COMP206-08S Test 2

15 February 2008

First name: _____ **Last name:** _____

ID Number: _____

Instructions

1. Write your name and ID number into the spaces provided above.
2. There are seven questions of equal value.
3. Time allowed is 100 minutes.
4. This test is open book; but NO computers, pdas, cell phones, or similar, are allowed.
5. Write your answers in the spaces provided. Do not use your own paper! There is a blank page at the end of the test that you can use. You can use the backs as well. You can also get extra paper from me if necessary.

DO NOT TURN THE PAGE UNTIL YOU ARE ASKED TO DO SO!

Question 1: Java Trivia

Circle the correct answer. All statements are about the Java language and its libraries.

Swing is thread-safe.	YES	NO
Reflection is useful for building flexible tools like debuggers, shells, etc.	YES	NO
The Collection classes use generic types.	YES	NO
A RandomAccessFile only allows read access.	YES	NO
Using reflection one can access even private data fields of any class.	YES	NO
Serialization handles shared structure correctly.	YES	NO
Writes to a RandomAccessFile will always immediately update the file on disk.	YES	NO
ActionListeners must call paint() to update the screen.	YES	NO
MouseAdapter is an abstract utility class simplifying MouseListeners.	YES	NO
Generic types help the compiler catch more errors.	YES	NO
Threads stop once the run method finishes.	YES	NO
Callables are like Runnables, but also return a result.	YES	NO
You cannot not have more threads than CPU cores in a ThreadPool.	YES	NO
Synchronized methods are used to avoid inconsistencies.	YES	NO
Readers and Writers are for textual IO only.	YES	NO
If two objects have the same hash-code, they are also equal.	YES	NO
JPanel objects can be used inside other JPanel objects.	YES	NO
JFrame objects can be used inside other JFrame objects.	YES	NO
Every class in Java can only "extend" one other class.	YES	NO
Every class in Java can only "implement" one interface.	YES	NO
Object is the ultimate superclass of every other class.	YES	NO

Question 2: Implementing some class

Assume you are given the following Interface definition for accounts in general:

```
public interface Account {
   public void deposit(int amount);
   public void withdraw(int amount);
   public int getBalance();
   public String getOwner();
}
```

First, implement a class "BankAccount" that implements the "Account" interface. Your class has two private data fields: _owner and _balance. Define one constructor BankAccount(String owner) which will remember the owner and will set the balance to 0. BankAccount also needs a compareTo method which should compare owners first (class String supports "compareTo"); if two accounts have the same owner the accounts should be compared using their balance. toString should print something like: <BankAccount, owner = Bob, balance = 100> for Bob's account, when his balance is 100 (fill in code for the dots below).

```
public class BankAccount implements Account, Comparable<Account> {
       private int _balance = -1;
       private String _owner;

       public BankAccount(String owner) {
           ...
       }

       public int compareTo(Account otherAccount) {
          ...


       }

       public String toString() {
         ...
       }
       // all other methods needed for implementing Account
       ...
```

Question 3: Inheritance

Now we need a class "CheckingAccount" which will be a subclass of "BankAccount". As deposits and withdrawals incur fees, we need to track the number of these transactions to be able to compute the appropriate fee. Implement deposit and withdraw which use the deposit and withdraw method of BankAccount, but also increment numberOfTransactions by 1 for every withdrawal and for every deposit. toString should print something like: <CheckingAccount, owner = Bob, balance = 100> for Bob's account, when his balance is 100. Also make sure the constructor does the right thing.

```
public class CheckingAccount extends BankAccount {

     private int _numberOfTransactions = 0;

     public int computeFees(int costPerTransaction) {
       int fee = costPerTransaction * _numberOfTransactions;
       _numberOfTransactions = 0;
       return fee;
     }

     public CheckingAccount(String owner) {
         ...
     }

     public void deposit(int amount) {
        ...


     }

     public void withdraw(int amount) {
        ...


     }

     public String toString() {
         ...


     }
}
```

Question 4: Composition

Now implement a class "CheckingAccountByComposition" which will have the same functionality as the CheckingAccount class from the previous question, but will use composition instead of inheritance. toString should print something like: <CheckingAccountByComposition, owner = Bob, balance = 100> for Bob's account, when his balance is 100.

```
public class CheckingAccountByCompostion implements Account {
     private BankAccount _account;
     private int _numberOfTransactions = 0;

     public CheckingAccountByCompostion(String owner) {
         ...
     }

     public int computeFees(int costPerTransaction) {
       int fee = costPerTransaction * _numberOfTransactions;
       _numberOfTransactions = 0;
       return fee;
     }

     public void deposit(int amount) {
        ...


     }

     public void withdraw(int amount) {
        ...


     }

     public String toString() {
         ...
     }

     public String getOwner() {
         ...
     }

     public int getBalance() {
         ...
     }

}
```

Question 5: Constructors

What output will the main method of class TestABC below produce?

```
class A {
  private String _name;
  public A() { setName(); }
  public void setName() { _name = getClass().getName();}
  public String toString() { return _name; }
  public String fullName() { return toString(); }
}

class B extends A {
  public B() { setName(); }
  public String fullName() { return toString() + toString(); }
}

class C extends B {
  public C() { setName(); }
  public String fullName() { return toString() + toString() + toString(); }
}

public class TestABC {
  public static void main(String[] args) {
    A[] objects = new A[]{new C(), new B(), new A()};
    for(A o: objects) {
      System.out.println(o);
    }
    for(A o: objects) {
      System.out.println(o.fullName());
    }
  }
}
```

Question 6: Polymorphism

For the following code fragment, for all statements circle the correct answer: this statement is fully ok (*Ok*), or it will not compile (*Does not compile*), or it will compile, but cause an exception at runtime (*runtime exception*):

```
import java.awt.Rectangle;

interface Edible {}

public class Sandwich implements Edible {

   public static void main(String[] args) {

       Sandwich sub = new Sandwich();
       // Ok  /  Does not compile  /  runtime exception

       Edible e = sub;
       // Ok  /  Does not compile  /  runtime exception

       Rectangle cerealBox = new Rectangle(0,0,20,30);
       // Ok  /  Does not compile  /  runtime exception

       Edible f = cerealBox;
       // Ok  /  Does not compile  /  runtime exception

       f = (Edible) cerealBox;
       // Ok  /  Does not compile  /  runtime exception

       sub = e;
       // Ok  /  Does not compile  /  runtime exception

       sub = (Sandwich) e;
       // Ok  /  Does not compile  /  runtime exception

       sub = (Sandwich) cerealBox;
       // Ok  /  Does not compile  /  runtime exception

       f = null;
       // Ok  /  Does not compile  /  runtime exception

       sub = (Sandwich) f;
       // Ok  /  Does not compile  /  runtime exception
   }
}
```

Question 7: Concurrency

This is a modified version of Race.java, answer questions about it on the next page, please.

```
public class Race {

  public static void main(String[] args) throws Exception {
    new Race();
  }

  public Race() throws Exception {
    ExecutorService pool = Executors.newFixedThreadPool(2);
    Future<Date> racer1 = pool.submit(new Racer("Anne", 100));
    Future<Date> racer2 = pool.submit(new Racer("Bob", 100));
    Date date1 = racer1.get();
    Date date2 = racer2.get();
    pool.shutdown();
    if (date1.compareTo(date2) < 0) {
      System.out.println("Anne wins");
    } else {
      System.out.println("Bob wins");
    }
  }

  public synchronized void report(final String name, final int count) {
    System.out.println("Here is " + name + ": " + count);
  }

  private class Racer implements Callable<Date> {

    public Racer(final String name, final int count) {
      _name = name;
      _count = count;
    }

    public Date call() {
      while (_count-- > 0) report(_name, _count);
      return new Date();
    }

    private int _count;
    private String _name;

  }
}
```

1. Describe how the program above differs from the version we saw in class which was a code example using wait() and notify() for synchronisation.

2. Which version do you find easier to understand, explain why?

3. How does the version above ensure that both Racers have finished?

4. Do you think that the program above is still indeterministic, i.e. that sometimes Anne will win, and sometimes Bob will win?

Extra space for answering questions