Week 3 Day 3

Java OOP and Algorithms

REVATURE

Java: Object Oriented Programming

REVATURE

- There are four pillars of object-oriented programming
  - Remember APIE
- Abstraction
  - Hiding details of implementation
- Polymorphism
  - Object taking many forms
- Inheritance
  - Objects taking states and behaviors from a super class
- Encapsulation
  - Data hiding

Java OOP: Encapsulation

REVATURE

- Hiding data, but giving access through mutators
- Java uses access modifiers to hide data
- Mutators
  - Methods available to other classes to share the states we are hiding
  - Denoted by getters and setters,
    - getName()
    - setName()

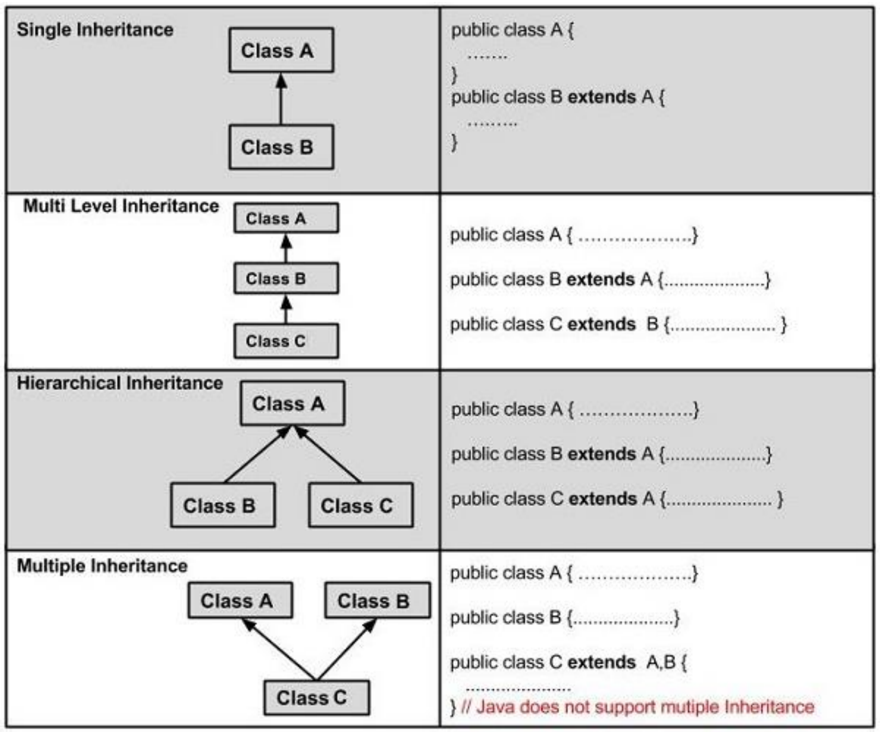
Java OOP: Inheritance

Base class (parent) passes traits and behaviors to sub-class (child)

- All non-private fields/methods are passed to the sub-class
- Shadowing
  - Variables with the same name in super and sub-classes will be shadowed
  - The variable of the parent class is overridden by the value of the child class

Java OOP: Types of Inheritance

REVATURE



- Single Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- No multiple inheritance allowed with classes,
  - Possible with interfaces

Java OOP: Abstraction

Centralize common characteristics and generalize behavior into conceptual classes

- Abstract Classes
  - General classes that cannot be instantiated
  - Created with the abstract keyword
  - Contains abstract and concrete methods
  - Use the extends keyword
- Interfaces
  - Contracts for classes with methods to implement
  - Inherently public abstract
  - All methods are public abstract
  - Uses the implements keyword

Java OOP: Polymorphism

REVATURE

The ability for an object to take on many forms

- Method Overloading
  - Methods with same name, but different signatures
- Method Overriding
  - Method in child class with same name and signature as method in parent class
- Covariant return type
  - While overriding you can change the return type, access modifier, and exception type
- Casting
  - Upcasting assigns child object to parent
  - Downcasting assigns parent object to child

Time Complexity

Study of how efficient an algorithm is

- Big O
- Omgea
- Theta

Big O is the most typical

- Worst case for the algorithm
- How many possible operations

Time Complexity: Big O Notation

Denote n as the number of operations

- O(1): constant time
- O(log n): logarithmic
- O(n): linear
- O(n log n): n logarithmic
- O(n^x): exponential

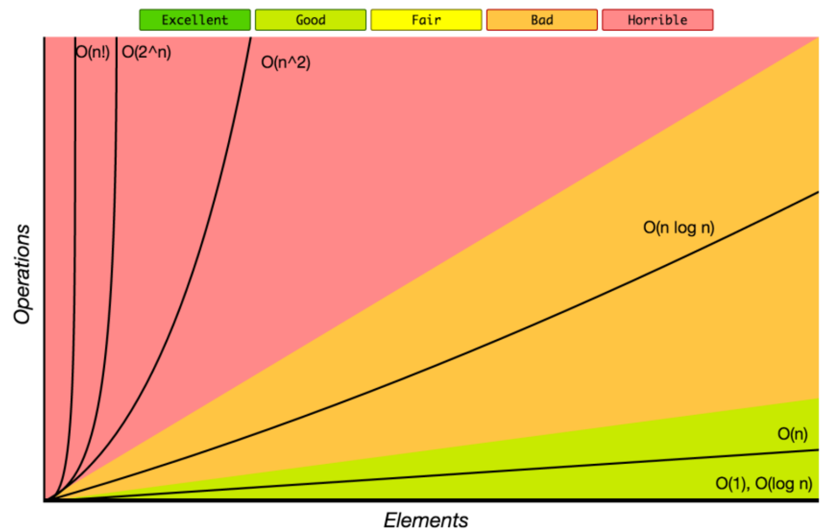
Time Complexity: Determining Big O

- General rules for determining Big O
  - No loops: Constant time
  - Dividing work/Divide and Conquer: logarithmic
  - Single loops: linear
  - Nested loops: exponential

Big O Sheet

REVATURE

Big-O Complexity Chart



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Algorithms: Linear Search

Most efficient algorithm to search an unsorted list

- Loop through the list once
- Return the index you found the item, or -1 if not found
- Algorithm time complexity
  - O(n)

Algorithms: Binary Search

REVATURE

Efficient algorithm to find an element in a sorted list

- MUST be a presorted list
- Split the list in half and check the value
  - Search the left or right half depending on the value
  - Continue splitting in half until you find the value or determine its not there
- The algorithm splits its work over an over, meaning it has a time complexity of:
  - O(log n)