Final Exam Studyguide

0. Design
    A. Constants
    B. Cohesion: the degree to which components of a class belong together to fit a particular role
        1.) Aim for High cohesion where a class is specialized in aspect.
        2.) A class that tries to do many things comes with higher maintenance and lower reusability.
    C. Coupling: the degree to which one class has dependency upon another
        1.) Aim for LOW coupling because any changes to one class could impact the other and the person making the changes may be completely unaware of this and unknowingly break the class.
    D. Encapsulation: restricting access to the internal workings of a class
        1.) As a general rule, hide all design decisions and things that are likely to change
        2.) Decreases coupling: program less likely to fail/crash
        3.) Increases abstraction: decreases complexity
        4.) Indicates a good level cohesion
    E. Problem Solving: decomposing problems into simpler problems
        1.) Use Design Patterns: proven solutions that work
    F. Testing and Reliability
        1.) Validation: build the system the user asked for
        2.) Verification: build the system correctly without flaws
        3.) Refactoring: “cleaning up the code” to fill in short-cuts, eliminate duplication and dead code, and to make the design and logic clear
        4.) Fix code with Debugger using test cases

I. Java Objects
    A. Constructors: allow us to instantiate our objects via declaration, assignment and creation
        1.) must have the same name as the class
        2.) Overloading: constructors with the same name but have different parameters
        2.) Access Modifiers:
            a. public: accessible to all other classes everywhere
            b. protected: accessible to the package the implementation is in
            c. private: can only be constructed from within its own class
    B. toString: Returns a string representation of the object.
    C. equals: Indicates whether some other object is "equal to" this one.
    D. Static: member of a class that isn’t associated with an instance of a class.
        1.) belongs to the class itself which allows access the static member without first creating a class instance.
        2.) you can’t access a nonstatic method or field from a static method because the static method doesn’t have an instance of the class to use to reference instance methods or fields.
    E. Access Modifiers: set access levels for classes, variables, methods and constructors

1.) Private: can only be accessed within the declared class itself
   a. Class and interfaces cannot be private
   b. main way that an object encapsulates itself and hides data
2.) Public: can be accessed from any other class
   a. all public methods and variables of a class are inherited by its subclasses
   b. if declared in superclass, must be declared public in all subclasses
3.) Protected: can be accessed only by the subclasses
   a. class and interfaces cannot be protected
   b. Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private

II. Debugging
A. Finding the bugs
1.) Debugging: strategy when things go wrong:
   a.) drop ALL your assumptions
   b.) divide and conquer
   c.) find location/situation that fails
   d.) write a test case for THAT situation
   e.) fix it and confirm that it now works!
B. Debugger
1.) Make different test cases for every possible scenario. Try to make code fail.
2.) Watch your variables for unexpected values!
C. Breakpoints
1.) Breakpoints: specify where the execution of the program should stop
2.) Once the program is stopped: investigate variables, change their content, etc
3.) Single Step Execution: allows you to identify specific statements that cause logical errors

III. Testing
A. Principles
1.) Regression Testing: keep all test cases so new code is rechecked
2.) Good tests: unique scenarios that try to make the program fail
3.) Test Driven Development: the way to force us to think about the implementation before writing the code
   a. Write a test
   b. Run all tests
   c. Write the implementation code
   d. Run all tests
   e. Refactor
B. Test Suites
1.) Test suite: bundle a few unit test cases and run it together

IV. Stacks
A. Stack: a collection of objects in which the order of access is defined by Last-In, First-Out
1.) Access to items on the stack is limited to the “top” item.

B. API
1.) isEmpty ( ): returns boolean
2.) push (item): put item onto the stack
3.) pop ( ): returns & removes value on top
4.) peek ( ): optional - only looks at the top item, does not remove

B. Uses
1.) “Call Stack” / “Program Stack”
a. Keeps track of calling/returning from methods
b. “Calling” a method PUSHes address of next instruction on the stack
c. “Return” from a method POPs address into PC and execution continues
2.) Example: Backtracking - Maze exploration, “Depth-first”

V. Queues

A. Queue: a collection of objects in which the order of access is defined by First-In, First-Out
1.) Add to items to the “end”; Remove items from the “front”

B. API
1.) int count ( ): returns number of items in queue
2.) enqueue (item): put item onto the queue at end
3.) dequeue ( ): returns & removes value from front
4.) peek ( ): only looks at the front item, does not remove

C. Uses
1.) Keeping list of things that need first-come, first-served priority
2.) Buffer between systems of different speeds
a. one system enqueues; other dequeues
3.) Examples:
a. Queue” up for the movies; or a bank teller;
b. Backtracking - Maze exploration, “Breadth-first”

VI. Inheritance

A. Principles
1.) Inheritance: the process where one object acquires the properties of another made manageable in a hierarchical order
2.) Purpose: Method Overriding, Code Reusability
3.) IS-A Relationship
a. Every class inherits from Object
b.Child class is more specific than parent class
c. Commonalities belong higher in hierarchy
d. Each class manages its own data (let super do its job)
4.) Subclass should:
a. override: equals (Object obj)
b. override: toString ( )
5.) Special Cases
a. Final: useful for security
1.) Class: that cannot be extended or used for inheritance
2.) Method: cannot be overridden

b. Abstract: idea/category; helps organize into hierarchy
1.) cannot be instantiated
2.) subclasses must implement all abstract methods, but may override non abstract methods

6.) Polymorphism: ability of an object to take on many forms
a. implements through method overloading and method overriding

7.) Late Binding: the type of the object is determined at run-time
a. the method used is determined by the most specialized version (subclass that overwrites it at bottom of hierarchy)

VII. GUI

A. Container: used to hold components or other containers in a specific layout
1.) JFrame: provides the "main window" for the GUI application
2.) JPanel: holds other graphical components for organizational purposes

B. Component: Elements used inside the container
1.) Examples: Button, TextField, Label

C. MouseListener: interface used for sensing mouse related data
1.) Must implement:
a. void mouseClicked(MouseEvent e)
b. void mouseEntered(MouseEvent e)
c. void mouseExited(MouseEvent e)
d. void mousePressed(MouseEvent e)
e. void mouseReleased(MouseEvent e)

VIII. Enumerated Types

A. Uses
1.) Enum Types: a special Java type used to define multiple related constants
2.) Example: High, Medium, Low, Off

B. Issues
1.) Not “type” safe - int val = MEDIUM;
2) No easy way to print all constants

IX. Generics

A. Purpose: allow a type or method to operate on objects of various types while providing compile-time type safety
1.) Provides compiler time safety
2.) Allows a class to manipulate a variety of types without having to rewrite duplicate the class

B. Implementation
1.) When defining the class, we “tag” it with a generic type
a. Example: public class Stack<Element>

C. Possible issues
1.) Generics work with Objects only, not fundamental types (int, double, char, etc)
2.) Must use “wrapper” classes for those (Integer, Double, Character, etc)

X. Exceptions

A. Concepts
1.) Exception: a problem that arises during the execution of a program
2.) Runtime system searches for someone to handle the exception:
a. going up the call stack (to whoever called this method)
3.) That code can either:
a. Handle the situation as it decides
b. Propagate (throw) the exception to who-ever called it (up the call stack)
c. Handle the exception with a catch

B. Types
1.) Checked: typically a user error or a problem that cannot be foreseen
a. Cannot be ignored at the time of compilation.
b. Handle by propagating the exception, or try-catch
c. Example: if a file is to be opened, but the file cannot be found, an exception occurs.
2.) Runtime: problem that probably could have been avoided by the programmer.
a. Ignored at the time of compilation
3.) Errors: problems that arise beyond the control of the user or the programmer.
a. Ignored at the time of compilation
b. Example: if a stack overflow occurs, an error will arise

C. Solutions
1.) Handle using try-catch options:
a. Catch that exception and do something (or nothing)
b. Make your own exception
c. Example: MyException myEx = new MyException( );
throw myEx;

XI. File I/O
A. File I/O: package containing nearly every class needed for input and output
B. Scanners
1.) Part of standard Java class library for reading and writing input values
2.) Source can be keyboard or data file
C. Streams
1.) Handle input/output: FileReader and FileWriter
D. Exceptions
1.) file might throw checked exception

XII. Object References
A. “Pointers” to objects are just memory addresses

XIII. Arrays
A. Examples:
1.) Create an array: float[ ] myNums;
myNums = new float[20];
2.) Array of Objects: Student[] studentArray = new Student[7];
B. 2D arrays: “Rows and columns”
1.) Example: Spreadsheet of user movie ratings

2.) Arrays of Arrays: int[ ][ ] myArray = { {236, 189, 189, 0},
{236, 80, 189, 189},
{236, 0, 189, 80},
{236, 189, 189, 80} };

XIV. Linked List

A. Concept: Abstraction of a list of items
1.) “front” and “back” of list
2.) No fixed size (dynamically grows/shrinks)
3.) get item at position N
4.) Insert an item at position N
5.) Remove an item at position N
6.) Find position of a particular item
7.) Allows me to store anything

B. Nodes and Pointers
1.) Each item holds the value and a reference (pointer) to the next item in the list
a. Sometimes called “nodes” or “list elements”
b. Example: Node <E>
E value;
Node next;

C. Advantages
1.) dynamic structure: memory allocated as needed
2.) add and remove don’t require “re-shuffling”
3.) Stacks and Queues are easily implemented

D. Disadvantages
1.) extra storage required for pointers (references)
2.) Sequential access
3.) Reverse listing

XV. Dictionary

A. Key/Value Pairs: Maps keys to values
1.) Like a real dictionary
a. key = word
b. value = definition

B. Uses: Anywhere key is not adequate for indexing directly
1.) Property lists
2.) Sets of values (no duplicates!)
3.) Word Frequency tracking (key is word, value is count)
4.) Structured data for books (keys are Title or author, or etc)

C. Time Complexity : takes O(N) or maybe O(log N) time
1.) Hashing a value h(x)
a. take input value and “transform” it into a value btwn 0 and m-1
b. Range of input values should hash equally throughout range 0 to m-1
c. Ex: “Tom” might hash to 48; “Susan” might hash to 20; “Voldemort” might hash to 0;
d. get/put/remove in constant time

XVI. Sorting

A. Selection Sort: Divides list into sorted and unsorted
    1.) Find smallest in unsorted list
    2.) put at front of unsorted list and change split of sorted/unsorted
    3.) repeat
B. Insertion Sort: Builds final order one item at a time
    1.) Efficient for small data sets
    2.) O(n2) Better in practice than Selection Sort
C. Shell Sort: Sorting pairs that are separated by a gap
    1.) Gap shrinks for each iteration
    2.) A series of Insertion Sorts on sub-lists of the full list
D. Merge Sort:
    1.) Divide and conquer algorithm
    2.) Divide lists into smaller and smaller lists
        a. Smallest (two item) sort it
        b. Now combine as (4 item); sort it
        c. Combine as (8 item); sort it
    3.) Each larger list is partially sorted
    4.) Small lists are quick to sort;
    5.) Partial sorted lists are quick to sort
E. Radix sort
    1.) Look up video
F. Quick Sort: “King” of Sorts
    1.) Divide and conquer also
        a. Pick a pivot value
        b. Partition the list so that < pivot on left; > pivot on right
        c. put pivot value in between them
        d. Now repeat for each side (left and right - leaving out original pivot)
        e. Repeating until lists divide into 0 or 1 (sorted)

XVII. Time Complexity

XVIII. Inheritance

A. Properties
    1.)Implements an “is-a” relationship
    2.) Allows hierarchies to be built
    3.) Allows manipulating heterogeneous collections
    4.) Polymorphism: the type of the object, not the type of the reference, is used (at runtime) to determine which method to invoke
B. Limitations
    1.) To manipulate these heterogeneous collections, they must have a common ancestor with the method defined

XIX. Interfaces

A. Properties
    1.) an “acts-as” relationship

2.) define how an object must behave
3.) Example: public interface Weighable {
public int weight ( ); }
4.) only specifies required methods; may have super interface

B. Summary
1.) A collection of abstract methods
2.) Specifies behavior only by specifying the methods that are needed
3.) Cannot instantiate an interface (since it is abstract)
4. All methods are abstract
5.) Can be part of a hierarchy (super interfaces)

XX. Recursion

A. Recursion: A process that solves larger problems by dividing the problem into similar, but simpler problems (solve that problem, by solving that problem!)

B. Steps
1.) Find the simple, base-case(s)
a. trivial cases that can simply return the answer
2.) Find relationship from large case to simpler cases
a. Each sub-division of problem moves closer to the base-case

XXI. Collections

A. Collection: A repository for other objects

B. Properties:
1.) May be ordered (or not)
2.) Support add and remove, plus others

C. Java Collections Framework: a unified way to represent and manipulate a collection of objects
1.) Examples: ArrayList, LinkedList, Set, SortedSet, Queue, Deque