General Notes on Java

This semester all projects are to be written in Java. The version on the detective cluster is 1.4 and can be downloaded at:

`http://java.sun.com/j2se/1.4/download.html`

The online version of the documentation is at

`http://java.sun.com/j2se/1.4/docs/api/index.html`

We highly recommend that you download the Java sdk and do most of your work from home-if nothing else this will lighten the load on the now overworked DC machines, which can become very slow as lower-level project due dates approach. However, you should be aware that this approach has inherent risks.

As in most CMSC classes, your projects will be graded according to how they execute on the detective cluster machines (dc.umd.edu). In fact, you will not be able to submit your CMSC 420 projects for grading *unless* they compile and execute the primary data correctly.

Similarly, you should make regular backups of your work to ensure that your grade will not suffer from any catastrophic failure of your home system.

While there should not be any portability issues long as you develop your code using the correct Java version, it's still a good idea to download your code regularly to be sure that a copy of your project exists on the cluster and works. In fact, you should make regular verification that your program executes correctly on the target platform as a part of your design and development process.

We'll eventually be working with graphics, and if you use ssh to access the detective cluster from home, you will have to figure out how to set up an X-server to support drawing on your own machine. The difficulty of this task will vary depending on your computer's configuration. So, even if you use JAVA on your local machine, and don't have to worry about messing around with an X-server, you should visit an on-campus machine to test your drawing functions.

IDEs

We **strongly** encourage you to use an IDE to develop your project code. Although you could develop this project using only emacs and a java compiler and virtual machine, it is in your best interest to use an integrated development environment (IDE). An IDE allows you to write, compile, test, debug, and run your program without having to go to the command line (or a shell in emacs). A good IDE is one that helps you find compilation errors and allows you to debug your program by stepping through it line-by-line while displaying a print out of all local variables.

Many java IDEs are available. Try out a few and find out that works for you. Some potential IDEs include but are not limited to:

- Eclipse [http://www.eclipse.org/]

- JCreator [http://www.jcreator.com]
- Dr. Java [http://drjava.sourceforge.net/]
- jbuilder [http://www.borland.com/jbuilder/] (free but registration required)
- NetBeans [http://wwws.sun.com/software/sundev/jde/index.html]
- SunOne [www.sun.com/sunone/] (Community Edition is a free download from Sun)

Do a Google search to find the URLs to download these IDEs or look for them on the WAM machines (I have no idea which of these are installed on the UMD networks I merely noted the popular IDEs that I have heard of). If you find another IDE which you like, post it to the newsgroup to earn class participation points and allow others to share in your wisdom at the same time.

While you are permitted to use any JAVA drawing facility you are comfortable with, a simple drawing package is available on the class web page. It is this package that will be most readily supported by the TA's should any problems arise. The package 'Canvas.java' provides a simple class which allows drawing of circles , squares, lines, captions, and other simple primitives in a java jframe. While this isn't being used in part 1, it will show up in the not to distant future so you may want to take a peak at it. A drawing package appropriate for the project can be downloaded from the class webpage.

Pass by Reference... but not really

Every semester a new group of students gets caught up by the same thing in Java. They start out hearing "Java is always pass by reference" and they do silly looking things like the following:

```
void foo(String t)
  {
  t = new String("World");
  }

String s = new String("Hello");
foo(s);

System.out.print(s); //prints "Hello".  Why didn't it change?
```

In true pass by reference C++ this would have worked. But what is happening is not really pass by reference, *it is pass by value*, except what is being passed is a pointer. If you were to transfer the above to C++ it would look like:

```
void foo(String *t)
  {
  t = new String("World");
  }

String *s = new String("Hello");
foo(s);

cout<<*s<<endl; //prints "Hello".  Hopefully obvious why
```

You can see in the second example that t is only a local copy of s. If you alter the value t is pointing at then s will see the change. However, if you point t at something else s will never know. In this example there is actually no way for foo to change s, since java Strings are *immutable* after creation. An error less obvious than the above is:

```
void foo(String t)
  {
  t = t+"World";
  }
```

This looks like concatenation, not reallocation, but that '+' operator actually allocates a new String. The above is actually just a shortcut in java for:

```
void foo(String t)
  {
  StringBuffer temp = new StringBuffer();
  temp.append(t);
  temp.append("World");
  t = temp.toString();
  }
```

It's important to realize what's going on in the background! Of course in the above example, foo still doesn't change t, but what you could do instead is:

```
void foo(StringBuffer t)
  {
  t.append("World");
  }
```

This time, since `t` always points to the same location, the original value really is modified. In java ,"pass by reference" as C++ programmers tend to think of it always requires some kind of wrapper. In the last example, StringBuffer is a wrapper for a dynamically sized character array. There is a quick and dirty hack to get a similar effect without building and entire class wrapper, pass a 1 element array instead:

```
void foo(String[] t)
  {
  t[0] = new String("World");
  }

String s[]=new String[1];
s[0] = new String("Hello");
foo(s); //s[0]= "World"
```

This works for a similar reason. t points to the same array in memory that s does. When an element of the array is updated by t, s will see the change as well. This ends my FYI on pass by reference, try not to get caught up by this common error :)

Comparators

JAVA has two basic tools for doing comparison: the `compareTo()` method of Comparable objects; and, the `compare()` method of the Comparator class. We will try to explain this with a quick example. Suppose you wanted to sort a collection of strings in alphabetical (ignoring case for the moment) order. One might use a TreeSet to do this:

```
SortedSet sorter = new TreeSet();
sorter.add("hello");
sorter.add("world");
```

```
sorter.add("cat");
sorter.add("dog");

Iterator i = sorter.iterator();
while(i.hasNext())
  System.out.print(i.next()+" ");//prints "cat dog hello world"
```

In the above example the string constants are automatically cast to String, which implements the `compareTo()` method (just like c/c++ strcmp). The sorted map assumes its elements are Comparable and uses `compareTo()` to sort them. Note that this is a unique case where `i.next()` can be used without a cast, since in JAVA *ALL* objects implement the `toString()` method which is automatically called here. Back on topic, what if I wanted to sort the words backwards? One way is to wrap the strings in another class that implements compareTo() backwards like:

```
class MyString
  {
  String s;
  MyString(String s){this.s=s;}
  public int compareTo(Object other){return -1*(s.compareTo(other));}
  }
```

This would work, but it is a bit of a mess. We can't extend String directly, since Sun has made it a final class. In any case it will not be obvious what it is you are doing. However, there is a better way that you can use with all of JAVA's sorted classes (and which you will implement in your own sorted maps later this semester). You can use a comparator:

```
class ReverseCompare implements Comparator
  {
  public int compare(Object a, Object b)
    {
    return(-1*((Comparable)a).compareTo(b));
    }
  }

SortedSet sorter = new TreeSet(new ReverseCompare());
sorter.add("hello");
sorter.add("world");
sorter.add("cat");
sorter.add("dog");

Iterator i = sorter.iterator();
while(i.hasNext())
  System.out.print(i.next()+" ");//prints "world hello god cat "
```

Because the TreeSet was given a Comparator in its constructor, it will no longer assume its elements are Comparable, and will use the Comparator for sorting instead. Comparators allow you to easily have sets with different types of objects which aren't natively comparable with each other, or to impose your own sorting rules on other people's classes (like String) with ease. Cool stuff. You'll hopefully find this useful for doing your coordinate checking in Part 1, as well as implementing a priority queue for your adjacency lists.

Interfaces

One of the goals of object oriented programming is to create modular code that can easily be reused for various tasks. One good way to achieve this goal in JAVA is to use an interface.

An interface defines a set of public methods that a class must implement. The joy of using an interface is that if two different data structure implement the same interface, you should be able to switch freely between them without any problems!

Consider two classes: LinkedList and ArrayList (both reside in the java.io package). The ArrayList class stores values using a private array while the LinkedList uses dynamic memory management. Both classes, however, implement the List interface, which allows for the following:

```
List list;
ArrayList arrayList = new ArrayList();
LinkedList linkedList = new LinkedList();

list = arrayList;  // compiler casts these for us automatically
list = linkedList; // because both implement the List interface
```

The real joy of interfaces is that you don't need to worry about the implementation of the class - you only need to know what the interface is. Say that you wrote a large program that has a large dictionary of information and you decide to store this information using the java.util.LinkedList class. You could pass this variable around in your program as a LinkedList, but you realize that you don't really care about the fact that you are using a *linked* list, you only care about the fact that you are using a List. So you pass the dictionary around as a List object (since LinkedList implements the List interface).

```
public static void main(String[] args)
{
List masterDictionary = new LinkedList();

// ... now 400,000 lines of code that does something fun and profitable
}
```

After 5 months of hard work you ship your product but your customer comes back to you and complains that the program is too slow - the O(n) access time of your LinkedList isn't fast enough! Thankfully you had the foresight to pass the LinkedList around as a List object, so you can quickly replace the LinkedList object with any other class that implements the List interface. You just happen to have a SkipList class in your code base that implements the List interface. By changing a single line of code you are able to drastically improve performance without the headaches of having to search and replace for every instance of LinkedList.

Of course, this only works if you had the foresight to use interfaces. ;)

Kevin Conroy