Computer Science Review

Chang-Hyun Mungai
v2019

General Primitives

Boolean:

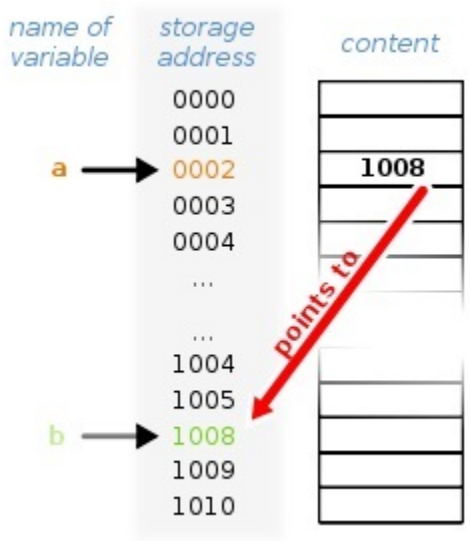
- true or false
- some languages 0 is false
  - C
  - not Java

Floating-Point Number:

- float
  - faster
  - only use to operate on a lot of floating- point numbers (think in the order of thousands or more) and analysis of the algorithm has shown that the reduced range and accuracy don't pose a problem
  - scientific notation in base 2
- double
  - more precise
  - use in default
  - long double can be used if you need more range or accuracy than double
- real

Character:

Pointers



Definition:

- object whose value refers to another value stored elsewhere in the computer memory using its memory address

---

Syntax:

- C
  - `int *ptr;`
    - This declares ptr as the identifier pointer that points to an object of type int

```
int a = 5;
   int *ptr = NULL;
   ptr = &a;
```

  -
    - Assigns the value of the address of a to ptr
    - example: if a is stored at memory location of 0x8130 then the value of ptr will be 0x8130 after the assignment
    - To dereference the pointer, an asterisk is used again
  - `*ptr = 8;`
    - This means take the contents of ptr (which is 0x8130), "locate" that address in memory and set its value to 8

---

sources:

- https://en.wikipedia.org/wiki/Pointer_(computer_programming)

Ints

Representation:

- Binary
- Two's Complement
- Hexadecimal

Division and Modulus:

- Division is floored

Notes:

- Finite memory for infinitely many integers
- Arithmetic overflow (after max is -, before min is +)
  - when dividing Integer.Min by -1
  - in multiplication, addition, subtraction

Bit Manipulation

Basics:

And (&):	0 & 0 = 0	1 & 0 = 0	0 & 1 = 0	1 & 1 = 1
Or (|):	0 |0 = 0	1 |0 = 1	0 |1 = 1	1 |1 = 1
Xor ($\bigwedge$):	0 ^0 = 0	1 ^0 = 0	0 ^1 = 0	1 ^1 = 1

Shifts:

- Left Shift: If you run out of space the bits drop off
  - Both arith and logical shift in 0

  1. 00011001 <<2 = 01100100
  2. 00011001 <<4 = 10010000

- Right Shift if you run out of space the bits drop off
  - arithmetic shift - shift in sign bit (sticky shift), logical-shift in 0

  1. 00011001 >>2 = 00000110
  2. 00011001 >>4 = 00000001

Notes:

- Windows calculator can do operations in binary, view programmer

Arrays

Big O:

- space O(n)
- time
  - access worst O(1), average O(1)
  - search worst O(n), average O(n)
  - insert worst O(n), average O(n)
  - delete worst O(n), average O(n)

ArrayList (Dynamically Resizing Array):

```
public ArrayList<String> merge(String[] words, String[] more) {
   ArrayList<String> sentence = new ArrayList<String>();
   for (String w : words) sentence.add(w);
   for (String w : more) sentence.add(w);
   return sentence;
 }
```

Notes:

- Index starts with 0

Big-O Cheat Sheet

Preface

This is a LaTeX'ed version of http://bigocheatsheet.com/ (as of 17 February 2015).

Legend: Good Fair Poor

Searching

Algorithm	Data Structure	Time Complexity Average	Worst	Space Complexity Worst
Depth First Search (DFS)	Graph of $\|V\|$ vertices and $\|E\|$ edges	-	$O(\|E\| + \|V\|)$	$O(\|V\|)$
Breadth First Search (BFS)	Graph of $\|V\|$ vertices and $\|E\|$ edges	-	$O(\|E\| + \|V\|)$	$O(\|V\|)$
Binary search	Sorted array of n elements	$O(\log n)$	$O(\log n)$	$O(1)$
Linear (Brute Force)	Array	$O(n)$	$O(n)$	$O(1)$
Shortest path by Dijkstra, using a Min-heap as priority queue	Graph with $\|V\|$ vertices and $\|E\|$ edges	$O((\|V\| + \|E\|) \log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$	$O(\|V\|)$
Shortest path by Dijkstra, using an unsorted array as priority queue	Graph with $\|V\|$ vertices and $\|E\|$ edges	$O(\|V\|^2)$	$O(\|V\|^2)$	$O(\|V\|)$
Shortest path by Bellman-Ford	Graph with $\|V\|$ vertices and $\|E\|$ edges	$O(\|V\|\|E\|)$	$O(\|V\|\|E\|)$	$O(\|V\|)$

Sorting

Algorithm	Data Structure	Time Complexity Best	Average	Worst	Worst Case Auxiliary Space Complexity Worst
Quicksort	Array	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$
Mergesort	Array	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heapsort	Array	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Bubble Sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bucket sort[a]	Array	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(nk)$
Radix sort[b]	Array	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

[a] Only for integers with range k
[b] Constant number of digits 'k'

Graphs

Node/Edge Management	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency list	$O(\|V\|+\|E\|)$	$O(1)$	$O(1)$	$O(\|V\|+\|E\|)$	$O(\|E\|)$	$O(\|V\|)$
Incidence list	$O(\|V\|+\|E\|)$	$O(1)$	$O(1)$	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$
Adjacency matrix	$O(\|V\|^2)$	$O(\|V\|^2)$	$O(1)$	$O(\|V\|^2)$	$O(1)$	$O(1)$
Incidence matrix	$O(\|V\|\|E\|)$	$O(\|V\|\|E\|)$	$O(\|V\|\|E\|)$	$O(\|V\|\|E\|)$	$O(\|V\|\|E\|)$	$O(\|E\|)$

Data Structures

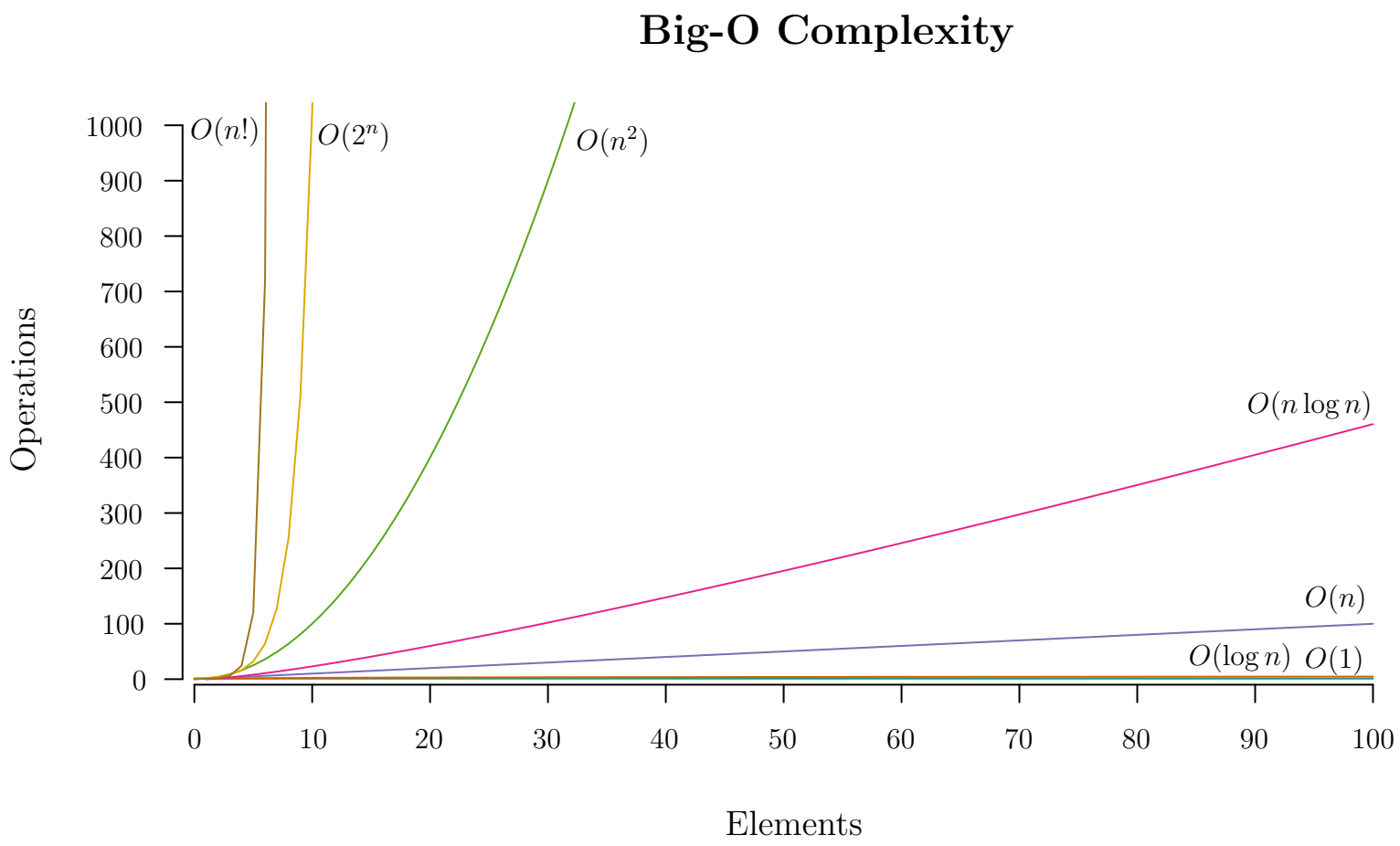
Data Structure	Time Complexity Average Indexing	Search	Insertion	Deletion	Worst Indexing	Search	Insertion	Deletion	Space Complexity Worst
Basic Array	$O(1)$	$O(n)$	-	-	$O(1)$	$O(n)$	-	-	$O(n)$
Dynamic Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Singly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
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	-	$O(1)$	$O(1)$	$O(1)$	-	$O(n)$	$O(n)$	$O(n)$	$O(n)$
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	-	$O(\log n)$	$O(\log n)$	$O(\log n)$	-	$O(n)$	$O(n)$	$O(n)$	$O(n)$
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	-	$O(\log n)$	$O(\log n)$	$O(\log n)$	-	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

Heaps

Heaps	Time Complexity Heapify	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Linked List (sorted)	-	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(m+n)$
Linked List (unsorted)	-	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Binary Heap	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m+n)$
Binomial Heap	-	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Fibonacci Heap	-	$O(1)$	$O(\log n)$[a]	$O(1)$[a]	$O(1)$	$O(\log n)$[a]	$O(1)$

[a] Amortized

Big-O Complexity Chart



Linked Lists

Big O:

- space O(n)
- time
  - search worst O(n), average O(n)
  - insert worst O(1), average O(1)
  - delete worst O(1), average O(1)

Advantages:

- Linked lists are a dynamic data structure, allocating the needed memory while the program is running
- Insertion and deletion node operations are easily implemented in a linked list
- Linear data structures such as stacks and queues are easily executed with a linked list
- They can reduce access time and may expand in real time without memory overhead

Disadvantages:

- They have a tendency to use more memory due to pointers requiring extra storage space
- Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access
- Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list
- Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards[1] and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer

Uses:

- Stack
- Queue

- Memory Allocation

Creating a Linked List:

```
class Node {
   Node next = null;
   int data;
   public Node(int d) { data = d; }
   void appendToTail(int d) {
      Node end = new Node(d);
      Node n = this;
      while (n.next != null) { n = n.next; }
      n.next = end;
   }
}
```

Deleting a node:

```
Node deleteNode(Node head, int d) {
   Node n = head;
   if (n.data == d) {
      return head.next; /* moved head */
   }
   while (n.next != null) {
      if (n.next.data == d) {
         n.next = n.next.next;
         return head; /* head didnt change */
      }
      n = n.next;
   }
}
```

Notes:

- Alternative to array to implement stack and queue
- Allows any length

Queues

Implementing a Queue:

```
class Queue {
   Node first, last;
   void enqueue(Object item) {
      if (!first){
         back = new Node(item);
         first = back;
      } else {
         back.next = new Node(item);
         back = back.next;
      }
   }
   Node dequeue(Node n) {
      if (front != null) {
         Object item = front.data;
         front = front.next;
         return item;
      }
      return null;
   }
}
```

Notes:

- First in First out

Stacks

Implementing a Stack:

```
class Stack {
   Node top;
   Node pop() {
      if (top != null) {
      Object item = top.data;
      top = top.next;
      return item;
      }
      return null;
   }
   void push(Object item) {
      Node t = new Node(item);
      t.next = top;
      top = t;
   }
}
```

Notes:

- Last in First out

Hash Tables

Big O:

- space O(n)
- time
  - search worst O(n), average O(1)
  - insert worst O(n), average O(1)
  - delete worst O(n), average O(1)

Advantages:

- faster than other structures on large entries
- efficient when max entries known (dont have to resize)

Disadvantages:

- have to resize for more data

Uses:

- associative arrays (arrays index through arbitrary strings)
- database indexing
- caches
- sets (?)
- object rep (key is method or object, value is pointer to member or method)

Properties:

- keys have to be hash able (able to compute numeric value from it)
- entries in no particular order

Creating a Hash Table:

```
public HashMap<Integer, Student> buildMap(Student[] students) {
   HashMap<Integer, Student> map = new HashMap<Integer, Student>();
   for (Student s : students) map.put(s.getId(), s);
   return map;
}
```

Notes:

- Alternative to array to implement stack and queue
- Allows any length
- can be made more efficient with better fit hash function

Dictionary/Map

Operations:

- Add(K key, V value) adds given key-value pair in the dictionary. With most implementations of this class in .NET, when adding a key that already exists, an exception is thrown.
- Get(K key) returns the value by the specified key. If there is no pair with this key, the method returns null or throws an exception depending on the specific dictionary implementation.
- Remove(key) removes the value, associated with the specified key and returns a Boolean value, indicating if the operation was successful.
- Contains(key) returns true if the dictionary has a pair with the selected key
- Count returns the number of elements (key value pairs) in the dictionary

Notes:

- Abstract Data Structure
- "map" or "associative array"
- maps keys to values
- hash table is one implementation

Tries

Associated trees

Advantages (over BST):

- no collisions
- no hash function

Disadvantages:

- slower than hash table
- some keys can be meaningless (floating point nos)

Applications:

- dictionary (autocomplete)

Operations:

- look-up
- insert

Notes:

- bitwise vs compressive implementation

Heap

Binary Heap Operations:

- findmax (O(1))
- insert (O(log n) (Binary Imp))
  - insert item into next place in the BT
  - Swap item up with parent until heap invariant is maintained
- remove-max (O(log n) (Binary Imp))
  - Remove the root (and store it to return)
  - Place the last element inserted at the root
  - Swap item down with child until heap invariant is maintained

Applications:

- heapsort
- graph algorithms
- order stats
- Priority Queue

Notes:

- tree-based structure that satisfies heap property (max heap parent greater than or equal to children)

sources:

- `http://interactivepython.org/runestone/static/pythonds/Trees/BinaryHeapImplementation.html`

Common Binary Search Trees

Big O:

- space O(n)
- time
  - search worst O(n), average O(log(n))
  - insert worst O(n), average O(log(n))
  - delete worst O(n), average O(log(n))

BST Node:

- Data
- Left Child
- Right Child

Comparison:

- advantages
  - related sorting algorithms
  - search algorithms
  - inorder traversal
- disadvantages
  - shape depends on insertions
  - keys has to be compared when inserting or searching
  - height grows n, which grows much faster than log n
- uses
  - sets, multisets, associative arrays, priority queue

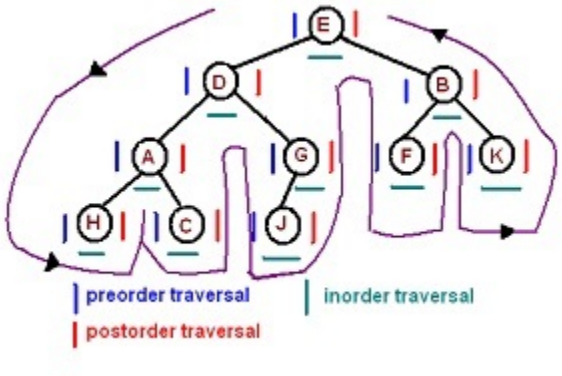
Operations:

- Look-Up

  - compare to current node: go left if less, go right if greater

- Insert
  - Same as look-up but replace node where it should be
- Delete
  - no children: just delete
  - 1 child: remove node and replace it with the child
  - 2 children: find in-order successor or predecessor R to the current node N, switch it with N then call delete on the respective child with N)

Traversals:



- In-Order
  - Traverse left node, current node, then right
  - everything in order
- Pre-Order
  - Traverse current node, left node, then right
  - while duplicating nodes and edges can make duplicate binary tree
- Post-Order
  - Traverse left node, right node, then current
  - while deleting and freeing can delete and free an entire tree

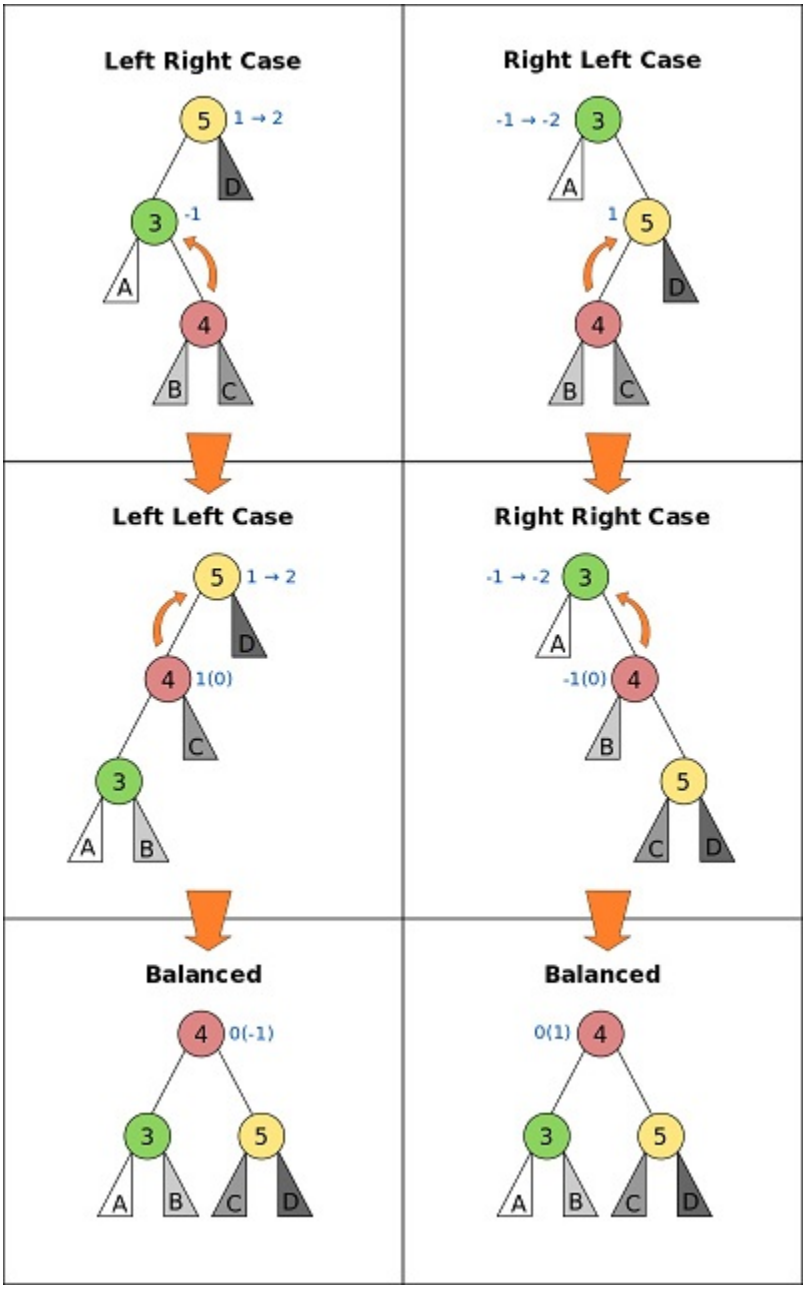
---

AVL:

- General
  - rebalance when insertion so that height never exceeds O(log n)

  - shape of tree changes during insertion and deletion
  - the height of an AVL tree is at most 1.44log(N)
  - AVL tree may need O(log(N)) operations to rebalance the tree

- searching-BST
- insertion - check balance factor for all ancestors if | balance factor | > 1 then rebalance
  - balance factor = height(left sub tree) - height(right sub tree)
  - rebalance starts from the inserted node up (from bottom up)
- deletion
  - Let node X be the node with the value we need to delete, and let node Y be a node in the tree we need to find to take node X's place, and let node Z be the actual node we take out of the tree
  - Steps to consider when deleting a node in an AVL tree are the following:
    * If node X is a leaf or has only one child, skip to step 5 with Z:=X.
    * Otherwise, determine node Y by finding the largest[citation needed] node in node X's left subtree (the in
    * order predecessor of X it does not have a right child) or the smallest in its right subtree (the in
    * order successor of X it does not have a left child).
    * Exchange all the child and parent links of node X with those of node Y. In this step, the in
    * order sequence between nodes X and Y is temporarily disturbed, but the tree structure doesn't change.
    * Choose node Z to be all the child and parent links of old node Y = those of new node X.
    * If node Z has a subtree (which then is a leaf), attach it to Z's parent.
    * If node Z was the root (its parent is null), update root.
    * Delete node Z.
    * Retrace the path back up the tree (starting with node Z's parent) to the root, adjusting the balance factors as needed.



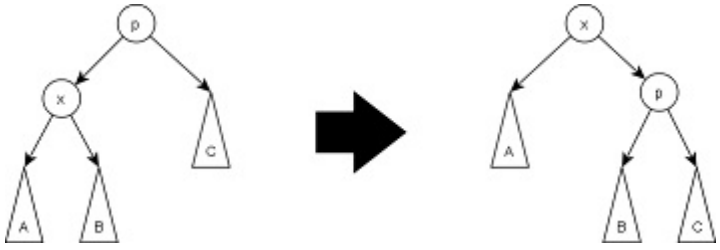
Red/Black:

- General
  - the maximum height of a red-black tree, $\sim$ 2log(N)
  - red-black tree needs O(1) operations to rebalance the tree
  - each node has an extra bit for color red or black
- In addition to the requirements imposed on a binary search tree the following must be satisfied by a redblack tree
  - A node is either red or black

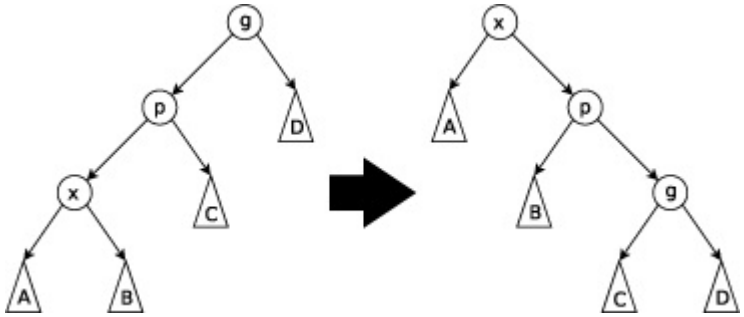
  - The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis
  - All leaves (NIL) are black
  - If a node is red, then both its children are black
  - Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes. The uniform number of black nodes in the paths from root to leaves is called the black height of the redblack tree

- searching-BST
- insertion O(log n)
  - Insert as BST
  - Fix any red-black vialations starting with inserted node continuing up the path
- deletion O(log n)
  - Delete as BST
  - Restore red-black properties

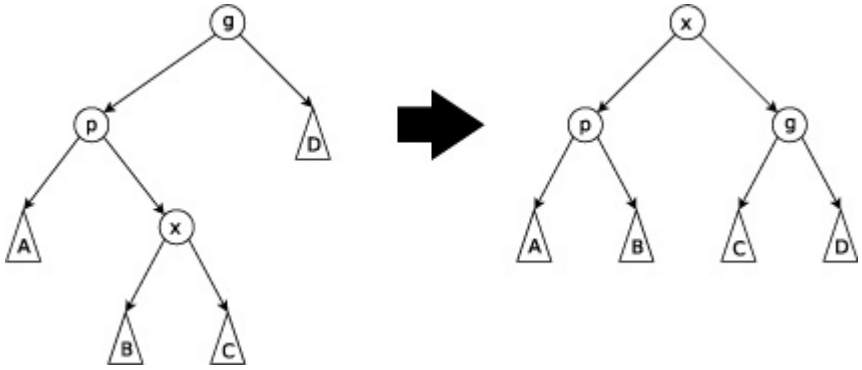
Splay:

- General
  - rebalance when look-up so frequently accessed elements move up
  - rebalances on look-up
  - shapes is not constrained and depends on look-ups
- Advantages
  - ave case is efficient as others
  - small memory
  - works with duplicate keys
- Disadvantages-height can be linear
- splaying - x=accessed node, p=parent of x, g parent of p
  - when p is the root tree is rotated on edge between p and x
  - 

  - p is not the root and x and p are either both right children or are both left children



- p is not the root and x is a right child and p is a left child or vice versa



Comparisons:

- AVL trees
  - AVL trees gurantee fast lookup (O(log n)) on each operation
  - AVL more useful in multithreaded environ with lots of look-ups because can be done in parallel (splay not in parallel)
  - Real time (and more) look-ups AVL is better
- Red Black
  - red black better with more inserts and deletes
- Splay trees
  - Splay trees gurantee any sequence of n operations take at most O(nlog n)
  - Splay faster on average on more look-ups
  - Splay better for splitting and merging efficiently
  - Splay more memory efficient (no need to store balance info)
  - Splay more effective when you only access a small subset
  - Splay rotation logic easier therefore easier to implement

Notes:

- Not all binary trees are BST
- Balancing
- Height: longest path from root to leaf
- Depth: The depth of a node is the number of edges from the node to the tree's root node

---

sources:

- https://en.wikipedia.org/wiki/AVL_tree
- https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
- https://www.topcoder.com/community/data-science/data-science-tutorials/an-introduction-to
- https://en.wikipedia.org/wiki/Splay_tree
- https://attractivechaos.wordpress.com/2008/10/02/comparison-of-binary-search-trees/

Treap

Big O:

- space O(n)
- time
  - search worst O(n), average O(log(n))
  - insert worst O(n), average O(log(n))
  - delete worst O(n), average O(log(n))

Advantages:

- Treap is same shape regardless of history
  - security: cant tell history
  - efficient sub tree sharing
  - useful for sets

Notes:

- heap invariant (children less or equal to parent)?
- formed by inserting nodes highest priority first into a BST without rebalancing
- each node has priority (heap) and key (BST)

Graphs

Notation:

G graph

V Vertices

E edges

Representation:

- Adjacency List
  - Storage O(|V| + |E|)
  - Add vertex O(1)
  - Remove vertex O(|V| + |E|)
  - Add edge O(1)
  - Remove edge O(|E|)
  - Query O(|V|)
- Adjacency Matrix
  - Storage O($| V |^2$)
  - Add vertex O($| V |^2$)
  - Remove vertex O($| V |^2$)
  - Add edge O(1)
  - Remove edge O(1)
  - Query O(1)

Searches:

- DFS
  - stack
  - DFS Numbering (entrance(d)/exit (f))
    - d value of a vertex u is lesser than the d value of all the descendants of u.
    - f value of a vertex u is higher than the f value of all the descendants of u.
  - visited set
  - cycles?
  - DFS tree?

        - forward edge (node to descendant)
        - back (node to ancestor)
        - cross (neither forward nor back)

- BSF
    - queue

Notes:

- undirected vs directed

General

Cycle Detection:

- Tortoise and hare
  - tortoise pointer moves 1 at each step
  - hare moves 2 at each step
  - if they ever meet their is a cycle
  - you can also use this algorithm (with more steps) to find start of cycle and the length of the cycle
- Applicable data structures
  - Linked List
  - Graph

General Common Issues:

- Space vs Indexing speed vs Sorting speed

sources:

- `https://en.wikipedia.org/wiki/Cycle_detection`

Classification

Classification by purpose:

- each algorithm has a goal
- kind of purposes
  - Sorting a list

Classification by implementation:

- Recursive or iterative
  - a recursive algorithm calls itself repeatedly until a certain condition matches
  - a iterative algorithm uses looping statements such as for loop, while loop or do-while loop
  - every recursive version has an iterative equivalent iterative, and vice versa
- Logical or procedural
  - an algorithm may be viewed as controlled logical deduction
  - a logic component expresses the axioms which may be used in the computation
  - a control component determines the way in which deduction is applied to the axioms
- Serial or parallel
  - in serial algorithms, computers execute one instruction of an algorithm at a time
  - parallel algorithms take advantage of computer architectures to process several instructions at once
- Deterministic or non-deterministic
  - deterministic algorithms solve the problem with a predefined process
  - non-deterministic algorithm must perform guesses of best solution at each step through the use of heuristics

Classification by design paradigm:

- Divide and conquer
- Contraction (Reduction/transform and conquer)

- Dynamic programming
- Greedy method
  - similar to dynamic programming but solutions to subproblems do not have to be known at each stage
  - a "greedy" choice can be made of what looks the best solution for the moment
  - Kruskal
- Linear programming
- Graphs
- The probabilistic and heuristic paradigm
  - Probabilistic
    - Those that make some choices randomly
  - Genetic
    - Attempt to find solutions to problems by mimicking biological evolutionary processes
    - a cycle of random mutations yielding successive generations of "solutions"
    - thus, they emulate reproduction and "survival of the fittest"
  - Heuristic
    - whose general purpose is not to find an optimal solution, but an approximate solution where the time or resources to find a perfect solution are not practical

---

Sources:

- https://www.quora.com/Which-are-the-10-algorithms-every-computer-science-student-must-imp

Sorts

Parameters:

- inplace
- stability-maintain relative order for "equal" keys

Accolades:

- Simplest-selection
- Efficient-MergeSort
- insertion best n average n^2 worst n^2
- selection best n^2 average n^2 worst n^2
- merge best $nlogn$ average $nlogn$ worst $nlogn$
- heap best $nlogn$ average $nlogn$ worst $nlogn$
- quick best $nlogn$ average $nlogn$ worst n^2

Classifications:

- Simple (insertion, selection)
- Efficient (Merge, Heap, quick)
- Bubble
- Distribution (bucket)
- Exchange sorts (bubble sort, quicksort)
- Selection sorts (shaker sort, heapsort)

insertion:

- take one element at a time insert into place in another list
- first element
- 2nd after or before first

- 3rd in between after or before

Selectionsort:

- look for smallest, place
- next smallest, place

Quickort:

- Pick an element, called a pivot, from the array.
- Reorder the array so that all elements with values less than the pivot come before the pivot, all elements with values greater than the pivot come after it (equal values can go either way)
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values

---

merge:

- Divide into two lists
- Recursively sort 2 lists
- merge two lists (add in least of both comparing them first)

---

bucketsort:

- Create buckets by range (0-9,10-19,20-29,etc)
- Separate into different buckets
- Sort buckets (using either bucketsort, quicksort, what-ever)

---

heapsort:

- Create heap of all elements
- Keep removing max (min) from heap to place into sorted array

bubblesort:

- iterate through array swapping consecutive elements if in the wrong order
- run until an interation with no swaps

sources:

- http://betterexplained.com/articles/sorting-algorithms/
- https://en.wikipedia.org/wiki/Sorting_algorithm

Recursive/Iterative

Definitions:

- Recursive
  - a function calls itself again and again till the base condition(stopping condition) is satisfied
  - common to functional programming
- Iterative
  - iterative is used to describe a situation in which a sequence of instructions can be executed multiple times
  - each time an iteration

Recursive Rules:

- each recursive call should be on a smaller instance of the same problem, that is, a smaller subproblem
- recursive calls must eventually reach a base case, which is solved without further recursion
  - Base case
  - Recursive Case
- Dynamic Programming and recursion
  - use when ever you compute a recursive input multiple times
  - memoization (caching previosly computed results, use result next time computation is needed)

Example problems:

- drawing fractals
- factorial
- towers of hanoi

Comparison:

- factorial
  - Iterative

```
def factorial(n):
    factorial = 1
    for i in range(2,n+1):
        factorial *= i
    return factorial
```

  - Recursive

```
def factorial(n):
    if (n < 2):
        return 1
    else:
        return n*factorial(n-1)
```

- Generally recursion more elegant, iteration better performance and debugability
- Iterative algorithm
  - more lines of code
  - faster
- Recursive algorithm
  - complex to implement
  - code will be elegant and easy to read
  - tracing is difficult
  - takes more time because of overheads like calling functions and registering stacks repeatedly
  - some complex problems can be solved easily and effectively in recursion

Divide and Conquer

Steps:

- divide: I into some number of smaller instances of the same problem p
- recurse: on each of the smaller instances to get the answer
- combine: the answers to produce an answer for the original instance I

Notes:

- Inductive proof
- Work and span by recurrences
- Naturally parallel

Contraction (Reduction, Transform and Conquer)

Steps:

- contract: "contract" (map) instance of problem p to smaller instance
- solve: solve smaller instance recursively
- expand: use the solution to solve original instance

Notes:

- Inductive proof
- Work by recursive (inductive) relation
- Efficient if reduce the problem size geometrically (constant factor < 1)

Dynamic Programming

Dynamic Programming:

Dynamic programming is a technique for solving problems recursively and is applicable when the computations of the subproblems overlap

DP Tools:

- Memoization (Top down)
  - an optimization technique where you cache previously computed results, and return the cached result when the same computation is needed again
  - storing the results of expensive function calls and returning the result when the same inputs occur again
- Tabulation (Bottom Up)
  - using iterative approach to solve the problem by solving the smaller sub- problems first and then using it during the execution of bigger problem
- Comparison
  - memoization usually requires more code and is less straightforward, but has computational advantages in some problems
    - mainly those which you do not need to compute all the values for the whole matrix to reach the answer
  - tabulation is more straightforward, but may compute unnecessary values
    - if you do need to compute all the values, this method is usually faster, though, because of the smaller overhead

Examples:

- Longest Common Subsequence problem
- Knapsack
- Travelling salesman problem

Sources:

- http://stackoverflow.com/questions/12042356/memoization-or-tabulation-approach-for-dynami

Greedy

Greedy:

- algorithm that makes the locally optimal choice at each stage
- a greedy algorithm never reconsiders its choices
  - choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem

Greedy components:

- a candidate set: from which a solution is created
- a selection function: chooses best candidate to be added to the solution
- a feasibility function: determines if a candidate can be used to contribute to a solution
- an objective function: assigns a value to a solution (or partial solution)
- a solution function: indicates when we discover a complete solution

Examples:

- Traveling Salesman
  - Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city
- min coins to give change
  - pick biggest coin possible, next biggest possible, etc.
- minimum spanning tree
  - Kruskal's
  - Prim's
- optimum Huffman trees

Sources:

- https://en.wikipedia.org/wiki/Greedy_algorithm

Shortest Path

Dijkstras:

Implementation with PQ

- O(|E| + |V| log |V|)
- Make source current node with its distance 0
- Repeat until no elements in PQ
  - If current node is not in distance map or has a greater value than computed value place the current node in the distance map (mapped to distance)
  - Place its neighbors in the PQ with their distances to current node + current node distance
  - Dequeue min element from PQ and make current node

Bellman Ford:

psuedo java code

```
for i=1 to size(vertices)-1
   for each edge (u,v)
      if distance[u]+w < distance[v]
         distance[v]=distance[u]+w
         predecessor[v]=u
```

c++ code

```
function BellmanFord(list vertices, list edges, vertex source)
   ::distance[],predecessor[]

   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (distance and predecessor) with shortest-path
   // (less cost/distance/metric) information

   // Step 1: initialize graph
   for each vertex v in vertices:
       if v is source then distance[v] := 0
       else distance[v] := inf
       predecessor[v] := null

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) in Graph with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
               predecessor[v] := u
```

```
    // Step 3: check for negative-weight cycles
    for each edge (u, v) in Graph with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

Notes

- O(|V| |E|)
- Allows negative weights

Minimum Spanning Tree

Boruvkas:

Algorithm for minimum spanning tree (smallest weight subgraph with all vertices) of a graph O(Elog V) where E=edges, v=vertices in the graph

- Find min edge for all vertices
- Connect those edges
- Loop until all connected
  - Find min edge out of all trees (connected vertices)
  - Connect those edges

O(Elog V) where E=edges, v=vertices in the graph

Notes:

- Prims
- Kruskal

Graph Contraction

Types:

- edge: two vertices connected by an edge are contracted
- star: one vertex center of stars and all vertices directly connected are contracted
- tree: disjoint tree identified and contraction performed on trees

Notes:

- Can be used to find min span tree

Back Propagation

Definitions:

Backpropagation is the central mechanism by which neural networks learn. It is the messenger telling the network whether or not the net made a mistake when it made a prediction.

Sources:

- http://neuralnetworksanddeeplearning.com/chap2.html
- https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-w
- https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-w
- https://pathmind.com/wiki/backpropagation

Convolutional (CNN)

Definitions:

Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms.

Sources:

- https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the
- http://deeplearning.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

Dynamic Programming

Dynamic Programming:

Dynamic programming is a technique for solving problems recursively and is applicable when the computations of the subproblems overlap

DP Tools:

- Memoization (Top down)
  - an optimization technique where you cache previously computed results, and return the cached result when the same computation is needed again
  - storing the results of expensive function calls and returning the result when the same inputs occur again
- Tabulation (Bottom Up)
  - using iterative approach to solve the problem by solving the smaller sub- problems first and then using it during the execution of bigger problem
- Comparison
  - memoization usually requires more code and is less straightforward, but has computational advantages in some problems
    - mainly those which you do not need to compute all the values for the whole matrix to reach the answer
  - tabulation is more straightforward, but may compute unnecessary values
    - if you do need to compute all the values, this method is usually faster, though, because of the smaller overhead

Examples:

- Longest Common Subsequence problem
- Knapsack
- Travelling salesman problem

Sources:

- http://stackoverflow.com/questions/12042356/memoization-or-tabulation-approach-for-dynami

Probability

Probability numbers between 0 and 1, 0 is impossible, 1 is certain

---

Interpretations:

- Frequentist-proportion of heads if we toss a coin many times
- Propensity-tendency of a coin to land heads
- Subjectivist-how strongly we believe that a coin will land heads

---

Notes:

- Probability vs statistics (prob-likelihood of certain events vs stat-observe results and determine probabilities from which they might have originated)
- Random isnt really randon just chaotic (underlying principles very complicated and tiny changes affect result, just really hard to predict)
- true randomness does exists-radioactive decay
- Quantum mechanics is the only known effect in nature that produces true randomness
- As we roll dice more and more often, the observed frequencies become closer and closer to the frequencies we predict using probability theory. This principle always applies in probability and is called the Law of large numbers.
- As we increase the number of dice rolled at once, we also see that the shape of the probability distribution changes from a triangular shape to a bell-shaped curve. This is known as the Central Limit Theorem.

---

sources:

- `https://en.wikipedia.org/wiki/Probability`

Combinatorics

Factorial:

- factorial (5*4*3*2*1=5!)
- 0!=1

r-s Principle:

- 5 pairs of pants to go with 2 shirts 5*2 options
- ordered pair-pair of 'things' arranged in a certain order

Permutations and combinations:

- Permutations (care about order)
  - place n objects in k positions
  - $$ {}^nP_k = \frac{n!}{(n-k)!} $$
- combinations (dont care about order)
  - divide by n! because compared to permutation n places(order) to place first choice, n-1 to place second...
  - $$ {}^nC_k = \frac{n!}{k!(n-k)!} = \binom{n}{k} $$

Bonomial Identity:

$$ (x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} $$

$$ \binom{n}{0} = \binom{n}{n} = 1 \quad \text{for all integers} $$

sources:

- `http://world.mathigon.org/Combinatorics`

Set Theory

Definitions:

- an object either belongs or does not belong
- set-collection of things that are brought together because they obey a certain (well defined) rule
  - ex: numbers, people, shapes
- element- "thing" that belongs to a given set

Symbols:

- {...}-the set of ...
  - ex:{-3,-2,-1,0,1,2}, {integers between -3 and 3 inclusive}, {x|x is an integer amd |x| <4}
- ($\in$) symbol means element of
- set usually uppercase A,B and elements lowercase x,y
- U-universal set, all things under discussion
- {},($\varnothing$)-empty set, null set
- N-natural numbers, whole numbers starting at 1
- Z-integers
- R-real numbers

Set Operations:

- (A $\cap$ B)intersection (two sets overlap)
- (A $\cup$ B) union (elements in either)
- (A - B) or (A B) difference (elements in A but not B)
- (A') or(A^C) or complement (everything not in A is in A')
- cardinality (if A={lowercase letters of the alphabet}, | A| =26)
- P(A) powerset-set of all subsets (including empty) of A

  - if | A |=k then | P(A)| =2^k proof (for each element we can choose to include element or not)

- Cartesian Products
  - if we have n sets: A1 , A2, ..., An, then their Cartesian product is defined by: $A1A2...An$ = { (a1, a2, ..., an) | a1 ∈ A1, a2 ∈ A2, ..., an ∈ An) } and (a1, a2, ..., an) is called an ordered n-tuple.

Relationships:

- Equality = (same elements, repeats ignored)
- subsets (A⊆B) all elements of A are also elements of B
  - A ⊆ B and B ⊆ A, then A = B
  - proper subset (A ⊂ B) if B contains at least one element that isn't in A
- disjoint no elements in common

Foundational Rules of Set Theory:

- The Laws of Sets
  - Commutative Laws
    - ∩ B = B ∩ A
    - A ∪ B = B ∪ A
  - Associative Laws
    - (A ∩ B) ∩ C = A ∩ (B ∩ C)
    - (A ∪ B) ∪ C = A ∪ (B ∪ C)
  - Distributive Laws
    - A ∩ (B ∪ C) = (A ∩ B) ∪ (A ∩ C)
    - A ∪ (B ∩ C) = (A ∪ B) ∩ (A ∪ C)
  - Idempotent Laws
    - A ∩ A = A
    - A ∪ A = A
  - Identity Laws
    - A ∪ ∅ = A
    - A ∩ U = A
    - A ∪ U = U
    - A ∩ ∅ = ∅
  - Involution Law

    - $(A')' = A$
  - Complement Laws
    - $A \cup A' = U$
    - $A \cap A' = \varnothing$
    - $U' = \varnothing$
    - $\varnothing' = U$
  - De Morgans Laws
    - $(A \cap B)' = A' \cup B'$
    - $(A \cup B)' = A' \cap B'$
    - proof
      - $(A \cup B)' \subseteq A' \cap B'$
      - $A' \cap B' \subseteq (A \cup B)'$

---

sources:

- https://en.wikibooks.org/wiki/Discrete_Mathematics/Set_theory

General Tips

Functional:

- Invariants that can be used to prove function

Administrative:

- small informative variables
- Comments in function definition, if, while, for, etc.
- pre/post conditions, loop invariant
- input/output types
- test null and all possible inputs (and outputs)
- Initial a variables you get from else where as null then get them (just in case there's an issue getting them) and test if null

Proving code:

- Preconditions, postconditions, assertions, loop invariants
- connect algorithmic ideas to imperative programs
- proofs + contracts = correctness and safety of code

Software Design

Software design and complexity:

- scale
- environment (I/O, Network)
- infrastructure (libraries, frameworks)
- evolution (design for change)
- correctness (testing, analysis tools, automation)

software qualities:

- sufficiency/func correctness
- robustness
- flexibility
- reusability
- efficiency
- scalability
- security

A simple process:

- Discuss the software that needs to be written
- Write some code
- Test the code to identify the defects
- Debug to find causes of defects
- Fix the defects
- If not done, return to first step

design tips:

- Think before coding
- Consider quality attributes (maintainability,extensibility, performance)
- Consider alternatives and make conscious design decisions

Preview: The design process:

- ObjectOriented Analysis
  - Understand the problem
  - Identify the key concepts and their relationships
  - Build a (visual) vocabulary
  - Create a domain model (aka conceptual model)
- ObjectOriented Design
  - Identify software classes and their relationships with class diagrams
  - Assign responsibilities (attributes, methods)
  - Explore behavior with interaction diagrams
  - Explore design alternatives
  - Create an object model (aka design model and design class diagram) and interaction models
- Implementation
  - Map designs to code, implementing classes and methods

Objects:

- A package of state (data) and behavior (actions)
- Can interact with objects by sending messages
  - perform an action (e.g., move)
  - request some information (e.g., getSize)
- Possible messages described through an interface

---

sources:

- `http://www.cs.cmu.edu/~charlie/courses/15-214/2015-fall/index.html#schedule`

Design for Change

Design principle for change: information hiding:

- expose little implementation as possible
- allows you to change hidden details later

Subtype Polymorphism:

- There may be multiple implementations of an interface
- Multiple implementations coexist in the same program
- May not even be distinguishable
- Every object has its own data and behavior

Interface:

- can implement start (defining required methods and classes)
- create class to implement interface

What to test:

- Functional correctness of a method (e.g.,computations, contracts)
- Functional correctness of a class (e.g., class invariants)
- Behavior of a class in a subsystem/multiple subsystems/the entire system
- Behavior when interacting with the world
  - Interacting with files, networks, sensors,
  - Erroneous states
  - Nondeterminism, Parallelism
  - Interaction with users
- Other qualities (performance, robustness, usability, security,)

Unit Tests (Junit good for JAVA):

- Unit tests for small units: functions, classes, subsystems
  - Smallest testable part of a system
  - Test parts before assembling them
  - Intended to catch local bugs
- Typically written by developers
- Many small, fastrunning, independent tests
- Little dependencies on other system parts or environment
- Insufficient but a good starting point
- extra benefits:
  - Documentation (executable specification)
  - Design mechanism (design for testability)

Test cases strategies:

- use specs
- representative cases
- invalid cases
- boundary cond
- think like attacker
- dificult cases

static methods:

- Static methods belong to a class
- global
- Direct dispatch, no subtype polymorphism
- Avoid unless really only a single implementation exists (e.g., Math.min)

Best practices:

- control access
  - fields not accessible from client code
  - methods only accessible in exposed interface
- contracts - agreement between provider and user
  - interface specification
  - functionality and correctness expectations
  - Performance expectations
- Visibility Modifiers
  - design principle for change: information hiding
  - expose little implementation as possible
  - allows you to change hidden details later

Notes:

- try to avoid setters
- Organize program functionality around kinds of abstract objects
  - For each object kind, offer a specific set of operations on the objects
  - Objects are otherwise opaque: Details of representation are hidden
  - Messages to the receiving object
- Distinguish interface from class
  - Interface: expectations
  - Class: delivery on expectations (the implementation)
  - Anonymous class: special Java construct to create objects without explicit classes: Point x = new Point() /* implementation */ ;
- Explicitly represent the taxonomy of object types
  - This is the type hierarchy (!= inheritance, more on that later): A CartesianPoint is a Point
- Design Patterns!!
  - Design Patterns by Gamma,Helm,Johnson,Vlissides

sources:

- http://www.cs.cmu.edu/~charlie/courses/15-214/2015-fall/index.html#schedule

Design for Reuse

Delegation:

- Delegation is simply when one object relies on another object for some subset of its functonality
  - Sorter is delegating functonality to some Comparator implementation
- Judicious delegation enables code reuse
  - Sorter can be reused with arbitrary sort orders
  - Comparators can be reused with arbitrary client code that needs to compare integers

Delegation and design:

- Small interfaces
- Classes to encapsulate algorithms
  - ex: the Comparator, the Strategy pattern

Inheritance:

- Typical roles:
  - An interface defines expectations/commitment for clients
  - An abstract class is a convenient hybrid between an interface and a full implementation
  - A subclass overrides a method definition to specialize its implementation

Benefits of Inheritance:

- Reuse of code
- Modeling flexibility
- A Java aside:
  - Each class can directly extend only one parent class
  - A class can implement multiple interfaces

Power of object oriented interfaces:

- Subtype polymorphism
  - Different kinds of objects can be treated uniformly by client code
- e.g., a list of all accounts
  - Each object behaves according to its type
- If you add new kind of account, client code does not change

Inheritance and subtyping:

- Inheritance is for code reuse
  - Write code once and only once
  - Superclass features implicitly available in subclass
- Subtyping is for polymorphism
  - Accessing objects the same way, but getting different behavior
  - Subtype is substitutable for supertype

Java details: final:

- A final field: prevents reassignment to the field after initialization
- A final method: prevents overriding the method
- A final class: prevents extending the class
  - e.g., public final class CheckingAccountImpl

Type Casting:

- Sometimes you wan a different type than what you have
  - ex: float pi=3.14; int indianapi=(int) pi;
- Useful if you have more specific subtype
- Advice: avoid downcasting types

  - Never downcast within superclass to a subclass

Behavioral Subtyping:

- Compiler enforced rules in Java:
  - Subtype (subclass, subinterface, object implementing interface) can add but not remove methods
  - Overriding method must return same type or subtype
  - Overriding method must accept same parameter types
  - Overriding method may not throw additional exceptions
  - Concrete class must implement all undefined interface methods and abstract methods
- A subclass must fulfill all contracts its superclass does:
  - same or strong invariants
  - same or stronger post
  - conds for all methods
  - same or weaker pre
  - conds for all methods

Parametric polymorphism via java generics:

- Parametric polumorphism is the ability to define a type generically to allow static type
- checking with out fully specifying types
- The java.util.Stack instead
  - A stack of some type T:

```
public class Stack<T> {
   public void push(T obj){...}
   public T pop() { ...}
}
```

- Improves typechecking, simplifies client code

Notes:

- Template method design pattern
- Decorator design pattern

Design for Robustness

Exceptions:

- Notify caller of exceptional circumstance (usually operational failure)
- Semantics
  - An exception propagates up the function
  - call stack until main() is reached (terminates program) or until the exception is caught
- Sources of exceptions:
  - Program throwing an exception
  - Exception thrown by the Java Virtual Machine

Java: Finally:

The finally block always runs after try/catch

Design choice: checked and unchecked:

- Unchecked exception: any subclass of RuntimeException
  - Error which is highly unlikely and/or unrecoverable
- check exception: any subclass of Exception that is not a subclass of RuntimeException
  - Error that every caller should be aware of and explicitly decide to handle or pass on
- Return values(- 1, false, null): If failure is common and expected possibility

Creating and throwing your own exceptions:

- Methods must declare any checked exceptions they might throw
- If your class extends java.lang.Throwable you can throw it:
  -
```
if (some__){
  throw new customException("Blah blah");
 }
```

Benefits of exceptions:

- High level summary of error and stack trace
  - Compare: core dumped in C
- Can't forget to handle common failure modes
  - Compare: using a flag or special return value
- Can optionally recover from failure
  - Compare: calling System.exit()
- Improve code structure
  -
  - Separate routine operations from error
  - handling
- Allow consisten clean
- up in both normal and exceptional operation

Guide for exceptions:

- Catch and handle all checked exceptions
  - Unless there is no good way
- Use runtime exceptions for programming error
- Other
  - Don't catch an exception without (at least somewhat) handling the error
  - When you throw an exception describe the error
  - if you re
  - throw an exception, always include the original exception as the clause

Modular Protection:

- Errors and bugs unavoidable but exceptions should not leak across modules (methods, classes) if possible
- Good modules handles exceptional conditions locally

  - Local input validation and local exception handling where possible
  - Explicit interfaces with clear pre/post conditions
- Explicitly documented and checked exceptions where exceptional conditions may propagate between modules
  - Information hiding -encapsulation of critical code (likely bugs, likely exceptions)

Problems when testing (sub -)systems:

- User interfaces and user interactions
  - Users click buttons, interpret output
  - Waiting/timing issues
- Test data vs. real data
- Testing against big infrastructure (database, web services,..)
- Testing with side effects (e.g. printing and mailing documents)
- Nondeterministic behavior
- Concurrency

Testing strategies in environments:

- Separate business logic and data representation from GUI for testing
- Test algorithms locally without large environment using stubs
- Advantage of stubs
  - Create deterministic response
  - Can reliably simulate spurious states (network error)
  - Can speed up test execution
  - Can simulate functionality not yet implemented
- Automate

Scaffolding:

- Catch bugs early: Before client code or service available
- Limit scope of debugging: Localize errors

- Improve coverage
  - System level tests may only cover 70
  - Simulate unusual error conditions
- Validate internal interface/API designs
  - Simulate clients in advance of their developement
  - Simulate services in advance of their development
- Capture developer intent (in absence of specification documentation)
  - A test suite formally captures elements of design intent
  - Developer documentation
- Improve low
- level design
  - Early attention to ability to test

General

Criticism:

- target wrong problem
- lacks formal foundations
- leads to inefficient solutions
- does not differ significantly from other abstractions

Sources:

- https://sourcemaking.com/design_patterns
- https://en.wikibooks.org/wiki/Computer_Science_Design_Patterns
- https://en.wikipedia.org/wiki/Software_design_pattern

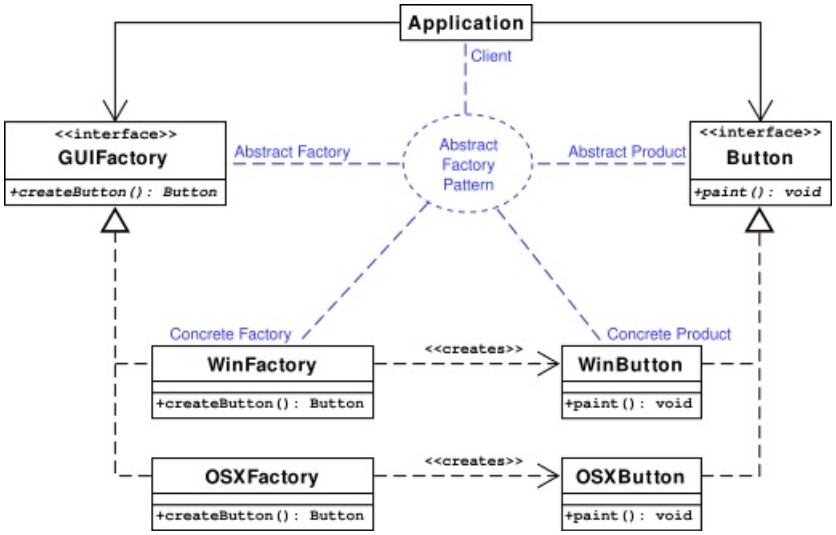
Creation

General:

- deal with object creation mechanisms, trying to create objects in a manner suitable to the situation
- These design patterns are all about class instantiation
- This pattern can be further divided into class
- creation patterns and object
- creational patterns
- While class
- creation patterns use inheritance effectively in the instantiation process, object
- creation patterns use delegation effectively to get the job done
- The creational patterns aim to separate a system from how its objects are created, composed, and represented
- They increase the system's flexibility in terms of the what, who, how, and when of object creation

Abstract Factory:

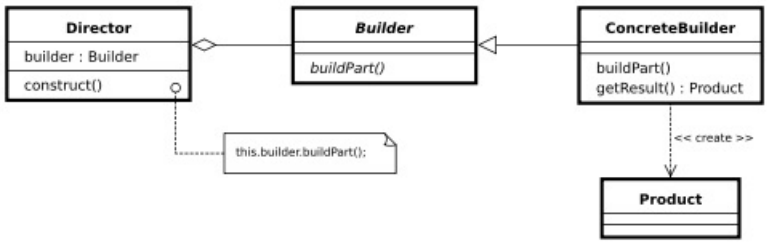
- Definition/Use
  - provides an interface for creating related or dependent objects without specifying the objects' concrete classes
  - provide an interface for creating families of related or dependent objects without specifying their concrete classes
  - encapsulates "new" ex: new product()
  - determines concrete type but returns abstract pointer
    - client code has no knowledge and isn't burdened by concrete type
    - adding new concrete types done by modifying client code to use different factory (1 line)
  - can determine concrete type from config file for example
- Structure



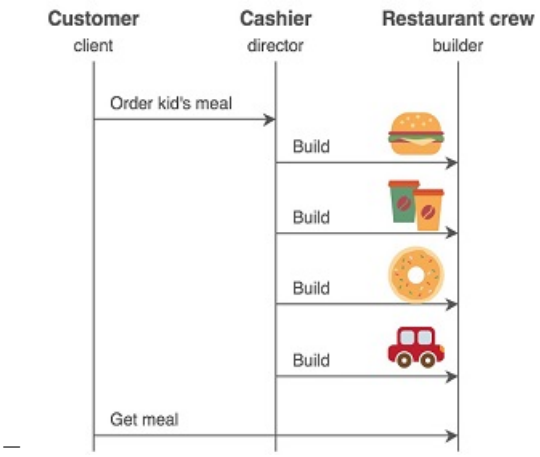
–

Builder:

- Definition/Use
  - separates the construction of a complex object from its representation so that the same construction process can create different representations
  - builder pattern is useful to avoid a huge list of constructors for a class
  - an application needs to create the elements of a complex aggregate
  - use builder to store parameters and then use that builder in constructor
  - separate the construction of a complex object from its representation so same construction can create different representations
- Structure



–

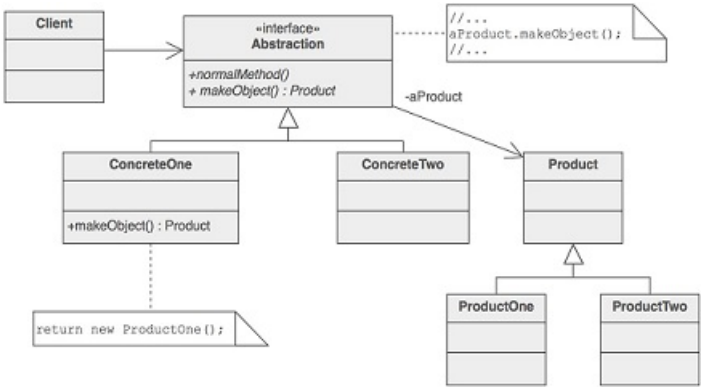


- Example
  - StringBuffer and StringBuilder
- Notes
  - put the builder term in the name of the builder class to indicate the use of the pattern to the other developers
  - if the target class contains flat data, your builder class can be constructed as a Composite that implements the Interpreter pattern

---

Factory Method:

- Definition/Use
  - allows a class to defer instantiation to subclasses
  - new operator considered harmful
  - define an interface for creating an object, but let subclasses decide which class to instantiate
  - provide a way for users to retrieve an instance with a known compile-time type, but whose runtime type may actually be different
  - an increasingly popular definition of factory method is: a static method of a class that returns an object of that class' type
    - unlike a constructor, the actual object it returns might be an instance of a subclass
- Structure



–

- Example
  - a factory method that is supposed to return an instance of the class Foo may return an instance of the class Foo, or an instance of the class Bar, so long as Bar inherits from Foo
  –

```
Color.make_RGB_color(float red, float green, float blue)
Color.make_HSB_color(float hue, float saturation, float brightness)
```

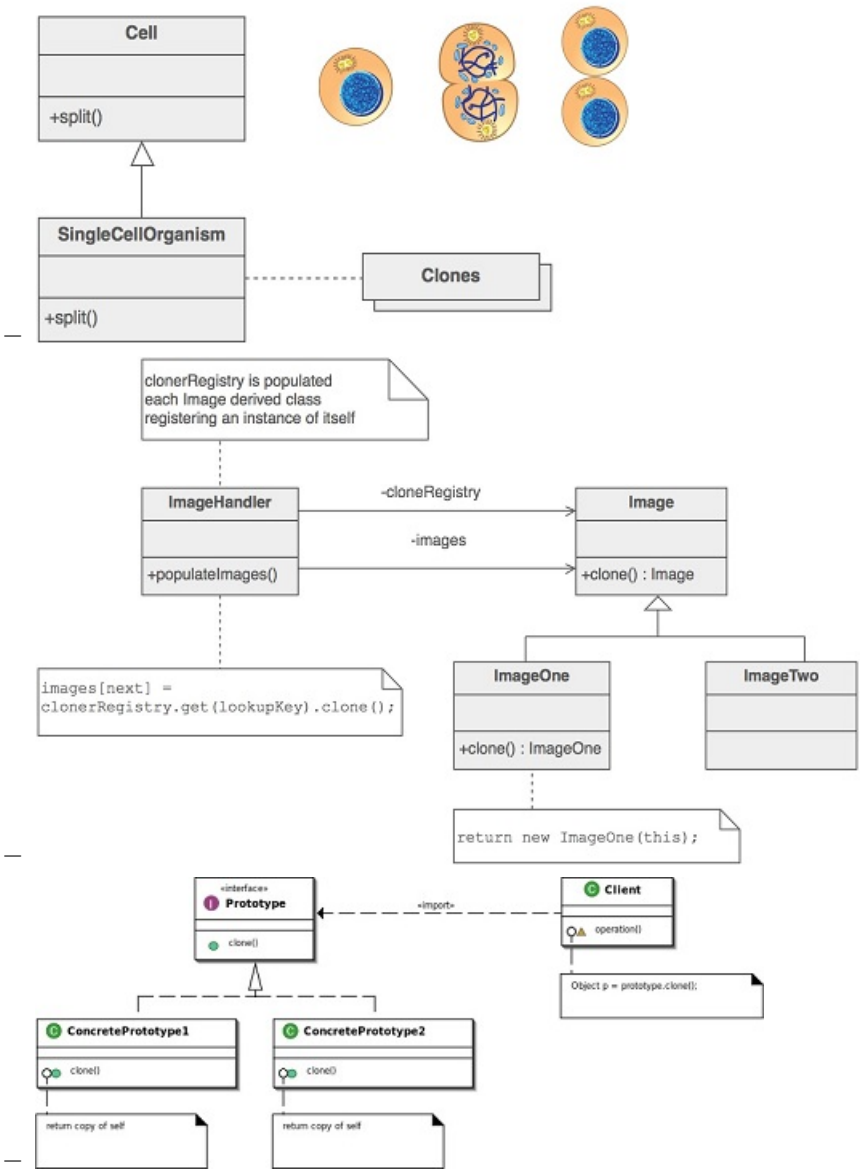
  –

```
Letter.getLetter(char) if vowel return Vowel(char) else Consonant(char)
    (vowel, consonant extend letter)
```

- Notes
  - consider making all constructors private or protected

Prototype:

- Definition/Use
  - specifies the kind of object to create using a prototypical instance, and creates new objects by cloning this prototype
  - when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects
  - application ”hard wires” the class of object to create in each ”new” expression
- Structure



- Notes
  - add clone() method
  - add registry
  - put the prototype term in the name of the prototype classes to indicate the use of the pattern to the other developers

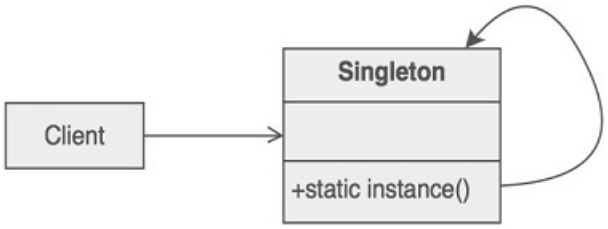
---

Singleton:

- Definition/Use
  - ensures that a class only has one instance, and provides a global point of access to it

  - ensure a class has only one instance, and provide a global point of access to it

- Structure

  - 

- Notes

  - make instantiation(private ThisSingleton()) private aka define all constructors to be protected or private
  - name the method getInstance() to indicate the use of the pattern to the other developers
  - define a private static attribute in the "single instance" class

---

Comparison:

- abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype
- factory method: creation through inheritance, prototype: creation through delegation

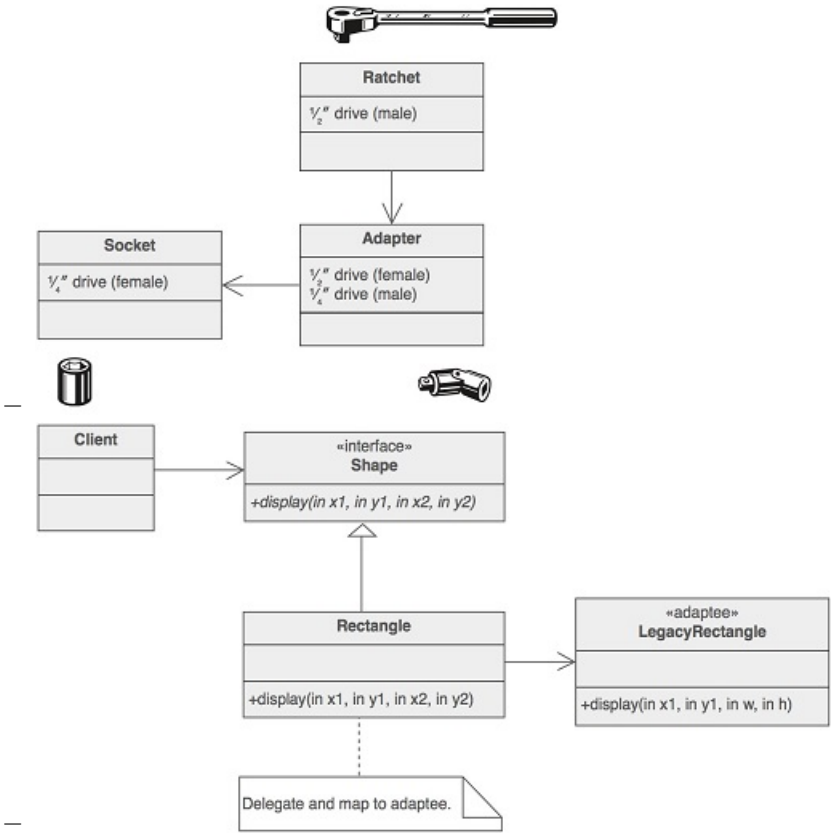
Structural

General:

- ease the design by identifying a simple way to realize relationships between entities
- about class and object composition
- use inheritance to compose interfaces
- define ways to compose objects to obtain new functionality

Adapter:

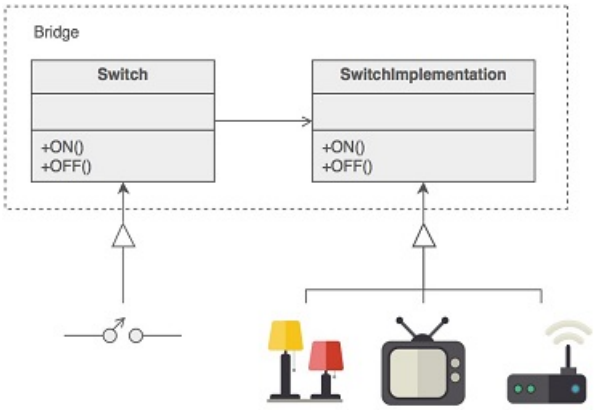
- Definition/Use
  - 'adapts' one interface for a class into one that a client expects
  - used when a client class has to call an incompatible provider class
  - an "off the shelf" component offers compelling functionality but its "view of the world" is not compatible
  - wrap an existing class with a new interface
- Structure



- Notes
  - put the adapter term in the name of the adapter class to indicate the use of the pattern to the other developers

---

Bridge:

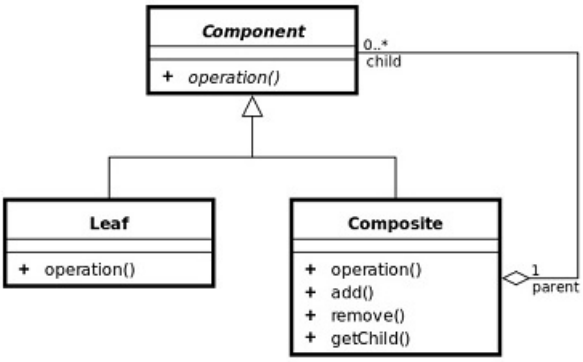
- Definition/Use
  - decouple an abstraction from its implementation so that the two can vary independently
  - useful when a code often changes for an implementation as well as for a use of code
  - decouple an abstraction from its implementation so that the two can vary independently
- Structure
  - 

- Notes
  - design the separation of concerns: what does the client want, and what do the platforms provide

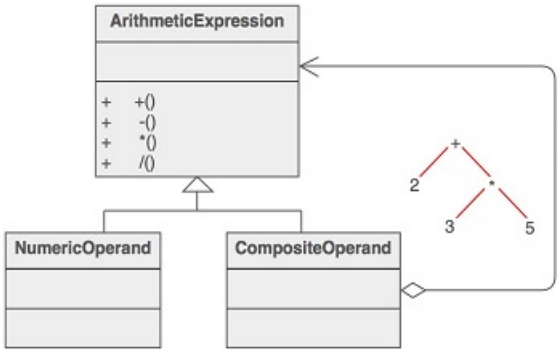
---

Composite:

- Definition/Use
  - a tree structure of objects where every object has the same interface
  - application needs to manipulate a hierarchical collection of "primitive"(leaf) and "composite" objects
- Structure



–

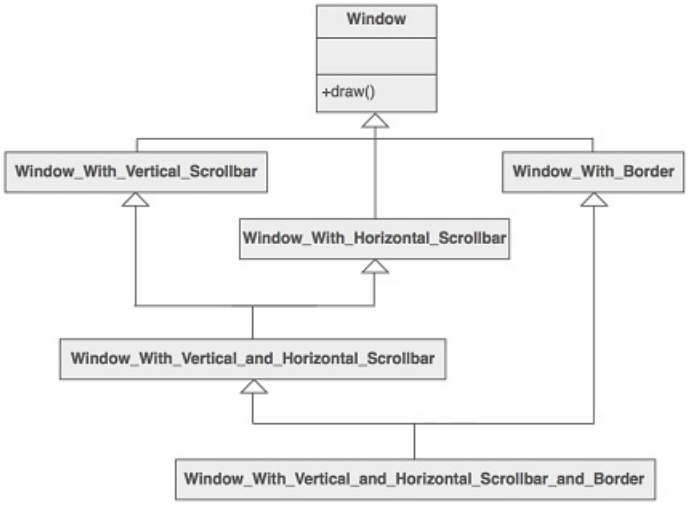


–

- Examples
  - GUI, widgets organized in a tree and operations (resize, repainting) on all widgets processed using pattern
- Notes
  - consider the heuristic, "containers that contain containees, each of which could be a container"

Decorator (Wrapper):

- Definition/Use
  - add additional functionality to a class at runtime where subclassing would result in an exponential rise of new classes
  - client-specified embellishment of a core object by recursively wrapping it
- Structure



–

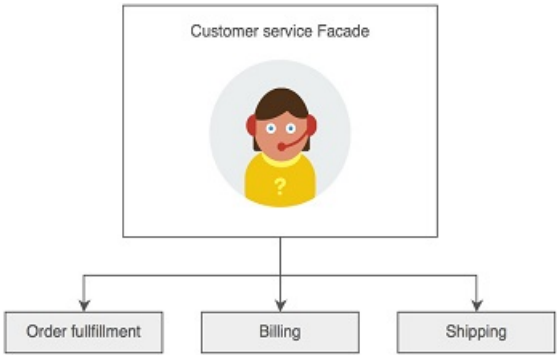
*
```
Widget* aWidget = new BorderDecorator(
  new HorizontalScrollBarDecorator(
    new VerticalScrollBarDecorator(
      new Window( 80, 24 ))));
aWidget->draw();
```

- Notes
  - ensure the context is: a single core (or non-optional) component, several optional embellishments or wrappers, and an interface that is common to all

---

Farcade:

- Definition/Use
  - create a simplified interface of an existing interface to ease usage for common tasks
  - hides the complexities of the system and provides an interface to the client from where the client can access the system
- Structure



–

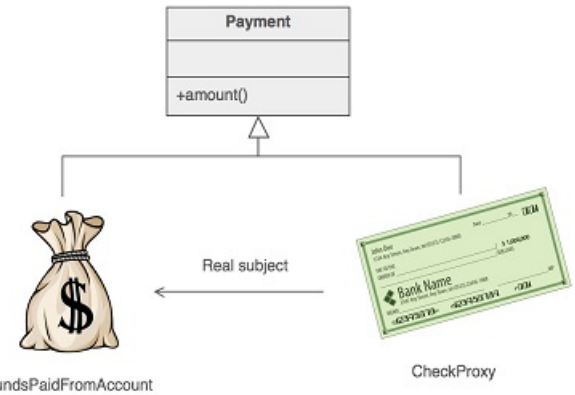
- Notes
  - often singletons because only one facade object is required
  - client uses (is coupled to) the facade only

Flyweight:

- Definition/Use
  - a large quantity of objects share a common properties object to save space
  - each "flyweight" object is divided into two pieces
    - the state-dependent (extrinsic) part: stored or computed by client objects, and passed to the Flyweight when its operations are invoked
    - the state-independent (intrinsic) part: stored (shared) in the Flyweight object
- Example
  - in video games, it is usual that you have to display the same sprite (i.e. an image of an item of the game) several times
    - it would highly use the CPU and the memory if each sprite was a different object
    - so the sprite is created once and then is rendered at different locations in the screen
    - this problem can be solved using the flyweight pattern
    - the object that renders the sprite is a flyweight

Proxy:

- Definition/Use
  - a class functioning as an interface to another thing
  - provide a surrogate or placeholder for another object to control access to it
- Structure



- Example
  - ProxyImage and RealImage

Comparison:

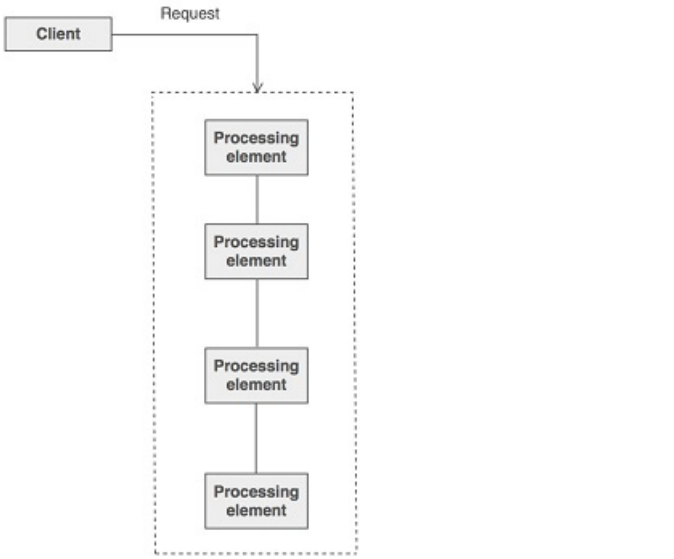
- adapter makes things work after they're designed, bridge makes them work before they are
- composite and decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects
- decorator and proxy have different purposes but similar structures

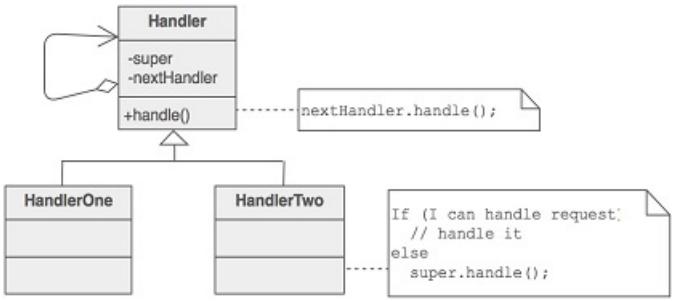
Behavioral

General:

Identify common communication patterns between objects and realize these patterns

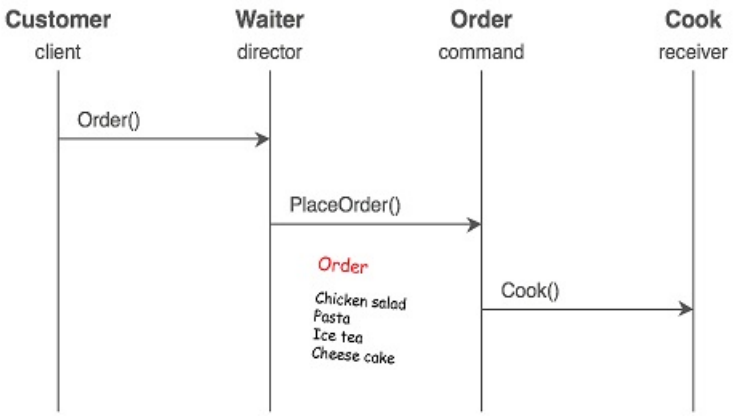
Chain of responsibility:

- Definition/Use
  - command objects are handled or passed on to other objects by logic-containing processing objects
  - avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
- Structure
  - 

  - 

- Notes
  - base class maintains a next pointer
  - if the request needs to be "passed on", then the derived class "calls back" to the base class, which delegates to the "next" pointer

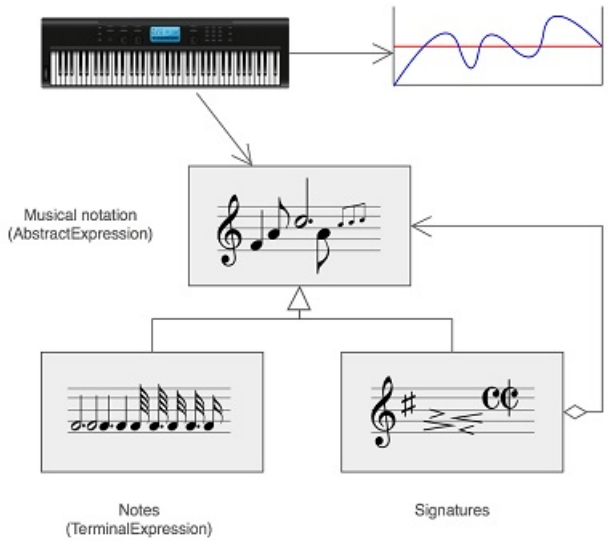
Command:

- Definition/Use
  - command objects encapsulate an action and its parameters
  - Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request
  - separation provides flexibility in the timing and sequencing of commands
  - command objects can be thought of as "tokens",created by one client that knows what need to be done, passed to another client that has the resources for doing it
- Structure
  - 

- Notes
  - define a Command interface with a method signature like execute()

Interpreter:

- Definition/Use
  - implement a specialized computer language to rapidly solve a specific set of problems
  - map a domain to a language, the language to a grammar, and the grammar to a hierarchical object
  - oriented design
- Structure



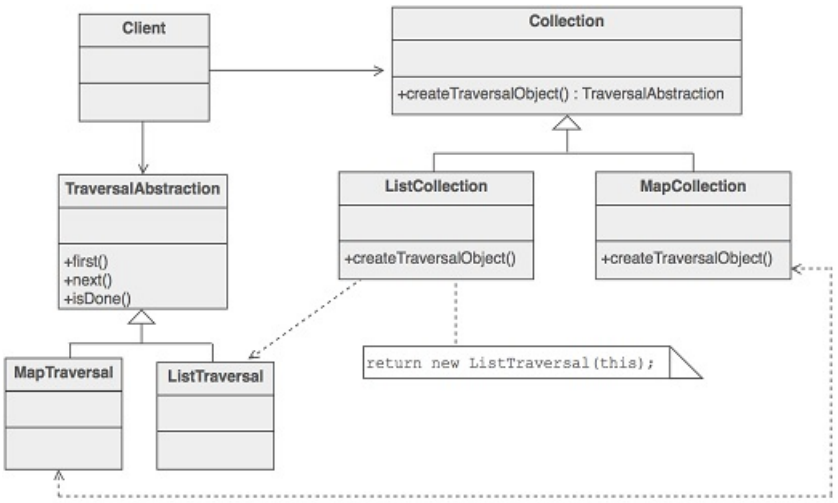
_

- Notes
  - the pattern doesn't address parsing. When the grammar is very complex, other techniques (such as a parser) are more appropriate

---

Iterator:

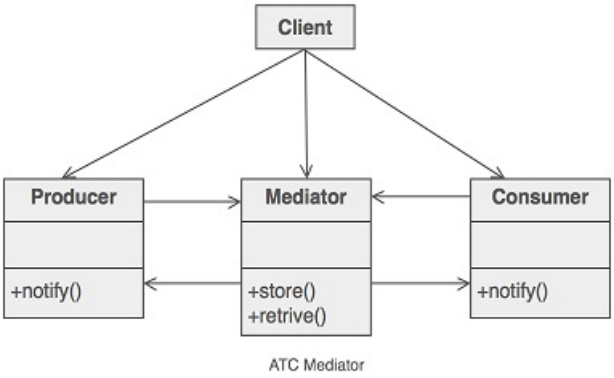
- Definition/Use
  - iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation
  - need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently
- Structure

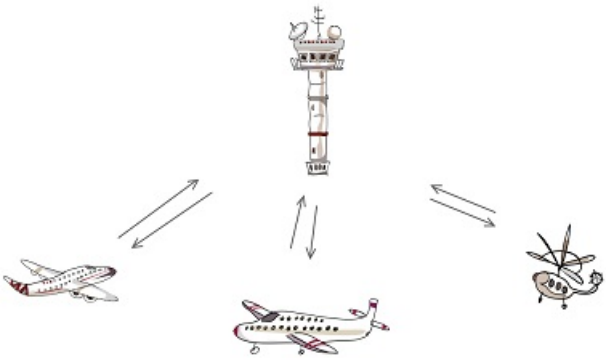


_

- Notes
  - ```
    clients use the first(), is_done(), next(), and current_item() protocol to
        access the elements of the collection class
    ```

Mediator:

- Definition/Use
  - provides a unified interface to a set of interfaces in a subsystem
  - promotes loose coupling by keeping objects from referring to each other explicitly
- Structure
  - 

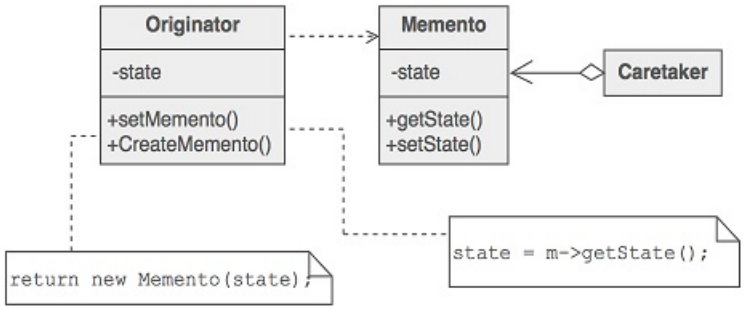
ATC Mediator
  - 
- Notes
  - be careful not to create a "controller" or "god" object

Memento:

- Definition/Use
    ```

  - provides the ability to restore an object to its previous state (rollback)
  - pattern defines three distinct roles
    * originator : the object that knows how to save itself
    * caretaker : the object that knows why and when the originator needs to save and restore itself
    * memento : the lock box that is written and read by the originator, and shepherded by the caretaker

- Structure



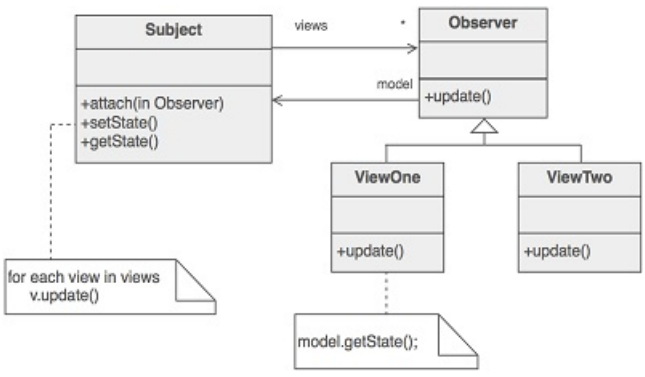
  –

- Notes
  - identify the roles of caretaker and originator

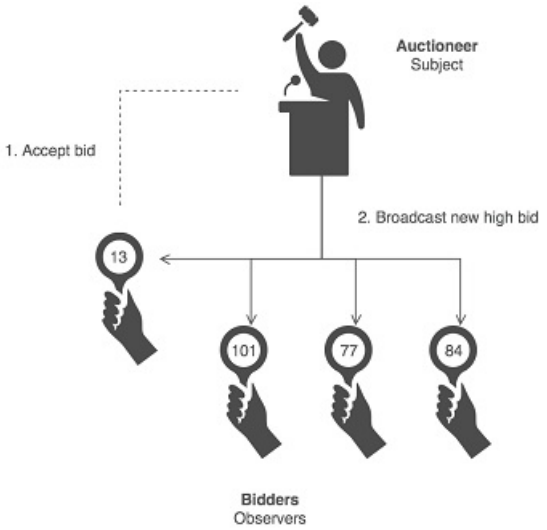
Observer(Publish/Subscribe or Event Listener):

- Definition/Use
  - objects register to observe an event that may be raised by another object
  - defines a one
  - to
  - many relationship so that when one object changes state, the others are notified and updated automatically

- Structure



  –

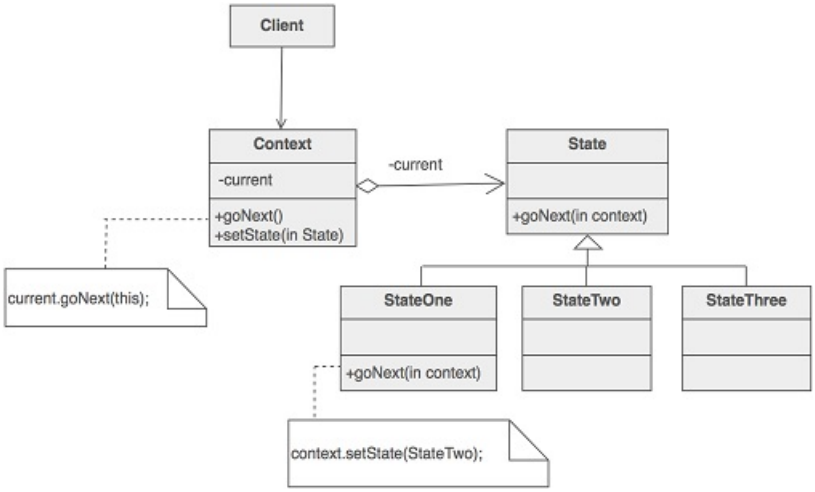


–

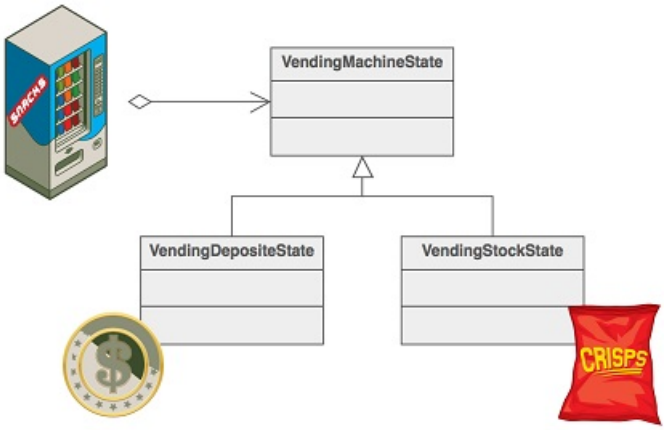
- Notes
  - subject broadcasts events to all registered observers

State:

- Definition/Use
  - a clean way for an object to partially change its type at runtime
  - a monolithic object's behavior is a function of its state
- Structure



–

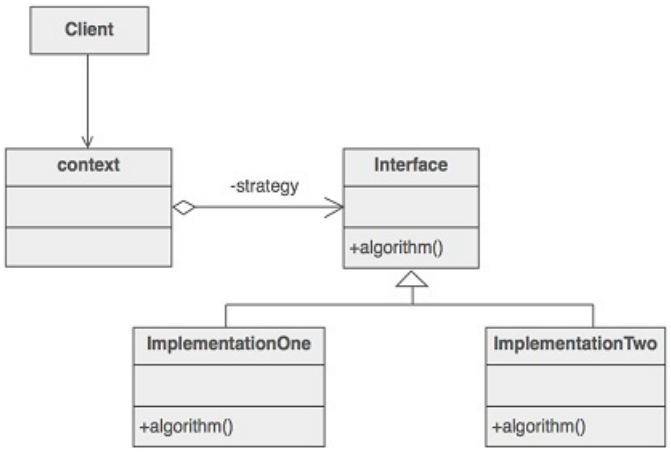


- Notes
  - pattern does not specify where the state transitions will be defined
    - the "context" object
    - each individual State derived class
      - advantage is ease of adding new State derived classes
      - disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses

---

Strategy:

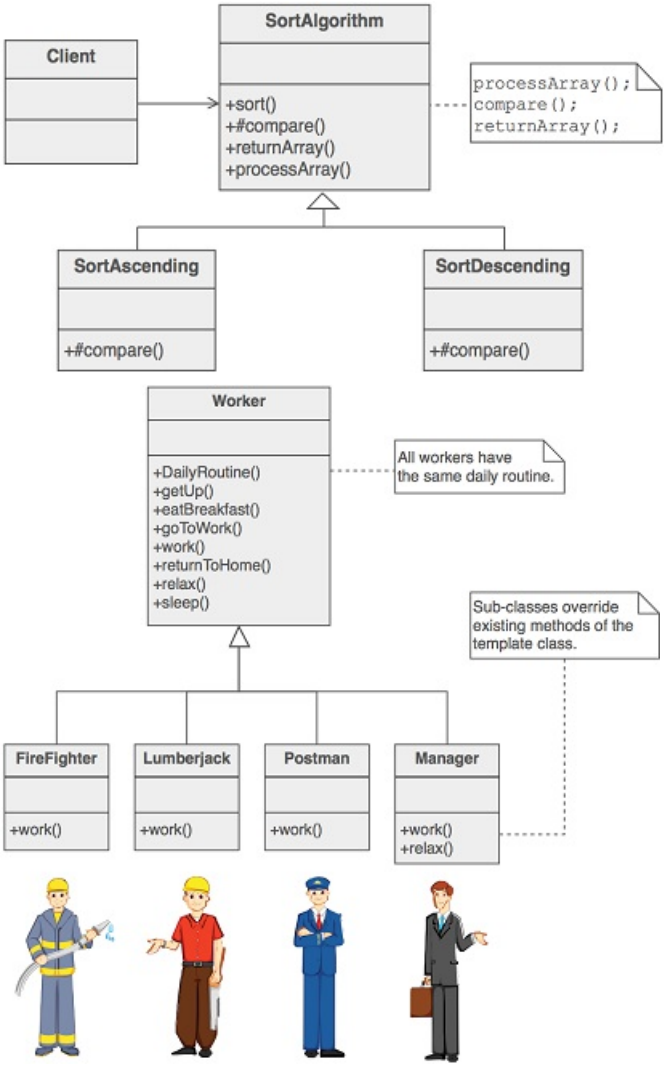
- Definition/Use
  - algorithms can be selected on the fly
  - defines a set of algorithms that can be used interchangeably
- Structure



- Notes

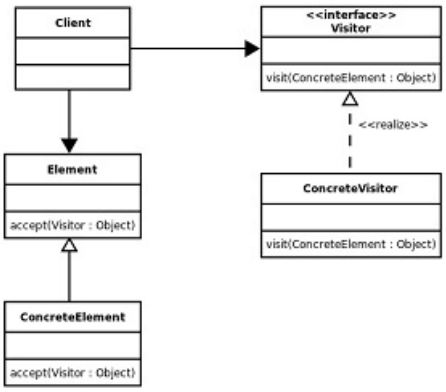
- identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point"

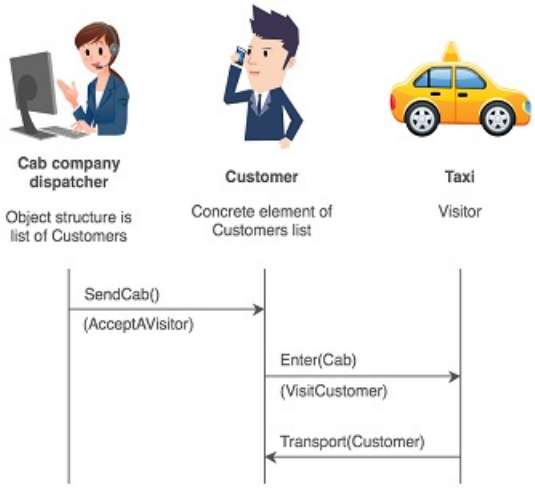
Template method:

- Definition/Use
  - describes the program skeleton of a program
  - component designer decides which steps of an algorithm are invariant (or standard), and which are variant (or customizable)
- Structure
  -


  -
- Notes
  - examine the algorithm, and decide which steps are standard and which steps are peculiar to each of the current classes

Visitor:

- Definition/Use
  - a way to separate an algorithm from an object
- Structure
  - 

  - 

- Example
  - ```
    public interface CharacterVisitor {

      public void visit(char aChar);

    }

    public class MyString {

    // ... other methods, fields

      // Our main implementation of the visitor pattern
      public void foreach(CharacterVisitor aVisitor) {
    ```
    ```

```
    int length = this.length();
    // Loop over all the characters in the string
    for (int i = 0; i < length; i++) {
      // Get the current character, and let the visitor visit it.
      aVisitor.visit(this.getCharAt(i));
    }
  }

// ... other methods, fields

}// end class MyString

public class MyStringPrinter implements CharacterVisitor {

  // We have to implement this method because we're implementing the
      CharacterVisitor
  // interface
  public void visit(char aChar) {
    // All we're going to do is print the current character to the standard
        output
    System.out.print(aChar);
  }

  // This is the method you call when you want to print a string
  public void print(MyString aStr) {
    // we'll let the string determine how to get each character, and
    // we already defined what to do with each character in our
    // visit method.
    aStr.foreach(this);
  }

} // end class MyStringPrinter
```

- Notes
  - if you have and will always have only one visitor, you'd rather implement the composite pattern

Comparison:

- chain of Responsibility, command, mediator, and observer, address how you can decouple senders and receivers, but with different trade
- offs
  - chain of Responsibility passes a sender request along a chain of potential receivers
- command and memento act as magic tokens to be passed around and invoked at a later time
  - in command, the token represents a request

- in memento, it represents the internal state of an object at a particular time
- polymorphism is important to command, but not to memento because its interface is so narrow that a memento can only be passed as a value

Compiler

Compilation system:

- hello.c (source program, text)
- hello.i (modified source program, text) preprocessor (cpp)
  - modifies according to # directives (#include<stdio.h>)
- hello.s (assembly program, text) compiler (cc1)
  - assembly
  - language program (statement equals 1 low level machine language instruction)
  - assembly language common compiler output for different high level languages
- hello.o (relocatable object programs, binary) assembler (as)
  - to binary file (relocatable object program) whose bytes encode machine language instructions
- hello (executable object program, binary) linker (ld) (linked with printf.o)
  - linker merges standard c library functions (executable object file)

sources:

- Computer Systems: A Programmers Perspective pg 4-7

Hardware Systems

Hardware Organization:

- buses
  - carry bytes of info between componenets
  - words: fixed sized chunks of bytes (4 bytes/32 bits or 8 bytes/64 bits)
- I/O devices
  - system's connection to outside world by controller or adapter
  - display, disk, mouse
  - controller: chip sets in device itself or motherboard
  - adapter plugs into slots on motherboard
- Main Memory
  - temporary storage for both program and data it manipulates
  - physically dynamic random access memory (DRAM)
  - logically organized as linear array of bytes each with its own unique address
- Processor
  - central processing unit (cpu)
  - interprets (or executes) instructions stored in main memory
  - at its core is a word sized register called program counter (PC)
    - At any point points at machine language instruction in main memory
  - executes instruction and updates pc to next instruction
  - instructions revolve around main memory, register file, and arithmetic/logic unit
    - reigter file small storage device with word sized registers each with its own unique name
    - ALU computers new data and address values
  - Simple operations CPU carries out at the request of an instruction
    - Load: Copy a byte or a word from main memory into a register, overwriting the previous contents
    - Store: Copy a byte or a word from a register to a location in main memory, overwriting the previous contents
    - Operate: Copy the contents of two registers to the ALU, perform an arithmetic operation on the two words, and store the result in a register, overwriting the previous contents
    - Jump: Extract a word from the instruction, copy that word into the program counter (PC), overwriting the previous value

sources:

- Computer Systems: A Programmers Perspective pg 7-10

Operating System

Processes:

- instance of computer program
- abstraction for a running program
- single CPU can appear to execute multiple processes concurrently by having the processor switch among them
- OS performs this using contect switching
- OS keeps track all state information for a process (context)
  - values of PC, register file, main memory
- single processor can only execute code for single process
- when OS decided to transfer control to new process performs context switch
  - save contect of current process, restore contect of new process, pass control to new context
- system call: special function, passes control to OS and saves shell context, creates new context (hello) and gives it control
- scheduling: kernel decides to preempt the current process and restart a previously preempted process
  - handled by code in the kernel called scheduler
- sum: decision: scheduling, act: context switch

Threads:

- execution unit
- process can be made up of multiple threads that execute concurrently on process context with same code and global data
- requirement for concurrency in network servers make threads important
  - easier to share data between multiple threads than multiple processes
  - threads more efficient than processes
- multithreaing a method to make programs faster with multiple processors

Virtual Memory:

- abstraction provides each process the illusion that it has exclusive use of the main memory
  - heap: follows code and data areas (which are themselves fixed in size)
    - expands and contracts dynamically at runtime (malloc, free)
  - Shared libraies: near the middle
  - stack: at the top
    - used to implement function calls
    - expands and shrinks dynamically (grows when you call/shrinks when you return from a function)
  - kernel virtual memory

sources:

- Computer Systems: A Programmers Perspective

Concurrency

General:

- concurrency refers to the general concept of a system with multiple, simultaneous activities
- parallelism refers to the use of concurrency to make a system run faster

Resources:

- each thread has its own stack
- threads in a process share heap

Approaches:

- hyperthreading (thread level concurrency)
  - multiple copies of some of the CPU hardware (program counters,register files) while having only single copies of other parts (units that perform floating-point arithmetic)
  - decides which thread to run on a cycle by cycle basis
- instruction-level parallelism: modern processors can execute multiple instructions at one time
- Application level concurrency
  - Benefits
    - overlap useful work with I/O requests
    - separate concurrent logic flow for interactions with humans (resize window)
    - reducing latency by defering work (in Dynamic Storage allocator defer coalescing to concurrent flow at lower priority in CPU spare time)
    - separate logic flows for each client
    - partitioned applications with concurrent flows run better on multiprocessor
  - Methods
    - processes
      - each logic control flow scheduled and maintained by kernel
      - separate virtual address spaces
      - need explicit interprocess communication (IPC) mechanism to communicate
    - I/O multiplexing
      - applications explicitly schedule their own logical flows in the context of a single process

        - Logical flows are modeled as state machines that the main program explicitly transitions from state to state as a result of data arriving on file descriptors
        - all flows share the same address space
    * threads
        - run in the context of a single process and are scheduled by the kernel
        - hybrid of the other two approaches
        -
        - scheduled by the kernel like process flows
        -
        - sharing the same virtual address space like I/O multiplexing flows

---

Tools for sharing:

- lock: allows only one thread to enter the part that's locked and the lock is not shared with any other processes
- Mutex Used to provide mutual exclusion
  - ensures at most one process can do something (like execute a section of code, or access a variable) at a time
  - A famous analogy is the bathroom key in a Starbucks
    * only one person can acquire it, therefore only that one person may enter and use the bathroom
    * Everybody else who wants to use the bathroom has to wait till the key is available again
- Monitor: object designed to be accessed from multiple threads
  - member functions or methods of a monitor object will enforce mutual exclusion, so only one thread may be performing any action on the object at a given time
  - if one thread is currently executing a member function of the object then any other thread that tries to call a member function of that object will have to wait until the first has finished
  - no part of the instance can be touched by more than one process at a time
  - A monitor is like a public toilet
    * Only one person can enter at a time
    * They lock the door to prevent anyone else coming in, do their stuff, and then unlock it when they leave
- Semaphore: lower-level object
  - P() and V()functions
  - You might well use a semaphore to implement a monitor
  - A semaphore essentially is just a counter

    - When the counter is positive, if a thread tries to acquire the semaphore then it is allowed, and the counter is decremented
    - When a thread is done then it releases the semaphore, and increments the counter
  - Semaphores are typically used as a signaling mechanism between processes
  - A semaphore is like a bike hire place
    - They have a certain number of bikes
    - If you try and hire a bike and they have one free then you can take it, otherwise you must wait
    - When someone returns their bike then someone else can take it
    - If you have a bike then you can give it to someone else to return, the bike hire place doesn't care who returns it, as long as they get their bike back

---

Issues:

- Deadlock is a condition in which a task waits indefinitely for conditions that can never be satisfied
  - task claims exclusive control over shared resources
  - task holds resources while waiting for other resources to be released tasks cannot be forced to relinguish resources a circular waiting condition exists
- Livelock conditions can arise when two or more tasks depend on and use the some resource causing a circular dependency condition where those tasks continue running forever
  - this blocks all lower priority level tasks from running (these lower priority tasks experience a condition called starvation)
  - A real world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time

---

Implementing in Java:

- extend Thread
  - can't extend any more classes
  - write run() function with what ever you want thread to do
    - thread.start() starts new thread (calls run within start)
    - thread.run() will run
- implement runnable
  - can implement more interfaces

- write run() function with what ever you want thread to do and create thread with runnable in constructor

---

sources:

- Computer Systems: A Programmers Perspective
- https://www.quora.com/Semaphore-vs-mutex-vs-monitor-What-are-the-differences
- http://stackoverflow.com/questions/7335950/semaphore-vs-monitors-whats-the-difference
- http://stackoverflow.com/questions/2332765/lock-mutex-semaphore-whats-the-difference
- http://javarevisited.blogspot.com/2014/07/top-50-java-multithreading-interview-questions-html

Distributed Systems

General:

- organizing resources via network with more latency, less bandwidth, and higher error rate
- Vs parallel systems
  - Distributed has multiple (distributed users) vs parallel designed for single user or user process
  - Parallel has less security issues
  - Distributed Systems: cooperative work environment, Parallel Systems: environment designed to provide the maximum parallelization and speed-up for a single task

Definitions:

- task: collection of resources configured to solve a particular problem
  - contains not only the open files and communication channels, but also the threads (a.k.a. processes)
  - a task is a factory, all of the means of production scattered across many assembly lines

Migrating:

- Migrating Computation
  - need to move resources (memory, files)
  - move state of interaction with resources
  - need to restore communication (other side need to know new location)
- Migrating File State
  - keep track of the essential file state within the process and be ready to recreate it
  - migration system guarantees that higher
  - level file operations are atomic, operation opens the file, seeks as needed, performs a read or write, and then closes the file
  - or maintain the same file state, but only reopen the file when needed
- Migrating Communication Sessions
  - need to reestablish and map gloabl state, such as network sockets
  - higher level network name

  - edge cases, such as what happens when a process move mid
  - transmission

Networking:

- LAN: Local Area Network a homeogeneous network
  - to communicate broadcast (yell) and all stations hear the broadcast
  - station have station id or LAN Address and messages have sender and reciever id
    - everyone can hear other stations messages but ignore it unless diagnostic or malicious
  - size is self-limiting
    - longer wire the weaker the signal
    - greater the distance through the air the weaker the signal
    - network can be clogged with collision, can collapse with utilization as low as 30
- bridge
  - connect LANs
  - send message for station only to relevant LAN (hashtable)
  - bridge managed so if station changes LAN
  - bridges (through configuration) create a spanning tree of location to prevent cycles
    - if bridge fails form a different tree to get around failure
  - can't create bridges for the whole planet
- IP Address
  - IP4
    - Class A (huge): 8 bits(nework) + 24 bits (host) address begin with 0
    - Class B (big): 16 bits(nework) + 16 bits (host) address begin with 10
    - Class C (small): 24 bits(nework) + 8 bits (host) addess begin with 110
  - IP6
    - Classless, first few bits to describe the network/host division
    - 73.93.0.0/15
      - address with 15 network bits and 17 host bits

---

incomplete:

- Dynamic Host Configuration Protocol (DHCP) allows IP addresses to be assiged on a temporary or quasi-permant basis
  - DHCP is great for clients – but not for servers: Servers need to have well-known addresses

sources:

- http://www.andrew.cmu.edu/course/15-440-f14/index/lecture_index.html

General Languages

Classification:

- Abstraction
  - Declarative
    - Functional
  - Imperative
    - Procedural
- Behavior
  - Dynamic
  - Static

Declarative Languages:

- not imperative
- describes what computation should be performed and not how to compute it
- lacks side effects (referentially transparent)
  - an expression always evaluates to the same result in any context
  - instance can be replaced with its corresponding value without changing the program's behavior
- clear correspondence to mathematical logic

Functional Languages:

- Def
  - define programs and subroutines as mathematical functions
  - many functional languages are "impure", containing imperative features
  - many functional languages are tied to mathematical calculation tools
  - declarative: programming is done with expressions or declarations instead of statements
  - the output value of a function depends only on the arguments that are input to the function
    - calling a function f twice with the same value for an argument x will produce the same result f(x) each time

    - eliminating side effects(changes in state that do not depend on the function inputs) makes it much easier to understand and predict the behavior of a program

- Pure ex: Haskell
- Impure ex: SML

Imperative Languages:

- Def
  - uses statements that change a program's state

Scripting Languages:

Hierarchy of programming languages:

- High-Level language (C, Java, PHP, Python)
  - more complex than machine code
- Assembly language
  - machine code with names substituted for numbers
- Machine code
  - only numbers
- Hardware
- convert program into machine language
  - compile the program
  - interpret the program

Web Development Languages:

- HTML/XHTML
  - Defines content of web page
- CSS

  - appearance of web page

- HTML+CSS can create static web pages
  - static pages can interact with your visitor through the use of forms
  - form is fill, request submitted and sent back to the server, new static web page is constructed and downloaded into the browser
  - disadavantage of static web pages, only way visitor interact with the page is by filling out the form and waiting for a new page to load

- javascript
  - behavior of web page
  - can validate each of the fields as visitor enter and provide immediate feedback when they make a typo (vs after they filled everything and submitted)
  - add animations into the page which either attract attention to a specific part of the page or which make the page easier to use
  - provide responses within the web page to various actions that your visitor takes
  - load new images, objects, or scripts into the web page without needing to reload the entire page
  - pass requests back to the server and handle responses from the server without the need for loading new pages
  - not everyone visiting your page will have JavaScript and so your page will still need to work for those who don't have JavaScript

---

sources:

- `https://en.wikipedia.org`
- `https://wiki.haskell.org/Referential_transparency`

APIs

Summary:

- `Math..exact`
  - for arithmetic overflow
- Iterator
- Scanner
- String
- StringBuilderr
- Set (HashSet)
- List (Arraylist)
- LinkedList
- Stack
- Queue (LinkedList, PriorityQueue)
- PriorityQueue
- Map (HashMap)

Iterator:

- hasnext
- next
- remove

Scanner:

- findinline(Pattern)
- hasNext..int,byte,pattern()
- next...Int

String:

- charAt
- codePointCount(int begin,int end) returns unicode(ASCII) points within range
- valueOf
- toLowercase()

StringBuilder:

- append()..int,char,etc.
- charAt
- insert(int i, char,int, string...)
- toString

Set(Hashset):

- add
- clear
- clone
- contains
- isEmpty
- iterator
- remove
- size

List (ArrayList):

- add(E e),(int ind, E e)
- clear

- get(int x)
- indexOf
- set(int ind, E e)
- remove

LinkedList (LinkedList):

- add(E e),(int i, E e)
- addFirst
- addLast
- get(int i)
- peek
- remove
- set

Stack (Stack):

- empty()
- peek
- pop
- push
- search(E e)

Queue (LinkedList, PriorityQueue):

- Interface with multiple implementations

PriorityQueue (Binary Heap):

- add(E e)

APIs notes

- iterator
- poll()
- peek()

Map (HashMap):

- put
- size
- get
- contains

Comparator:

- `implements Comparator<T>`
- override compare method

Java Nuances

General Tips:

- Getter and setter
- Override and super
- Java outmatically collects garbage
- &&/|| checks left first
- + strings makes a new string every time, if you want to do in a loop use stringbuilder(reduce memory)
- Everything in Java not explicitly set to something, is initialized to a zero value
  - references (anything that holds an object):null
  - int/short/byte:0
  - float/double:0.0
  - booleans: false.
  - array of something, all entries are also zeroed

Useful built in functions:

- Arrays
  - Arrays.binarySearch(arr, target)
    - Negative value shows where it should be
  - Arrays.sort(arr)

Switch Statement:

- Example
  -
```
switch (month) {
            case 1: monthString = "January";
                    break;
            case 2: monthString = "February";
                    break;
            case 3: monthString = "March";
                    break;
            case 4: monthString = "April";
```

```
                break;
            case 5: monthString = "May";
                break;
            case 6: monthString = "June";
                break;
            case 7: monthString = "July";
                break;
            case 8: monthString = "August";
                break;
            case 9: monthString = "September";
                break;
            case 10: monthString = "October";
                break;
            case 11: monthString = "November";
                break;
            case 12: monthString = "December";
                break;
            default: monthString = "Invalid month";
                break;
        }
```

Breaking out of for loops:

- if you want to skip a particular iteration, use continue

```
for(int i=0 ; i<5 ; i++){

    if (i==2){

      continue;
    }
}
```

- if you want to break out of the immediate loop use break

```
for(int i=0 ; i<5 ; i++){

    if (i==2){

        break;
    }
}
```

- if there are 2 loop, outer and inner.... and you want to break out of both the loop from the inner loop, use break with label

```
lab1: for(int j=0 ; j<5 ; j++){
     for(int i=0 ; i<5 ; i++){
```

```
        if (i==2){

          break lab1;
        }
      }
}
```

Generics:

- Definition
  - generics are a facility of generic programming
    - a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters
  - ex: compiletime: List¡String¿ runtime:List
- Notes
  - in java, generics are only checked at compile time for type correctness
  - generic type information is then removed via a process called type erasure, to maintain compatibility with old JVM implementations, making it unavailable at runtime
- Sources
  - `https://en.wikipedia.org/wiki/Generics_in_Java`

Type Classifications:

- Concrete Types
  - concrete types describe object implementations, including memory layout and the code executed upon method invocation
  - the exact class of which an object is an instance not the more general set of the class and its subclasses
  - beware of falling into the trap of thinking that all concrete types are single classes!
  - Set of Exact Classes
  - ex: List x has concrete type ArrayList, LinkedList, ...
- Abstract Types
  - Abstract types, on the other hand, describe properties of objects
  - They do not distinguish between different implementations of the same behavior

  - Java provides abstract types in the form of interfaces, which list the fields and operations that implementations must support

---

Access Modifiers:

- public
  - any class can access
  - accessible by entire application
- private
  - only accessible within the class
- protected
  - allow the class itself to access them
  - classes inside of the same package to access them
  - subclasses of that class to access them
- package protected
  - default
  - the same class and any class in the same package has access
  - protected minus the subclass unless subclass is in package
- Static: Belongs to class not an instance of the class

Things to override in new object (for hashing and equality uses):

- public int hashCode()
- public boolean equals(Object object)

```
ex: Tiger
   @Override
   public boolean equals(Object object) {
      boolean result = false;
      if (object == null || object.getClass() != getClass()) {
         result = false;
      } else {
         Tiger tiger = (Tiger) object;
         if (this.color == tiger.getColor()
               && this.stripePattern == tiger.getStripePattern()) {
            result = true;
```

```
        }
      }
      return result;
  }
```

Sources:

- https://www.cs.utexas.edu/~scottm/cs307/codingSamples.htm