<u>1302 FINAL EXAM STUDY GUIDE</u>

May 1, 2014 | 08:00-11:00
Same Room As Lecture

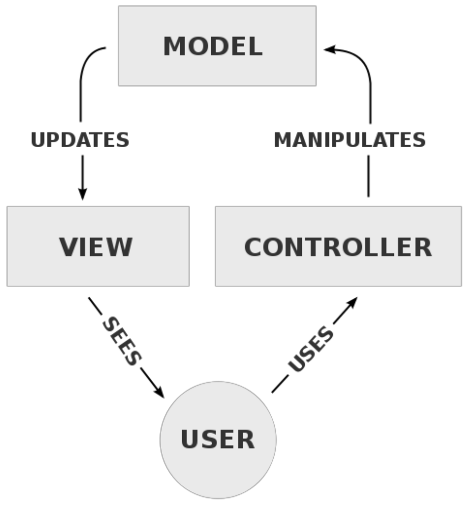
Random Questions:

Three part exam: true/false
short answer
general code-based questions
(+bonus punny question)

Topics:

1. <u>Model-View Controller & HCI Principles</u>

a. Purpose:
   i. Completely separates the Calculations and interface from each other;
      1. Model: Is for data, the methods used, and provides access to the data
      2. View: The interface (think gui; buttons clicked, text fields)
      3. Controller: Coordinates interactions between view and model
   ii. Extra explaination: https://www.youtube.com/watch?v=dTVVa2gfht8
b. Comprised Of 3 Components:
   i. Model - notifies associated views and controllers when there has been a change in its state
      1. Notification allows views to produce updated output and the controllers to change the available set of commands
   ii. View - requests information from the model that it needs for generating an output representation to the user
   iii. Controller - sends commands to the model to update the model's state
         a. Ex. editing a document
      2. Can also send commands to its associated view to change the view's presentation of the model
         a. Ex. Scrolling through a document



2. Information Visualization

a. InfoVis' Goal
    i. Transform the data into information (understanding, insight) thus making it useful to people
    ii. Provide tools that present data in a way to help people understand and gain insight from it
    iii. Cliches
        1. "Seeing is believing"
        2. "A picture is worth a thousand words"
    iv. Tasks: analysis, assimilation, monitoring, awareness
b. InfoVis' Tasks
    i. Search
        1. Finding a specific piece of information
    ii. Browsing
        1. Look over or inspect something in a more casual manner, seeking interesting information
    iii. Analysis
    iv. Monitoring
    v. Awareness

3. Swing

a. GUI - the user is not limited to responding to prompts in a particular order and receiving feedback in one place

b. Three kinds of objects are needed to create GUI:
    i. Components
        1. An object that defines a screen element used to display information or allow the user to interact with the program

            a. A *container* is a special type of component that is used to hold and organize other components
        2. Example:
            a. Text Fields - Allows user to enter typed input
                i. Generates an action event when the Enter or Return key is pressed
                    1. Push button and text field generate the same kind of event (action event)
                ii. An alternative implementation could involve adding a push button to the panel which causes the conversion to occur when the user pushes the button
            b. Check Boxes - A button that can be toggled on or off
                i. Generates an item event when it changes state from selected to deselected (or vice versa)
            c. Radio Button - A button that can also be toggled on or off
                i. Used with other radio buttons to provide a set of mutually exclusive options
                ii. Produce an action event when selected
                iii. ButtonGroup class is used to define a set of related radio buttons
            d. Sliders - Allows the user to specify a numeric value within a bounded range
                i. Can be presented either vertically or horizontally
                    1. Features: tick marks or label indicating the range of values
                ii. Produces a change event; indicating that the position of the slide and the value it represents has changed
            e. Combo Boxes - Allows the user to select one of several options from a pull down menu
                i. When pressed a list of options is displayed from which the user can choose one
                ii. Generates an action event whenever the user makes a selection from it
            f. Timer - Helps manage an activity over time, has no visual representation
                i. Defined by the Timer class and are provided to help manage an activity over time
                    1. Generates an action event at regular intervals

ii. *Events*
    1. An object that represents some occurrence in which we may be interested
        a. Often is used to user actions
            i. ie: mouse button press, keyboard, key press
        b. Most GUI components generate events to indicate a user action related to that component
        c. *Event-driven* is a program that is oriented around GUI, responding to user events

iii. *Listeners*
    1. An object that "waits" for an event occur and responds in way when it does

            a. In designing a GUI-based program we need to establish the relationships between the listener, the event it listens for, and the component that generates the event

c. Key elements to creating a java program that uses GUI:
    i. Instantiate and set up the necessary components
    ii. Implement listener classes that define what happens when particular events occur
    iii. Establish the relationship between the listeners and the components that generate the events of interest
    iv. The java components and other GUI-related classes are defined primarily in two packages:
        1. java.awt
            a. The Abstract Window Toolkit was the original java GUI package
        2. javax.swing
            a. Provides components that are more versatile than those of AWT

d. Containers are either:
    i. HeavyWeight
        1. Managed by the underlying operating system
            a. Examples: a frame, an applet(used to display and execute a java application through a Web browser)
    ii. LightWeight
        1. Managed by the java program; More complex than heavy weight containers
            a. Example: a panel

e. Example methods:
    i. setDefaultCloseOperation
        1. Determines what will happen when the close button in the corner of the frame is clicked
    ii. getContentPane
        1. The content pane of the frame is obtained
    iii. add
        1. The content pane is added; also allows a component to be added to the panel
    iv. pack
        1. sets the frame size according to the components inside (sets the frame size to perfectly fit all the components inside the frame).
    v. PushCounter
        1. it displays label (can be used to also display an image)
        2. push button (generates an action event)
    vi. ButtonListener
        1. Represents the action listener for the button
        2. Was created as an inner class (defined inside of another class)
            a. They have access to the members of the class that contains it
        3. ButtonListener implements the ActionListener interface
    vii. ActionListener interface

1. Only method listed is the actionPerformed method
2. The button component generates the action event resulting in a call to the actionPerformed method, passing an ActionEvent object

f. Layout Manager
   i. The default layout manager for a panel simply displays components in the order they are added, with as many components on one line as possible
      1. Determines the size and position of each component
      2. Each Layout Manager has its own rules and properties governing the layout of the components it contains
         a. setLayout method of a container is used to change its layout manager
   ii. Every container has a default layout manager, but we can replace it if desired
   iii. Predefined Managers:
      1. Border
         a. Organizes components into five areas (North, South, East, West and Center)
            i. The four outer areas are as large as needed in order to accommodate the component they contain
               1. The Center area expands to fill any available space
            ii. If no components are added to a region, the region takes up no room in the overall layout
      2. Box
         a. Organizes components into a single row or column
            i. When combined with other layout managers it can produce complex GUI designs
            ii. Components are organized in the order in which they are added to the container
            iii. There are no gaps between the components UNLESS you specify it by adding an area using JPanel.add(Box.createRigidArea(new Dimension(x,y)).
      3. Card
         a. Organizes components into one area such that only one area is visible at a time
      4. Flow
         a. Organizes components from left to right, starting new rows as necessary
            i. One of the easiest managers to use
            ii. Puts as many components as possible on a row at their preferred size;
               1. When a component can not fit on a row it is put on the next row
      5. Grid
         a. Organizes components into a grid of rows and columns
            i. One component is placed in each cell, and all cells are the same size

                ii. The number of rows and columns in the layout is established by using parameters to the constructor when the layout manager is created
                iii. As components are added to the layout, they fill the grid from ***left to right, top to bottom***
                iv. There is no way to explicitly assign a component to a particular location in the grid other than the order in which they are added to the container
        6. GridBag
            a. Organizes components into a grid of cells, allowing components to span more than one cell

g. Containment Hierarchies:
    i. Definition: The way components are grouped into containers, and the way those containers are nested within each other
        1. Generally one primary (Top-level) container (such as frame or applet)
            a. The top-level container often contains one or more contains (such as panels)
                i. These panels may contain other panels to organize the other components as desired

h. Mouse and Key Events:
    i. Mouse Events
        1. Mouse Events: Occur when the user interacts with another component via the mouse (implement the MouseListener interface class)
            a. Event Examples: mouse-pressed, released, clicked, entered, exited
        2. Mouse Motion Events: Occur while the mouse is in motion (implement the MouseMotionListener interface class)
            a. Motion Examples: mouse - moved, dragged
    ii. Key Events
        1. Key Event: is generated when the user presses a keyboard key
            a. Allows a program to respond immediately to the user while they are typing or pressing other keyboard keys
    iii. Extending Adapter Classes:
        1. An alternative technique for creating a listener class is to use inheritance and extend an adapter class
            a. Each listener interface that contains more than one method has a corresponding adapter class containing empty definitions for all methods in the interface
        2. Example: The MouseAdapter class implements the MouseListener interface class and provides empty method definitions for the five mouse event methods
            a. By subclassing MouseAdapter, we can avoid implementing the interface directly
                i. Can save time and keep easy readability for our source code

i. Dialog Boxes
   i. *Dialog box* is a graphical window that pops up on top of any currently active window so that the user can interact with it
      1. Can serve a variety of purposes:
         a. Conveying information
         b. Confirming an action
         c. Permitting the user to enter information
   ii. JOptionPane dialog boxes fall into three categories:
      1. Message
         a. Displays an output string
      2. Input
         a. Presents a prompt and a single input text file into which the user can enter one string of data
      3. Confirm
         a. Presents the user with a simple yes-or-no question
   iii. *File choosers* is a specialized dialog box used to select a file from a disk or other storage medium
      1. Automatically presents a standardized file selection window
      2. Filters can be applied to the file chooser programmatically

j. Borders
   i. A *border* is not a component but defines how the edge of a component should be drawn
      1. Provides visual cues as to how GUI components are organized
         a. BorderFactory class is useful for creating borders for components
   ii. *Tool tip* is a short line of text that appears over a component when the mouse cursor is rested momentarily on top of the component
      1. Usually inform the user about the component
      2. button.setToolTipText("INSERT TEXT");
   iii. A *mnemonic* is a character that allows the user to push a button or make a menu choice using the keyboard in addition to the mouse
      1. Once set (setMnemonic method) a character in the label will be underlined to indicate that it can be used as a shortcut

k. GUI Design
   i. Fundamental ideas of good GUI design:
      1. Knowing the user
      2. Preventing user errors
      3. Optimizing user abilities
      4. Being consistent
   ii. The design focus should be that the interface will cause the user to make as few mistakes as possible

---

4. Applets

a. There are two flavors of Java
    i. Applications
        1. standalone program
        2. executed using an interpreter or virtual machine
    ii. Applet (means small application)
        1. Program that is intended to be embedded into an HTML doc, transmitted across a network, and executed using a web browser
        2. Another type of media that can be transmitted across the WWW
        3. Objects are derived from class JApplet (the object is a container)

b. Principles of Applets
    i. Applets do not use constructors
    ii. Initialization code placed in method init()
        1. Executes once an applet is first loaded (you should initialize data)
    iii. Normal way to run an applet is as part of a web page or with an applet viewer
        1. Appletviewer deals only with applet code and not HTML
    iv. Web page structure:
        1. All pages consist of a header and body
            a. Header contains:
                i. Page title
                ii. Keywords (to guide search engines)
                iii. Style information to control appearance

c. Applet Security
    i. Applets that are not signed using a security certificate are considered to be untrusted and referred to as unsigned applets
    ii. When running on a client, unsigned applets operate within a security sandbox that allows only a set of safe operations
    iii. Applets can be signed using a security certificate to indicate that they come from a trusted source
        1. Signed applets operate outside the security sandbox and have extensive capabilities to access the client
            a. Runs outside the security sandbox only if the user accepts the security certificate
    iv. Unsigned Applets Can:
        1. Make network connections to the host they came from
        2. Display HTML documents
        3. Invoke public methods of other applets on the same page
        4. Read secure system properties
    v. Unsigned Applets Cannot:
        1. Cannot access client resources such as the local file system, executable files...
        2. Cannot connect to or retrieve resources from third party servers
        3. Cannot load native libraries
        4. Cannot change the SecurityManger
        5. Cannot create a ClassLoader
        6. Cannot read certain system properties

d. Fractals:
    i. *Fractals* are geometric shape that are made up of the same pattern, but at different orientations and different sizes
        1. They lend well to recursion and are easy to generate

---

5. <u>Java Graphics</u>

a. Terms:
    i. Pixels: picture elements
    ii. Resolution: how many pixels
    iii. Color depth: how many bits per pixel
    iv. Image file size: is equal to resolution times color depth

b. How many colors are possible with 8 bits (1 byte) that represent each component of the RGB triple? 256 x 256 x 256 = 16,777,216 colors

c. Graphics:
    i. Has Methods for: drawing shapes, lines, rectangles, ovals and methods for filled or unfilled

d. PaintComponent
    i. public void paintComponent(Graphics g)
        1. The g is a graphics context that you interact with
    ii. super.paintComponent
        1. It overrides the parent method's paintComponent method, which means that all commands interacting with "g" after that call will paint over what has already been drawn on the component

e. Texture Mapping
    i. A method to apply detail or texture to the surface of a shape or polygon

f. InfoVis
    i. Goal of InfoVis (seeing is believing):
        1. Transform the data into information thus making it useful to people
        2. Provide tools that present data in a way to help people understand and gain insight from it
    ii. Task of InfoVis
        1. Finding a specific piece of information
        2. Browsing
            a. Look over or inspect something in a more casual manner, seek interesting information
        3. Analysis
        4. Monitoring

        5. Awareness
    iii. Components of InfoVis
        1. Taking items without a direct physical correspondence and mapping them to a 2-D or 3-D space
        2. Giving information a visual representation that is useful for analysis and decision making
        3. Scale
        4. Interactivity

---

6. <u>Complexity</u>

a. Growth Functions and Big-Oh
    i. For every algorithm we want to analyze, we need to define the size of the problem
        1. Downloading a file: the size of the file is the size of the problem
        2. Searching for a target value: the size of the search pool is the size of the problem
    ii. The overall amount of time spent at a task is directly related to how many times we have to perform the key processing step
    iii. *Algorithm efficiency* can be defined in terms of problem size and the key processing step
    iv. A *growth function* represents the time complexity (or space complexity) of an algorithm
        1. It is not typically necessary to know the exact growth function for an algorithm
            a. We are mainly interested in the asymptotic complexity of an algorithm (the general nature of the algorithm as *n* gets bigger)
                i. *Asymptotic complexity* (called the order of the algorithm) is based on the dominant term of the growth function; the term that increases most quickly as *n* increases
                ii. The chart below represents Growth Functions and Big-Oh

Growth Function	**Order**	**Complexity**
t(n) = 17	O(1)	Constant time
t(n) = 20n - 4	O(n)	Linear time
t(n) = 12n log n + 100n	O(n log n)	Logarithmic Time
t(n) = 3n	O(n^2)	Quadratic Time
t(n) = 2	O(2^n)	Exponential Time

b. Comparing Growth Functions
    i. Optimizing an algorithm by tweaking it so it is more efficient is good, however, finding a better algorithm that takes you into a different Big-Oh category is better (has a bigger impact)
    ii. Faster processors and most optimizations add constant factors to the timing function, which we throw away when determining Big-Oh efficiency

c. Loop Analysis
   i. When loops are nested, we must multiply the complexity of the outer loop by the complexity of the inner loop
      1. Example:
```
for(int count = 0; count < n; count++)
    for(int otherCount = 0; otherCount < n; otherCount++)
    {
        //Some sequence of O(1) steps
    }
```
      Both the inner and outer loops have complexity of O(n);
      And multiplied together the order becomes O(n^2)

d. Recursive Analysis
   i. Determine the order of the recursion (The number of times recursive definition is followed) and multiply it by the order of the body of the recursive method
      1. Example:
```
public int sum(int num)
{
        int result;
        if(num == 1)
        {
                result = 1;
        }
        else
        {
                result = num + sum (num - 1);
        }
        return result;
}
```
      When we have an if-else statement we determine the average and worst case scenarios. In this particular example, worst case would be O(n) due to the else statement.

e. Caching
   i. A processor's cache can influence execution times
      1. Cache is a form of fast memory near the processor
      2. When a processor generates address of a data item it needs to access, it asks the cache if it contains a copy
   ii. Temporal locality
      1. An assessed element is likely to be accessed again in the near future
   iii. Spatial locality
      1. Items close together in space (memory) are likely to be accessed close together in time

7. Iterators

a. An *iterator* is an object that is used to process each element in a collection one at a time
   i. The Iterable interface is implemented by a collection to formally commit to providing an iterator when it is needed

ii. The Iterator interface is implemented by an interface and provides methods for checking for, accessing, and removing elements.

iii. For-each loops and Iterators

1. A for-each loop can be used only with collections that implement the Iterable interface. It is a syntactic simplification that can also be accomplished using an iterator explicitly.
2. You may need to use an explicit iterator rather than a for-each loop or you don't plan on processing all elements in a collection or if you may use the iterator's remove method

iv. Fail-Fast

1. An iterator is fail-fast when it will fail quickly and cleanly if the underlying collection has been modified by something other than the iterator itself
2. An iterator notes the modifications count of the collection when it is created and on subsequent operations makes sure that the value hasn't changed. If it has, the iterator throws a CurrentModificationException

---

8. Collections

a. **Collection**: is an object that gathers and organizes other objects; serves as a repository for other objects; provides services to add, remove and manage the elements it contains

i. Can be separated into two broad categories:

1. linear
   a. one in which the elements of the collection are organized in a straight line
2. nonlinear
   a. is one in which the elements are organized in something other than a straight line, such as a hierarchy or a network; may not have any organization at all

ii. Organization of the elements in a collection, relative to each other are determined by one of two things:

1. the order in which they were added to the collection
2. Some inherent relationship among the elements themselves

b. **Abstraction**: a collection is an abstraction where the details of the implementation are hidden.

c. **Data Type**: is a group of values and the operations defined on those values; primitive data types defined in java are the primary examples;

d. **Abstract data type (ADT)**: a data type whose values and operations are not inherently defined within a programming language; a collection is an abstract data type

e. **Data Structure**: the collection of programming constructs used to implement a collection;

i. Example: a collection might be implemented using a fixed size structure such as an array.

f. **Java Collections API** (application programming interfaces): is a set of classes that represent a few specific types of collections, implemented in various ways;

i. Reasons to learn how to design and implement collections even though they are already provided:

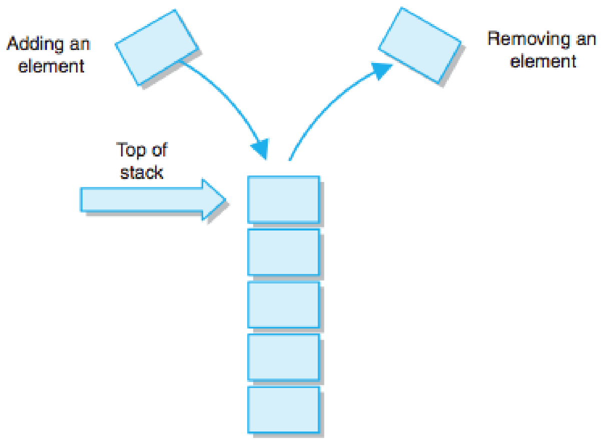
1. Java API provides only a subset of the collections you may want to use
2. Classes that are provided may not implement the collections in the way you desire

3. The study of software development requires a deep understanding of the issues involved in the design of collections and the data structures used to implement them

---

9. <u>Stacks</u>

a. Stack (14.1-14.3): A stack is a linear collection whose elements are added and removed from the same end
   i. It is a LIFO: last in, first out; That is the last element to be put on a stack will be the first one that gets removed (Think of the undo operation)
   ii. Usually a stack is depicted vertically, and the end which elements are added and from which they are removed is known as the top of the stack
      1. Refer to Figure 1



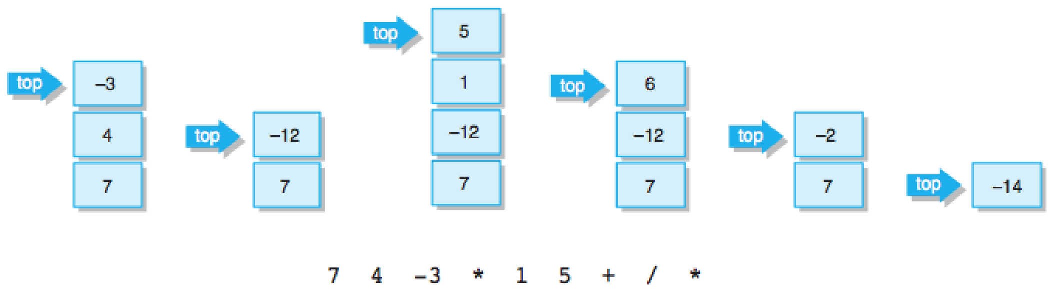
   iii. Operations for a stack
      1. push - Adds an element to the top of the stack
      2. pop - Removes an element from the top of the stack
      3. peek - Examines the element at the top of the stack
      4. isEmpty - Determines if the stack is empty
      5. size - Determines the number of elements on the stack
   iv. SUBTOPICS:
      1. A polymorphic reference uses the type of the object, not the type of the reference, to determine which version of a method to invoke.
      2. Generics: we can define a class so that it stores, operates on, and manages objects whose type is not specified until the class is instantiated

b. A Stack ADT (14.4)
   i. Stack interface is defined as Stack<T>, operating on a generic type T.

1. the type of various parameters and return values are often expressed using the generic type T. When this interface is implemented, it will be based on a type that is substituted for T
2. By using the interface name as a return type, the interface doesn't commit the method to the use of any particular class that implements a stack.

c. *Using Stacks: Evaluating Postfix Expressions (14.5)*

   i. Traditionally, arithmetic expressions are written in infix notation, meaning that the operator is placed between its operands in the form:
      1. ***<operand><operator><operand>*** 4+5
   ii. In a postfix expression the operator comes after its two operands like:
      1. ***<operand><operand><operator>*** 6 9 - which is equivalent to 6-9
   iii. A **postfix** expression is generally easier to evaluate than a **infix** expression because precedence rules and parentheses do not have to be taken into account. Thats why compiler often use postfix expressions in their internal calculations for this reason.
      1. The process of evaluating a postfix expression can be easy following this rule:
         a. scan from left to right, apply each operation to the two operands immediately preceding it and replace the operator with the result
         b. EXAMPLE: 4 + 5 * 2 in postfix notation would be 4 5 2 * +
            i. using the rules scan left until we encounter the * operator; Then apply this operator to the two operands that are immediately preceding it (5 and 2); Replace it with 10 leavin 4 10 +; Scan left again until we encounter + symbol then apply the operator to the two operands immediately preceding it (4 and 10) yielding 14.
         c. EXAMPLE 2: (3 * 4 - (2 + 5)) * 4/2 in postfix notation it would be: 3 4 * 2 5 + - 4 * 2 /
            i. Applying the rule it would produce:
               1. ORIGINAL - <u>**3 4 ***</u> 2 5 + - 4 * 2 /
               2. THEN 12 <u>**2 5 +**</u> - 4 * 2 /
               3. THEN <u>**12 7 -**</u> 4 * 2 /
               4. THEN <u>**5 4 ***</u> 2 /
               5. THEN <u>**20 2 /**</u>
               6. RESULTING = 10
   iv. A stack is the ideal data structure to use when evaluating a postfix expression
      1. Using a stack to evaluate a postfix expression; Figure 2



      2. The algorithm can be expressed as follows:
         a. Scan the expression from left to right

b. Identify each token (operator or operand) in turn
c. If it is an operand push it onto the stack
d. If it is an operator pop the top two elements off the stack, apply the operation to them, and push the result onto the stack
e. When we reach the end of the expression the element remaining on the stack is the result of the expression.
    i. If at any point we attempt to pop two elements off of the stack but there are not two elements on the stack, then our postfix expression was not properly formed
    ii. Or, if we reach the end of the expression and more than one element remains on the stack, then our expression was not well formed.

d. *Exceptions (14.6)*
    i. Example of stack exceptions (used from the postfix evaluation):
        1. if the stack were full on a push
            a. conceptually speaking, there is no such things as a full stack; It technically would be a data structure issue
        2. if the stack were empty on a pop
            a. Use EmptyCollectionException
        3. if the stack held more than one value at the completion of the evaluation

e. *Implementing A Stack: With Arrays (14.7)*
    i. The implementation of the collection operations should not affect the way users interact with the collection.
    ii. Managing Capacity: Capacity is the number of cells in an array; the capacity of an array cannot be changed once the array has been created

f. *The ArrayStack Class (14.8)*
    i. ArrayStack: represents a stack with an underlying array-based implementation.
        1. to be more precise, we define a class called ArrayStack<T> that represents an array-based implementation of a stack collection that stores objects of generic type T.
            a. When we instantiate an ArrayStack object, we specify that the generic type T represents.
    ii. For efficiency, an array-based stack implementation keeps the bottom of the stack at index 0.
    iii. An array implementation of a stack can be designed by making four assumptions:
        1. the array is an array of object references (type determined when the stack is instantiated)
        2. The bottom of the stack is always at index 0 of the array
        3. The elements of the stack are stored in order and contiguously in the array
        4. There is an integer variable top that stores the index of the array immediately following the top element in the stack.

iv. From the assumptions above: we can determine that our class will need a constant to store the default capacity, a variable to keep track of the top of the stack, and a variable for the array to store the stack

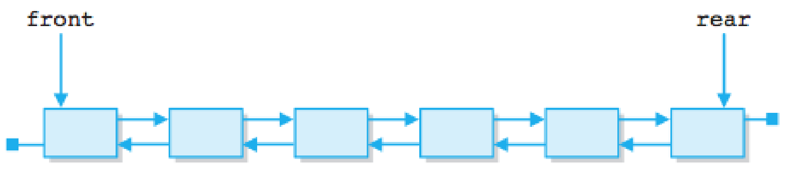
g. *References As Links (14.9)*

i. Linked Structure: is a data structure that uses object reference variables to create links between objects;
   1. They are the primary alternative to an array-based implementation of a collection

ii. Object reference variables can be used to create linked structures
   1. Usually the specific address that an object reference variable holds is irrelevant; while it is important to be able to use the reference variable to access an object, the specific location in memory where it is stored is unimportant
   2. Therefore, we usually depict a reference variable as a name that points to an object; A reference variable, used in this context is called a pointer

iii. A linked list is composed of objects that each point to the next object in the list; Linked list is a linked structure in which one object refers to the next, creating a linear ordering of the objects in the list.
   1. A linked list is only one kind of linked structure; it can have multiple references to objects making a more complex structure that is created;
   2. Often the objects stored in a linked list are referred to generically as the nodes of the list.
   3. A linked list dynamically grows as needed and essentially has no capacity limitations
      a. A linked list is considered to be a dynamic structure because its size grows and shrinks as needed to accommodate the number of elements stored

iv. Doubly Linked Lists:
   1. Objects that are stored in a collection should not contain any implementation details of the underlying data structure
   2. An alternative implementation for linked structures is the concept of a doubly linked list



   3. Each node in the list stores both a reference to the next element and a reference to the previous one
      a. In a doubly linked list, two references are maintained:
         i. one to point to the first node in the list and another to point to the last node in the list.
         ii. Each node in the list stores both a reference to the next element and a reference to the previous one

h. *Managing Linked Lists (14.10)*

i. Stacks only update one end of a list, but lists can be used for many collections

ii. There are a few basic techniques involved in managing nodes on the list, no matter what the list is used to store

iii. Special care must be taken when dealing with the first node in the list

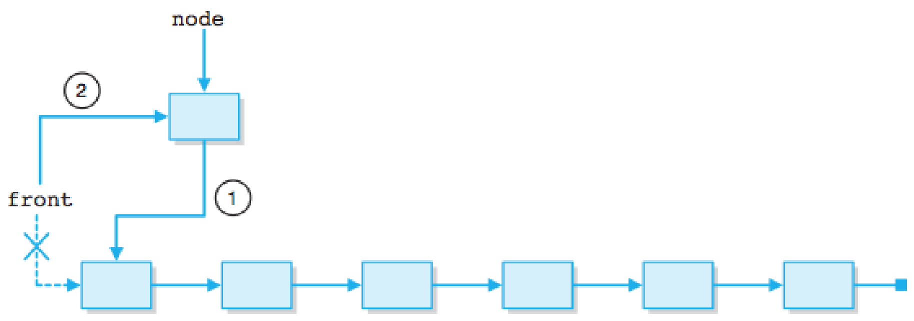
iv. Stacks only update one end of a list, but for other collections a node may be inserted or deleted anywhere in the list

v. Accessing Elements:

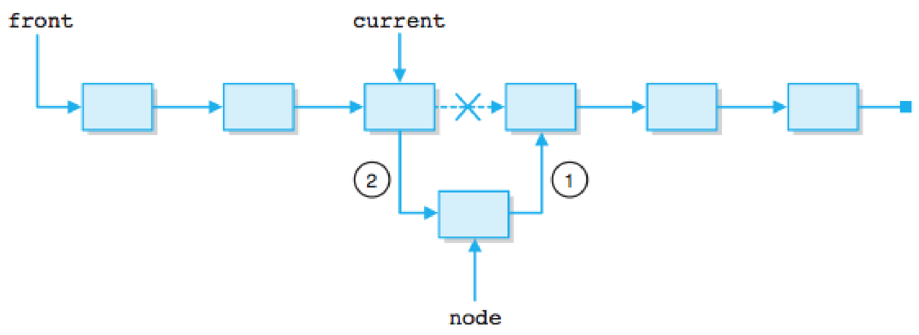
  1. Special care must be taken when dealing with the first node in the list so that the reference to the entire list is maintained
     a. When using linked lists we maintain a pointer to the first element in the list
     b. To access other elements, we must access the first one and then follow the next pointer from that one to the next one and so on.

vi. Inserting Nodes:

  1. The order in which references are changed is crucial to maintaining a linked list.
  2. A node may be inserted into a linked list at any location:
     a. Front
        i. First the next reference of the added node is set to point to the current first node in the list. Second, the reference to the front of the list is reset to point to the newly added node



     b. Middle
        i. First we have to find the node in the list that will immediately precede the new node being inserted
           1. Unlike an array, in which we can access elements using subscripts, a linked list requires that we use a separate reference to move through the nodes of the list until we find the one we want; This type of reference is often called current because it indicates the current node in the list that is being examined



c. End
   i. If the new node is inserted at the end of the list, the net reference of the new node is set to null

vii. Deleting Nodes
   1. Any node in the list can be deleted.
      a. To delete the first node in a linked list, the reference to the front of the list is reset so that it points to the current second node in the list.
      b. To delete a node from the interior of the list, we must first find the node in front of the node that is to be deleted.
         i. This processing often requires the use of two references:
            1. one to find the node to be deleted
            2. Another to keep track of the node immediately preceding that one
               a. Often called current and previous
         ii. Once these nodes have been found the next reference of the previous node is reset to point to the node pointed to by the next reference of the current node. The deleted node can then be used as needed.

i. *Implement A Stack: The Java.Util.Stack Class (14.13)*
   i. Class java.util.Stack is provided by Java API framework:
      1. The push operation accepts a parameter item that is a reference to an object to be placed on the stack
      2. The pop operation removes the object on top of the stack and returns a reference to it
      3. The peek operation returns a reference to the object on top of the stack
      4. The empty operation behaves the same as the isEmpty operation that mentioned earlier.
      5. The size operation returns the number of elements in the stack
   ii. The Java.util.Stack class is derived from Vector, which gives a stack inappropriate operations
      1. The stack is not everything a vector is(conceptually), the stack class should not be derived from the Vector class.

j. *Packages (14.14)*

i. java.util - contains classes for utility purposes; java.lang contains classes related to core aspects of the language; java.io encompasses input and output functionality
    1. We typically organize code into packages by function, for example all collections might be together in one package with all exceptions for those collections might be in a subordinate package
    2. Code for a particular package will be located in common directory/folder with subordinate packages arranged in subordinate directories/folders

ii. package javafoundations;
    1. this statement indicates that the class contained in the file is part of the javafoundations package. At compile time, the compiler will check the package declaration statement and the location of the source code file in the directory tree to ensure that the source code file is in the correct location

iii. An environment variable is a variable set up at the operating system level to establish a value needed by the overall computing system.

iv. The CLASSPATH environment variable is usually set in the command shell to indicate the list of paths where the compiler or run-time environment will look if it cannot find a class we reference in the source code.

v. CLASSPATH can be set in the shell or as a parameter during program execution

vi. The -cp option sets the CLASPATH to look for packages containing needed classes in two locations:
    1. C: \javafoundations
    2. the current working directory

vii. When a class is required at compile or run-time, if it is not in the API, the CLASSPATH locations are checked for the needed classes.

---

10. <u>Queues</u>

a. A Queue ADT (15.1)

  i. A queue is a linear collection whose elements are added on one end and removed from the other.
    1. FIFO - First in, first out; Elements are removed from a queue in the same order in which they are placed on the queue
        a. Think of waiting in line

  ii. Usually a queue is depicted horizontally. One end is established as the front of the queue and the other as the rear of the queue. Elements go onto the rear of the queue and come off of the front.
    1. Sometimes the front of the queue is called the head and the rear of the queue the tail
    2. COMPARE AND CONTRAST: In a stack, the processing occurs at only one end of the collection. In a queue, processing occurs at both ends
        a. Similar to stack, there are no operations that allow the user to “reach into” the middle of a queue and reorganize or remove elements.
        b. The principal purpose of a stack is to reverse order, the principle purpose of a queue is to preserve order.

iii. Terminology
1. Enqueue (can be called add or insert) - used to refer to the process of adding a new element to the end of a queue
2. Dequeue (can be called remove or serve - refers to removing the element at the front of a queue
3. The First Operation (can be called front) - allows the user to examine the element at the front of the queue without removing it from the collection
4. isEmpty - determines if the queue is empty
5. size - determines the number of elements on the queue

b. *Using Queues: Code Keys (14.2)*
   i. A queue is a convenient collection for storing a repeating code key
   ii. A Caesar Cipher - shift each letter in a message along the alphabet by a constant amount k.
      1. Example: vlpsolflwb is the encoded message; each letter is shifted the same number of characters backwards at k = 3
         a. The outcome would be: simplicity **(vlpsolflwb != simplicity)**
      2. An improvement to this can be made using repeating key; instead of shifting each character by a constant amount we can shift each character by a different amount using a list of key values;

c. *Implementing Queues: With Links (15.4)*
   i. A linked implementation of a queue is facilitated by references to the first and last elements of the linked list
   ii. Does it make a difference to which end of the list we add or enqueue elements and from which end of the list we remove or dequeue elements?
      1. If our linked list is singly linked....then yes

d. *Implementing Queues: With Arrays (15.5)*
   i. One array-based strategy for implementing a queue is to fix one end of the queue at index 0 of the array. The elements are sorted contiguously in the array.
   ii. Because queue operations modify both ends of the collection, fixing one end at index 0 requires that elements by shifted
   iii. The Shifting of elements in a non-circular array implementation creates an O(n)
   iv. Treating arrays as circular eliminates the need to shift elements in an array queue implementation
   v. FINISH LATER

---

11. Sorts(Insertion, Bubble, Quick, and Merge)

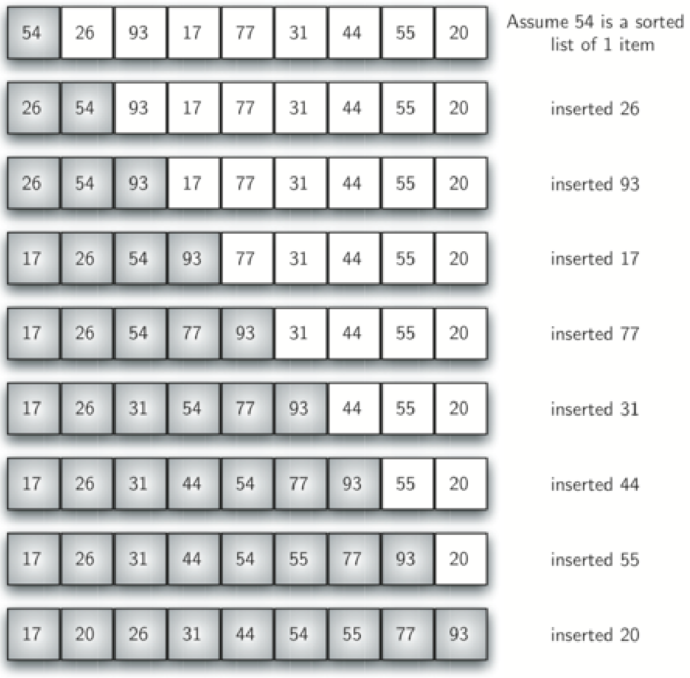
a. Definition:
   i. Sorting is the process of arranging a group of items into a defined order, either ascending or descending, based on some criteria.

b. Insertion Sort
   i. Definition:
      1. Algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted. One at a time, each unsorted

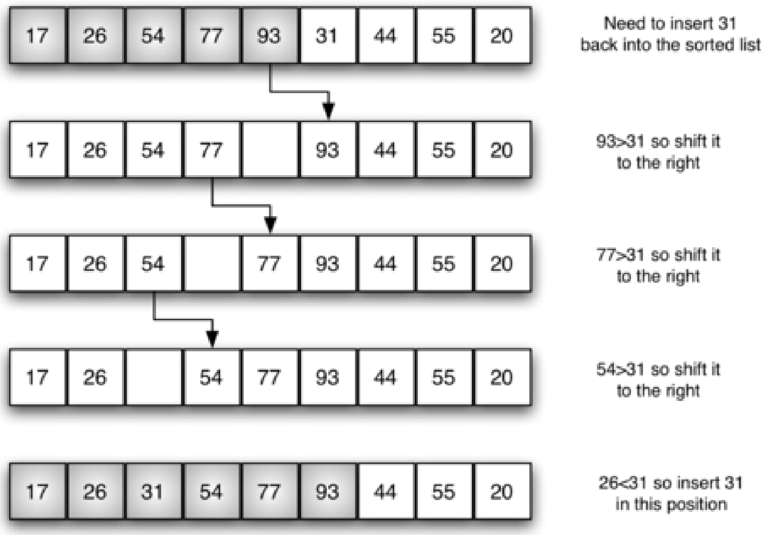
element is inserted at the appropriate position in that sorted subset until the entire list is in order

ii. General Strategy
   1. Sort the first two values in the list relative to each other by exchanging them if necessary.
   2. Insert the list's third value into the appropriate position relative to the first two (sorted) values
   3. Insert the fourth value into its proper position relative to the first three values in the list.
   4. Each time an insertion is made the number of values in the sorted subset increases by one.
   5. Continue this process until all values in the list are completely sorted

iii. Pictures:



54 26 93 17 77 31 44 55 20 — Assume 54 is a sorted list of 1 item
26 54 93 17 77 31 44 55 20 — inserted 26
26 54 93 17 77 31 44 55 20 — inserted 93
17 26 54 93 77 31 44 55 20 — inserted 17
17 26 54 77 93 31 44 55 20 — inserted 77
17 26 31 54 77 93 44 55 20 — inserted 31
17 26 31 44 54 77 93 55 20 — inserted 44
17 26 31 44 54 55 77 93 20 — inserted 55
17 20 26 31 44 54 55 77 93 — inserted 20



17 26 54 77 93 31 44 55 20 — Need to insert 31 back into the sorted list
17 26 54 77 _ 93 44 55 20 — 93>31 so shift it to the right
17 26 54 _ 77 93 44 55 20 — 77>31 so shift it to the right
17 26 _ 54 77 93 44 55 20 — 54>31 so shift it to the right
17 26 31 54 77 93 44 55 20 — 26<31 so insert 31 in this position

c. Bubble Sort

i. Definition:
   1. sorts values by repeatedly comparing neighboring elements in the list and swapping their position if they are not in order relative to each other.

ii. General Strategy:
   1. Scan through the list comparing adjacent elements and swap them if they are not in relative order.
   2. This has the effect of "bubbling" the largest value to the last position in the list, which is its appropriate position in the final, sorted list.
   3. Then scan through the list again, bubbling up the second-to-last value.
   4. This process continues until all elements have been bubbled into their correct positions.

iii. EXAMPLE:
   1. [9, 6, 8, 12, 3, 1, 7]

        a. first compare 9 and 6 ;finding them not in the correct order, swap them...
    2. [6, 9, 8, 12, 3, 1, 7]
        a. then compare 9 and 8; finding them not in the correct order, swap them...
    3. [6, 8, 9, 12, 3, 1, 7]
        a. the compare 9 and 12; they are in the correct order, we <u>don't</u> swap them...
        b. because they are compare 12 and 3; they're not in order, swap them...
    4. [6, 8, 9, 3, 12, 1, 7]
        a. now compare 12 and 1; swap them...
    5. [6, 8, 9, 3, 1, 12, 7]
        a. now compare 12 and 7; swap them...
    6. [6, 8, 9, 3, 1, 7, 12]
        a. This completes one pass through...After the first pass 12 (largest value) is in the correct position; The next pass will bubble the value 9 up to its final position.
        b. After the first pass, the last value no longer needs to be consider in future passes. After the second pass, we can forget about the last two and so on.

d. Quick Sort
    i. Definition:
        1. Begin by choosing a pivot point. Then reorder the list until all elements greater than the pivot are above the elements less than the pivot. Then apply this technique recursively to the other partitions until the sort is complete.
        2. Time complexity: worst O(n^2), avg O(n log(n)).
    ii. Pseudocode:
        1. Choose one element in the list to be the partition element
        2. Organize the elements so that all elements less than the partition element are to the left and all greater are to the right
        3. Apply the quick sort algorithm (recursively) to both partitions

e. Merge Sort
    i. Definition:
        1. Begin by breaking down 1 unsorted list with n elements into n lists with 1 element. Then use the merge operation to recombine. What this does is compare two sorted lists and compares the first element of each. Whichever is smaller is placed first into the new sorted list. Then the next smallest element from the two original lists is added. This continues until everything is sorted and relies of the fact that you are merging two sorted lists.
        2. Time complexity: worst, best, and avg: O(n log(n))

---

12. <u>Contracts</u>

a. *Non-Disclosure agreements*: Defines confidential information that two parties will be sharing with one another (How long information must remain confidential)
    i. Are used when the client is giving the developer design secrets.

b. *Non-Compete agreements*: Are used by employers to ensure that an employee will not directly compete with them, and this can limit getting jobs in the same field and starting own businesses (most states limit geographical area and time period)

c. *End User License Agreement*: It grants the software publisher the permission to monitor the use of the product and update it automatically. Also, no liability to the software publisher for damages to the user's computer.
   - i. Prevents:
     1. Distribution of copies
     2. Public criticism
     3. Benchmarking
     4. Reverse engineering
   - ii. Grants permissions:
     1. Monitor its own use
     2. Update automatically

d. Warranties
   - i. Warranty Provisions
     1. Software will work as developer has stated
     2. Developer will fix any errors noticed within X time
     3. Guarantee non-infringement of other's copyrights
   - ii. Shrinkwrap Warranties
     1. Accept software "as is"
     2. Some offer 90 day replacement or money back guarantee
     3. None accept liability for harm caused by use of software

e. Moral Responsibility
   - i. If vendors were responsible for harmful consequences of defects:
     1. companies would test software more
     2. They would purchase liability insurance
     3. Software would cost more
     4. Start-ups would be affected more (than large companies)
     5. Less innovation in software industry
     6. Software would be more reliable (duh)

f. Dispute Resolution
   - i. Arbitration
     1. Person or panel will render a decision on any arguments (gives up right to go to court)
   - ii. Mediation
     1. A 3rd party mediator tries to settle dispute

g. Software Patents
   - i. Exclusive rights to patent holder for 20 years
   - ii. Owner often grants usage license for royalties

iii. Can't patent an algorithm / math formula
iv. Can't patent anything obvious (though this is getting looked over more and more by the patent office...blergh)
v. Often uses vague language

---

13. <u>Design Patterns</u>

a. Pattern: Each pattern is a three part rule, which expresses a relation between a certain context, a problem, and a solution - C Alexander

b. Keys For Patterns:
   i. A reusable solution to a common software problem
      1. Experienced software designers reuse solutions that worked well previously
   ii. A general template that can be followed in different situation
   iii. Well structured object oriented systems have recurring patterns of classes and objects
   iv. Speeds up development
   v. Improves readability for coders familiar with design patterns

c. Types of Software Patterns
   i. Conceptual
      1. Pattern whose form is described by means of terms and concepts from the application domain
   ii. Design
      1. Pattern whose form is described by means of software design constructs, such as objects, classes, inheritance and aggregation
   iii. Programming
      1. Pattern whose form is described by means of programming language constructs

d. Abstraction
   i. The gang of four design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

	More Abstract
Complex design for an entire application/subsystem	↓
<u>Solution to a general design problem to a particular context</u>	↓
Simple reusable design class (ie. linked list)	↓
	More Concrete

e. Design Pattern Essentials
   i. Pattern name
      1. Have a concise, meaningful name for a pattern

ii. Problem
    1. What is the problem and context of use of pattern?
    2. What are the conditions that must be met?

iii. Solution
    1. Description of elements that make up pattern
    2. Emphasis on relationships, responsibilities, and collaborations
    3. Not a concrete design

f. Creational Patterns - Various mechanisms for object creating
   i. Abstract Factory
      1. Factory - location in the code where objects are created
         a. The factory determines the concrete type of product object to instantiate
      2. Can only interact with objects through abstract interface
      3. Client code is only aware of abstract product type, not concrete type
   ii. Object Pool
      1. Set of initialized objects kept ready to use instead of creating and destroying them on demand
      2. Client code requests objects from the pool and returns them when finished
      3. The pool must return objects to a safe, default state before giving the object to another client

---

14. Android

a.

---

15. JavaFX

a.

Midterm Questions:

True/False Section

- **super() is always the first statement in a subclass constructor**
  - False, super() is the first command executed, but it doesn't have to be a typed statement
- **Abstract classes cannot have non-abstract methods**
  - False, declaring a class abstract just means you can't instantiate it
- **You exit vi without saving by issuing the :x command**
  - False(:x is with saving, `:q!` is without saving)
- **A finally clause is required in a try-catch block**
  - False- It's optional

- **There is only one base case in the recursive solution to finding a path through a maze**
    - False- there are two; finding the end and hitting a dead end
- **The following Unix command changes to the parent of the parent directory: cd ./../..**
    - True
- **@author and @return are the two required Javadoc tags for classes and interfaces**
    - False- @Author and @Version
- **Github is best described as a decentralized version control system**
    - True
- //Syntax errors are generally easier to fix than logic errors
    - Thrown Out. Very subjective. Much bad question. Wow.
- **In Unix, the > redirection operator appends the command line output to the end of a file without deleting the information that is already in place**
    - False: >> command
- **In vi, you move one character to the left by pressing the L key**
    - False....Depends actually...He was looking for H but left arrow works also
- **In a makefile, the character "#" is used to label comments**
    - True .
- **The octal notation that would assign permissions of read and write to the user, group, and others would be "666"**
    - True .
- **Private instance variables in a base class are inherited in a subclass**
    - False - technically they are but not accessable; He just let these go because its so vague
- **Classes are a grouping of related type providing access protection and name space management**
    - False. Packages

Code Reading Section:

```
public static void doStuff(String s)
{
        for(int i = s.length() - 1; i >= 0; i--)
        {
                System.out.print(s.charAt(i));
                System.out.print(s.charAt(i));
        }
}
```

What does it do?

Prints a word backwards but every letter twice

Now, write it recursively...

```
public String reverseString(String s)
{
        if(s.length() == 0)
        {
                return s;
        }

        return reverseString(s.substring(1)) + s.charAt(0) + s.charAt(0);
}
```

Error/Exception Reading

```
1       int x = doStuff();
        try {
2               x = fancyMath(x);
3               manipulate(x);
4               printStuff();
        }
        catch(IOException ioe) {
5               System.err.println(ioe);
        }
        catch(IndexOutOfBoundsException oob) {
6               System.err.println(oob);
        }
        catch(Exception e) {
7               System.err.println(e);
8               System.exit(1);
        }
        finally {
9               System.out.println("Out of tries");
        }

10              System.out.println("Finish!");
```

Questions pertaining to code above:

1. Which lines will run if there is an unchecked exception in doStuff()?
   a. Line 1
2. Which lines will run if there is an IO Exception in fancyMath()?
   a. Lines 1, 2, 5, 9, 10
3. Which lines will run if there is a NullPointer in manipulate
   a. Lines 1, 2, 3, 7, 8
4. Which lines will run if there is no exceptions?
   a. Lines 1, 2, 3, 4, 9, 10

More Code Reading Section:

```
public class Laptop {
        public void shoutOut() {
                System.out.println("I'm a laptop!");
                printLines();
        }

        public void printLines() {
                System.out.println("****");
        }
}

public class MacBookPro extends Laptop {
        public void shoutOut() {
                System.out.println("Hello, I'm a Mac!");
        }
```

```
	public static void main (String[] args) {
		Laptop one = new MacBookPro();
		Laptop two = new Laptop();
		one.shoutOut();
		two.shoutOut();

		((Laptop)one).shoutOut();
		two = one;
		two.shoutOut();

		Can someone explain this one, I forgot how it works and I got it wrong on the final and now I
feel stupid
	}
}
```

What is the output?

Hello, I'm a Mac!
I'm a laptop!
****
Hello, I'm a Mac!
Hello, I'm a Mac!

Last Question:

```
public interface MyInterface {
	public void aMethod(void);
}
public abstract class ClassOne Implements MyInterface {
	protected int x;

	public ClassOne(int x) {
		this.x = x;
	}

	public void printMethod() {
		System.out.println(x);
	}

	public void aMethod(int y) {
		x = x + y;
	}
}
public class ClassTwo extends ClassOne {
	private int x, y;
	public ClassTwo (int x)
	{
		super(x);
		this.x = 5;
	}
	public void aMethod(float x) {
		System.out.println(x);
	}
	public void aMethod(int z) {
		x = x * z;
	}
```

```
        public void printMethod() {
                System.out.println(x + y);
        }
}
```

I unfortunately don't know what was asked about this...I think this was the question that Plaue asked a class before and they did really bad. I think he asked something about what is wrong with this code? Is there anything wrong with this code? I'm not sure though. If anyone can remember please finish this for me.

-- Didn't he ask about what methods are overriding and overloading here?

Answer:

Final Exam Study Guide

(posted as a questions pdf and an answers pdf under "Content" on elc new)

True or False:

1. Model View Controllers are reusable: when the problem reoccurs, there is no need to invent a new solution; we just have to follow the pattern and adapt it as necessary, and expressive: by using the MVC design pattern our application becomes more expressive.

1. True.

2. A stack is processed in a first out, last in manner.

2. False: A stack is processed in a last in, first out (LIFO) manner

3. The pop() method of the stacks interface removes the element that has spent the most time on the stack and returns it as a value of the function, i.e. a stack is a first-in first-out (FIFO) data structure.

3. False, stacks are last-in first-out (LIFO) - the last element you push on will be popped off.

4. A queue is a linear collection whose elements are added on one end and removed from the other. Therefore, the queue elements are processed in FIFO manner.

4. True.

5. Queues process element in a LIFO(Last In First Out) manner. The Last element in is the First element out.

5. False, Queues process in a FIFO manner First element in First element out manner.

6. Stacks are processed in a FIFO manner while queues are processed in a LIFO manner.

6. False, stacks are LIFO (last in, first out) and queues are FIFO (first in, first out)

7. Queues are processed in a last in, first out order.

7. FALSE! They are processed in a first in, first out order.

8. The Nyquist rule state that the sampling rate must be at least twice as fast as the fastest frequency.

8. TRUE

9. When loops are nested, we add the complexity of the outer loop to the complexity of the inner loop.

9. False: When loops are nested, we multiply the complexity of the outer loop by the complexity of the inner loop.

10. In order to determine the time complexity of a recursive method, one would multiply the # of times the recursive definition is followed by the order of the body of the recursive method.

10. True.

11. Quicksort is the ideal and fastest sorting algorithm, running in O(n log n) time in all cases.

11. False - While true that quicksort is faster in the general case, it does not run in O(n log n) time in all cases; in the worst case, it runs in O(n^2) time.

12. Insertion sort and Quick sort have the same worst case time complexity.
12. True (n^2)
13. Many types of sorting method swap two array elements by calling the swap method, which works by assigning each array element equal to each other.
13. False a third variable is used as place holder for the swap method, so 3 assignment statements are used.
14. Non-Compete agreements are used when client is giving developer design secrets.
14. False, answer is non-disclosure agreements.
15. Open source software is also free software.
15. False; not all open source software is free and neither is free software always open source. For example, the Android platform is open source, but access to the source code is restricted in certain ways. Android also uses some nonfree libraries and firmware within its design.

Short Answer

1. Briefly describe the three components of the Model View Controller.
1. **Model:** The model object knows about all the data that need to be displayed. It is model who is aware about all the operations that can be applied to transform that object. It only represents the data of an application. The model represents enterprise data and the business rules that govern access to and updates of this data. Model is not aware about the presentation data and how that data will be displayed to the browser.
View: The view represents the presentation of the application. The view object refers to the model. It uses the query methods of the model to obtain the contents and renders it. The view is not dependent on the application logic. It remains same if there is any modification in the business logic. In other words, we can say that it is the responsibility of the of the view's to maintain the consistency in its presentation when the model changes.
Controller: Whenever the user sends a request for something then it always go through the controller. The controller is responsible for intercepting the requests from view and passes it to the model for the appropriate action. After the action has been taken on the data, the controller is responsible for directing the appropriate view to the user. In GUIs, the views and the controllers often work very closely together.
2. What are some of the operations in stack? Name two and explain their functions.
2. The pop operation: removes and returns the element at the top of the stack
The peek operation: returns a reference to the element at the top of the stack without removing it
from the array.
3. Explain the difference between a properly implemented peek() method and pop() method of the Stacks interface.
3. peek() returns but does not remove the last element pushed onto the stack, pop() removes and returns it.
4. Discuss briefly on Queue: Linked List Implementation and its advantage.
4. In addition to a reference pointing to the first element in the list, it also has a second reference pointing to the back of the list. There is a variable to keep track of the number of elements in the list. The advantage of this is when adding something to the front or the back the time it takes is instant because it is always keeping track of the first and last element of the list.
5. Which implementation of Queues has the worst time complexity? Which implementation has the most space complexity?
5. The non-circular array implementation with an O(n) dequeue or enqueue operation has the worst time complexity. Both of the array implementations waste space for unfilled elements in the array. The linked implementation uses more space per element stored.
6. Define 3 of the 5 main operations of a queue and 3 of the 5 main operations of a stack as defined in chapters 14 & 15
6. s-push- Adds element to top of stack

q-Enqueue- Adds element to rear of queue
s-pop -Removes element to top of stack
q-dequeue- Removes element from front of queue
s-peek- Examines element to top of stack
q-first- Examines element at the front of the queue
s- isEmpty -Determines if the stack is empty
q- isEmpty- Determines if the queue is empty
s- size -Determines the number of elements in the stack
q- size -Determines the number of elements in the queue
7. Briefly explain some of the advantages of queues as opposed to other alternatives to solve programming problems.
7. One possible answer is that queues allow you to easily handle situations where you need to be aware of the order things are inputted/processed. Because they use a first-in-first-out processing order, this makes it very easy for the programmer to make sure items are processed and taken from the stack in the right place at the right time.
8. Does an analog or digital sound wave have better quality and why?
8. The analog sound wave because a digital sound wave uses sampling to convert the wave into 1's and 0's and then back into a wave, reducing quality.
9. What is the order and complexity of this growth function t(n) = 4n + 23n2 + 5n?
9. Order is O(2^n). The complexity is exponential.
10. Rank the following functions in terms of their rates of growth: O(n^2),O(n), O(n^3), O(nlogn), O(logn). Rank them from largest rate of growth to smallest.
10. O(n) ,O(log n), O(nlog n) , O(n^2), O(n^3)
11. Which of the following sequences is sorted correctly in terms of computation speed in the average case, from fastest to slowest?
a) Shaker Sort > Insertion Sort > Bubble Sort > Heap Sort > Quick Sort
b) Quick Sort > Heap Sort > Insertion Sort > Shaker Sort > Bubble Sort
c) Heap Sort > Quick Sort > Insertion Sort > Shaker Sort > Bubble Sort
d) Quick Sort > Shaker Sort > Insertion Sort > Bubble Sort > Heap Sort
11. C
12. Explain how the quickSort algorithm works. Do not write actual code, just explain what happens.
12. QuickSort works by continuously dividing the list into two parts and moving the lower items to one side and the higher items to the other. It starts by picking one item in the entire list to serve as a pivot point. The pivot could be the first item or a randomly chosen one. All items that compare lower than the pivot are moved to the left of the pivot; all equal or higher items are moved to the right. It then picks a pivot for the left side and moves those items to left and right of the pivot and continues the pivot picking and dividing until there is only one item left in the group. It then proceeds to the right side and performs the same operation again.
13. Explain what it means for a sorting algorithm to be stable.
13. A sorting algorithm is stable if it does not reorder equal elements in a data set, while an unstable algorithm will swap equal elements.
14. Describe how the bubble sort algorithm works.
14. Bubble sort starts by scanning through a list of array elements, while adjacent elements are swapped if they are not in relative order. Then it scans through the list again up till the second to last element. This process continues until it has scanned through the list the number of times of the length of the array minus 1.
15. What is an abstract factory?
15. An abstract factory is a location in the code where objects are created. The factory determines the concrete type of the object, but the client is only aware of the abstract type.
16. What is the importance of a EULA (End User License Agreement).
16. EULA prevents distribution of copies, public criticism, benchmarking and reverse engineering. It grants the software publisher permission to monitor the use of the product and update it automatically. Also no liability to the software publisher for damages to the user’s computer.

17. Briefly describe the importance of a non-compete agreement.
17. Prevents an employee from directly competing with their employer. It limits getting another job in same field and starting a separate business.
18. List the four essential freedoms that comprise free software.
18. A) The freedom to run the program, for any purpose.
B) The freedom to study how the program works, and change it so it does your computing as you wish. Access to the source code is a precondition for this.
C) The freedom to redistribute copies so you can help your neighbor.
D) The freedom to distribute copies of your modified versions to others.
19. List three ways in which free open source software can cost corporations money.
19. (note: more than three answers are given for sake of review):
a) Documentation can be sparse or cost money, which would preserve the aspect of "free open source" but in practice would cost the company money directly for the documentation or the increased overhead in hiring experts on the software to train the users.
b) A company is essentially responsible for its own tech support when dealing with free open source software as there is no central entity responsible for providing expert support. There may be community forums and other similar avenues for seeking support with any given software, but this is not acceptable for a large company.
c) Companies are responsible for integration of the software throughout their respective workplace. d) Companies are responsible for customizing the software to suit their specific needs rather than outsourcing this work to a contractor.
e) Companies are responsible for handling upgrades of the software