Table of Contents

Purpose

This document is meant primarily for new hires at the company who will either be developing backend code or documenting the backend code. While we expect new hires to at least have a basic understanding of JavaScript, we don't expect you to know the internal conventions that we use when writing code. This document will explain the company's internal conventions while using code that has been used for our clients.

We will use code whose purpose is to set up a new gaming world with game rooms that are indexed by zones and room types. Let's start breaking up the code and looking at the company's standards.

Creating a New World with Zones

As you can see from the first line, the purpose of this overall code is to create a *world* for a game instance. This will be started by building the *world*. The highlighted word, *constructor*, is the function that tells us that a *world* will be built.

```
class World {
    constructor() {
        //a list of all instantiated rooms, indexed by zone and roomtype
        this.roomObjs = {}
        //a list of all instantiated items, indexed by zone and roomtype
        this.items = {}
    }
```

Figure 1: Constructor Explanation

The company follows the convention of providing simple explanations of how the code is being used. For each screenshot of code in this document, the explanation sections will be emphasized by red brackets, as shown in Figure 1. The explanation here tells us that *Constructor* will utilize the list of instantiated (objects) for the rooms and items, and then:

- Index the rooms by zone and room type;
- Index the items by zone and room type.

Now that *constructor* has been defined, the parameters and explanations for the creating zones, rooms, and items need to be dealt with.

Setting Up the Zone(s)

```
/**
 *
 * @param {string} zone
 * @param {Array<roomtype>} zoneData
 */
addZone(zone, zoneData) {
    //initialize a list of rooms for this new zone
    this.roomObjs[zone] = []
    //initialize a list of items for this new zone
    this.items[zone] = []
    //iterate through list of all included room types and spawn new rooms
    zoneData.forEach(roomType => {
        //instantiate a room of the provided type and add it into the world
        this.roomObjs[zone][roomType.name] = new roomType()
        //instantiate an empty list of items that this room contains
        this.items[zone][roomType.name] = []
    })
}
```

Figure 2: Zone Parameters + Explanation

Figure 2 shows the parameters (@param) for the *zones*. These code lines are followed by the explanation for adding the *zone(s)* and the data that will be used. Looking at the explanation lines, we can see that the following will occur:

- A list of rooms will be initialized for the new zone;
- A list of items will be initialized for the new zone;
- The list of rooms will be checked for all room types and new rooms will be created for each roomType that meets the criteria and added to the *world*; and
- The code will also instantiate an empty list of items for each room (to be populated later in the code).

Room Objects

It is time to set up the *room object(s)*.

```
/**
 *
 * @param {string} zone
 * @param {roomtype} room
 * @returns instantiated room object
 */
getRoom(zone, room) {
    //lookup the instantiated room using the zone and room type as keys
    return this.roomObjs[zone][room.name]
}
```

Figure 3: Room Objects

Figure 3 shows the code for listing the *room object(s)*. This code is very straightforward. The @param will check the parameter type for zone, room, and room object. The instantiated room will be looked up using the keys for zone and room type and will return the *room object*.

Room Items

We will add *room items* to the item list that was created earlier. The code to do this is shown in Figure 4.

```
/**
 *
 * @param {string} zone
 * @param {roomtype} room
 * @param {Array<itemtype>} roomItems
 */
addItems(zone, room, roomItems) {
    //iterate over all of the room types
    for (let i = 0; i < roomItems.length; i++) {
        //instantiate a new item of this type
        let roomItem = new roomItems[i]()
        //add the item into the item list for the specified room
        this.items[zone][room.name].push(roomItem)
    }
```

Figure 4: Room Items

The parameters are defined in the highlighted text. Items will now be added to zone, room, and roomItems. We see that the following will occur:

- All room types will be iterated using the equation highlighted in green;
- The identified new roomItem will be added to the item list for the specified room.

Once the *room items* have been identified and added to the list, we'll need to execute code to obtain a list of all the item instances in the specified room. Figure 5 shows the code that will produce the items list.

```
/**
 *
 * @param {string} zone
 * @param {roomtype} room
 * @returns a list of the item instances in this room
 */
getItems(zone, room) {
    return this.items[zone][room.name]
```

Figure 5: List of Item Instances for a Room

Confirm Needed Items

There will be times when all of the known items might not be needed. The code shown in Figure 6 is used to determine if all the items currently in a room are still needed. This will be run against the @param of the itemKey and playerPosition.

```
 *
 * @param {itemtype} itemKey
 * @param {playerPosition} pos
 * @returns the item if found or null
 */
takeItem(itemKey, pos) {
    //get all of the items in the current room
    let roomItems = this.getItems(pos.zone, pos.room)

    //iterate over each item
    for (let i = 0; i < roomItems.length; i++) {
        //if this item can be identified by the provided key
        if (roomItems[i].keys.includes(itemKey)) {
            //remove the item from the room's item list
            let item = roomItems.splice(i, 1)
            // return the item instance
            return item[0]
        }
    }
    //if no item matches, return null
    return null
```

Figure 6: Items Needed

Note: if there aren't any item matches, the value of "null" will be returned (see highlighted explanation).

Final Code to Create New World

We have now reached the final segment of coding needed to create a *newWorld*. As shown in Figure 7, the final coding utilizes the parameters for item and playerPosition. The item instance is pushed into the room's item list. Once this is completed, the *newWorld* has now been created (highlighted text).

```
    /**
     *
     * @param {item} item
     * @param {playerPosition} pos
     */
    giveItem(item, pos) {
        //push the item instance into the room's item list
        this.items[pos.zone][pos.room.name].push(item)
    }
}

var world = new World()
```

Figure 7: Create newWorld

Congratulations! You have now walked through an example of actual code used at the company. You should now have a clear understanding of the internal conventions that the company follows. To sum up, good code will show the following:

- Items that are being created dependent on parameters and determined values.
- After each logical segment of code, an explanation (//) will provide information on what is being accomplished with that segment of code.

For more information on the company's internal conventions and good coding practices, you have the following resources at your disposal:

- Your manager and coworkers
- The company's depository of how-to documents
- The company's website