Com S 227
Fall 2020
Topics, review problems, and process for final exam
Monday, November 23, 7:00 - 9:00 pm

General information

The format of the exam is similar to Exam 2. It will again be conducted on Canvas using the lockdown browser, and proctored via Zoom. Do the following immediately if you haven't already:

1. *Check Zoom:* Make sure you have the Zoom app on your phone. Join a test meeting using your phone at https://zoom.us/test and make sure the microphone and camera are working. Make sure you know how to "raise your hand". You'll need to keep the microphone un-muted during the exam, but you can turn down/off your speaker. (We'll use the chat window to get your attention if we have to make an announcement.)

2. *Check Location:* Figure out where you're going to take the exam and how you will prop up your phone so that you are visible. You will need to be in a private room, at a desk or table that is clear except for your phone and computer and one piece of scratch paper. It needs to be a quiet place, because you'll have your microphone on during the exam. Make sure your phone is set up to use the local wifi.

3. *Check Canvas:* Try the "practice exam" on Canvas (under "Quizzes) to make sure you can successfully use the Respondus lockdown browser.

- Check-in and check-out process:
  - Once everyone is in the zoom room, we will ask you all to simultaneously provide a panorama of your room and desktop[1] - after that, we'll give out the exam password.
  - You'll show your ID at the *end* of the exam, before leaving the zoom room. **Important:** *If you leave the meeting without showing your ID, we will not grade your exam.*
  - Exam time limit 120 minutes.

If you cannot run the lockdown browser or if you cannot participate in the zoom meeting with your smartphone, please contact your instructor right away to make an alternative arrangement.

[1] If you uncomfortable about other people seeing your room, please send a private chat to the TA in your zoom meeting, and we can anonymously move you to a breakout room for that portion of the check-in.

Exam topics

The final exam is comprehensive, though it will emphasize topics covered since Exam 2. A reasonable guess is that about 75% 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.

- Sorting basics, selection sort
- The merge sort algorithm
- Inheritance, polymorphism, and interfaces
    - 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
    - The `protected` keyword
    - Abstract methods and classes
- The `Comparable` interface, writing and using a `compareTo` method
- Exceptions – reporting, handling, and declaring exceptions
- `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 by moving all common code into an abstract superclass. Do not modify the public API and do not use any non-private instance variables.

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

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

  public RandomPlayer(String givenName) {
    name = givenName;
  }

  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);
  }
  public String getName() {
    return name;
  }
}

public class AlternatingPlayer implements IPlayer {
  private String name;
  private int runlength = 1;
  private int count = 0;
  private ArrayList<Integer> history = new ArrayList<Integer>();

  public AlternatingPlayer(String givenName) {
    name = givenName;
  }

  public int play(){
    int move;
    // does runlength many 0's, then a 1 and a 2,
    // and then runlength increases by 1
    if (count < runlength) {
      move = 0;
      count += 1;
    }
    else if (count == runlength){
      move = 1;
      count += 1;
    }
    else {
      move = 2;
      runlength += 1;
      count = 0;
    }
    history.add(move);
    return move;
  }
  public int getPreviousMove(int movesAgo){
    return history.get(history.size() - movesAgo);
  }
  public String getName() {
    return name;
  }
}
```

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 any kinds of shapes and has these two 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 might this be an undesirable solution?

3. (15 pts) Suppose you have the interface **Transformation** shown in the box at right. a) Write a method that, given an array of numbers and a **Transformation**, applies the transformation to each element of the array (that is, each number in the array is replaced by its transformed value).

```
public interface Transformation
{
  double transform(double d);
}
```

```
public static void applyToAll(double[] numbers, Transformation t)
{
```

b) Create a class **HalfTransformer** implementing the **Transformation** interface whose **transform()** method divides its argument in half. (*For example*: if **t** is an instance of **HalfTransformer** and **arr** is the array [10.0, 20.0, 30.0], then after **applyToAll(arr, t)**, **arr** contains the elements [5.0, 10.0, 15.0].)

4. 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;
  }
}
```

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

6. In lab 2, checkpoint 2, we wrote several different implementations of a class called **`RabbitModel`**. Refer to the lab pages if you don't remember. 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 any non-private instance variables. If you want to try out your code, you can find the **`plotter.IRabbitModel`** interface and the updated **`SimulationPlotter`** here:
http://web.cs.iastate.edu/~smkautz/cs227f20/temp/rabbits_with_interface.jar

7. 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;
  }
```

8. There are many variations of the mergesort algorithm having different strategies to reduce memory usage and array copying. Suppose that you are *given* a **merge** method with the following declaration:

```
/**
 * Merges two sorted subarrays of a given array, storing the result back in
 * the given array. That is, when the method is called,
 *
 *      arr[start] through arr[mid] is already sorted, and
 *      arr[mid + 1] through arr[end] is already sorted
 *
 * When the method returns,
 *
 *      arr[start] through arr[end] is sorted.
 */
private static void merge(int[] arr, int start, int end, int mid)
```

Suppose that you are also given the public method:

```
public static void mergeSort(int[] arr)
{
  mergeSortRec(arr, 0, arr.length - 1);
}
```

Write the following recursive helper method that will sort a given subarray using the merge sort algorithm:

```
/**
 * Performs a recursive merge sort of the subarray consisting of
 * arr[start] through arr[end].
 */
private static void mergeSortRec(int[] arr, int start, int end)
```

Note: This problem is NOT asking you to rewrite the merge() method! You can find a sample solution for the problem above, along with another other variation of mergesort, in the week 11 code examples for Sections A and B (link #6 on the Canvas front page).

9. Rewrite the base case of your mergesort implementation above so that whenever the subarray has size 5 or less, it directly sorts it using a selection sort algorithm.

10. 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...
}
```

In addition to a possible **FileNotFoundException** as seen in the method declaration, **printPercentages** will throw a **NumberFormatException** if the file format is incorrect.

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

```
public static void main(String[] args) throws FileNotFoundException
{
  System.out.println("Enter filename for votes: ");
  String filename = new Scanner(System.in).next();
  printPercentages(filename);
}
```

Rewrite the main method *without* the **`throws`** declaration. Add error handling so that if **`printPercentages`** throws a **`FileNotFoundException`**, the message "no such file" is printed to the console, and if **`printPercentages`** throws a **`NumberFormatException`**, the message "invalid file format" is printed to the console. Add a loop so that the main method will allow the user to re-enter another filename as many times as needed until it is successfully read.

11. 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 your previous commented-out lines do not execute)

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

12. 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();
}
```

```
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;
  }
}
```