CS 220 Software Design II | Spring 2022

In-class Exercises 02

University of Wisconsin - La Crosse | *Date: February 11*

1. Mark **true** or **false** on the statements below. Be prepared to explain the reasoning for your answer to your peers.

(i) ___true___ Abstract classes can have constructor methods.

(ii) ___false___ New objects of an abstract class type can be instantiated.

(iii) ___true___ A non-abstract class implementing an interface **must** provide implementations for the interface methods in the class.

(iv) ___true___ A class can both extend a class and implement an interface.

(v) ___false___ Interfaces can have attributes.

(vi) ___true___ A class can implement multiple interfaces.

2. Write a new interface `LoginAuth` that has two methods: a method `encryptPassword` that takes a single `String` password as an argument and returns a `String`, and a method `checkLogin`, which takes two `String` parameters representing the username and password, and returns a `boolean`.

Solution:

```
public interface LoginAuth {

    public String encryptPassword(String pw);
    public boolean checkLogin(String username, String pw);

}
```

3. Consider the following code. Assume that execution for the code on the right is paused on line 14 (i.e., just before the method returns). Draw out what the stack and the heap will look like at this point in time.

```
1  public class Airplane {
2
3      protected int numPass;
4
5      public Airplane(int n){
6          numPass = n;
7      }
8
9      public void board(int n){
10         numPass += n;
11     }
12 }
```

```
1  public static void main(String[] args) {
2      Airplane a = new Airplane(5);
3      check(a);
4  }
5
6  public static void check(Airplane air) {
7      int num = 3;
8      add(air, num);
9  }
10
11 public static void add(Airplane a, int n) {
12     n *= 2;
13     a.board(n);
14 }
```

Solution:

STACK

```
add : 14
  a : <points to Airplane object on heap>
  n : 6
--------------
check : 8
  air : <points to Airplane object on heap>
  num : 3
--------------
main : 3
  a : <points to Airplane object on heap>
```

HEAP

```
  Airplane
------------
numPass : 11
```

4. Consider the code below. Explain any differences between calls to methods `swap1` and `swap2` with respect to the issues discussed in class this week. Draw the stacks and heaps to illustrate your explanation.

```
1  public class Airplane {
2
3      protected int numPass;
4      protected String aircraft
       ;
5
6      public Airplane(int n,
       String s){
7          numPass = n;
8          aircraft = new String
       (s);
9      }
10
11     public void board(int n){
12         numPass += n;
13     }
14 }
```

```
1  public static void main(String[] args) {
2      Airplane a1 = new Airplane(5, "Boeing
       787");
3      Airplane a2 = new Airplane(7, "Airbus
       A380");
4
5      // call to swap1 or swap2 would go here
6
7      System.out.println(a1);
8      System.out.println(a2);
9  }
10
11 public static void swap1(Airplane x,
       Airplane y) {
12     Airplane temp = x;
13     x = y;
14     y = temp;
15 }
16
17 public static void swap2(Airplane x,
       Airplane y) {
18     String temp = x.getName();
19     x.setName(y.getName());
20     y.setName(temp);
21 }
```

Solution:

`swap1` reassigns the references for `x` and `y`. Because `x` and `y` are local variables in the method, these reassignments are local to the method. Thus, in `main` the `a1` and `a2` are unchanged. `swap2` instead changes attributes in the objects referenced by `x` and `y`. Because `x` and `y` are aliases for `a1` and `a2`, the swap changes the names in `a1` and `a2`.

Consider the classes and interfaces on the handout for the problems below.

5. Write a new, non-abstract class `Aibo` that extends `PetBot`. The animal type should be set to `"dog"`, and the model should be set to `"aibo"`; do not have these as parameters for the constructor. There should also be a new `String` attribute `name`, the value of which will be a parameter for the constructor, and then the attribute set to that value. No need to provide *implementations* for other methods, but the signature should be there if required (i.e., don't worry about overriding already implemented methods), and then `{...}` if it is going to be implemented, or `;` if it is abstract.

Solution:

```
public class Aibo extends PetBot {

    protected String name;

    public Aibo(String n) {
        super("aibo", "dog");
        name = n;
    }

    public void move() { ... }
}
```

6. Determine whether the code below is syntactically valid or invalid. **Explain your reasoning.**

```
1 Nao n = new Nao();
2 Robot r = n;
3 r.speak();
```

Solution: invalid - cannot call the method speak() on a variable of type robot

```java
public abstract class Robot {

    protected String model;

    public Robot(String m) {
        model = m;
    }

    public abstract void move();
}
```

```java
public abstract class PetBot extends Robot {

    protected String animalType;

    public PetBot(String m, String at) {
        super(m);
        animaType = at;
    }
}
```

```java
public abstract class HumanBot extends Robot{

    protected String name;

    public HumanBot(String m, String n) {
        super(m);
        name = n;
    }

    public abstract void speak();

    public void move() { /*HumanBot walks*/ }
}
```

```java
public class Nao extends HumanBot {
    ...

    public Nao() { ... }

    public void speak() { ... }
}
```