30 September

Classes

either: - a program - a template for new types of objects

object is an "entity" that combines state and behavior

object-oriented program: programs that perform as interactions between objects. Keep state changes in well-defined objects

Classes contain: - fields: variable, part of object state. Each object has its own copy of each field. - access fields by *dereferencing* with the dot notation (ex: `s.length()`) - methods

```
public type name(parameters) {
  statements;
}
```

Methods can be:

- accessor
- mutator

(point class example) (how to access fields from inside same class)

Inheritance

Formalizes hierarchies of how data is structured

```
public class Animal {
  String name;
  int happiness;
  boolean newDay = true;

  public int getName() {
    return name;
  }

  public int getHappiness() {
    return happiness;
  }

  public void interact() {
    happiness = happiness + 1;
    newDay = false;

  public void sleep() {
    newDay = true;
  }
}
```

To create a *subclass* inheriting this *superclass*:

```
public class Cow extends Animal {
  boolean milked = false;

  public void milk() {
    milked = true;
  }
  public void interact() {
    happiness = happiness + 2;
  }

  public void grumpy_interact() {
    super.interact();
  }
}
```

- Multiple levels of inheritance are allowed; multi-inheritance is not.
- Constructors are not inherited: if the super-class has a constructor defined, so must the sub-classes

Linked List

- intro to new friend
- talk about what you know about linked lists
- assume we're going to implement a linked list with two classes: `MyLL` and `ElementLL`

```
public class MyLL {
  public ElementLL first;

  public MyLL(int[] inputList) {

  }
}
```

- what fields and/or methods does the `ElementLL` class need?
    - data, next `ElementLL`
- what steps would you take to implement the constructor for `MyLL`?

October 2

Housekeeping

- Don't come to class / lab if you're sick!
- HW1 is out
- Lab1 is being graded
  - grades and feedback will be on Canvas
- If you're not solid on how to run code locally on your own computer and run tests, come talk to me
  - Run through my process
  - Will send out instructions on getting private/public key authentication set up with GitHub

Makefiles

- look at example from HW1
- don't edit these

Stacks & Queues Refresh

- Stacks
  - first in, last out (last in, first out) "LIFO"
  - can only access the "top" of the stack
  - push, pop, peek
  - Some uses: compilers, code linters (keeping track of parenthesis), search with backtracking, "undo"
  - Usually implemented with arrays, which you will do for homework
- Queues: first in, first out "FIFO"
  - add, remove, peek
  - Uses: job queue for printer, commands in scripting language
  - Java quirk for declaring new Queue objects
  - `Queue<Integer> q = new LinkedList<Integer>();`

What are the special cases that we need to handle to implement these?

- Stack / queue is empty
  - Builtin queues will usually have an `s.isEmpty()` method that returns `True/False`
  - Both have a `s.size()` variable

Given queue `q`, if we want to examine each element exactly once:

```java
int size = q.size();
for (int i=0; i < size; i++) {
    elem = q.remove();
    // operate on elem, possibly adding it back to the queue
}
```

Why do we need a separate variable for `size`? If we directly access q.size in the loop, may cause bugs because actual size of q is changing.

How to perform an operation on each element of a stack? Need backup data structure (another stack).

Exceptions

- Say we've implemented good data structures for queues and stacks. The programmer can still use the data structures in incorrect ways (ex: popping from empty stack).
- In this case, our data structure should "throw an error"
- Errors vs. exceptions
  - error usually cannot be managed inside the program. ex: JVM out of memory
  - exception examples: ClassNotFoundException, IOException, FileNotFoundException

- Two types of built-in exceptions: Checked and Unchecked
    - checked at compile-time vs not
    - if checked, the method containing the exception must either handle the exception or use the keyword `throws`
    - `public static void main(String[] args) throws IOException {...}`
    - unchecked example: divide by zero will compile but throw `ArithmeticException` when run
- Handle exceptions using `try` + `catch` blocks
- Can define custom extensions:

```
public class StackException extends Exception {
...
}
```

Have to implement on your own for homework, but I will tell you that Exception (the superclass) has two constructors:

```
public Exception(String errorMessage) {...}
public Exception(String errorMessage, Throwable err) {...}
```

October 4

Brief intro to runtime

Example: searching in linked list

O(n)

Example: travelling salesman

O(n!)

Example: matrix multiplication

O(nˆ3)

Bonus, show most recent work on improving this bound: divide and conquer, hashing methods

First improvement: divide and conquer, 1969. Algorithm still used today for matrices with n > 500

1990: O(nˆ2.3755) 2024: O(nˆ2.371552)

galactic algorithms

Hash Tables

- Sets: collections of unique objects, with operations `add`, `remove`, and `contain`.
- Let's consider integers to start.
- If implemented in regular array, storing new elements in the next available index, add is O(1), contains is O(n), remove O(n)
- In sorted order? Add is O(N), contains is O(log N), remove O(N)
- Crazy idea: store integer i at index i
    - very efficient if we have unlimited memory! and only positive numbers!
- **hash (def.):** to map a large domain of values to a smaller fixed domain
    - want to map to integer indices
    - hash table: an array that stores elements via hashing
    - hash function: maps values to indices
    - hash code: the output of the hash function for a given value

Reminder: - for a function f: X -> Y, we define - injective function (one-to-one): maps distinct elements of its domain to distinct elements in codomain - x1 != x2 -> f(x1) != f(x2) - surjective function (onto): for every element y in the codomain, there is at least one element in the domain such that f(x) = y. - (not wasting any memory) - bijective: one-to-one correspondence (invertible), relation between two sets

We want: as small of a codomain as possible, but still retain the ability to distinguish f(x1) and f(x2).

How to fix **collisions**: when hash function maps two values to same index!?!?

```
private int hash(int value) {
    return Math.abs(value) % hashtable.length;
}
```

- **Probing**: store exact value in array, resolve collision by moving to another index
    - linear probing: move to next available index (wrapping if needed).
    - quadratic probing: move increasingly far away (+1, +4, +9)
    - how to search?
        - use hash function to find index of value; if f(x) = 0, we know it's not there
        - if we find the value, we know it is there
        - if we find a *different* value, this space has already been filled by probing. Perform probe until we either find the value or find 0.
    - how to remove?

            - if we just set to zero, might break a probe sequence
                - if time: example, ten element array, add 54, then 14, then remove 54
                - use special value to signify “removed”, skip this during “add” and “contains”
    - problem: full table! how to solve?
- **Re-hashing**: moving data to larger array when our hash table is too full.
    - can’t simply copy in-place to larger array; why not?
    - often use prime numbers as table size to reduce collisions
- **Separate chaining:** Solve collisions by storing a list at each index
    - trade multiple probes for traversing lists
    - impossible to “run out” of indices
    - have to check for duplicates in list before adding, and implement proper linked-list access

How to handle objects?

- all objects have built in `hashCode()` method (based on memory address)
- have to be careful with generics

```
public class HashSet<E> implements Set<E> {
  private Node[] elements;

  public HashSet() {
    elements = (Node[]) new HashSet.Node[10];

  private class Node {
    public E data;
    public Node next;
  }

  private int hash(E e) {
    return Math.abs(e.hashCode()) % elements.length;

  ...
}
```

Hash Map

- Similar to hash set, but stores key/value pairs instead of just values
- always hash keys
- `add` method is now `put`: if given key already exists in table, replace value