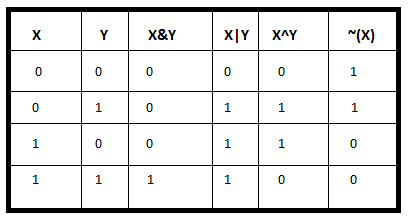
Technical Interview Cheatsheet

<https://hackernoon.com/14-patterns-to-ace-any-coding-interview-question-c5bb3357f6ed>

## Terms:

1. Static vs dynamic programming language:
   1. Dynamic performs common programming behavior at runtime while static does them at compile time
   2. Behaviors include extension of the program by adding new code, by extending objects and definitions, or by modifying the type system
   3. Most dynamic languages are also dynamically typed but not all are
   4. Most dynamic languages referred to as scripting languages
   5. Dynamic languages: JavaScript, Python, Ruby, PHP
   6. Static languages: C, C++, Java
2. Polymorphism (OOP):
   1. When an object can take many forms. In Java, every object “is-a” their own type as well as an Object class so they are all polymorphic.
   2. Example: public class Deer extends Animal implements Vegetarian (interface). Deer is-a: animal, vegetarian, deer, and object.
   3. Method Overriding vs. Overloading:
      1. Override: inheritance, different class but same method name/parameters
      2. Overload: same class, same method name, different parameter
      3. Virtual keyword in C++ marks the method as inheritable and overridable
3. Inheritance vs. composition (OOP):
   1. extends (is-a) vs. implements (has-a)
   2. Both can be used with classes, usually implement interfaces.
   3. Class cannot extend multiple classes in Java but can implement multiple interfaces.
   4. Class A,B,C → B extends A or B extends A, C extends B or B extends A, C extends A
   5. In has-a relationship, there is an instance of a class in another class so code does not need to reused. E.g. class speed, class van. Inside van there is private Speed sp object.
4. Interface: (abstraction, abstract classes)(OOP)
   1. Can never be extended by a class
   2. Hides details of implementation, just has method headers
   3. If class has an abstract method, it must be abstract class
   4. Cannot instantiate an abstract class, must inherit it
   5. Interface can extend multiple interfaces
5. Encapsulation (OOP):
   1. A way of data hiding so that a variable is declared private and can only be accessed through public setters and getters.
   2. Singleton, Factory design patterns
6. Virtual:
   1. In C++ used to help with polymorphism and overriding
7. Memory (stack vs. heap):
8. Iterator:
9. Struct:
10. Value type vs. reference type:
11. Scope/Variable:
    1. Public: visible to all classes
    2. Private: visible only to class they belong, highest level of encapsulation, private method cannot be overridden, making ctor private is how to prevent it from being subclassed (singleton is example)
    3. Protected: visible only to class they belong and any subclasses (classes that extend your class)
    4. Static: lifetime of entire run of the program, memory allocated at compile time before program is executed, can be accessed before class is even instantiated
    5. Final: assigned permanently and can’t be changed, reassigned, subclassed, inherited
    6. Const: in C, a variable that is read-only, compile-time, may change its value during run time
    7. Local: in the function or block where it was declared
    8. Global: visible throughout the program
    9. Volatile: value may change between different accesses, even if it does not appear to be modified, primarily arise in hardware access (memory-mapped I/O)
12. Dynamic programming: breaking a problem into smaller problems and solve recursively
13. UML (unified modeling language)
14. Pointers:
15. Common memory and CPU time optimization techniques:
16. Strings: methods in java → charAt(int index), boolean contains(CharSequence s), char[] toCharArray()
17. Bit manipulation:



Two’s Complement:

* A negative # is represented as the two’s complement of its absolute value with 1 in ints sign bit to indicate negative

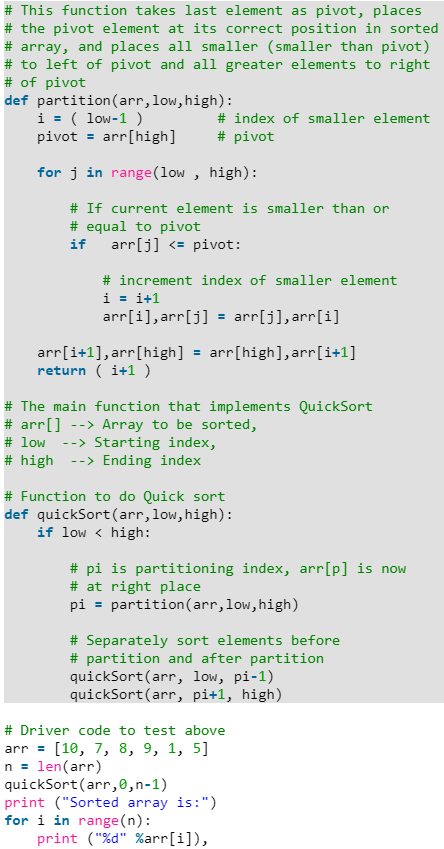
1. Take a neg number: -7
2. Bit representation of abs(-7)=7: 0111 (0\*2^3 + 1\*2^2 + 1\*2^1 + 1\*2^0 = 0 + 4 + 2 + 1)
3. Flip bits: 0111 → 1000
4. Add one: 1001
5. Put 1 in sign bit to make negative (if it’s not already there): 1001 = -7 in binary

Bit Shifting:

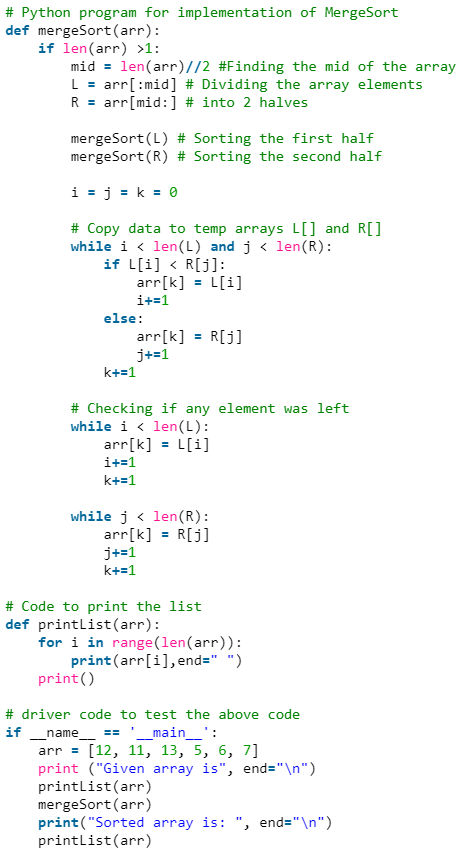
* >>> logical right shift: visually shifting all bits to the right, add 0 to most significant bit
* >> arithmetic right shift: shift bits to the right, fill in new bits with value of sign bit. Roughly divides the number by 2.

## Sorting:

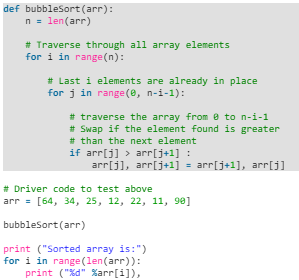
1. Quicksort:
   1. Pick a pivot and sort array so everything smaller than pivot is to left and bigger is to right. Do the same on first half and second half recursively.
   2. Best: O(n), avg: O(nlogn), worst is O(n2) but rare



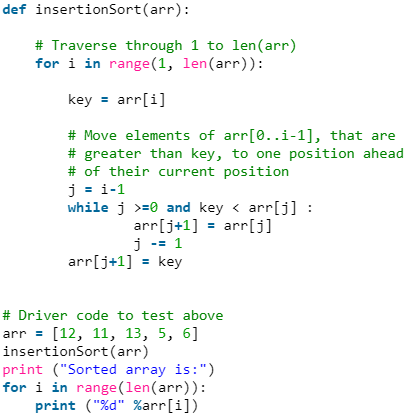
1. Merge sort:
   1. Split the array in half until each one element, compare two at a time, sort, merge → then merge 2 and 2 by sorting.
   2. Best case: O(n), avg/worst case: O(nlogn)



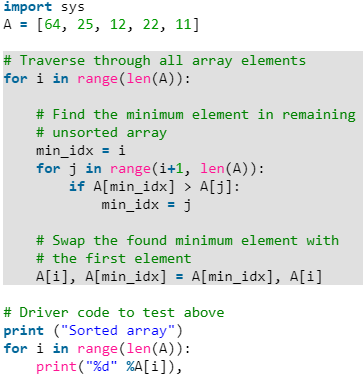
1. Bubble sort:
   1. Start from front and compare 2 adjacent, swap if wrong order. Then do next two. Keep repeating.
   2. Best: O(n), avg/worst: O(n2)
   3. Easy to implement but worst performance



1. Insertion sort:
   1. Start with first element, iteratively compare to next one then sort. Move on to next element, compare with first 2, sort.
   2. Less efficient on larger data like merge, quick, heap but simpler to implement
   3. N2 - best case n



1. Selection sort:
   1. Go through and find smallest (or largest), move to front (or back). Essentially having sublist of sorted and unsorted. Keep going through and finding next smallest and moving to 2nd position in front
   2. N2



## Big O:

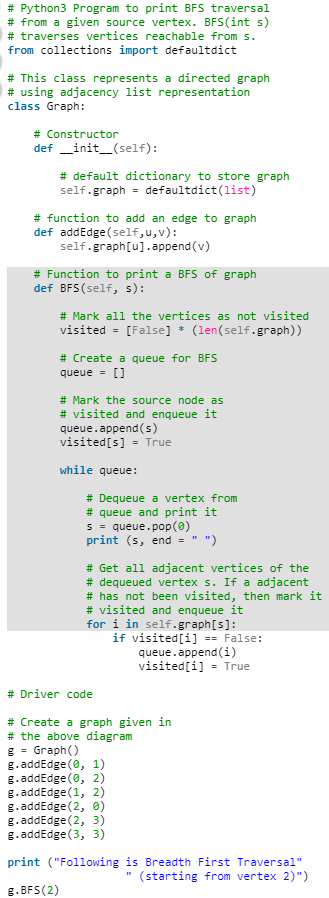
|  |  |
| --- | --- |
| O(1) | If statements, copying an element |
| O(logn) |  |
| O(sqrt(n)) | isPrime |
| O(n) | One loop |
| O(mn) | Nested loops with different array lengths |
| O(nlogn) | sorting |
| O(n^2) | Nested loops with same array length |
| O(2^n) | Recursive, with trees = O(calls to rec branches^depth), fibonacci |
| O(n!) | factorial |

## Data Structures:

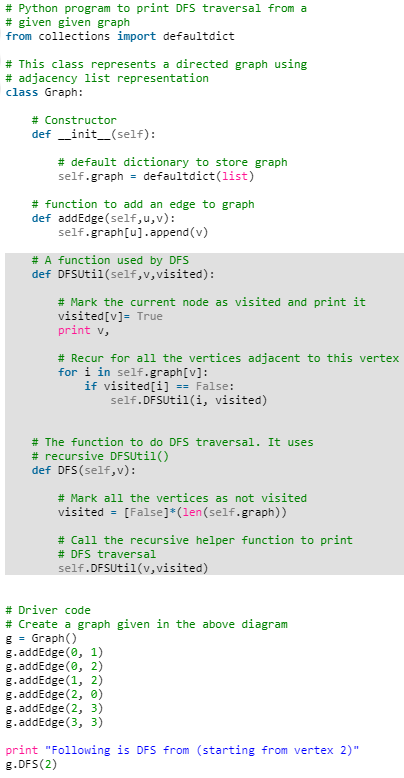
1. Hash table/map:
   1. Map key to values, array as memory, can have collision if hash is bad
   2. Access, insert, search: O(1)
   3. Implement:
      1. Array of linked lists
      2. Hash code
      3. Map hash code to index in array e.g. hash(key) % array.length
2. Array:
   1. Java: int [ ] A = new int[10]; //// C++: int A [10];
   2. Fixed size, indexed, contiguous memory, elements all same type
   3. Access by index: O(1)
   4. Search, insertion: O(n)
3. Linked list:
   1. List of nodes with data and pointer to next (and prev)
   2. Index, search: O(n)
   3. Insertion: O(1)
4. Vector/arraylist:
   1. Variable size, growing array, ArrayList, backed by fixed size array and resize as needed
   2. Index, search, insert: O(log n)
5. Trees:
   1. A type of graph but simpler to implement than graph. It’s a connected graph without cycles.
   2. Cannot contain cycles
   3. Has a root node that has 0+ child nodes which in turn have 0+ child nodes
   4. Typically has to do with recursion
   5. Data can be in no order
   6. Any data type
   7. Binary tree
      1. Each node can have up to 2 children
      2. Binary search tree (BST):
         1. Has the specific rule that left < root < right
      3. Traversals: Inorder: L, Root, R; PreOrder: Root, L, R; PostOrder: L, R, Root (all are forms of DFS graph search)
      4. Perfect binary tree has 2^k - 1 nodes (all levels full)
   8. Can be balanced or unbalanced
      1. Balanced
         1. Means O(log n) for insert and find
         2. Red-black trees
         3. AVL trees
   9. Heaps (min/max):
      1. Getting min/max is top element O(1)
      2. Inserting at rightmost spot then bubble up to correct position O(log n)
   10. Tries (prefix tree)
       1. N-ary tree in which characters are stored in each node
       2. Paths may represent a word (problems related to prefixes of words, lists of valid words)
       3. Might have a special way to signify end of complete word: special node or boolean flag
       4. Node could have anywhere from 0 to ALPHABET\_SIZE children (or 1 - A\_SIZE - 1 if using a special terminating node)
       5. As fast as hash table when checking if string is valid prefix. O(k) where k = length of string
6. Graph:
   1. Collection of nodes with edges between them
   2. Directed or undirected
   3. Can contain multiple subgraphs. Connected graph is when there’s a path between every pair of vertices.
   4. Cyclic or acyclic
   5. Representation:
      1. Adjacency list
         1. Most common way
         2. Every vertex (node) has list of adjacent vertices (in undirected graph, edges would be stored twice: once for each vertex’ list)
         3. Can be represented using array (or hash table) of lists
      2. Adjacency matrix
         1. NxN (N=# of nodes) boolean matrix where if matrix[i][j] = true, there’s an edge from node i to j. In undirected graph, matrix will be symmetric.
         2. Less efficient than adjacency list because need to iterate through all nodes to find neighbors instead of just looking at nodes list of neighbors
7. Stacks:
   1. LIFO, implement with array or linked list
   2. Abstract data type mean its implementation is hidden and we can put any data type in it
   3. Pop, push, peek
   4. Made with linked list where head is only place for inserting and deleting
   5. Uses: recursion/iterative, find matching pair
8. Queues:
   1. FIFO, ADT, use doubly linked list that only removes from head and adds to tail
   2. Enqueue, dequeue, peek
   3. Uses: BFS, implementing a cache

## Algorithms:

1. BFS:
   1. Searches tree (or graph) by levels starting at root
   2. Explore each neighbor before going on to any children
   3. Good for searching trees that are more wide than deep
   4. Good for finding shortest path (or any path): start a BFS on start and dest node and where they collide you found the shortest path
   5. Uses queue, more memory than DFS
   6. Search: O(|E|+|V|)
   7. Looping algorithm



1. DFS:
   1. Searches tree (or graph) by depth first starting at root
   2. Traverse left until hits leaf node, then goes back up looking at right and all its left children
   3. Good for searching trees that are deeper than wider
   4. Often preferred if we want to visit every node in the graph and it’s a bit simpler
   5. Uses stack, less memory than BFS since it doesn’t need pointers to keep track of next node
   6. Search: O(|E|+|V|)
   7. Recursive algorithm



1. Binary search:
2. Divide and conquer:
   1. Algorithm that involves recursively splitting a problem into smaller problems until they can be solved directly and merging the solutions (quicksort, merge)
3. Dijkstra’s
4. Other greedy algo
5. Topological sort
   1. Can only be done on a directed acyclic graph
   2. First element has zero in-degree, nothing depends on it so it can be first
   3. DFS with a stack
   4. Used for questions involving scheduling, something depends on something else
   5. Algo
      1. Init visited list of size numVertices
      2. Init empty stack
      3. Loop numVertices
         1. If not visited current vertex
            1. Call topSortUtil(vertex, visited, stack)
      4. Print stack
      5. In topSortUtil:
         1. Mark vertex as visited
         2. Loop thru all neighbors of vertex
            1. If neighbor not visited

Recurse call topSortUtil

* + - 1. When all neighbors visited and none left, insert current vertex into stack at 0
  1. Time: O(V+E) since DFS, space: O(V) for the stack

## Operating Systems:

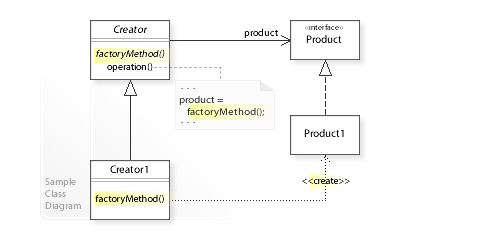
1. Semaphore: generalized mutex that has a lock (flag)
2. Mutex: for mutual exclusion, processes take turns (producer, consumer)
3. Process: instance of program in execution (web browser)
   1. States: running, ready, waiting
   2. Only one process can be running/have CPU
4. Thread: single sequence stream within a process (browser has multiple tabs)
5. Context switch: time spent switching between 2 processes (multitasking). Need to save the state info of current process and bring in state info from waiting processes into memory.
6. Deadlock: 2 or more processes wait for each other but never end. All 4 of these makes deadlock occur:
   1. Mutual exclusion: cannot share a resource, one thread of execution never enters its critical section at the same time that another concurrent thread enters its own
   2. Hold and wait: process holding resource while waiting for another
   3. No preemption: OS cannot take a resource back
   4. Circular wait: set of processes waiting for each other in circular form
      1. In order to prevent deadlock, get rid of one of these.
7. Thrashing: when system crashes by spending time processing page faults rather than executing transactions

## Linux/UNIX:

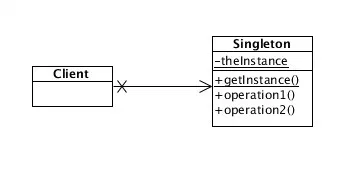
1. Grep: search for string in a file
2. Find: search file by name
3. Ssh: login to remote host
4. Diff: output differences between two files
5. Sort: sort files in ascending order (use -r for descending)
6. Ls: list files
7. Cd: change directory
8. Mkdir: make directory
9. Rm: remove
10. Cp: copy file
11. Mv: rename file
12. Cat: print to screen
13. man : manual of a command
14. Chmod: change permissions (chmod ug+rwx filename.txt)
15. Clear: clears the screen of commands
16. kill : stop a process
17. Pwd: display pathname to current directory
18. Who: shows who is logged on

## Software Design:

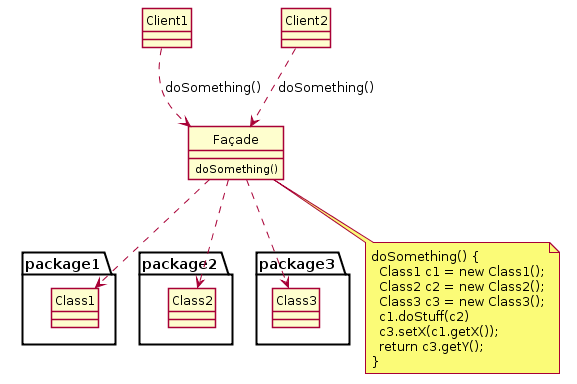
1. Factory
   1. Uses factory methods to deal with problem of creating objects without having to specify the exact class of the object that will be created
   2. Lets a class defer instantiation it uses to subclasses that can override them to specify the derived type of product that will be created



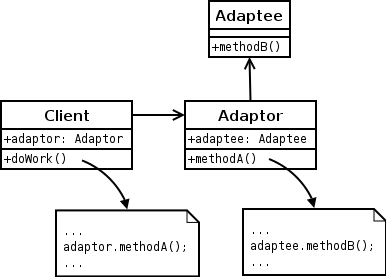
1. Singleton
   1. One instance of the class exists
   2. Global access to that instance
   3. All constructors to the class are private
   4. Static method that returns a reference to instance



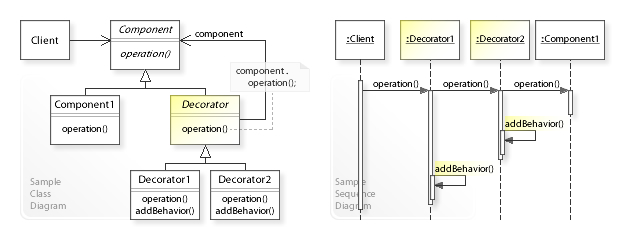
1. Facade:
   1. Hides the complexities of larger system and provides simpler interface to client
   2. Single wrapper class that contains a set of members required by client, members access the system on behalf of the facade client and hide the implementation details



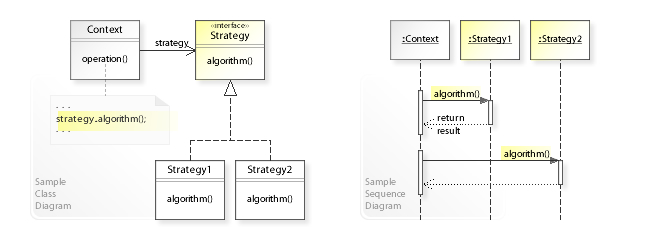
1. Adapter
   1. Converts one interface to another so that it matches what the client is expecting
   2. Polymorphic behavior
   3. Used to make existing classes work with others without modifying their source code
   4. Defines a separate adapter class that converts an incompatible interface of a class into another interface that clients require, works through the adapter to reuse classes that do not have the required interface



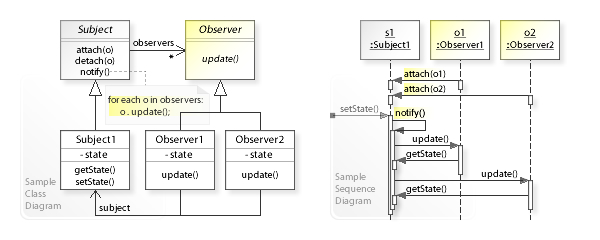
1. Decorator
   1. Dynamically adds responsibility to the interface by wrapping the original code
   2. Add or alter behavior of individual object at run-time without affecting behavior of other objects from same class
   3. Follows Single Responsibility Principle since it allows functionality to be divided between classes with unique areas of concern
   4. Decorator class maintains reference to decorated object and forwards all requests to it so it is invisible to clients of the decorated class



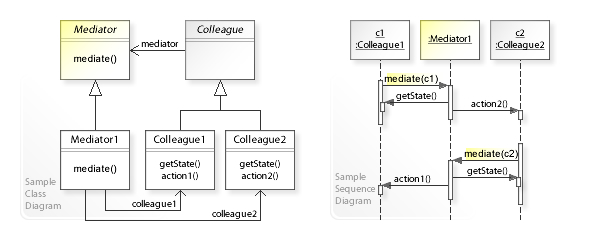
1. Strategy
   1. Enables selecting an algorithm at runtime
   2. Lets algorithm vary independently from clients that use it
   3. Typically stores a reference to some code in data structure and retrieves it



1. Observer
   1. One-to-many dependency between objects so when one object changes state, all its dependents are notified by calling one of their methods
   2. Mainly used to implement distributed event handling systems in “event driven” software
   3. Suits process where data arrives through I/O
   4. Loosely coupled subject and observer(s), no explicit knowledge of each other
      1. Subject maintains list of observers and notifies them of state changes by calling their update() method
      2. Observers (un)register themselves on a subject and to update their state (synchronize state with that of their subject) when notified. Observers can be added and removed independently at run-time



1. Mediator
   1. Defines an object that encapsulates how a set of objects interacts
   2. Communication between objects is encapsulated within mediator object rather than communicating directly
   3. Reduces dependencies → reduces coupling



Sample Question:

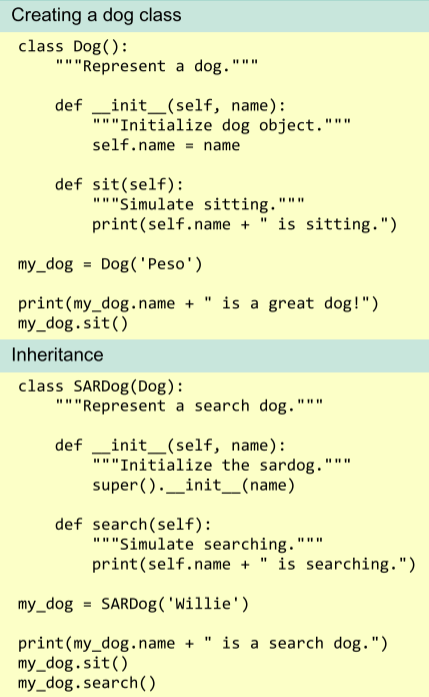
* Design a parking lot

## SOLID:

1. **Single responsibility**: every module, class, or function should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class
2. **Open-closed principle**: software entities should be open for extension but closed for modification
3. **Liskov substitution**: if S is a subtype of T, then objects of T may be replaced with objects of type S without altering any of the desirable properties of the program
4. **Interface-segregation**: no client should be forced to depend on methods it does not use, splits interfaces that are very large into smaller and more specific ones so that clients will only have to know the methods that they care about
5. **Dependency inversion**: decouples software modules, high-level modules should not depend on low-level modules and both should depend on abstractions, abstractions should not depend on details and details should depend on abstractions

## Python:

1. strings
   1. str.find(sub[, start[, end]] )
   2. String[::-1] // reverses string
   3. String[start:stop:step] // not inclusive of stop
   4. reversed(string) // reverses string
   5. int(str(num)[::-1]) // reverse number
   6. string.find(string[::-1]) // find palindrome
2. Lists
   1. List.count(2) // returns how many 2 in List
   2. “ “.join(LIST)
   3. max(LIST) // max num
   4. list.copy()
   5. list.append() → insert at end
   6. list.insert(0, ‘hi’) → insert at index 0
   7. del list[0] → delete at index
   8. list.remove(‘hi’) → delete by value
   9. list.pop(#) → delete last inserted value or with #, at index
   10. list.reverse()
   11. list.sort() or list.sort(reverse=True)
   12. Nums = list(range(1001)) → list from 0-1000
   13. min(list), max, sum
3. List Comprehension
   1. Squares = [x\*\*2 for x in range(1,11)] or [x\*\*2 for x in numbers] where numbers is another list with nums
   2. common\_num = [a for a in list\_a for b in list\_b if a == b] where 2 lists of nums
   3. square\_cube\_list = [ [a\*\*2, a\*\*3] for a in list\_a] which produces list of lists with squares and cubes of nums in list\_a
   4. Slicing: list[:2] gets first 2 elements
4. Dictionaries
   1. For key, value in dict.items()
   2. For key in dict.keys()
   3. For val in dict.values()
5. Classes
   1. Inheritance



1. Other
2. A, b = b, a // Swapping two variables
3. collections.Counter([1, 2, 2]) → Counter({2:2, 1:1})
4. M = collections.OrderedDict((str(x),x) for x in range(10)) // remembered order
5. B = a // Shallow copy
6. Fibonacci: lambda x: x if x<=1 else fib(x-1) + fib(x-2)
7. User input:
   1. Name = input(“what’s your name?”) → always stored as a string, must convert if num

## Java:

1. Collection framework: arraylist is dynamically resizing array, vector is similar to arraylist but is synchronized, LinkedList is built in class, and hashmap is builtin
2. Final statement: for var, value cannot change; for var ref, it cannot point to other object; for method, it cannot be overridden; for class, it cannot be subclassed
3. Finally keyword: optional block with try/catch statement that will always be executed. Usually for clean-up code right after the block.
4. finalize(): auto garbage collector calls this just before destroying object

## General questions:

1. Describe object oriented programming
   1. Focus on manipulating objects and data rather than logic
   2. Objects are nouns, methods are verbs that manipulate the objects
   3. 4 facets: polymorphism, encapsulation, interface, inheritance
2. Difference between C and C++
   1. C++ can basically do what C does, similar syntax but C++ has an exception handling system (try, catch, throw), is object-oriented so can make use of classes, C++ has constructors (C user has to make structs)
   2. C is procedural (routines, subroutines, instructions of code executed one after another), C++ is multi-paradigm, Java is object oriented
   3. No explicit pointers in Java
   4. C/C++ allocate memory using malloc and free, Java has garbage collector
   5. C++ has operator overloading, Java has method overloading, no overloading in C

## Coding Portion:

* Restate problem
* Can input be null? Empty? Negative? Zero?
* What type are the inputs and outputs?
* Size restrictions?
* Account for overflow/underflow?
* Space/time constraints?
* Returning or print?
* TEST CODE