Final Exam Review Packet

COP 3330, Spring 2023

Exam Date: Friday, April 28 (10:30 AM – 12:30 PM)

This packet gives an overview of topics to study as you prepare for the final exam. The anticipated time limit for the exam is 2 hours. You will need a pen or pencil and your UCF ID. You cannot use calculators, notes, or any other aids.

I anticipate that the final exam will have four to seven questions. It will focus primarily on topics covered since Exam #2, but there will likely be an explicitly cumulative component, as well. (The nature of the material covered this semester is of course inherently cumulative in many ways, as well.) The primary topic areas for the exam are:

1. Generics in Java
2. Packages and Access Modifiers
3. Exception and Error Handling
4. Interfaces
5. Java's Container Classes
6. Problem Solving, Object-Oriented Concepts, Design Principles, and Java Tidbits

In addition to reviewing all the topics and questions in this packet, I strongly encourage you to work through all the examples, practice exams problems, and quiz questions I've posted in Webcourses as part of your preparation for the final exam. You should also be studying and replicating any code we've written in class. You can gain additional coding practice by completing practice problems on CodingBat on a regular basis and revisiting the programming assignments in the course. You should also consider studying in varied conditions.

The exam covers material presented from the beginning of the semester through Monday, April. 24. Unless otherwise specified in this packet, all information presented in class is fair game for the exam, even if I forgot to include certain topics in this packet, and even if I haven't created practice problems related to a certain topic.

The exact structure and point breakdown for your exam will be posted as an announcement in Webcourses as soon as I finish writing the exam (probably very late in the evening of Thursday, Apr. 27).

Final Exam Review Packet

General Expectations

On the exam, you might encounter any or all of the following types of questions:

1. You might be asked to write Java code. You could be asked to write short snippets of code to demonstrate that you know basic Java syntax – e.g., "Write a line of code that creates an array list of strings," or, "Modify the following class declaration so that only objects that implement Java's *Comparable* interface are allowed into the *SnowGlobe* class: *public class SnowGlobe<T>*." You might also be asked to write longer chunks of code to solve novel problems on the exam. The "Mapping a Single Key to Multiple Values" example from the Apr. 14 notes is a good indication of the level of difficulty I would expect you to be able to tackle in a coding question on the final exam.

2. When coding on an exam, you should write clear, readable code. While you're not expected to memorize every style restriction imposed on the programming assignments, I will nevertheless expect you to conform as closely as possible to the style used in class. (I suspect this won't be a problem for anyone who has been attending class regularly and/or reading through all the notes regularly.) In all the code you write, you will also be expected to avoid features and components of Java that we have not yet covered in class.

3. On the exam, you will be asked to scrutinize existing code to determine its behavior. That could involve tracing through code to determine what its precise output will be, identifying errors that would prevent the code from compiling, or explaining errors that would cause a program to crash at runtime. You could also be asked to identify and fix errors in existing code or to fill in missing pieces from a chunk of code. Many of these questions might be similar to the questions in quizzes #10, #11, and #12. These questions could also mirror the style and structure of exercises we've gone over in class or practice exam questions posted at the end of each day's lecture notes in Webcourses.

4. You could be presented with some code and asked to identify areas where it violates key software design principles or object-oriented design principles covered in class (e.g., violations of the single responsibility principle (SRP) or the DRY principle, problems with encapsulation or abstraction, the inclusion of magic numbers, the use of meaningful variable names, and so on). You could also be asked to re-write code in such a way that it meets those design principles – e.g., re-factoring a method (or an entire class) to eliminate duplicate lines of code, as we did with the *Lunchbox* example (see "Functional Decomposition: Lunchbox" in the lecture notes from February 17).

5. The exam might include some problem solving components where you have to apply your knowledge of Java (along with critical thinking and creative problem solving skills) to solve a novel problem.

6. You could be asked conceptual or definitional questions related to Java specifically, object-oriented programming concepts (abstraction, polymorphism, inheritance, and encapsulation), software design principles covered in class, or broader programming-related topics we've discussed.

7. You might encounter questions about big-oh runtime analysis, but I will not harp on the big-oh runtimes associated with all of Java's container classes, since we haven't explored how red-black trees and hash tables work in detail.

8. You might be asked conceptual or coding questions related to the container classes we've covered so far this semester: sets, lists, and maps. You will be expected to know the differences between those three types of containers, including comparative advantages and disadvantages so you can choose appropriate containers to solve novel problems on the exam. You might be asked to write code with any of the container classes we've covered: *ArrayList*, *LinkedList*, *TreeSet*, *HashSet*, *TreeMap*, and *HashMap*. (See also: Quiz #13.)

Review Topics

1. **Generics:** What are generics, and how and why are they used in Java? On the exam, you could be given a class and asked to make it generic, much like you did with the *BlackBoxOfChaos* assignment (although any class you're asked to make generic on the exam would be significantly less complex than the *BlackBoxOfChaos*). What is the syntax for restricting a type parameter to a particular class? What is the syntax for restricting a type parameter to a particular interface? What is the syntax for imposing multiple such restrictions? What limitations do generics have with respect to static methods? What limitation do generics have with respect to array creation (and where has that cropped up, and what technique have we seen for getting around that)? (That limitation is mentioned in "Side Note: Arrays of Generic Type (and Vindication)" in the Mar. 29 notes in Webcourses.) Where have we seen the need for the wildcard type parameter ("?"), and how is it used?

2. **Packages:** What is the syntax for declaring a package, and what limitations are there on package declarations? (Three limitations that come to mind are: (1) each *.java* file can declare at most one package; (2) if there's a package declaration in a *.java* file, it must be the first line of actual code (although it can be preceded by comments and whitespace); and (3) our package name must match the directory name where the *.java* file is located.) What conventions do we follow for naming packages? How do we compile and run a class that's contained within a package, and what special considerations have we seen when doing that from the command line? Syntactically, what are some options for importing different classes from different packages? Recall that one option is to use an import wildcard ("*"). How can we use a class from some other package without using any *import* statements, and why might we want to do that?

3. **Access Modifiers:** What are the differences between the *public*, *protected*, default, and *private* access modifiers? (In particular, be familiar with the table below, which shows the visibility of members with different access modifiers.) What does it mean for a class member to "package-private?" Recall that with the default access modifier, we do **not** write "default" explicitly; instead, we just leave the modifier off entirely.

	Class	Package	Subclasses	World
public	**yes**	**yes**	**yes**	**yes**
protected	**yes**	**yes**	**yes**	no
(default)	**yes**	**yes**	no	no
private	**yes**	no	no	no

   Note: There are some special exceptions involving protected *constructor* methods in Java, which I mention in the Mar. 31 notes, in a section titled, "(Supplementary) Unexpected Behaviors of Protected Members". You do **not** need to be familiar with those unexpected behaviors for the exam.

4. **Interfaces:** What is an interface? What do interfaces do, and why are they useful? Be familiar with the syntax for writing a custom interface. Syntactically, how do we enumerate the list of required methods for an interface? (Answer: We write method signatures with no bodies.) What naming conventions have we seen for interfaces? How do we have a class implement an interface, and what exactly does it mean to implement an interface? What happens if we claim to implement an interface but don't write all of that interface's required methods? What happens if we write all the required methods from some interface, but don't declare that our class implements that interface? Recall that we can sometimes use the name of an interface as if it were a superclass. We saw that specifically when we used *FancilyPrintable* to create an array, and we saw that again when using interfaces as generic type restrictions. With interfaces, when do we use *implements*, and when do we use *extends*? Be familiar with Java's *Comparable* interface. Remember that the *Comparable* interface is generic, so if you want class *X* to be *Comparable*, you write the following: *public class X implements Comparable<X>*. This specifies that *X* objects will only be compared to other *X* objects (and not to other types of objects). **Note:** There are some notes in Webcourses (Apr. 5, in the section titled, "(Supplementary) Other Fields and Methods in Interfaces") about special behaviors of fields and static methods in interfaces, which you do **not** have to be familiar with.

5. **Errors and Exceptions:** How do *try-catch* blocks work in Java? Be familiar with their syntax, as you might be asked to code them on the exam. What does a *finally* block do? What is the difference between *throw* and *throws* in Java? What is the different between *try-catch* blocks and simply throwing an exception? What is the syntax for throwing an exception? (Recall that we need to use the *new()* keyword to create and throw a new exception, since exceptions are objects.) Why does Java force us to explicitly declare some exceptions to be thrown by the methods that throw them, while others don't have to be declared as thrown? What is the difference between *checked* and *unchecked* exceptions? What are some specific examples of exceptions that we've seen in class so far? How do we create a custom exception? (See, e.g., the *ChaosException* from the *BlackBoxOfChaos* assignment.) In particular, be familiar with how to create an exception class and how to set up a constructor to tuck a custom message inside a hand-crafted exception. Also, if we create a new exception class, what is the practical implication of extending *RuntimeException* instead of just extending *Exception*? What is one "best practice" we've discussed for exception handling?

6. **File I/O:** There isn't much to know here. You should be able to create a new *Scanner* and set it up to read from either a file or from *System.in*. How do we read until the end of a file? How do we read an integer a *Scanner*, and how do we read a string? The only Scanner methods you need to know about (in addition to its constructor) are *hasNext()*, *next()*, and *nextInt()*. What happens if we use the *nextInt()* method when the next thing in our input stream isn't an integer? (See the Apr. 17 notes for an example.)

7. **Container Classes:** Be familiar with the differences between Java's *List*, *Set*, and *Map* containers. Recall that these are interfaces in Java (not classes). What are some of the comparative abilities of each of those container types? Given a novel problem to solve, could you select the most appropriate container type for solving that problem? For example, see Question #4 on Quiz #13. Note that there will likely be a mix of conceptual and applied questions related to container classes. You should be prepared to write code with all of the container classes we've covered so far: *ArrayList*, *LinkedList*, *TreeSet*, *HashSet*, *TreeMap*, and *HashMap*. What are some differences between *TreeSet* and *HashSet*? What about *TreeMap* vs. *HashMap*? How do we iterate through the elements in a list, set, or map? In particular, recall that we iterate over the key set in a map, which is fetched via the *keySet()* method. What is the difference between the *get()* methods for lists and maps? Recall that there is no *get()* method for sets. Recall also that sets and lists have *contains()* methods, but maps use *containsKey()* and *containsValue()*. Between the *containsKey()* and *containsValue()* methods for maps, which one is expected to be faster in the average case, and why? What restrictions are there on the types of elements we can place in a *TreeSet* or *TreeMap*? What restrictions are there on the types of elements we can place in a *HashSet* or *HashMap* (provided we want those containers to work properly and effectively)? What methods do we need to be sure to write if we create a custom class that we want to throw into a *HashSet* or *HashMap*? (You do not need to know exactly how to write those methods, but you should be aware that they exist and are essential to the functioning of a hash container.) Recall that a Java program is able to compile and run if we throw custom classes into a *HashSet* or *HashMap*, even if we don't write the two methods that allow hash containers to operate effectively on those classes. How does Java accomplish that (i.e., where does Java get those methods from if we don't write them ourselves)? Which container classes are amenable to *Collections.sort()* and *Collections.shuffle()*? What do we need to do if we want to ensure we can call those methods on one of our containers without crashing or failing to compile?

8. **Nested Containers:** You will want to be familiar with nested containers. What is the syntax for creating and manipulating, for example, an array list that contains sets of strings? What about a tree map that maps strings to array lists of integers? What about a map of integers to integers? For this, you'll want to be familiar with the "Mapping a Single Key to Multiple Values" example from the Apr. 14 notes , as well as the *PotionMaster* methods (Program #6).

9. **Other Things:** As mentioned in class and on pg. 2 of this PDF, there might also be questions on problem solving, object-oriented concepts, design principles, and Java tidbits.

Important Classes and Members to Commit to Memory

This section lists several classes and class members (both fields and methods) that you should be familiar with on the exam. You might have to produce these from memory and use them in code that you write on the exam in order to solve any number of problems. So, you need to not only remember that the classes and members below are available to you; you must also know what they do and the proper syntax for using them in code.

1. **The *System* Class*:*** Be familiar with *System.out.println()* and *System.out.print()*. What is the difference between these methods? What happens when we pass a reference to one of these methods? What about a *String* reference? What happens if we pass a string literal to one of these methods, and how do we get our output to include the values of variables? More specifically, you should be familiar with what the following line of code does:

   System.out.println("Max: " + maxValue);

   How does order of operations affect the output of a call to *println()* or *print()* when we're mixing arithmetic operations with string concatenation? For example, what is the output of each of the following?

   *System.out.println(5.1 + 3 / 2 + " Hello! " + 3 + 2.0 * 9);*

2. **The *Integer* Class:** You must be familiar with how to use *Integer.MAX_VALUE*, *Integer.MIN_VALUE*, *Integer.min()*, *Integer.max()*, and *Integer.parseInt()*. What common task have we used *Integer.parseInt()* for in class?

3. **The *Arrays* Class:** Why do we need an *import* statement in order to use the *Arrays* class in Java, but not for the other classes we've been using (*System*, *Integer*, *Math*, etc.)? What is the exact syntax for the *import* statement that allows us to use the *Arrays* class in Java? You must be familiar with the following methods from the *Arrays* class: *Arrays.sort()*, *Arrays.toString()*, *Arrays.copyOf()*, and *Arrays.fill()*.

4. **The *Math* Class:** You must be familiar with the following methods from the *Math* class: *Math.abs()*, *Math.ceil()*, *Math.floor()*, *Math.max()*, *Math.min()*, *Math.pow()*, and *Math.random()*. What is the difference between the *max()* and *min()* methods in the *Integer* class and the *max()* and *min()* methods in the *Math* class? (That was not covered explicitly in class. Take a look at the Java Docs, though, and see if you can spot a key difference.) What exactly does *Math.random()* return? How could you use *Math.random()* to generate integers on the range 0 through *n*? What about *m* through *n*? How could you use *Math.random()* to generate random alphabetic characters from 'a' through 'z'? What special typecasting considerations are there when using *Math.random()*? Given a line of code involving *Math.random()*, could you determine the precise range of values it generates (as in Questions #9 and #10 on Quiz #2)?

5. **The *Character* Class:** You do not have to memorize any of the methods in Java's *Character* class. If I want you to use any of those, I will give the Java Docs for those methods on the exam (including return type, method name, a list of method arguments, and a description of what the method does). You might want to glance through the following methods and play around with them to see what they do if you find the descriptions in the Java Docs inadequate: *isDigit()*, *isLetter()*, *isLetterOrDigit()*, *isLowerCase()*, *isUpperCase()*, *toLowerCase()*, and *toUpperCase()*.

6. ***The String Class:*** You must be familiar with the following methods from the *String* class: *String.charAt()*, *String.equals()*, and *String.length()*. Ideally, I would also like you to be familiar with *String.contains()*, *String.toLowerCase()*, *String.toUpperCase()*, and *String.valueOf()*.

7. **Container Classes:** You will want to be familiar with the methods we've talked about in the following container classes: *ArrayList*, *LinkedList*, *TreeSet*, *HashSet*, *TreeMap*, and *HashMap*.