

PRACTICAL 1 - IMPLEMENTATION

AIMS

- To being the process of implementing the program planned in practical 0.
- Learn about implementing abstract data types.

BEFORE THE PRACTICAL:

- Read the specifications from practical 0.
- Review your solution to practical 0.

ACTIVITY 1: IMPLEMENTATION

- Develop the three classes (Network, Vertex, Edge) and implement them in Java.
- This will include implementing container classes for Vertex and Edge

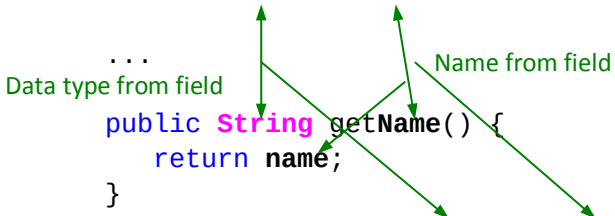
Implementing the Classes:

- It's a good idea to design and implement your classes in the following order:
  - Write down the member fields (variables)
  - Write down getters/setters (accessors/mutators) for those member fields
  - Write down the constructors, which should initialise *all* of the member fields
  - Write down other methods
- This way, it becomes obvious that you can use the setters as appropriate in the constructors, simplifying your code since the constructor can then let the setter do input validation for it (*i.e.,* then you code input validation *only once* – in the setter).
- However, continue to *organise* your classes in code as follows:
  - Member fields
  - Constructors
  - Getters/Setters (keep the getXYZ() with the setXYZ())
  - Other methods
- Getters and setters should become no-brainers to you. In fact, here is a recipe you can follow for almost any getter/setter:

```
public class Vertex {
```

```
private char deviceType;
private String name;

...
public String getName() {
   return name;
}
public void setName(String inName) {
   if ( (inName == null) || (inName.equals("")) )     // Validation
      throw new IllegalArgumentException("Name must not be blank");
   name = inName; // Assignment
}
```



NOTE: We do **NOT** want to return *copies* in the getter for this system (which is important for objects).

- Constructors should also become quite straightforward. Usually you consider three constructors immediately:
  - Default constructor (no parameters)
  - 'Alternate' (standard) constructor (one parameter for each member field)
  - Copy constructor (a single parameter: an object of the same type as the current class which is to be duplicated).
- Still, you need to decide whether you actually need them all (particularly the default constructor).
- The 'alternate' or 'standard' constructor (there's no good name for it!) takes the parameters and uses them to initialise the member fields. Often it's a case of simply assigning the member field to its associated parameter, but sometimes it's a little different (*e,g,.* you have an array as a member field: the parameter would be maxSize of the array, and you initialise the member array to be maxSize in capacity).
  - Also, you should have a close look at the standard constructor to see if you really should be passing in parameters for *all* member fields, or if some member fields are really internal-only and shouldn't be initialised by parameters.
- **COMPILE EARLY AND OFTEN** – this avoids errors piling up into a huge list.

Vertex Notes:

For Vertex, you will need to define a deviceType field that stores the device represented by the vertex. In practice this would be used to decide aspects like the reliability of the component and its cost. We'll ignore such real-world issues for now and also limit the types of devices. For this project we'll restrict all of our devices to be either 'RV325' or '2921/K9'. You will need to check that a vertex has a valid type, which in our case means one of these two strings.

One way to define this field is to make it a **char** and then define two constants that are the only valid values for that field. In Java, constants are marked as **static final** (*i.e.*, static=global and final="cannot be changed"), and are usually also public. For example:

```
public class Vertex
{
   public static final char DEVICETYPE_RV325 = 'R';
   public static final char DEVICETYPE_2921K9 = '2';

   private char deviceType;
   private String name;

   public Vertex(char inDeviceType, String inName) {
      if ( (inDeviceType != Vertex.DEVICETYPE_RV325) &&
            (inDeviceType != Vertex.DEVICETYPE_2921K9) )
         throw new IllegalArgumentException("Invalid device type provided");

      setName(inName);
      deviceType = inDeviceType;
   }
   ... and the other methods
}
```

inName must be a valid string but can be empty (*i.e.*, ""). Notice also that the setter setName() is used in the constructor since it will do input validation for us. On the other hand, since we don't have a setDeviceType() we must validate in the constructor. Don't forget to perform import validation on *everything*! Strings must be checked for null as well as a .equals() check against invalid values.

Edge Notes:

An edge connects two vertices, and thus must contain some way to reference these. There are a number of different ways that this referencing can be achieved, and the best way of doing this will require some thought.

Recall also that edges, like vertices, have a reliability which is a floating point number. Be aware of possible issues with floating point numbers.

Network Notes:

The network should contain all of the edges and vertices. In general, both will be read from a file, sorted and then not changed further during the running of the program. As such it is safe to use structures that aren't ideal for changes but are good at accessing data (such as arrays).

Note that the algorithm will need the Network class to pass requests on to both the Edge and Vertex classes that it contains. This means that you'll need to have accessors for both of these classes in Network as well, which will delegate the call to the appropriate object.

ACTIVITY 2: TESTING

It's usually a good idea to test your classes individually after you initially write them. Otherwise you end up building a full system from untested components, and hit dozens of bugs. This is called *unit testing*, (covered in OOPD and Intro to Software Engineering) and involves creating a main() that acts as a test harness to test each method of a given class for correctness. If you don't do it now, you'll end up doing it the hard way later on anyway – ***TEST EARLY AND OFTEN!!!***.

Appendix 1 (a separate file on Blackboard) contains a test harnesses for Vertex, although it may not work for your specific version if you've made different choices. Use this as a guide to write your own for Edge and Network. In particular, make sure that you test every method, and check for the 'edge' cases on the line between a valid situation and an invalid one.

SUBMISSION DELIVERABLE:

Your Vertex, Edge and Network classes are due at the beginning of your next tutorial. The classes developed here will also form the foundation for the whole system, which you will also submit when an extended version is completed. Please only submit the *.java files.

You do not need to submit your test harness if everything is fine. If there are doubts about some part of your program though, the test harness could be quite useful but it won't be available if you haven't submitted it. I will not allow you to bring code from your account during the testing since this adds uncertainty – we'll work with what you've submitted.

SUBMIT ELECTRONICALLY VIA BLACKBOARD, under the *Assessments* section.

If you finish early, use the rest of the practical to start the next worksheet, because that will be due later on.