IT 313 – Advanced Application Development

Final Exam -- March 13, 2016

Part A. Multiple Choice Questions. Answer all questions. You may supply a reason or show work for partial credit. 5 points for each question.

1. Which of these is not a primitive datatype?
   a. `Character` b. `long` c. `int` d. `short`

2. Which statement checks whether the String objects **`s`** and **`t`** contain the same characters?
   a. `s.eq(t)` b. `s.equals(t)` c. `s = t` d. `s == t`

3. What is the output of this code fragment? Remember that integer division throws away the decimal part of the quotient.

```
int n = 50;
while (n > 1) {
    if (n % 2 == 0) {
        n /= 2;
    }
    else {
        n /= 3;
    }
    System.out.print(n + " ");
}
```

   a. `13 70 35` b. `25 8 4 2 1` c. `50 16 5 2` d. `50 25 8 4 2`

4. The ordered list of data types for a method is called the method _____ .
   a. instance variables b. local variables c. post list d. signature

5. Which symbols mark the beginning of a Javadoc comment?
   a. `/*` b. `/**` c. `//` d. `///`

6. In the Java Class Library, Observer is a(n)
   a. base class b. derived class c. interface d. package

7. In the Java Class Library, Observable is a(n)
   a. base class b. derived class c. interface d. package

8. In a HashMap collection, objects are inserted by
   a. exception code b. index c. key d. value

9. What kind of Java object is used to connect to a relational database?
   a. Connection b. Query c. ResultSet d. Select

10. What is the output of the recursive method call f(3, 4)? The method f is defined as

```
public static void f(int n, int m) {
    if (n > 0 && m > 0) {
        f(n – 1, m – 2);
        System.out.print(m + n + " ");
        f(m – 2, n – 2);
    }
}
```

a. 1 0 4 1 -1 3 7 b. 1 4 0 7 1 3 -1 c. 4 3 7 d. 4 7 3

11. What kind of Java object is used to represent a page in an Android Studio app?
    a. Activity b. View c. Leaf d. Segment

12. What software is used to develop server-side web applications using Java?
    a. Ant b. AWT c. JSP d. Swing

Part B. Predict the Fill Order. Enter numbers starting at 1 to indicate the order in which the cells are filled in the grid with the floodFill algorithm if the method call `floodFill(3, 3)` is used (the start point is x=3, y=3). 10 points. Here is the definition of the `floodFill` method:

```
public static void floodFill(int x, int y) {
    if (grid[y][x] == 0) {
        grid[y][x] = "*";
        floodFill(x, y-1);
        floodFill(x, y+1);
        floodFill(x+1, y);
        floodFill(x-1, y);
    }
}
  //   The variable x references the columns.
  //   The variable y references the rows.
```

	0	1	2	3	4	5	6
0	*	*	*	*	*	*	*
1			*		*		*
2	*	*					*
3	*					*	*
4	*	*			*		*
5	*		*	*			*
6	*	*	*	*	*	*	*

Part C: Unit Tests. On Page 4, write unit tests for the Dog methods `getBreed`, `setAge`, and `toString` methods. 5 points for each unit test. Here is the source code for the Pet and Dog classes.

```
//------- Source code file: Pet.java
package it313.finalexam.petsalon;
public class Pet {
    private String _name;
    private String _owner;
    private int _age;

    public Pet(String theName, String theOwner, int theAge) {
        super();
        _name = theName;
        _owner = theOwner;
        _age = theAge;
    }
    public String getName() {
        return _name;
    }
    public String getOwner() {
        return _owner;
    }
    public int getAge( ) {
        return _age;
    }
    public void setAge(int theAge) {
        _age = theAge;
    }
    @Override
    public String toString() {
        return String.format("%s %s %d", _name, _owner, _age);
    }
}
//------- Source code file: Dog.java
package it313.finalexam.petsalon;
public class Dog extends Pet {
    private String _breed;

    public Dog(String theName, String theOwner, int theAge,
               String theBreed) {
        super(theName, theOwner, theAge);
        _breed = theBreed;

    }
```

```
//------- Source code file: Dog.java (continued)
    public String getBreed() {
        return _breed;
    }
    @Override
    public String toString( ) {
        return super.toString( ) + String.format(" %s", _breed);
    }
}

//------- Source code file: DogTest.java
public class DogTest {
    Dog _d;

    @Before
    public void setUp() {
        _d = new Dog("Duke", "Maxwell", 3, "German Shephard");
    }

    @Test
    public void testGetBreed() {




    }

    @Test
    public void testSetAge() {




    }

    @Test
    public void testToString() {




    }
}
```

Part D: Correct the Errors. Correct the errors in Problem 1 (Main1) or Problem 2 (Main2, TornadoSimulator, TornadoObserver), but not both. Correcting a pair of ( ), [ ], { }, < >, single quotes, or double quotes only counts as one error.

1. The Main1 class populates the dogs database table using the fields from Dog objects in a HashMap collection. There are about 10 errors in the Main1 class. Correct them.

```
//------- Source code file Main1.java.

package it313.finalexam.petsalon;

import java.sql.*;
import java.util.HashMap;
require java.util.Map.Entry;

public class Main1 {

    public static void main(String[] args)
        throws ClassNotFoundException, SQLException  {

        // Create HashSet that contains 4 Dog objects.
        HashMap<String, Dog> col = new HashMap( );
        col.put("D1", new Dog("Duke", "Maxwell", 3,
            "German Shephard"));
        col.put("D2", new Dog("Sparky", "Rogers", 1, "Dachshund");
        col.put("D3", new Dog("Mimi", "Sanchez", 1, "Pomeranian"));
        col.add("D4", new Dog("Ginger", "Saari", 4, "Collie"));
        System.out.println(col);

        // Create Connection and Statement objects.
        Class.forName("org.sqlite.JDBC");
        Connection c = DriverManager.
            getConnection("jdbc:sqlite:dogs.db");
        Statement s = c.createStatement;

        // Create new dogs table, if it doesn't already exist.
        String sql1 = "create table if not exists dogs(" +
                        "name varchar(10), " +
                        "owner varchar(10), +
                        "age integer, " +
                        "breed varchar(10));";
        s.executeUpdate(sql1);
```

```
//------- Source code file Main1.java (continued)

// Populate table with dogs from collection.
     for(Entry<String, Dog> e : col.entrySet()) {
          Dog d = e.getKey( );
          string sql2 = String.format(
               "insert into dogs values('%s', '%s', %d, %s);",
               d.getName( ), getOwner( ), d.getAge( ),
               d.getBreed);
          s.executeUpdate(sql2);
     }
}
```

2. The Main2 class of the Tornado project creates observables (TornadoSimulator objects), which simulate tornado warnings in Cook county and the collar counties DuPage, Kane, Lake, McHenry, and Will. The probability that a tornado warning occurs on a given day in a given county for this simulation is 0.1%. The Main2 class also creates an observer (TornadoObserver object), which monitors the tornado warnings in these counties and records any such warnings that occur. There are about 10 total errors in the Main2, TornadoSimulator, and TornadoObserver classes. Correct them.

   The bottom of Page 8 and pages 9 and 10 have Java Class Library documentation for the Observer interface and Observable class.

```
// ------------ Source code file Main2.java

import java.io.FileNotFoundException;

public class Main2 {

    public static void main(String[ ] args)
        throws FileNotFoundException {

        TornadoSimulator[ ] simulators =
            { new TornadoSimulator("Cook"),
              new TornadoSimulator("DuPage"),
              new TornadoSimulator("Kane"),
              new TornadoSimulator("Lake"),
              new TornadoSimulator("McHenry"),
              new TornadoSimulator("Will") };

         observer = new
             TornadoObserver(logfile.txt);
```

```
            for(TornadoSimulator sim : simulators) {
                sim.addObserver(observer);
            }

           for(int day == 1; day <= 1000; day++) {
               for(TornadoSimulator sim : simulators) {
                   sim.checkForTornado( );
               }
           }
       }
 }

// ----- Source code file TornadoSimulator.java

import java.util.Observable;
import java.util.Random;

public Class TornadoSimulator super Observable {

    private String _county;
    private Random _r;

    public TornadoSimulator(String theCounty) {
        _county = theCounty;
        _r = new Random( );
    }

    public void checkForTornado( ) {
        if _r.nextInt(1000) = 999 {
            notifyObservers("County: " + _county);
            setChanged( );
    }
}
```

```
// ----- Source code file TornadoObserver.java

import java.io.File;

import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Observable;
import java.util.Observer;

public class TornadoObserver implements Observer {

     PrintWriter _pw;

     public TornadoObserver(String the_file_name)
          throws FileNotFoundException {

          File f = new File(theFileName);
          PrintWriter _pw = new PrintWriter(f);
     }

     @Override
     public void update(Observable theSimulator, Object theMessage) {
         _pw.println(theMessage);
         pw.flush( );
     }
```

---

public interface Observer

Method Detail

`void update(Observable o, Object arg)`

This method is called whenever the observed object is changed. An application calls an Observable object's notifyObservers method to have all the object's observers notified of the change.

Parameters:
`o` – the observable object.
`arg` – an argument passed to the notifyObservers method.

---

public class Observable extends Object

This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed.

An observable object can have one or more observers. An observer may be any object that implements interface Observer. After an observable instance changes, an application calling the Observable's notifyObservers method causes all of its observers to be notified of the change by a call to their update method.

The order in which notifications will be delivered is unspecified. The default implementation provided in the Observable class will notify Observers in the order in which they registered interest, but subclasses may change this order, use no guaranteed order, deliver notifications on separate threads, or may guarantee that their subclass follows this order, as they choose.

Note that this notification mechanism has nothing to do with threads and is completely separate from the wait and notify mechanism of class Object.

When an observable object is newly created, its set of observers is empty. Two observers are considered the same if and only if the equals method returns true for them.

Constructor Detail

`public Observable( )`

Construct an Observable with zero Observers.

Method Detail

`public void addObserver(Observer o)`

Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set. The order in which notifications will be delivered to multiple observers is not specified.

Parameter: o – an observer to be added

`public void deleteObserver(Observer o)`

Deletes an observer from the set of observers of this object. Passing null to this method will have no effect.

Parameter: o - the observer to be deleted.

`public void notifyObservers(Object arg)`

If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.

Each observer has its update method called with two arguments: this observable object and the arg argument.

Parameter: arg – any object

`public void deleteObservers( )`

Clears the observer list so that this object no longer has any observers.

`protected void setChanged()`

Marks this Observable object as having been changed; the hasChanged method will now return true.

`protected void clearChanged()`

Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the hasChanged method will now return false. This method is called automatically by the notifyObservers methods.

`public boolean hasChanged( )`

Tests if this object has changed.

Returns:

true if and only if the setChanged method has been called more recently than the clearChanged method on this object; false otherwise.

`public int countObservers( )`

Returns the number of observers of this Observable object.
