QUEEN MARY, UNIVERSITY OF LONDON
DCS128 ALGORITHMS AND DATA STRUCTURES

Class Test Monday 27th March 2006 11.05-12.35

Please fill in your Examination Number here: **MODEL ANSWERS**

Student Number here: .....

All answers to this test should be written on the test sheet, but you may use spare paper for rough working. Answer as many of the questions as you can.

1) In the space below explain how the **selection sort** algorithm works (you do not need to give Java code for it, but your explanation should be a complete description). Also explain why the selection sort algorithm is categorised as $\mathbf{O(N^2)}$.

The selection sort algorithm is an algorithm for "sorting in place", that is it changes the position of items in an indexable collection destructively (that is, changes their positions in the actual object, rather than creating a new object with the items in the different positions). It leaves the items in some sort of ordering, with no item added or removed from the collection.

The algorithm is as follows. An index variable p is set up and at any point in the algorithm, the items in the collection before position p are in their final place where they will stay until the algorithm has finished, while the items at position p and after may have their position moved during the further execution of the algorithm. Initially p is set to 0, indicating that none of the collection has been sorted. Then, repeatedly, the lowest item in the ordering being used to sort is found in that part of the collection between that indexed by p up to the highest index. The item at that position is swapped in position with the item at position p. Then p can be increased by 1, since we know the item at position p is the lowest item from the rest of the collection and is in its correct sorted position. When p reaches one less than the size of the collection, we know the whole collection must be sorted.

When we are sorting a collection, the main operation which takes place is the comparison of two items in the collection to check their ordering in the ordering being used to sort it. If the number of items in the collection is N, it can be seen that at first, N-1 comparisons are made to find the lowest item in the ordering in the whole collection. Then N-2 comparisons are made to find the lowest item in all of the collection except the first position, then N-3 comparisons to find the lowest item in all but the first two positions. This goes on until 1 position is made in the last pass of the final two items in the collection.

So, it can be seen that the total number of comparisons made is (N-1)+(N-2)+(N-3)+… all the integers down to 1. We know this sums up to $N^2/2$ comparisons. In the categorisation of algorithms in the "big-O" notation, we consider the number of main operations needed, ignoring constant multiplying factors, and considering only the highest term. So this means selection sort is categorised, by this reasoning, as of $O(N^2)$.

2) The Java code library contains two generic classes, **`ArrayList<E>`** and **`LinkedList<E>`**. They appear to work quite similarly. In the space below, explain how they are related, and how they differ, and say why Java provides these two different classes.

These are both classes that hold an indexable mutable collection of objects. They are generic classes, so they take a type argument (hence the `<E>` annotation) which gives the type of object they hold a collection of. "Indexable" means the collection these classes represent has the concept of items in it occurring in a particular position in the collection, given by an integer ranging from 0 to one less than the size of the collection (total number of items in it). The collections are "mutable" in two ways, firstly the item stored in a particular position may be changed, secondly items may be inserted at a particular position or removed from a particular position, causing items following them to be moved one place up or down in position.

The common nature of `ArrayList<E>` and `LinkedList<E>` is fixed by both classes extending the interface type `List<E>`. This specifies all the operations necessary to give a mutable indexable collection type, and means `ArrayList<E>` and `LinkedList<E>` must necessarily provide those operations.

The two classes differ in the data structure they use to implement their operations.

The class `ArrayList<E>` uses an "array and count" data structure. This means that internally it has an array storing the items of the collection in the order they occur in the collection, but the array may be longer than the current size of the collection, so the count states how much of the array is currently used to represent the collection. An item is stored in the same position in the array as it is in the collection, so accessing an item at a particular position can be done in one step. The count is changed and items moved up or down positions in the array to implement those operations which add or remove items from the collection.

The class `LinkedList<E>` uses a "doubly-linked list" data structure. This consists of cell objects each of which has two links to further cell objects, which are set so that one link points to the next item in the list, and the other points to the previous item in the list. The order of items in the doubly-linked list is the same as the order of items in the collection it represents. This means that accessing an item at a particular position involves following down the links in the list until that position is reached. Although this is inefficient compared to the one-step access of an item at a particular position in `ArrayList<E>`, adding or removing an item at a particular position in `LinkedList<E>` involves linking in a new cell but doesn't require changing the items stored in other cells, so that is more efficient than adding or removing an item from an `ArrayList<E>` object.

Java provides both types in order to allow programmers to make a choice. In most cases, `ArrayList<E>` will be the type chosen to implement a mutable indexable list, but `LinkedList<E>` might be chosen if it is known that operations on the list objects will be mainly of the sort which `LinkedList<E>` implements more efficiently than `ArrayList<E>`.

3) a) Explain briefly how a **static method** in Java differs from an **object method**.

A static method is not called on an object, and operates only on values passed to it as arguments through its parameters. An object method must always be called attached to an object (except when it is called inside another object method, in which case if it is not attached to another object, it is assumed attached to the same object as that method call is attached to). An object method call has access not only to the values passed to it as arguments to its parameters, but also to the variables inside the object it is called on.

b) Explain briefly the two uses of the keyword **`this`** in Java.

One use is as a method call on the first line of a constructor. In this case it has the effect of calling the code in another constructor in the same class whose number and type of parameters match the arguments to the call of 'this' to set variables in the new object being constructed.

The other, and main use of the keyword 'this', is that in any object method the word 'this' means "the object this method is being called on".

c) Explain what is meant by **dynamic binding** when a method is called in Java.

In Java, a variable may be set to refer to an object of a class which extends the class the variable is declared to be of. In that case, given a variable which refers to an object, the "apparent type" of the object is the type the variable is declared to be of, while the "actual type" is the type of the constructor used to create the object. It is possible the actual type contains code for a method which "overrides" a method in the apparent type (that is, has same name and number and types of parameters). If code indicates a method being called on a variable, when the program is executed the code for that method call is the code of the actual type rather than the apparent type. This principle is known as "dynamic binding", because the actual type of a variable may only be known when the program is actually running.

d) Explain what is meant by the **natural ordering** of a class of objects in Java.

The "natural ordering" of a class of objects is the ordering which is given by the compareTo object method which is in that class, or which it inherits. If obj1 and obj2 refer to two objects of that class, then the call of obj1.comparesTo(obj2) returns 0 if they are considered equal in their natural ordering, a negative integer if obj1 is considered less than obj2 in their natural ordering, and a positive integer if obj1 is considered greater than obj2 in their natural ordering.

4) Consider the following three Java classes which form part of a program for a fantasy world game:

```
class Being
{
 protected int strength;
 public Being(int s)
 {
  strength=s;
 }
 public int getStrength()
 {
  return strength;
 }
 public String toString()
 {
  return strength+" strength being";
 }
}


class Monster extends Being
{
 private String colour;
 private int legs;
 public Monster(int s,int l,String c)
 {
  super(s);
  legs=l;
  colour=c;
 }
 public String toString()
 {
  return colour+" "+legs+"-legged monster";
 }
}


class Hero extends Being
{
 private Weapon holds;
 private String name;
 public Hero(int s,String nm,Weapon w)
 {
  super(s);
  holds=w;
  name=nm;
 }
 public int getStrength()
 {
  return strength+holds.getPower();
 }
 public String toString()
 {
  return name+" holding "+holds;
 }
 public void pickup(Weapon w)
 {
  holds=w;
 }
}
```

Question continued on next page

Suppose we have variables `h` of type `Hero`, `m` of type `Monster`, `w` of type `Weapon`, `b` of type `Being`, `ab` of type `ArrayList<Being>`, `am` of type `ArrayList<Monster>`, `strs` of type `String[]` and `n` of type `int`.

For each of the following code fragments, say whether it is valid or invalid (where "invalid" code is code which would cause a Java compiler error):

		Valid/Invalid
a)	`h.pickup(w);`	Valid (a)
b)	`b=h;`	Valid (b)
c)	`if(h.getStrength()>m.getStrength())` `  System.out.println(h);`	Valid (c)
d)	`ab.remove(m);`	Valid (d)
e)	`for(int i=0; i<am.size(); i++)` `  System.out.println(am.get(i));`	Valid (e)
f)	`m=b;`	Invalid (f)
g)	`for(int i=0; i<n; i++)` `  am.add(new Monster("green"));`	Invalid (g)
h)	`for(int i=0; i<n; i++)` `  ab.add(new Hero(10,strs[i],w));`	Valid (h)
i)	`ab=am;`	Invalid (i)
j)	`am=ab;`	Invalid (j)
k)	`m=am.get(n);`	Valid (k)
l)	`n=am.remove(m);`	Invalid (l)
m)	`m=am.remove(n);`	Valid (m)
n)	`for(int i=0; i<ab.size(); i++)` `  am.add(ab.get(i));`	Invalid (n)
o)	`for(int i=0; i<am.size(); i++)` `  ab.add(am.get(i));`	Valid (o)
p)	`for(int i=0; i<ab.size(); i++)` `  strs[i]=ab.get(i).toString();`	Valid (p)
q)	`if(am.get(n))` `  System.out.println("Got monster "+n);`	Invalid (q)
r)	`if(ab.get(n) instanceof Hero)` `  ab.get(n).pickup(w);`	Invalid (r)
s)	`h = (Hero) (ab.get(n));`	Valid (s)
t)	`w = (Weapon) (strs[n]);`	Invalid (t)

Note, this question will be marked on the basis of 1 mark for each correct answer, -1 mark for each incorrect answer.

5) This question uses the same code that was used for question 4).

a) Write a static method which takes an `ArrayList` of `Hero` or `Monster` objects, and returns a reference to whichever object in the `ArrayList` is the strongest (i.e. returns the highest value when `getStrength` is called on it).

```
static <T extends Being> T strongest(ArrayList<T> a)
{
 T strongestSoFar = a.get(0);
 for(int i=1; i<a.size(); i++)
    {
     T next = a.get(i);
      if(next.getStrength()>strongestSoFar.getStrength())
         strongestSoFar=next;
    }
 return strongestSoFar;
}
```

b) Write a method `exchange` to go inside class `Hero` which works such that if `h1` and `h2` are two variables referring to objects of type `Hero`, calling `h1.exchange(h2)` will cause the objects to swap the values of their `holds` variables.

```
void exchange(Hero h)
{
 Weapon temp = this.holds;
 this.holds = h.holds;
 h.holds = temp;
}
```

c) Write a method `equals` to go inside class `Monster` which overrides the default `equals` method, so that if `m1` and `m2` are two variables referring to objects of type `Monster`, `m1.equals(m2)` will return `true` if both objects have an equal `strength` variable, an equal `legs` variable and an equal `colour` variable, and `false` otherwise.

```
boolean equals(Object obj)
{
 if(!(obj instanceof Monster))
    return false;
 Monster m = (Monster) obj;
 return this.legs==m.legs&&
         this.strength==m.strength&&
         this.colour.equals(m.colour);
}
```