Adventure AMA

February 12, 2021

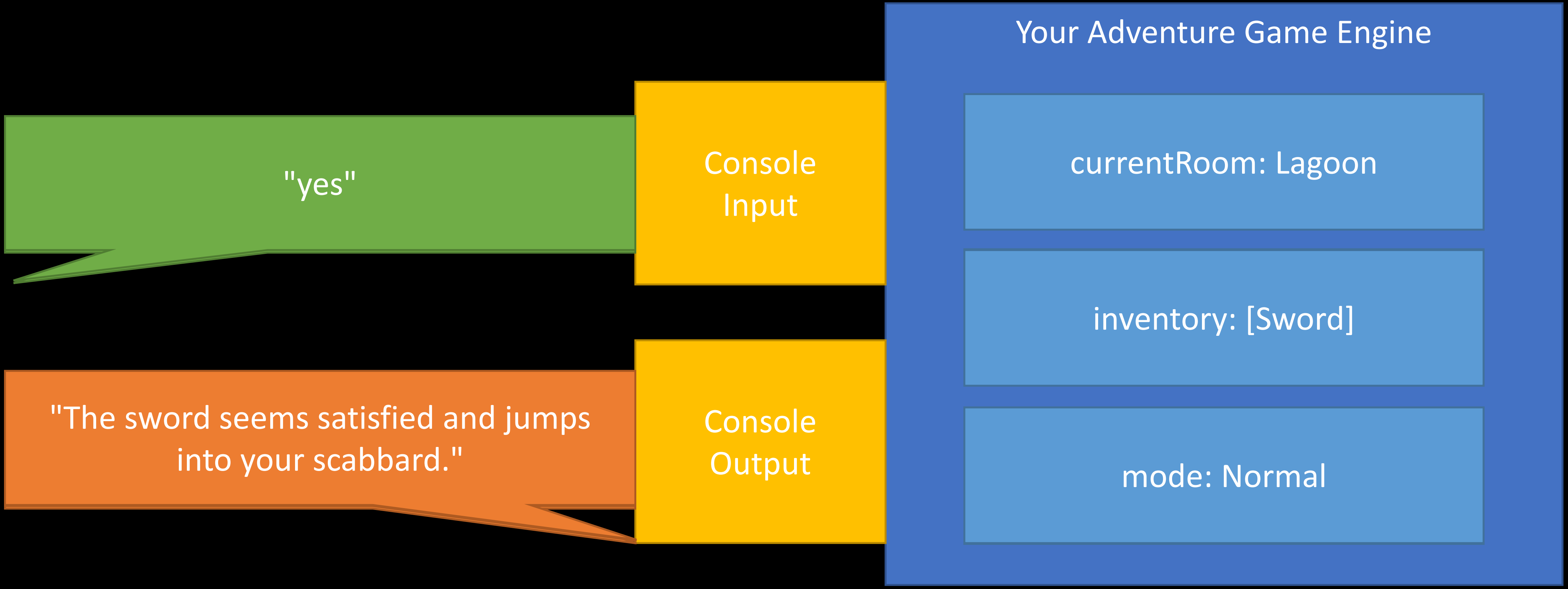
Assignment Objectives

- Separating presentational logic from core logic
- Working with Java input/output
- Designing effective objects
- More work with JSON

Adventure Design

- Week 2: Your design will be plugged into a website
- State-based design
  - Be able to advance the game **one step** at a time
  - Avoid needing to store previous command
- A step of your game includes
  - Accepting an input
  - Changing the state in your engine, if applicable
  - Providing some feedback to the user
    - Optional in some cases

Adventure Design – Visualized


"yes"
Console Input
Your Adventure Game Engine
currentRoom: Lagoon
inventory: [Sword]
mode: Normal
"The sword seems satisfied and jumps into your scabbard."
Console Output

New Rubric Items

- Testing: different test files for source files
- Object Decomposition
  - Member variables & avoiding duplicate storage
  - Placement of member functions
  - What methods should be public?
    - A Connect-Four method that sets a slot to a player color
    - A Connect-Four method that takes a column index and "drops" a player token

Customizability

- Concrete requirements:
  - JSON must be 10+ rooms
  - Layout mustn't be a straight line; there should be some different paths you can take
- Recommended:
  - Have fun with the theme!
- Where is the sample JSON?
  - ./src/main/resources/siebel.json
- Can I extend the command keywords / output text?
  - Yes, but be sensible
  - Retain given command words and add new ones

Testing Adventure

- **Avoid** redirecting streams
  - Consider creating methods that can take in strings
- Tests should reflect what the player can interact with
  - Players don't give their commands as separated lists;
    They give them as fully typed out lines

Code Review Digest – Return Values

- Return values should be optimally useful for *any* potential caller
  - Don't try to assume how the user wants their data
  - Don't assume the user will understand what an arbitrary value means

```
public String generateValues() {
  // get values in "list" variable
  return String.join(",", list);
}
```

```
public List<String> generateValues() {
  // get values in "list" variable
  return values;
}
```

```
public int determineWinner() {
        if (checkRedWins()) {
          return 0;
  } else if (checkYellowWins()) {
    return 1;
  }

        return -1;
}
```

Code Review Digest – Over-modularizing

- Short functions are not always modular!
- Masking most of the functionality behind one function doesn't make your code modular; see below

```
public Evaluation evaluate() {
  if (Math.abs(numO - numX) >= 2) {
    return Evaluation.UnreachableState;
  }
  if (getWinner() == 'X') {
    return Evaluation.Xwins;
  } else if (getWinner() == 'O') {
    return Evaluation.Owins;
  } else if (getWinner() == 'R') {
    return Evaluation.UnreachableState;
  }
  return Evaluation.NoWinner;
}
```

Code Review Digest – Over-modularizing (cont.)

- What to look for when making code more modular
  - Small code blocks that perform a distinct function
  - Any time a method name includes "and", i.e. has **too many responsibilities**
  - Repeated sections / sections that share similarity
- What to avoid
  - Copying the code as-is into "sectioning" functions
  - Sacrificing ease-of-understanding or data-redundancy

Code Review Digest – Testing Collection Equality

- Checking if two collections have the same number of elements in tests is **insufficient.**
- If order does matter (checking that the collections are an *exact* match)
  - Use assertEquals as you normally do; assertArrayEquals for arrays
- If order doesn't matter (checking that they contain the same elements)
  - Easy trick: use Collections.sort() on both expected and actual, then do the same as above

AMA

Visit the pinned thread on Campuswire to ask your questions

If you see a question you'd like to hear the answer to, upvote it