User Input, Object Decomposition
Test/Verify-As-You-Go

Writing a lot of code before you verify it:

- results in more total time spent before completion.
- is masochistic. Don’t make your life more difficult.

Make sure you verify your code as you are writing it. Do not wait until you have a lot of code. You are setting yourself up for unnecessary stress.
Object Decomposition

Well-Defined Classes

SOLID Principles

Object Types
Well-Defined Class*

- Has only one purpose that is clearly defined
- Does it well
  - Does everything it needs to do
  - Does nothing more than it needs
- Loosely coupled
- High cohesion

*this looks a lot like “single-responsibility principle”
SOLID Principles*

- **S** - Single-responsibility Principle
- **O** - Open-closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

*no, i don’t think you need to memorize this. Just remember that it exists for when you are ready for it.*
Types of Objects

To help you get started with a design, think in terms of these:

1. Entity Object
2. Control Object
3. Boundary Object
Bonus: Value Object
1. Entity Object

This object generally corresponds to some real-world entity in the problem space.

* Tic tac toe board
Control Object

2. Control Object
Control objects (sometimes also called Manager objects) are responsible for the coordination of other objects. These are objects that control and make use of other objects.

E.g. - game loop
Boundary Object

3. Boundary Object
These are objects which sit at the boundary of your system.

Any object which takes input from or produces output to another system — regardless if that system is a User, the internet or a database — can be classified as a boundary object.

Console, data files, etc
Value Object

Bonus: Value Object

Value objects represent a simple value in your domain. They are immutable and have no identity.
Dealing With User Input

Taking input from the console (Java):

**Scanner Class** - getting input from the keyboard:

**Scanner Next() method**

```java
import java.util.Scanner;
```
Example Command Line Input

mwoodley$ my_application -f initial_data.json -u imperial
mwoodley$ my_application - u metric -f initial_data.json
mwoodley$ my_application -U MEtrIC -F initial_data.json

*you may have a need for -u and -U to have distinct meanings*
public class MyApplication {
    public static void main(String[] args) {
        for (String s: args) {
            System.out.println(s);
        }
    }
}
Output

-U
MEtrIC
-F

initial_data.json
How We Might LIKE to Access Data

Using public members*

String fName = initialData.filename;
int units = initialData.units.

What are the implications of this approach?
Is this safe?
How We Might LIKE to Access Data

Using Getters

String fName = initialData.getFilename();
int units = initialData.getUnits();

Is this better?
How We Might LIKE to Access Data

What data structure would be convenient?

Is this better?
What If You Have a LOT of Initialization Data

INI file

https://en.wikipedia.org/wiki/INI_file
What If You Have a LOT of Initialization Data

To the point where it would be cumbersome to put on the command line.

mwoodley$ my_application -f initialization_data.json

“Initialization_data.json” would contain all the initial values for your application

1) If you already have the JSON parser as part of your application this might be an OK solution
2) This is not really what JSON is all about
Tangent

JSON is a “clear text” format which is easy to read so that’s why it exists?

What is “serialization”? 
That was EASY!!

If you get your user’s input into this form you are halfway there!

...or are you?

How rigid do you want your system to be?
“Condition Your Data”

Recall Friday when we talked about conditioning our data before sending it to our inner methods to streamline our logic?

- Excess whitespace - “...periods....are.like.spaces....”
- Dealing with cases - “PeriOds aRe NOT like spaces”
Make Your Data Convenient for You

Let’s not reinvent the wheel. We have examples of how we might like our input “formatted” already:

1) Strip off each word, “condition” it, store in a list. This is like Java “args” (with some additional functionality)
   https://www.javatpoint.com/command-line-argument

2) Parse user input from left to right
   This is like parsing a word.
Decoupled Design

User Interface

Well-defined (single responsibility)

1. Handles all the interaction with the user (nothing more)
2. “Defends” our inner logic (from “bad” commands)
3. Passes along a proper message
User Action

Here are some options for handling user input*:

1. String of a fixed pattern (CSV-style) - primitive (string)
2. List of strings - data structure
3. Data Object - object

*These three options represent three different degrees of abstraction and three different degrees of “convenience”.
User Action

String of a fixed format (CSV-style):

“Verb,item”

“Verb: action”

Strengths?

Weaknesses?
User Action

List of strings where:

```java
msgList[0] = “verb”;
msgList[1] = “item”;```

Strengths?

Weaknesses?
User Action

Data Object:

```c
    string verb;
    string item;
```
Which is most loosely-coupled?

Options:

1. String of a fixed pattern (CSV-style)
2. List of strings
3. Data Object
Which is most loosely-coupled?

Options:

1. String of a fixed pattern (CSV-style) (concrete)
2. List of strings (concrete)
3. Data Object (abstract)

The Data Object as depicted in example 3 allows for other objects that interact with the object to ignore specifics of implementation (data representation). The data object could use either example 1 or example 2 for its internal representation.
Which of the previously mentioned Object types would our “message” be?

1. Entity
2. Control
3. Boundary
4. Value
Finite State Machines

Both strategies end up with you creating a FSM to process the input.
Finite State Machines

In either of the previously mentioned cases, you are creating a finite state machine which can be represented by a graph.

- **Finite State Machine** (in general):
  

- **ATM** (User Interface and other examples):
  
  https://people.engr.ncsu.edu/efg/210/s99/Notes/fsm/
FSM Graph

In the graph, we can think of each

- **Node** as a *conditional*
- **Edge** as a *statement or group of statements* that are executed without interruption
The case of parsing a word is very strict. That is, the characters must follow in exact order to form the word.

- **Graph** (parsing a word):

Adventure User Interface FSM

Processing user input is tied to the user experience:

- Improper input -> reject input, prompt user again
- Proper input -> display room data, prompt user
Game Map: FSM

Your “game engine” will allow you to:

- Make a wide variety of FSMs for user experience
- Allow the user to traverse it

Your JSON data will describe a FSM.

It may be in your best interest to draw out the map before creating your JSON data.
In the graph, we can think of each
- Node as a conditional
- Edge as a statement or group of statements that are executed without interruption

The case of parsing a word is very strict. That is, the characters must follow in exact order to form the word.
Game Map: FSM

In the GAME graph, we can think of each

- Node as a room
- Edge is a “direction” and is unidirectional

With proper design, your engine should be able to easily traverse complicated graphs.
Some Thoughts on Maps

Use directionality of edges for interesting puzzles:

- Railroad room - a node that only connect two nodes
- Dungeons - no escape (you should use some form of marking or warning)
- Can only enter from one direction but may be able to exit in multiple directions.
- Can enter from multiple adjacent rooms but can only exit to one other
- Shortcuts - do they do the user any favors?

*be sure to consider user experience
Fun for YOU

Make sure you look at the optional “Custom Feature” part of the assignment.

There is a lot of room for YOU to have fun with your creation!