

## 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:

```
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) {...}
```