Com S 227
Fall 2017
Topics and review problems for Final Exam

Wednesday, December 13, 7:00 - 9:00 pm.

Locations:
Last name A-D: Carver 0001
Last name E-N: Curtiss 0127
Last name O-Z: Physics 0005

Note: locations are NOT the same as the midterms!

***** You must bring your ID *****

General information

This will be a 115-minute, timed, pencil-and-paper written exam. No books, no notes, no electronic devices, no headphones, no collaboration. The problems will primarily involve writing Java code or reading and interpreting Java code.

Exam topics

The final exam is comprehensive, though it will emphasize topics covered since Exam 2. A reasonable guess is that 40 to 60% of the exam will be on new material, but this is difficult to measure precisely since all new material really depends on earlier material. We strongly recommend that you review anything you had trouble with from earlier exams. The list below is a rough overview of the highlights for this **new** material. Numbers in parentheses refer to the **current** textbook. See http://www.cs.iastate.edu/~cs227/section_number_mapping.pdf for previous editions.

- Using the `compareTo` method (5.2.3, 10.3)
- Interfaces and polymorphism (10.1, 10.2)
- Inheritance (9.1 – 9.5)
  - Defining subclasses
  - Overriding methods
  - Calling the superclass constructor
  - The class `Object`
  - Polymorphism and dynamic binding of methods
  - The declared type and the runtime type of a variable
  - Downcasts (10.2.3)
  - The `protected` keyword
  - Abstract methods and classes
- Exceptions – reporting, handling, and declaring exceptions (11.4)
- `throw, throws, try/catch, try/finally, try/catch/finally`

You do not have to memorize methods from the Java API. You should know how to use `String`, `Scanner`, `Math`, `Random`, `File`, and `ArrayList<E>`, but for specific methods from these classes that might be needed, we'll provide you with the minimal one-sentence descriptions from the API. You should know how to read input from `System.in` and how to read and write text files. *The following pages contain some practice problems related to the topics above. Remember that the review materials for Exams 1 and 2 are still relevant. You are encouraged to post your sample solutions on Piazza for discussion.*

1. An interface is shown below that represents the players in a simple game. A “move” in the game consists of a player choosing an integer 0, 1, or 2 (like a Rock-Paper-Scissors game). Each player also keeps a history of past moves that can be examined with the getPreviousMove() method. Below you will find the code for two concrete implementations of IPlayer. Notice there is some duplicated code. Modify the implementation so that most of the duplication is eliminated.

```
interface IPlayer {
  int play();  //Returns the player's move, which is always 0, 1, or 2
  int getPreviousMove(int movesAgo); // Returns a previous move
}

class RandomPlayer implements IPlayer {
  private Random rand = new Random();
  private ArrayList<Integer> history = new ArrayList<Integer>();

  public int play(){
    int move = rand.nextInt(3); // randomly chooses 0, 1, or 2
    history.add(move);
    return move;
  }

  public int getPreviousMove(int movesAgo) {
    return history.get(history.size() - movesAgo);
  }
}

class AlternatingPlayer implements IPlayer {
  private int state = 0;
  private Random rand = new Random();
  private ArrayList<Integer> history = new ArrayList<Integer>();

  public int play(){
    // usually returns 0, but every third move randomly chooses 1 or 2
    int move = 0;
    if (state % 3 == 2){
      move = rand.nextInt(2) + 1;
    }
    ++state;
    history.add(move);
    return move;
  }

  public int getPreviousMove(int movesAgo){
    return history.get(history.size() - movesAgo);
  }
}
```

(*Some possible solutions are posted in the November 30 entry for the Section B topics page:*
http://web.cs.iastate.edu/~smkautz/cs227f16/topics.html
See *IPlayer, Solution1, and Solution2; discussion of solutions is in the javadoc.)*

2. Suppose we have the interface at right. You might have classes such as Line or Circle or Square that implement the interface, as in the example below:

```
public interface Shape
{
  double getArea();
  void draw();
}
```

```
class Circle implements Shape {  // example only
  private Point center;
  private double radius;
  public Circle(Point p, double r) { radius = r; center = p; }

  public double getArea() {
    return Math.PI * radius * radius;
  }
  public void draw() {
    // does graphical stuff to draw, details not shown...
  }
}
```

Create a class `Picture` that contains a list of shapes and has these public methods, plus a constructor that creates an empty `Picture`:

```
// adds a Shape to this Picture
public void add(Shape s)

// finds the total area of all shapes in this Picture (zero if empty)
public double findTotalArea()
```

3. A student who is enthusiastic about inheritance decides implement the `Picture` class like this:

```
public class Picture extends ArrayList<Shape>
{
  public Picture()
  {
      super();
  }

  public double findTotalArea()
  {
    double total = 0.0;
    for (Shape s : this)
    {
      total += s.getArea();
    }
    return total;
  }
}
```

a) Does this work? b) Why would this be an undesirable solution?

4. The class below is a general utility for processing values in an array to gather information about them (the array is not modified):

```
public class ArrayProcessor {
  // Processes values in an array using the given Processor
  public static double runProcessor(double[] arr, Processor p) {
    for (int i = 0; i < arr.length; ++i) {
      p.process(arr[i]);
    }
    return p.getResult();
  }
}
```

a) Suppose you know that `Processor` is a Java interface. Based on its usage above, write the definition for the interface (remember that interfaces do not have method bodies)

b) Create a class `MaxProcessor` implementing the `Processor` interface that will return the maximum value in an array, e.g.:

```
Processor maxFinder = new MaxProcessor();
double max = ArrayProcessor.runProcessor(myData, maxFinder);
```

c) Create a class `RangeCounter` implementing the `Processor` interface that will count how many values in the array are in the range [*min, max*], where *min* and *max* are arguments to the `RangeCounter` constructor.

5. Consider the class `Point` below. Modify it so that it implements the interface `Comparable<Point>`. Implement the `compareTo` method so that points are sorted according to their distance from the origin, $\sqrt{x^2 + y^2}$.

```
class Point {
  private int x;
  private int y;
  public Point(int givenX, int givenY)
  {
    x = givenX;
    y = givenY;
  }
  public int getX()
  {
    return x;
  }
  public int getY()
  {
    return y;
  }
}
```

6. Suppose you have a class `Foo` that implements the interface `Comparable<Foo>`. Write a method that returns the maximal `Foo` in an array of `Foo`.

7. In lab 3 we wrote several different implementations of a class called **`RabbitModel`**. Refer to the lab pages if you don't remember: http://web.cs.iastate.edu/~cs227/labs/lab3/page09.html
It was really awkward that each time we created a new one, we had to name it exactly **`RabbitModel`** and we had to rename or delete any previous implementations having the same name. The problem was that the **`SimulationPlotter`** class was written to explicitly depend on the **`RabbitModel`** class.

Now that we know about interfaces, we can do better. Suppose instead that we start with an *interface* called (say) **`IRabbitModel`**, defined as follows to represent the capabilities required by the **`SimulationPlotter`**. Now the **`SimulationPlotter`** can be written so that it can plot any model, no matter what the class is called, as long as it implements the **`IRabbitModel`** interface

```
public interface IRabbitModel
{
  public int getPopulation();
  public void simulateYear();
  public void reset();
}
```

Your task is: Reimplement the four models described in Checkpoint 2 of that lab. This time, you can name the four classes whatever you want; just make sure each one implements the IRabbitModel interface (directly or indirectly). Use what you know about inheritance and abstract classes to minimize code duplication. Do it without using protected instance variables. If you want to try out your code, you can find the **`plotter.IRabbitModel`** interface and the updated **`SimulationPlotter`**, along with the javadoc, here:
http://web.cs.iastate.edu/~cs227/exams/final/rabbits/

8. What is the result of the method call **`tryStuff("10 20 23skidoo 30 foo bar")`** ? (Recall that **`Integer.parseInt()`** will throw a **`NumberFormatException`** if you attempt to parse any non-numeric string.)

```
  public static int tryStuff(String text) {
    int total = 0;
    int i = 0;
    Scanner scanner = new Scanner(text);
    while (scanner.hasNext())
    {
      try
      {
        String s = scanner.next();
        i = Integer.parseInt(s);
        total += i;
      }
      catch (NumberFormatException nfe)
      {
        total -= i;
      }
    }
    return total;
  }
```

9. Suppose you have a method that opens and processes a file. For example, this method is supposed to tally up a file with total votes for several candidates and print the percentages.

```
// prints each name in the file followed by the total percentage of votes
public static void printPercentages(String filename)throws FileNotFoundException
{
  // ...details not shown...
}
```

A possible main method that *uses* this method looks like this:

```
  public static void main(String[] args) throws FileNotFoundException
  {
    printPercentages("votes.txt");
  }
```

Rewrite the main method *without* the **`throws`** declaration. Instead, add error handling so that if **`printPercentages`** throws a **`FileNotFoundException`**, the message "no such file" is printed to the console.

10. On the next two pages there are some declarations for a class hierarchy, along with the big box below containing some code. For each line of code in the box, indicate one of the following:

- If it is a compile error, comment out the line and state why; or
- if there is an exception at runtime, comment out the line and state which exception; or
- if the code works and produces output, indicate what the output is (assume that previous commented-out lines do not execute)

(*The first three lines are done for you as an example.*)

11. Without changing the *public* API, and without adding any non-private variables, modify the code from problem 10 so that the method **`getCheckOutPeriod()`** is implemented only in **`LibraryItem`**.

```
Item i = new Book("Treasure Island");                // OK
System.out.println(i.getTitle());                    // Output:  "Treasure Island"
// System.out.println(i.getCheckOutPeriod());        // Compile error, Item has no such method
Book b = new ReferenceBook("How to Bonsai Your Pet");
System.out.println(b.getTitle());
System.out.println(b.getCheckOutPeriod());
LibraryItem li = null;
li = new LibraryItem("Catch-22");
System.out.println(li.getTitle());
System.out.println(li.getCheckOutPeriod());
li = new DVD("Shanghai Surprise", 120);
System.out.println(li.getTitle());
System.out.println(li.getCheckOutPeriod());
System.out.println(li.getDuration());
i = b;
b = i;
System.out.println(i.getTitle());
ReferenceBook rb = (ReferenceBook) b;
System.out.println(rb.getTitle());
rb = (ReferenceBook) new Book("Big Java");
System.out.println(rb.getTitle());
```

```
public interface Item
{
  String getTitle();
}

public abstract class LibraryItem implements Item
{
  private String title;
  protected LibraryItem(String title)
  {
    this.title = title;
  }

  public String getTitle()
  {
    return title;
  }

  public abstract int getCheckOutPeriod();
}
```

```java
public class Book extends LibraryItem
{
  public Book(String title)
  {
    super(title);
  }

  public int getCheckOutPeriod()
  {
    return 21; // three weeks
  }
}

public class ReferenceBook extends Book
{
  public ReferenceBook(String title)
  {
    super(title);
  }

  public String getTitle()
  {
    return "REF: " + super.getTitle();
  }

  public int getCheckOutPeriod()
  {
    return 0; // reference books don't circulate
  }
}

public class DVD extends LibraryItem
{
  private int duration; // duration in minutes

  public DVD(String title, int duration)
  {
    super(title);
    this.duration = duration;
  }

  public String getTitle()
  {
    return "DVD: " + super.getTitle();
  }

  public int getCheckOutPeriod()
  {
    return 7; // DVDs check out for one week
  }

  public int getDuration()
  {
    return duration;
  }
}
```