CS100J About Prelim III:
Tuesday, 7:30–9:00PM, 14 November, Olin 155

Review session: Sunday, 12 November, 1:00--3:00. Philips 101

You should know everything that you needed to know for the first two tests —we review this below. Also:

While loops and for loops. We will give you a precondition, postcondition, and loop invariant, and you will have to develop the loop, with initialization, from it. The grade will depend on how well you deal with the four loopy questions **and the given invariant**. Please remember that the only variables that should be used (outside of variables declared in the repetend) are variables that are mentioned in the loop invariant.

Arrays. Everything on Sects 8.1 and 8.2 of the class text —these pages discuss the technical details for using arrays in Java and for reasoning about arrays. Two dimensional arrays: You should know how to declare and use rectangular arrays and ragged arrays; this requires knowing how arrays are stored. Sects. 9.1, 9.2, 9.3.1, 9.4.

Algorithms. You should know the following algorithms. For example, if we ask "what is algorithm partition", you should be able to write the precondition and postcondition (we won't give them to you), write down the invariant, and then develop the loop with initialization. These algorithms are discussed in the text, on the ProgramLive CD, or in lectures notes:

Algorithm Linear search (Sec. 8.5.1)
Algorithm Binary search (Sec. 8.5.3)
Algorithm Find minimum (Sec. 8.5.1)
Algorithm Partition (PLive activity 8-5.5)
Algorithm Selection sort (Sec. 8.5.4)
Algorithm Insertion sort (Sec. 8.5.5)

On selection sort and insertion sort, we will NOT want to see the inner loops. The repetends of these algorithms should be written at a high level, stating *what* is done and not how it is done, as discussed in the text and in lecture. Since you know that at least one of these will be on the test, please *practise* developing all of them. Answering a question on one of these should take 5 minutes, because you *know* how to develop them.

Classes String and Vector. We use these all the time. You are expected to know the basic methods of these classes: For String, charAt(i), substring(i), substring(i,j), length(). For Vector: add(ob), set(i, ob), get(i), size(). If a question calls for other methods of the classes, we will define them for you. You should know that **new** Vector<Cat>(...) creates a Vector whose elements are of class <Cat>.

With regard to classes and subclasses. You will have to write parts of classes/subclasses as on prelims II and II. Two ways to study:

(1) Make sure you know the definitions below, and

(2) Practice writing classes and subclass definitions.

Apparent and real types of a variable are important in understanding what components can and cannot be referenced. Take a look at other texts on Java and see what exercises they have for writing classes. Also, know that the definitions of the fields, together called the "class invariant", has to be kept true by all methods. Finally, when writing classes, write them in DrJava so you can test your syntax.

GUIS. We use three containers for components: JFrame (BorderLayout manager), JPanel (FlowLayout), and Box (BoxLayout). You should know the default layout manager of each and how to place elements in the container. You don't have to know the names of the components we might want to place (like JButton or JLabel). But you should be able to write a sequence of code to put a component into each of the three containers mentioned above. You do not have to know how to listen to a GUI event.

Definitions. Below is a collection of definitions. You are expected to know these backward and forward. On the test, wishywashy definitions will not get much credit. They must be the same as, or similar to, the ones below —and they must be correct. Learn these not by reading but by practicing writing them down, or have a friend ask you these and repeat them out loud. You should be able to write programs that use the concepts defined below, and you should be able to draw folders and execute method calls, drawing the frames for the calls.

Variable: A named box that can contain a value of some type or class. For a type like **int**, the value is an integer. For a class, it is the name of (or reference to) an instance of the class —the name that appears on the folder.

Declaration of a variable: a definition of the name of the variable and the type or class of value it can contain. Basic syntax: *type variable-name* ;

Four kinds of variables: parameter, local variable, instance variable (or field), static variable (or class variable).

Parameter: A variable that is declared within the parentheses of a method header. The variable is drawn in a frame for a call on the method.

Local variable: A variable that is declared in the body of the method. The variable is drawn in a frame for a call on the method.

Instance variable: A variable that is declared in a class without modifier static. An instance variable is placed in every folder of the class.

Static variable: A variable that is declared in a class with modifier static. A static variable is placed in the file drawer for the class in which it is declared.

Three kinds of method: procedure, function, constructor:

A procedure definition has keyword **void** before the procedure name. A procedure call is a statement.

A function definition has the result type in place of **void**. A function call is an expression; its value is the value returned by the function.

Argument: An expression that occurs within the parentheses of a method call (arguments are separated by commas).

A constructor definition has neither keyword **void** nor a type, and its name is the same as the name of the class in which it appears. The constructor call is a statement, whose purpose is to initialize (some of) the fields of a newly created folder.

Folder (manila folder, object, or instance) of a class. An entity that is drawn like a manila folder. It has a name or label on its tab. Its contents are the instance methods and instance fields defined in the class definition.

New-expression. An expression of the form "**new** class-name (arguments)". It is evaluated as follows:

(1) create a new folder of class class-name and put it in class-name's file drawer.

(2) Execute the constructor call "class-name (arguments)"; where the method called appears in the newly created folder.

(3) Yield as the result of the new-expression the name of the folder created in step (1).

Frame for a method call. The frame for a method call contains:

(1) the name of the method and the program counter, in a box in the upper left,

(2) the scope box (see below),

(3) the local variables of the method,

(4) the parameters of the method.

The scope box for a call contains: For a static method, the name of the class; for an instance method, the name of the folder in which the instance appears.

To execute a method call:

1. Draw a frame for the call. (Fill in its name and program counter in the upper left box. Fill in its scope box on the upper right with either the name of the class in which the method is defined (static method) or the name of the folder in which the method appears (non-static method). Draw local variables and parameters.)

2. Assign the values of the argument to the parameters.

3. Execute the method body. When a name is used, look for it in the frame for the call. If it is not there, look in the place given by the scope box.

4. Erase the frame for the call.

Folder. We assume you can draw a folder, or instance, of a class. For subclasses, remember that the folder has more than one partition. Look at the homework we had on drawing folders.

Class Object. Every class that does not explicitly extend another subclass automatically extends class Object. Class Object has at least two instance methods: toString and equals.

Calling one constructor from another. In one constructor, the first statement can be a call on another constructor in the same class (use keyword **this** instead of the class-name) or a call on a constructor of the superclass (use keyword **super** instead of the class-name).

Overriding a method. In a subclass, one can redefine a method that was defined in a superclass. This is called overriding the method. In general, the overriding method is called. To call the overriden method m (say) of the superclass, use the notation **super**.m(...).

Real and apparent class. A variable x defined using, say, C x; has apparent class C. *Apparently*, the type is C. The apparent class is used in determine whether a reference to a field or method is syntacally legal or not. One can write x.m(...), for example, if and only if method m is declared or is referenceable in class C.

The real class of x is the class of an object that is in x. *Really*, it contains the name of an object of class x. It could be a subclass. If x.m(...) is legal, then it calls the method that is accessible in the real class, not the apparent class.

Casting and instanceof. Just as one can cast an **int** i to another type, using, say, (**byte**) i or (**double**) i, one can cast a variable of some class-type variable to a superclass or subclass. You should know how to use operator **instanceof**. See Sect. 4.2 and 4.3 of the text.