CS125 Section 9: *I'll finally inherit a classy instance object then construct and return an immutable super deep copy!*

Rate Your Understanding of each sub-item out of 5. Mastery of Midterm II, "Attack of the OO copies"

Task 1. Write small pieces of erroneous code on a blank sheet, with errors that fall under any four of the following topics

Multi-dimensional Arrays and nested loops

1. Allocating and accessing 2D arrays. 2D arrays are just arrays of arrays.
2. new *SomeClassegString*[10] does not create any objects, rather 10 pointers, initially null.

Method invocation

1. Temporary call (execute) another piece of code, passing in values as parameters.
2. Primitive (ints,doubles...) parameters: Values are copied into the parameters.
3. Reference (aka Zombie) parameters: Memory addresses are copied but the objects they point to are not copied.
4. Methods temporarily stop executing the current piece of code, create a new scope for the method, and execute. When the method completes use the return value. For example,
   `int life = 1 + blah(); // life will be 42` elsewhere ... `public static int blah() {return 41;}`

Static methods aka "Class methods"

1. Called without an object reference e.g. `TextIO.putln(123);`
2. Include the class name if calling a method on a different class. e.g. `TextIO.methodName(param1, param2,...);`
3. The class name is optional if calling a method within the same class. e.g. `methodName(param1,param2,...);`

Object methods aka "Instance methods"

1. Need a reference to a specific object on which to call the method. e.g. `loneRangersHorse.neigh()`, `line.length()`
2. Internally have a special variable named "this".

Static variables aka "Class variable"

1. Single value of the data for the whole program - it's not stored in an object.
2. To access static data use the class name, e.g., `Horse.numLegs` ; it exists *before* any horses are constructed.

Object variables aka "Instance variable"

1. One value per object. Needs a reference (memory address of a specific object) to access.
2. Use an object reference and the 'dot' operator to access e.g. myhorse.name , of if inside a method this.name

Immutable vs. Mutable

Immutable objects can't be changed once they've been allocated. To make a type immutable ensure there's no way it can be changed by another object:
1) make all of its instance fields private, and
2) no public methods other than constructors should modify instance fields.

final object references

The keyword *final* means the bit pattern (value) of a variable cannot be changed after it's initialized.
`final Dog k9 = ...;` Now k9 will always refer to the same Dog object (which may or may not be mutable).

Public vs. Private

1. A public method/variable means it is accessible from anywhere, from other classes.
2. A private method/variable can only be used in the same class where it is defined. You stop other programmers from accessing your internal implementation e.g. helper methods and instance variables.
3. Don't write private with local (temporary) variables and parameters.

Shallow vs. Deep Copy

1. Depth of copying when copying a memory structure, such as a list of objects.
   a. When copying an object, if you don't clone its internal objects, then it is a shallow copy.
   b. If you create and use new versions of its internal objects then it's a deep copy.

Constructors

Small piece of code used to initialize objects. Parameters passed to `new` are passed to corresponding constructor .e.g.
`new ABC("!");` needs a constructor : `class ABC { ABC(String comment) { TextIO.putln("You said "+comment);} }`
Defining a constructor invalidates the default constructor.

0. Explain to your partner why you can't use "this" inside a static (aka class) method.
1. What kind of variables are never private or public? e.g. When would the following be incorrect?

```
private int score = 5; // ERROR
```

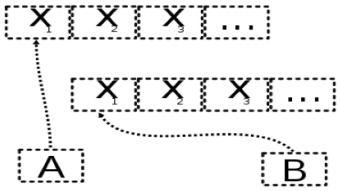
2. Explain the difference between `new Chipmonk[2]` and `new Chipmonk(2)`
3. Insert **two** lines of code to prevent a *NullPointerException*. The correct code will create an array of length 10, and 10 ghosts. Assume the Ghost class has a default constructor and defines a *setEdible* instance method.

```
Ghost[] array;
for (int i = 0 ; i < array.length ; i ++) {
   array[i].setEdible(true);
}
```

4. Why are both the following lines invalid?

```
int result = null;
if (result == null) TextIO.putln("No!");
```

5. The following picture demonstrates deep copy. How would you change it to illustrate a shallow copy? Hint: In a shallow copy only the memory pointers are copied. In a deep copy everything is copied ie. the two data structures are independent.
(image from http://en.wikipedia.org/wiki/Object_copy)



6. Here's some code inside Account.java. The class has a constructor.
i) Fix the *two* errors in the constructor. Hint you'll need 'this.'
ii) Write an example of how to create a new account, using the first constructor: `Account a = _____`
iii) Complete the *transfer* method to move the given amount from this account to the other account.
iv) Write a second constructor that takes a string and int value to initialize both instance variables.
v) What would happen if *funds* was declared as static (class) variable?
vi) If we removed the transfer function why are Account objects still mutable?

```
class Account {
    public int funds = (int) (999 * Math.random());
    public String name = "Yogi Bear";

    public void Account(String name) { name = name; } // Construct an account with a given name
    public Account(Account a) { funds = a.funds; name = a.name; } // Copy Constructor
    public void transfer (Account other, int amount) {
    }
    public int getFunds()   { return this.funds; }
    public String getName() { return this.name; }
    public equals(Object o) { // MP Hint!!
         if (o instanceof Account) {
              Account acct = (Account) o;
              return (acct.funds == this.funds) && acct.name.equals(this.name);
         }
         return false; // o was not pointing to an Account.
    }
}
```

```
class AccountList {
   public static void main(String args[]){
      Account[] acc_list = new Account[10];
      for (int i = 0 ; i < acc_list.length ; i ++)
           acc_list[i] = new Account(TextIO.getln());

      //perform deep copy of acc_list – Hint : Copy constructor




      //perform shallow copy of acc_list


}
```