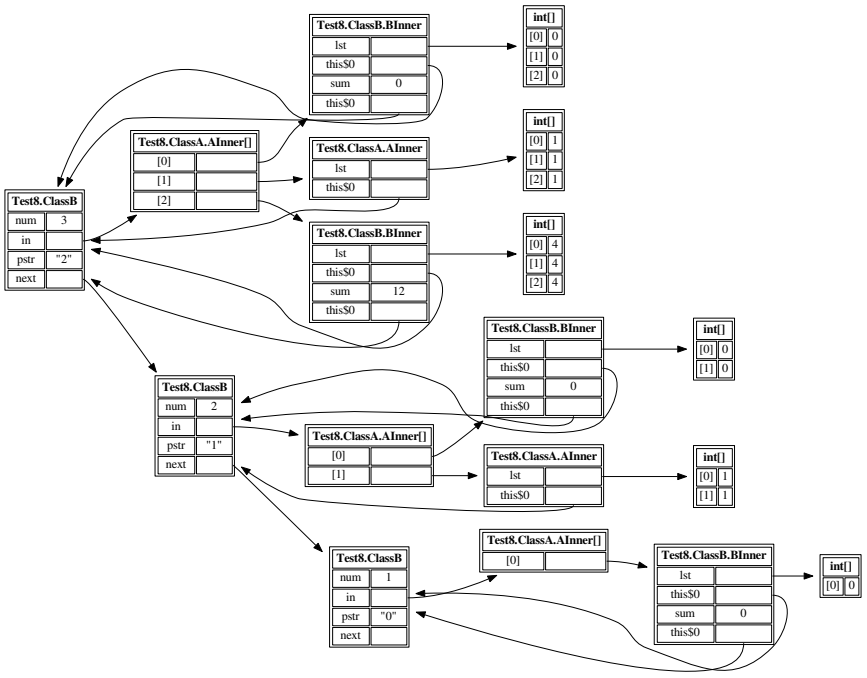
# Programming Assignment C
## Due on Wednesday, November 13, 2013 at 11:59 pm

The goal of this assignment is to be able to generate at runtime a diagram like the following that depicts a data structure.




The code will have three main components:

- `Formatter`: Formatter is an interface for how an object should be formatted. It has an inner class `GenObject` that can store any object (essentially a wrapper around `Object` with a boolean to indicate if the object should actually be a primitive type). It has another inner class `NamedObject` that pairs a `GenObject` with a `String` name.

  The purpose of this interface is to allow different objects to be rendered differently. Instantiations of `Formatter` can look at an object (`applies` method) and state whether they can deal with that type of object. For objects they deal with, they can state whether they would prefer to render it as a string or as a general structure (with fields and associated values). In either case, the `Formatter` can generate the associated string or fields.

- `GraphDrawer`: GraphDrawer is an interface for a general way to draw a graph in which the nodes are structures with a name and fields where each field has a name and either a string value or a pointer to another node.

  An implementation has been provided for you as `TxtDrawer.class`. Later in the assignment you will write this implementation.

- `DataDrawer`: DataDrawer is a class that uses objects of the above two types. In particular, it takes a `GraphDrawer` in its constructor and multiple `Formatters` can be added through the `addFormatter` method. Then, objects can be added with the `addObject` method. It handles finding the right `Formatter` for each object and traversing any objects they point to. An implementation has been provided in `DataDrawer.class` and later in the assignment you will write this yourself.

**NOTE:** This assignment provides `.class` files that you will later implement. You **may not** decompile these class files. Doing so is considered cheating. They are provided so that you can more easily implement the assignment (without having to write it all at once).

**part a.** [15 points]
Implement a class `FieldFormat` that implements `Formatter`. It should apply to all objects. If the object is primitive, it should prefer to output it as a string. If the object is not, it should list all of its fields. Recall that all objects have a `getClass` method that returns an object of type `Class` that describes its type. You can read how to use this at `http://docs.oracle.com/javase/9/docs/api/java/lang/Class.html`. You will need to return a `List` from `getDeclaredFields`. `List` is an interface. I would suggest using `java.util.LinkedList` as an implementation. You can find information about these at `http://docs.oracle.com/javase/9/docs/api/`. You will need to import at least `java.lang.reflect.*`

When accessing a field, you will have to deal with it possibly being private (and therefore throwing an `IllegalAccessException`). Just ignore private fields for now (do not add them).

Use the provided `TxtDrawer` class (which implements `Formatter`) and verify that you get the following output for the provided `Test1` main function:

```
node 0: Test1.A
        i: 3
        d: 3.5
```

**part b.** [5 points]
To access private members, you will have to turn off security checking. To do this, look at the `setAccessible` method of the class `Field` (which you can get to from the above URL). Changing this will throw exceptions. Just catch them and ignore them. You should now be able to run Test2 (with the same output as above).

If you have done things correctly, you can now run Test3 with the output

```
node 0: Test3.A
        i: 3
        d: 3.5
        next: -> node 1

node 1: Test3.A
        i: -1
        d: -0.5
        next:
```

You should also have it ignore static fields (or the string examples below will have lots of extra members).

**part c.** [10 points]
You must also access superclass members. Implement this and check that for Test4, you obtain the output

```
node 0: Test4.C
        i: 3
        d: 3.5
        next: -> node 1
        e: 100.75
        f: 103.75

node 1: Test4.B
        i: -1
        d: -0.5
        next:
        e: -3.14
```

**part d.** [30 points]
Until this point you have been using `DataDrawer.class` which is a compiled version of code I wrote. Now you need to write this code. Rename the file `DataDrawer-stub.java` to `DataDrawer.java` and implement the code therein. You will probably want to use a hash table (java.util.Hashtable or java.util.Hashmap)

You should now be able to run the examples above, using your own `DataDrawer` code. You should also be able to run `Test5` and get the following output (using your own `DataDrawer`).

```
node 0: Test5.B
        i: 3
        d: 3.5
        next: -> node 1
        e: 4.5

node 1: Test5.A
        i: -1
        d: -0.5
        next:

node 2: Test5.B
        i: 0
        d: 0.5
        next: -> node 1
        e: -100.3
```

**part e.** [10 points]
If you run `Test6` you will probably get something like

```
node 0: Test6.A[]
```

because the `FieldFormat` class does not know how to deal with arrays. Write a new formatter called `ArrayFormat` that formats arrays by placing each element in its own "field." This class should *only* handle arrays (`applies` reports `false` on other objects). You may find it difficult to access the elements of the array (because the code doesn't know it is an array). You can use the static methods of the `Array` class (see `https://docs.oracle.com/javase/9/docs/api/java/lang/reflect/Array.html`) Your output (uncommenting the line in `Test6.java` that adds this formatter) should be

```
node 0: Test6.A[]
        [0]: -> node 1
        [1]: -> node 2
        [2]: -> node 3

node 1: Test6.A
        i: 0
        d: 0.5
        next: -> node 1

node 2: Test6.A
        i: 1
        d: 1.5
        next: -> node 1

node 3: Test6.A
        i: 2
        d: 2.5
        next: -> node 1
```

You can also now run `Test7`. Note that you can see the internal structure of Java `String`s. However, often this is distracting. Therefore, the supplied `StringFormat` class can hide some or all of it. Try uncommenting one or the other of the commented lines in `Test7.java`.

**part f.** [25 points]
Finally it is time to implement `TxtDrawer`. As a hint, it should have a nested class that implements `GraphDrawer.Node`. You cannot change the input type for method `addPtrField`, but you can cast the `Node` (which is an interface type) to your own type. A stub is provided as `TxtDrawer-stub.java`. This shows you how to open a file. You will need to keep track of all of the nodes, as you must wait until `draw` is called to output the result.

You should now be able to run the examples above with your own `TxtDrawer` implementation.

**part g.** [5 points]
`Test8` is provided as a last large example. It is a little messy in the given output. Try using the provided `DotDrawer` instead. You can view the output as `java Test8 | dot -Tpng | display -` on the lab machines. `DotDrawer` outputs a text file that can be read by the graph-layout package `dot` that can produce an image of a graph. The graph at the beginning of this assignment is the output of `Test8`

Note that there is a strange field called `this$0`. Can you explain why? Can you explain why some records have two fields with this name? Include a file `partg.txt` with your answer (it should be brief).

**Submitting:**
You should submit the following files: FieldFormat.java, DataDrawer.java, ArrayFormat.java, TxtDrawer.java, and partg.txt.