QUEEN MARY, UNIVERSITY OF LONDON
DCS128 ALGORITHMS AND DATA STRUCTURES

Class Test Monday 13th February 2006 11.05-12.35

Please fill in your Examination Number here: ----------

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) Suppose that in Java class `DrinksMachine` and `Can` there are public methods with the following signatures:

`DrinksMachine` methods:

```
void insert(int n)
int getBalance()
int getPrice()
boolean cokesEmpty()
Can pressCoke()
void loadCoke(Can can)
void setPrice(int p)
```

`Can` methods:

```
boolean isFull()
void drink()
int volume()
```

Suppose also we have variables `mach1` and `mach2` of type `DrinksMachine`, and `c1` and `c2` of type `Can`, and you can assume they have been initialised to refer to objects of the appropriate type, and suppose also we have variable `n` of type `int` which has been initialised.

For each of the code fragments below, state whether it is valid or invalid code, where invalid code is code which would cause a compiler error.

Valid/Invalid

a)
```
n = mach1.getPrice();
```
…Valid…

b)
```
n = mach2.setPrice(n);
```
…Invalid…

c)
```
c1 = mach1.pressCoke();
if(c1.isFull())
   c2 = mach2.pressCoke();
```
…Valid…

d)
```
if(insert(n))
   c2 = mach2.pressCoke()
```
…Invalid…

e)
```
while(c1.volume()>0)
   c1.drink();
```
…Valid…

[Question continued on next page]

f)
```
while(mach1.cokesEmpty())
   mach1.loadCoke(mach2.pressCoke());
```
…Valid…

g)
```
n = mach1.pressCoke().volume();
```
…Valid…

h)
```
mach1.pressCoke();
if(c1.isFull())
   c2 = mach2.pressCoke();
```
…Valid…

i)
```
if(mach1.getPrice()>n)
   mach1.setPrice() = n;
```
…Invalid…

j)
```
while(mach1.cokesEmpty())
   {
    int p = mach2.getPrice();
    if(p>n)
        mach1.loadCoke(c1.volume());
   }
```
…Invalid…

Use the space below for rough working

2) a) Write a Java static method `slide` which takes as its argument an array of integers and alters it destructively so that the last integer is put in the first place, and all other integers are moved up one place. So if the array starts off as:

5	7	9	3	7	4	1	2	8	7	6

it becomes:

6	5	7	9	3	7	4	1	2	8	7

```
public static void slide(int[] a)
{
 int last = a[a.length-1];
 for(int i=a.length-1; i>0; i--)
    a[i]=a[i-1];
 a[0]=last;
}
```

b) Write in the space below a Java static method which performs the same operation as in part a), but does it constructively.

```
public static int[] slide(int[] a)
{
 int[] b = new int[a.length];
 b[0] = a[a.length-1];
 for(int i=1; i<a.length; i++)
    b[i]=a[i-1];
 return b;
}
```

3) a) Write a Java static method which takes an arrayList and an object and returns an arrayList containing all items from the original arrayList which are not equal to the argument object in the reverse order to the one they had in the original arrayList.

For example, if the original arrayList contains Integer values as follows:

5	7	9	3	7	4	1	2	8	7	3

and the second argument is the Integer value 3, the arrayList that is returned is:

7	8	2	1	4	7	9	7	5

Your method should be generic to cover arrayLists of all base types.

```
public static <T> ArrayList<T> revDel(ArrayList<T> a,T obj)
{
 ArrayList<T> b = new ArrayList<T>();
 for(int i=a.size()-1; i>=0; i--)
    {
     T obj1 = a.get(i);
     if(!obj1.equals(obj))
        b.add(obj1);
    }
 return b;
}
```

b) Below is a Java method:

```
public static <T> int meth(ArrayList<T> a)
{
 int n=0
 for(int i=0; i<a.size(); i++)
   {
    T thing = a.get(i);
    int j=i+1;
    for(; j<a.size(); j++)
       if(a.get(j).equals(thing)
          break;
    if(j==a.size())
       n++;
   }
 return n;
}
```

State in English what value this method returns:

The number of items in the arrayList a, but counting only once items which coccur more than one time

4) a) Describe briefly the difference between a **mutable** and an **immutable** object.

An immutable object is one whose state cannot be changed once the object has been created. So the object variables are all declared as private and none of its methods changes the value of any of its object variables. A mutable object is any object which is not immutable.

b) Explain briefly what is meant by **aliasing** and why this may be a problem with mutable but not immutable objects.

Aliasing means a case where an object is referred to by two different variables. A problem with mutable variables is that if the object state is changed by a method called on it attached to one variable, it will automatically change the object as viewed through the other variable as they are the same thing. This change could be unexpected and result in problems if the program is written without taking into account the possibility of the two variables being aliases. If the object is immutable, there is no such problem since it can't be changed at all.

c) Explain what the `String` method `toUpperCase` does.

It is a method called on a String object which returns a new String. The charactes in the new String are the same characters in the same order as in the old String, except that all lower case characters are replaced by the equivalent upper case character.

d) Assuming `str1` is set to a string, and `ch` is set to a character which we know occurs at least twice in the string, write a fragment of code which sets variable `str2` to the portion of `str1` between the first and last occurrence of `ch`. So if `str1` is `"hello world"` and `ch` is `'o'`, it should set `str2` to `"o wo"`.

You may use the `String` methods `indexOf` which takes a character as its argument and returns the position of the first occurrence of that character in the string it is called on, and `lastIndexOf` which is similar, but it returns the position of the last occurrence of the character.

```
str2=str1.substring(str1.indexOf(ch),str1.lastIndexOf(ch)+1);
```

5) Write static methods to implement the following operations on objects of the generic type LispList as described in the course notes and lectures:

a) `addAfter` takes a list and two items and returns the list obtained by adding the second item after all occurrences of the first item in the original list. So, for example, with `[2,3,4,5,3,2,3,4,7]` and `3` and `8`, the result will be `[2,3,8,4,5,3,8,2,3,8,4,7]`.

```
public static <T> LispList<T> addAfter(LispList<T> list,T obj1,T obj2)
{
 LispList<T> list1 = LispList.empty();
 for(; !list.isEmpty(); list=list.tail())
    list1 = list1.cons(list.head());
 for(; !list1.isEmpty(); list1=list1.tail())
    {
     T h = list1.head();
     if(h.equals(obj1))
        list = list.cons(obj2);
     list = list.cons(h);
    }
 return list;
}
```

b) `drop`, takes a list and an item and returns a list consisting of all those items from the original list after the first occurrence of the item. So, for example, with `[2,3,4,5,3,2,3,4,7]` and `4` the result is `[5,3,2,3,4,7]`.

```
public static <T> LispList<T> drop(LispList<T> list,T obj)
{
 for(; !list.isEmpty()&&!list.head().equals(obj); list=list.tail()) {}
 if(!list.isEmpty())
    list=list.tail();
 return list;
}
```