CISC 3115 Midterm Review Guide

Readings

You are responsible for Chapters 1 through 9 of *Head-First Java, 2nd Edition*. The exam will focus, of course, on the new material from those chapters.

Study Tactics

Work as many exercises from the book as you can.
Review all the questions I've asked you in this class (see Application Activities appended below).
Review all code from exercises, CodeLabs, etc.
Re-read readings (and reading guides) now that you've worked with ideas in lecture/lab.

Topics and Targets

Unit 1

Learning Targets

- I can explain the motivation and structure of Java's "virtual machine" approach.
- I can distinguish among objects, references, and primitive values.
- I can explain the relationship between objects and classes.
- I can explain some of the advantages of object-oriented design and programming.
- I can write a simple class definition, as well as code that uses that definition.

Topics

Object-Oriented Design (Brad vs. Larry)
The Java Virtual Machine
primitive and reference types
writing a simple class (methods and instance variables)

Unit 2

Targets

- I can write a class definition that includes instance variables and methods that use those variables.
- I can distinguish between instance variables and local variables.
- I can argue for the importance of encapsulation in class design, and use the public and private keywords appropriately.
- I can write "enhanced" for loops and use them appropriately.
- I can describe the class development process, including the role of test code.
- I can find and the online Java API documentation in order to understand how to use a particular method in the API.

Topics

Pass-by-value
Encapsulation; `public` vs `private`.
More advanced object-oriented programming (e.g. developing the Dot-Com game)
enhanced `for` loops
casting

`ArrayList`
“short circuit” boolean operators
`import` and the Java package system

Unit 3

Targets

- I can explain what "polymorphism" means in the context of object-oriented programming, and I can explain how it is implemented in the Java language.
- I can use inheritance to create a class that is a modified form of another given class.
- I understand the usefulness of abstract classes, what "abstract" means, and how to use abstract classes to ultimately create "real" objects.
- I can create interfaces and write classes that implement those interfaces.
- I understand overloading and overriding.
- I can describe the "lifespan" of Java objects.
- I can design classes with constructors and implement those constructors efficiently and effectively.

Topics

Polymorphism—what is it, and why?
Inheritance: base class, superclass, subclass, IS-A, `extends`
Even more advanced object-oriented design (like pp. 170-174)
Overriding and overloading
`abstract` classes and methods
the `Object` class
Polymorphism and casting
Interfaces
Stack vs. heap
Constructors
Garbage collection

Exam Structure, Roughly

50%: short-answer/multiple-choice
30%: writing code
20%: analyzing code

Application Activities

(Note that some of these we didn’t do in class. The rest of the Unit 3 activities coming soon…)

1. Why is the Java Virtual Machine that runs Java programs called “virtual”?

   a) It is run over a network, rather than on the same physical machine on which it appears the Java program is running.
   b) It is a software machine, rather than the “real” hardware of the laptops and tablets in this classroom.
   c) It is a conceptual machine that is useful during the compilation process, but “real” hardware actually runs the program.
   d) It operates on *byte*codes, rather than low-level *bits*.

2. Which of these are advantages of the virtual machine approach?

a) Once a program is written, it can be run on any platform with a JVM.
b) More security checks can be made before a program runs.
c) More people can write Java programs.
d) Both a and b.
e) Both a and c.

3. If a variable is not primitive, it is a
a) Pointer
b) Object
c) Reference
d) Constant
e) Class

4. Which of the following types is not a primitive type?
a) `boolean`
b) `int`
c) `String`
d) `double`
e) `char`

5. A reference to an object is best thought of as
a) The address of the object in memory.
b) Another name for the object.
c) A way to get to the object in order to tell it to do something.
d) A way to restrict what the object can do.

6. Write a Java program that reads **exactly** 10 integers from the keyboard, stores them in an array, calculates the average, then outputs the list of numbers and their average. Write a method called `average()` that returns the average of an array of integers; use this to compute the average. (Assume the input is exactly 10 integers—don't worry about the possibility of bad input.)

9. OK, on to Brad and Larry. On the first iteration of their contest (p. 28) who wrote less code and was done more quickly?

a) Larry
b) Brad

10. On the second iteration (p. 29), it looks like they both wrote about the same amount of code. Who has the advantage, though, and why?

a) Larry: adding an if statement is much simpler than writing a whole new class.
b) Larry: his code is still in a single file, but Brad now has at least *four* files to deal with.
c) Brad: he can add new behavior (a whole new Shape!) without changing any previous code.
d) Brad: he knows what a `.hif` file is.

11. On the third iteration (p. 30), Brad pretty clearly has the upper hand. What fundamental

difference seems to be helping Brad here?

a) His initial focus on "things" gives him more flexibility when different kinds of things behave differently;
b) Brad tends not to work in a cubicle and therefore has a more artistic approach to writing code.
c) Brad's code is better at geometric calculations.
d) None of these; Brad is just a Smarter Programmer.

12. In the final analysis (pp. 31-2), why is Brad's approach more effective than Larry's?

a) Brad's design has a lot of duplicated code.
b) Brad's design is able to express relationships (like shared behavior) among different shape classes.
c) Brad's design can handle the Amoeba's different rotation, but Larry's can't.
d) Actually, in the final analysis, Brad's design *isn't* any better than Larry's.

13. The "Guessing Game," starting on p 38, is an example of a Java program organized around the idea of "objects talking to objects." Which of these object-to-object conversations does *not* happen in the program? (Note: some of these names are *object* names and some are *class* names, so technically this is a flawed question. After we discuss the answer, we'll discuss how to correct the question.)

a) GameLauncher — GuessGame
b) GameLauncher — Player
c) GuessGame — Player
d) GuessGame — System.out
e) Player — System.out

14. Write a class definition for the `Square` class. But this is not Brad and Larry's Square class. Our square is more concerned with geometry than with animation, so our squares "know" the length of their side, and they can "do" calculations of their area and their perimeter. Use the Dog class on p. 36 as a template.

---

15. When a Java program is running, how is information sent *to* a method (arguments/parameters) or *from* a method (return values)?

a) Java allows the calling method and the called method to share the values.
b) Java makes a copy: if it's a primitive value, it makes a copy of the value; if it's an object reference, it makes a copy of the object.
c) Java makes a copy of the bits, no matter what the bits represent.
d) All of a, b, c are true in different circumstances.

16. The basic principle of encapsulation is that generally, ...

a) Programmers should be able to change their minds.
b) Variables shouldn't take on inappropriate values.
c) Objects should hide their data.
d) All instance methods should be public.
e) All instance variables should be private.

17. Suppose I defined a `Counter` class this way:

```
class Counter {
      public:
            int value = 0; // not technically required, but good to be clear
            void increment() { value++; }
            void reset() { value = 0; }
            int getValue { return value; }
}
```

What, if anything, is the problem here?

a) Whoever's using a `Counter` object can reset the value of the `Counter` at any time!
b) The only way to change the value of the counter is to increment it—lame!
c) Someone could give the `Counter` a negative value.
d) There is no `setValue()` method.
e) Actually, there are no problems; this is a fine (if very simple) class.

18. Consider this code, which has some inadequate encapsulation:

```
class TenThings {
    int[] things;
    final int THINGCOUNT=10;
    void setThing(int index, int val) {
        things[index] = val;
    }
    int getThing(int index) {
        return things[index];
    }
}

class Thinger {
    public static void main(String[] args) {
        TenThings things = new TenThings();
        Arrays.fill(things.things, 27);   // put same value in all elements
        .
        .
        .
```

This code works, but it nonetheless has a serious encapulation-related problem. What is it?

a) Both the class and the `main` method have a variable called `things`.
b) The `THINGCOUNT` constant is not used by any of this code.
c) If the `TenThings` class is changed to use an `ArrayList` to store the things, `main` breaks.
d) Why would you intialize everything to 27??!
e) Relax… none of these are problems.

19. What is the best way of understanding the == operator?

a) It checks to see whether two things are equal.
b) It compares two patterns of bits to see if they're the same.

c) It can only be applied to primitive values.
d) It can only be applied to two values of the same type.
e) It can only be applied to `String` values.

20. When you start writing a Java program, what's the first thing you should do?

a) Identify all the "things" in the program's world, because these will become objects/classes.
b) Identify all the "procedures" in the program, because these will become methods.
c) Sketch how the program will run/behave to be sure you have a good grasp of what it's supposed to do.
d) Write pseudocode/prepcode so you can be sure you the right logic before you write "real" code.
e) Write test code.

21. Which of these strategies does the book use in its development of "Sink a Dot Com"?

a) Implement a simpler version of the application first.
b) Sketch a flowchart of the high-level behavior first.
c) Write code first.
d) Both a and b
e) Both b and c

22. The book suggests we write "prepcode" before we write test code, because it allows us to "focus on logic without stressing about syntax." Which of these are also good reasons to write prepcode before test code?

a) You might realize you need some additional methods.
b) You might realize you need some additional instance variables.
c) You might get ideas about tests to run (for example to make sure tricky logic is correct).
d) Both b and c.
e) All of a, b, and c.

23. The book describes "Extreme Programming (XP)" as "a newcomer to the software development methodology world." But the book was written in 2005. What happened to "Extreme Programming" since then?

a) Only a few companies were able to make use of it, and it pretty much died out.
b) It's popular among small companies, but no "big" software can be produced this way.
c) Many of the practices (e.g. in the box on p 101) have persisted, though the full "Extreme Programming" package isn't really used.
d) Extreme Programming has been totally replaced by Agile Programming.
e) Extreme Programming is one of several widely-used development methodologies.

24. So writing test code *very first thing* doesn't make sense, but the book does write test code before they start implementing the class. Why?

a) Writing test code helps you uncover errors in your design.
b) Writing test code is just more fun than writing classes.
c) Writing test code helps you focus on what the class code is supposed to do.
d) After writing test code, writing class code is incredibly pleasurable.
e) There is no good reason to write test code.

25. Which of these "Bullet Points" on p. 109 is incorrect?

a) Your Java program should start with a high-level design.
b) Choose `for` loops over `while` loops when you know how many times you want to repeat the loop code.
c) Use the pre/post decrement to subtract 1 from a variable (`x--`)
d) `Integer.parseInt()` works only if the `String` represents a digit ("0", "1", "2", etc.)
e) Use `break` to leave a loop early (i.e. even if the boolean test condition is still true).

26. What is the result of this code fragment?

```
int[] array = {1, 2, 3, 4, 5};
for (int i: array) {
      i = i + 1;
}

for (int i: array) {
      System.out.print(i + ", ");
}
```

a) `1, 2, 3, 4, 5,`
b) `1, 1, 1, 1, 1,`
c) `2, 3, 4, 5, 6,`
d) this doesn't compile
e) this has an error when it runs

27. Why do the book authors want to use `ArrayList` instead of regular arrays for their code?
a) Regular arrays can't hold negative numbers.
b) Regular arrays can't change their size.
c) Regular arrays make you use that annoying [square bracket] notation.
d) Regular arrays can only hold values of primitive types.
e) Regular arrays can't have zero size.

28. What is the best way to write an `if` statement that will do something only when the `ArrayList list` contains no elements or when its first element is `0`?

a)
```
if (list.isEmpty() == true || list.get(0) == 0)
```

b)
```
if (list.isEmpty() == true | list.get(0) == 0)
```

c)
```
if (list.isEmpty()  || list.get(0) == 0)
```

d)
```
if (list.isEmpty()  | list.get(0) == 0)
```

29. What is the purpose of the `import` statement?

a) It makes the code of other classes (like those in the Java API) available when your program runs.
b) It saves you from typing annoyingly long class names, like `javax.security.auth.kerberos.DelegationPermission`
c) It tells the compiler that your code relies on other classes.
d) It allows you to put your class in a package of your choosing.
e) It allows you to use classes that aren't part of the Java API at all.

30. Which of these is *not* an advantage of using packages in Java?

a) They help organize the code in an application or a library.
b) They help prevent problems if two people working on an application decide to use the same class name.
c) They help reduce the size of your (compiled) program.
d) They help improve security by allowing you to limit access to your code.

31. Imagine a program that reads an arbitrary number of `Strings` in from the input (where each line of input is treated as a `String`), then outputs: the *shortest* `String` (if multiple `Strings` have the shortest length, output the one that occurs first in the input), the *longest* `String` (again, if multiple `Strings` have the longest length, output the first-occurring one), and print the `Strings` in alphabetical order.

(A) Of those three outputs, which require me to store all the input?

a) Print the shortest
b) Print the longest
c) Print the sorted list
d) Both a and b
e) All of a, b, and c

(B) Now write this program. But DO NOT WRITE the sorting code yourself! Hint: what do you find if you use the API search box to search for "sort"?

32. Considering the diagram on p. 167, if Brad and Larry's boss were to add a Rectangle to the mix, where should it go?

a) On the same level as the other four shapes.
b) As a subclass of Square.
c) As a subclass of Shape but a superclass of Square.
d) Not enough information given.

33. Considering the three classes on p. 169, which of these are *not* methods of the `FamilyDoctor` class?

a) `treatPatient()`
b) `makeIncision()`
c) `giveAdvice()`
d) Neither a nor b are.
e) Neither b nor c are.

34. When designing an inheritance tree for a set of classes/types, what are the fundamental questions to answer?

a) What are the nouns/things in the problem?
b) What do the types have in common?
c) How are the types related?
d) All of the above
e) Both b and c

35. Earlier, we talked about how instance variables are what an object "knows" and methods are what an object "does." Do any of these `Animal` instance variables (from p. 171) seem not to fit with what an animal knows?

a) `picture`
b) `food`
c) `boundaries`
d) `location`
e) None of the above

36. Which of the following statements is supported by the diagram on p. 174?

a) Felines all make noise the same way.

b) Felines and Canines have no common behavior.
c) Felines all roam the same way.
d) Dogs and Wolfs (?) roam in the same pack.
e) Hippos eat grass.

37. Draw an inheritance diagram of the "Sharpen your pencil" classes on p. 176: Musician, Rock Star, Fan, Bass Player, Concert Pianist. Note that not every class needs to be in the same inheritance tree. Be prepared to defend your design decisions.

38. Suppose I implement some of the roam() methods in the Animal hierarchy:

```
class Animal {
      void roam() {    System.out.print("move."); }
}

class Feline {
      void roam() {   System.out.print("calculate vector away from other
                            Felines... ");
                      super.roam(); }
}

class Cat {
      void roam() {    System.out.print("visit litter box... ");
                       super.roam(); }
}
```

What is the output of

```
      Cat c = new Cat();
      c.roam();
```

a) `visit litter box...`
b) `move. calculate vector away from other Felines... visit litter box...`
c) `visit litter box... calculate vetor away from other Felines...`
d) `visit litter box... calculate vector away from other Felines... move.`
e) `visit litter box... move.`

39. Consider the two (very deep) statements

X IS-A Z.
Y HAS-A Z.

Find appropriate X and Y for each of the following Z. (For example, if Z is Doctor, then X could be Surgeon ("Surgeon IS-A Doctor" and Y could be Hospital ("Hospital HAS-A Doctor").

Employee.
Book.
File (i.e. a disk file).
Team.
Polygon.

40. In general, overridden methods should provide

a) Behaviors unique to the subclass.
b) Behaviors common to all subclasses.
c) Behaviors that may change in future implementations.
d) Behaviors that add to the base class behavior.
e) Behaviors that the JVM cannot support.

41. Consider the following code:

```
class Hamburger {
 protected int calories = 800;
      int getCalories() {
           return calories;
      }

}

public class Cheeseburger extends Hamburger {
      protected int cheeseSlices = 1;
      void setSlices(int cheeseSlices) {
           this.cheeseSlices = cheeseSlices;
      }

      int getCalories() {
           return super.getCalories() +
                      100 * cheeseSlices;
      }
}
```

What is the result of trying to compile and run the following code?

```
Hamburger[] breakfast = new Hamburger[4];
breakfast[0] = new Hamburger();
breakfast[1] = new Hamburger();
breakfast[2] = new Cheeseburger();
breakfast[3] = new Cheeseburger();

int calories=0;
for (Hamburger h : breakfast) {
      calories += h.getCalories();
}
System.out.println("Your total calories for the meal: " + calories);
```

a) 3200
b) 3300
c) 3400
d) 3600
e) failure

42. An abstract class cannot be

a) Subclassed
b) Changed
c) Instantiated
d) Overridden
e) Compiled

43. Why should the book's `Animal` class be declared `abstract`?

a) Because there are no real animals in the world.
b) Because there are too many real animals in the world.
c) Because `Animal` has no subclasses.
d) Because there's no way to give specific behavior for a general "animal."
e) Both c and d.

44. What's wrong with following code fragment?

```
ArrayList<Animal> myDogArrayList = new ArrayList<Animal>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
Dog d = myDogArrayList.get(0);
```

a) You cannot declare an `ArrayList` of `Animal`s.
b) `get()` returns an `Animal` reference that can't be assigned to a `Dog` reference.
c) It's illegal to add a `Dog` object to an `ArrayList` of `Animals`.
d) Dog Adog New Dog is a terrible name for a band.
e) There's nothing wrong.

45. Which of these is the best way to understand "interface"?

a) It's a collection of instance variables.
b) It's a completely abstract class.
c) It's a role that any class can play.
d) It's a universal polymorphic type.
e) It's a good way to avoid the problems of multiple inheritance