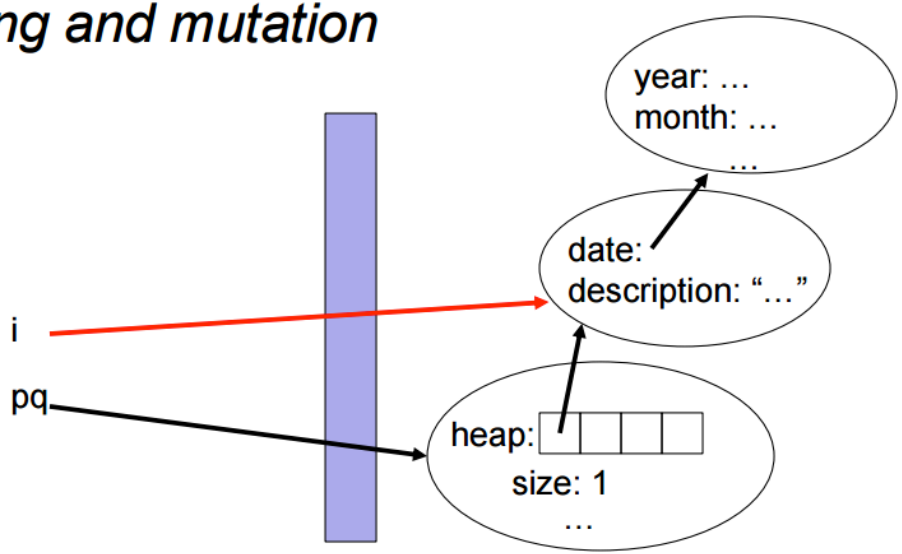
Final Review

CSE373 - Help Section

Preserving Abstraction

Aliasing and mutation



- Client was able to update something inside the abstraction because client had an alias to it!

Preserving Abstraction

```
class BankAccount {
        private Person owner;
    private float balance;
        public BankAccount(Person o, float b) {
                if(o == null || o.birthdate == null){
                        throw new IllegalArgumentException();
        }
        owner = o; balance = b;
    }
    public long getOwnerAge() {
        Date now = new Date();
        long millisecondsPerYear = 365*24*60*60*1000;
        return (now.getTime() - owner.birthdate.getTime()) / millisecondsPerYear;
    }
}
```

Checks not null.

Not null.

NullPointerException!!

Preserving Abstraction

```
class BankAccount {
        private Person owner;
    private float balance;
        public BankAccount(Person o, float b) {
            if(o == null || o.birthdate == null){
                throw new IllegalArgumentException();
        }
        owner = o; balance = b;
    }
    public long getOwnerAge() {
        Date now = new Date();
        long millisecondsPerYear = 365*24*60*60*1000;
        return (now.getTime() - owner.birthdate.getTime()) /
        millisecondsPerYear;
    }
}
```

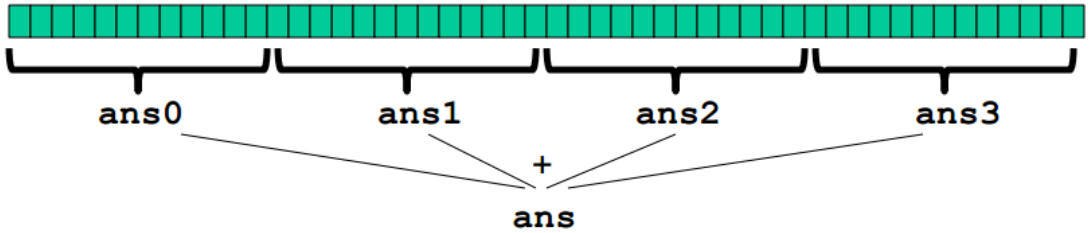
```
Person p = new Person();
p.name = "Bob";
p.birthdate = new Date(1988, 10, 17);
BankAccount acct = new BankAccount(p, 10.0);
p.birthdate = null;
acct.getOwnerAge();
```

Fixation:
The constructor of BankAccount should do a deep copy of the Person object passed in.

Multi-Threading

Parallelism idea

- Example: Sum elements of a large array
- Idea: Have 4 threads simultaneously sum 1/4 of the array
  - Warning: This is an inferior first approach, but it's usually good to start with something naïve works



  - Create 4 *thread objects*, each given a portion of the work
  - Call `start()` on each thread object to actually *run* it in parallel
  - *Wait* for threads to finish using `join()`
  - Add together their 4 answers for the *final result*

Multi-Threading

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … } // override
}
```

```
int sum(int[] arr){// can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++){// do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start();
  }
  for(int i=0; i < 4; i++) { // combine results
    ts[i].join(); // wait for helper to finish!
    ans += ts[i].ans;
  }
  return ans;
}
```

1. Create Threads
2. Call start() to run them in parallel
3. Wait for threads to finish with join()
4. Add together their returns to get the final result

Multi-Threading

```
class MaxThread extends java.lang.Thread {
    int lo; int hi; int[] arr; // arguments
    int ans = Integer.MIN_VALUE; // result
    MaxThread(int[] a, int l, int h) { ... }
    public void run() { // override
        if(hi - lo < SEQUENTIAL_CUTOFF)
            for(int i=lo; i < hi; i++)
                if (arr[i] > ans)
                    ans = arr[i];
        else {
            MaxThread left = new MaxThread(arr,lo,(hi+lo)/2);
            MaxThread right= new MaxThread(arr,(hi+lo)/2,hi);
            left.run();
            right.run();
            ans = Math.max(left.ans, right.ans);
        }
    }
}

int max(int[] arr){
    MaxThread t = new MaxThread(arr,0,arr.length);
    t.run();
    return t.ans;
}
```

Problem:
The current code is entirely sequential because a separate thread of execution is never created (i.e. start() is NEVER called).

Fixation:

```
left.start();
right.run();
left.join();
```

Data Structures

(a) While processing a list of objects, **check if you have processed a particular object** before.

Hashtable

(b) Store **a list of students and their grades**. You must also provide an efficient way for a client to see all students **sorted in alphabetical order by name**. Give the running time for this operation as well.

AVL Tree

(c) Process a digital image to divide the image up **into groups** of pixels of the same color.

Union-Find

Functions	insert()	find()	remove()
Stack	O(1) (push)	/	O(1) (pop)
Queue	O(1)	/	O(1)
Hashtable	O(1)	O(1)	O(1)
AVL Tree	O(logN)	O(logN)	O(logN)
Priority Queue	O(logN)	/	O(logN)
Union-Find	Union: O(1), Find: O(logN)		

Data Structures

(d) Compute a frequency analysis on a file. That is, **count the number** of times each character occurs in the file, and **store the results**.

Hashtable

(e) Store the activation records (i.e. objects containing the return address and local variable associated with a function call) for nested function calls.

Stack

Functions	insert()	find()	remove()
Stack	O(1) (push)	/	O(1) (pop)
Queue	O(1)	/	O(1)
Hashtable	O(1)	O(1)	O(1)
AVL Tree	O(logN)	O(logN)	O(logN)
Priority Queue	O(logN)	/	O(logN)
Union-Find	Union: O(1), Find: O(logN)		

Sorting

	Best Case	Worst Case	Average Case	Additional Space
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Heap Sort	~O(n*logn)	~O(n*logn)	O(n*logn)	O(1)
Merge Sort	~O(n*logn)	O(n*logn)	O(n*logn)	O(n)
Quick Sort (simple)	O(n*logn)	O(n^2)	O(n*logn)	O(1)
Quick Sort (good pivot)	O(n*logn)	O(n*logn)	O(n*logn)	O(1)
Bucket Sort	O(n+K)	O(n+K)	O(n+K)	O(n)
Radix Sort	O(n)	O(n)	O(n)	O(n)