Cheatography

Exam 1 Cheat Sheet

by megphibbs via cheatography.com/55461/cs/14748/

ArrayLists

Useful Code:	`contains()`
	`subList()`
General Notes:	Good for creating an array with variable size
	Necessary to turn an array into a set

Sets

Important Methods:	`add(Element)`
	`addAll(Collection)`
	`containsAll(Collection)`
	`remove(Element)`
	`removeAll(Collection)`
TreeSets:	Time complexity - O(log(n))
	Organized in order from least to greatest
	All elements need a compareTo() method
HashSets:	Time complexity - O(1)
	Faster than TreeSets - organized more efficiently
	All elements need a HashCode
General Notes:	All items are unique
	Can declare using a list
	Length is dynamic

Maps

Important Methods:	`containsKey()`
	`containsValue()`
	`entrySet()`
	`keySet()`
	`remove()`
TreeMaps:	Time complexity: O(log(n))
	Keys are stored in a specific order (key must have a .compareTo())
HashMaps:	Time complexity: O(1)
	Keys are stored based on hash codes (key must have a .hashCode() method)
General Notes:	Maps are useful for key-value pairs

Maps (cont)

Efficient way to add things to map: loop through and check if it contains the key already (then add) or if it doesn't (create new object and put key)

File Input

Useful Code:	`Scanner scan = new Scanner('filename.txt');`
	`hasNext()`
	`hasNextInt()`
	`nextInt()`
	`next()`
	`useDelimiter()`
Useful Delimiters	" "

Types of Analysis

Empirical Analysis:	Measure run times, then plot and fit a curve
	Useful for predicting, but cannot explain
Mathematical Analysis:	Analyze algorithm to estimate number of operations as a function of input size
	Useful for both predicting and explaining
	Independent of machine/compiler
	Where Big O comes into play

Big O

Use	Determines the algorithmic complexity of something
	Figure out which strategy is the most efficient/least timely
Determining Big O	1. Determine a general function for the algorithm
	2. Strip away all constants and only keep term with the highest order
Useful Formulas	$1 + 2 + 4 + 8 + \ldots = 2^{n+1} - 1$
	$1 + 2 + 3 + 4 + 5 + \ldots = n(n + 1) / 2$
Efficiency	Algorithms with the smallest big O are the most efficient
	n^2 takes significantly longer to execute than n or 1

Comparing Objects

`==`	Useful to see if two variables point to the same object or for comparing primitives
	Cannot determine if two objects have the same elements
`.equals()`	Useful for comparing contents of objects/testing equality for strings
	Determines if two objects contain the same elements
`a.compareTo(b)`	Useful for putting objects in a specific order
	Returns < 0 if a < b
	Returns 0 if a is equal to b
	returns > 0 if a > b

Hashing

Making Effective Hash Codes	Be sure to create a hash code that depends on the order of things - for example, {"a", "b", "c"} should have a different code than {"b", "a", "c"}
	For objects with multiple instance fields, ensure that each variable has influence over the hash code
	Generally, things are added to the hashcode
	Multiply by prime numbers (37)
	Avoid using 0 - can mess things up
Collisions	Occur when two objects have the same hashcode
	Decreases performance/efficiency, but still yields correct results
	Don't use hashcodes as keys for this reason - in this case, collisions will cause errors
	Can use .equals() to see if two objects with the same hashcode are actually equal
Conjunction with .equals()	Every object that overrides .equals() MUST also override .hashCode() to prevent errors
	Only overriding one leads to conflicts in code.

NBody

General Notes:	Small timestep means more accurate (to a degree - overly small causes issues)
	Large timestep doesn't update frequently enough, which causes errors

Markov

General Notes:	Comparing efficiency of TreeMaps vs. HashMaps
	Looking at Big O Time functions
EfficientMarkov	Declares and instantiates a map in an init method, then accesses that map later on
	Better than MarkovModel because MM iterates through every single time (VERY inefficient)
WordGram	Purpose: creating a comparable object (possible to use in TreeMaps)
	Made a hashCode as well
	Used for EfficientWordMarkov
EfficientWordMarkov	Keys are WordGram objects
	More efficient that WordMarkovModel for the same reason as EfficientMarkov
Benchmark	Used for testing efficiency
	**Note: this is an example of empirical analysis
	Seeing how different methods change how much time it takes
	Also can be used to compare tree and hash maps

APT 1

CirclesCountry	Tested for circles that lay within one another
	Good way to learn efficient programming
LaserShooting	Added up different angles
	Struggled with this a lot - taught importance of casting doubles etc.
Totality	Takes input of string - either "odd", "even", or "all"
	Returns # of odd, # of even, or total #
SandwichBar	Takes two arrays as inputs - list of ingredients and list of sandwiches
	returns the index of the first sandwich that can be made with the ingredients listed
	first use of sets in this class
ClassScores	Takes an array of ints as input
	Returns the mode score - if there are multiple modes, return the array of them in numerical order
	Where TreeSets become useful
Gravity	Teaches the ability to solve a simple equation using Java

APT 2

Thesaurus	Never figured this one out
	Tested ability of decomposition
	Used retainAll() method
Anonymous	Takes in two String arrays (list of headlines and list of messages)
	Returns the number of messages that can be constructed using only letters in the headlines
	Made use of String.trim()
SimpleWordGame	Takes in two String arrays (list of words in set, list of player guesses)
	Each correct guess receives a score of guess.length() * guess.length()
	Returns the sum of all of the players scores
MemberCheck	Takes in three string arrays (club1, club2, club3)
	Returns the list of members who attended more than one club
	Makes use of retainAll(), nested for loops, uniqueness of sets
ServiceNames	First time using maps
	Maps a specific input to the types of services it offers