0: Understanding This DOC.

This DOC is for basic documentation of map design and handling of Characters and items. There is a heavy lack of code as we are only concerned with basic understanding of the engine. The idea here, is that one with basic understanding of swift and programing logic should be able to modify the engine with help from this DOC to create their own text adventure game. More complete documentation for more experienced programers will be embedded within the code.

1: Creating the map and rooms:

In this engine creating maps and rooms is the most simple process, however the most crucial of the game, as without any rooms, there is no game. To create a room you simply need to come up with a description of the room, Let's create room 1. We will say that you are in a field surrounded by trees, to the north you can see a small opening in the forest. To add this we will simply use the library **DungeonText:**

```
let DungeonText: [Int:String] = [
        1:"You find yourself in a small clearing surround by forest, you see an opening in the brush to the north"

]
```

Now whenever the players starts the game they will see that message displayed. However typing "go north" will not allow the player to transverse in that direction. Why? Well this is true for two reasons,

1. There is no other room yet.
2. There code does not know what room is north of room 1.

1.1: Better Defining rooms:

To fix this, we need to to create another room and point to witch room is north. So let's first make a new room, let's use 69 because, why not?

```
let DungeonText: [Int:String] = [
        1:"You find yourself in a small clearing surround by forest, you see an opening in the brush to the north",
        69:"You have made it out of the forest! Thanks for playing!"
]
```

Great! So now we have another room! However we still cannot transverse to it because the code still doesn't define that room 69 is north of room 1. To define location of rooms we use 4 Libraries:

```
let whatsNorthOf: [Int:Int] = []
let whatsEastOf:  [Int:Int] = []
let whatsSouthOf: [Int:Int] = []
let whatsWestOf:  [Int:Int] = []
```

So in order to define That room 69 is north of room 1. We will need to put 2 Ints in library whatsNorthOf. We place into it roomnumber : RoomThatIsNorthOfThatRoom. So we will place 1:69 into it, I remember this by saying something like "North of room 1 lies room 69" To help me remember what order the numbers should be in. So our code will look like this:

```
let whatsNorthOf: [Int:Int] = [1:69]
```

So now, when the player types "go North" from room 1, they will enter into room 69.

1.2: Building Gates for the map:

Often you will find you may want to block paths with guards (see 3.3) or gates. So of course this engine has this feature baked right in as well.

This is handled with 5 Libraries called

PathBlockedText
NorthIsBlockedOf
EastIsBlockedOf
SouthIsBlockedOf
WestIsBlockedOf

To block a path for say north of room one we will add PathBlockedText with a message that will show that something is blocking the path. ie:

```
var PathBlockedText: [Int:String] = [
        1:"The path is blocked by a gate numbered is 647."
]
```

Then in the appropriate library, in this case it's **NorthIsBlockedOf** we will place the room number followed what item is needed to in order to pass, in this case will be gate 647.

```
var NorthIsBlockedOf: [Int:String] = [
                                        1:"key647"
]
```

This will declare that we need to obtain an item called "key647" in order to pass through the gate.

2: Creating and placing Items:

Creating, and placing items on the map is quite simple, defining how they act though is a different story as actual code must be written. But as far as defining, placing and stowing goes. It's pretty straight forward without a single line of code (relatively speaking) needing to be written.

To create a new item, we need to change the array of **allItems** Let's say we want to add a peanut into the game, we will simply add an item by the name of peanut.

```
let allItems: [String] = ["sand",
                          "Peanut"
]
```

So now, Peanut exists, but we need to do something with it so that we can fetch it or add it to our inventory later on using something like the *pickup* command The way that we do that is by modifying the library **itemLocation.** This works with a String:Int base, the String defines what item and the Int defines were the item is at.

Int:
0 – Item is in the players inventory
1–0xFFFFFEFF – Item is on the map in room X waiting to be picked up.
0xFFFFFF01 – Item is being held by a character to be given to the player.
0xFFFFFFFF – 0xFFFFFFFF – Reserved for future use.

So let us now place the peanuts on the map. This is done as such:

```
var itemLoction: [String:Int] = ["Peanut":1]
```

That code will place the the item on the map in room one, and upon using the pickup command, it will have it's status changed to 0 to represent being placed in the players inventory.

3: Adding a Character:

The text engine has a wide variety of variable to add characters. This can be achieved with as little as 2 variables or as many as 8 for more complex characters, At the writing, this makes adding characters the most complex part of the engines code.

To add a basic character the player can interact with we only have to modify the libraries **OriginChar** and **CharecterDescription.** If we want to say add a character in room 8, we must decide what we want to character to be. In this example we will use the old pruny man from ACT I. Add to character description the room number of 8 and his description like so:

```
let CharecterDescription: [Int:String] = [
        8:"In the center of the room sits a crusty raisin of a man in deep meditation."
]
```

So now when the player steps into room 8 they will be told that there is a character in it and be aware of his description. But the player will frown once they try to talk with the man as the man still can not interact. This is were **OriginChar** comes into play, this array holds the default message of the Character when the *talk* command is used. So we will add the our message text from room 8 like so:

```
let OriginChar: [Int:String] = [
        8:"In order to cross the bridge onto the main land you must take this broach."
]
```

So now when the player uses the talk command they will now see that the character has something to say. However now they will frown when there is no broach in they're inventory.

3.1: Giving players items:

So now we've set the stage to give the player an item. Now we need for the old man to actually give the player the stupid broach. Assuming the item is already in existence (See chapter 2 for creating and placing items) we can use the library **CharactersToGive** to give the player an item.

```
let CharactersToGive: [Int:String] = [
        8:"BroachOFHigs"
]
```

And with that the character in room 8 will give you the stupid broach when you talk to him. However, if we talk the old man again, we don't want him to say the same thing again. So we must also change **CharItemFlag** and **OptionChar** These two work in tandem to modify characters. These work quite simply, **CharItemFlag** tells the code that if the player has a specific item in the inventory to use the alternate text of **OriginChar** with **OptionChar.** So first let's modify **CharItemFlag** with the broach.

```
let CharItemFlag [Int:String] = [
        8:"BroachOfHigz"
]
```

The code above will be compared to the players inventory, if the broach is found in the inventory then
the code we instead display the text within **OptionChar**. Using **OptionChar** is the same as using **OriginChar** above.

```
let OriginChar: [Int:String] = [
```

```
        8:"Use that broach well my son."
]
```

And that will complete the code of the old man. But what if we only want to give a player an item once a condition is met?

3.2: Giving players items on conditions:

In the case of the old man we give the player a broach, but what about the crazy gibbering speaking guy outside of the old mans house? He only gives you the sword once you have picked up the cypher key to translate what he has to say! Well in this case we have code to handle that as well. To handle this we will need to:

1. Create a default line in **OriginChar**
2. Set a condition for **CharItemFlag** for the cypher
3. Set **OptionChar** as the alternate message.

Now all we need to do is set the library **CharactersToGiveIf** to give the sword, **CharactersToGiveIf** is an optional item prescriber that is automatically checked if **CharItemFlag** conditions are met. We will simply set it like:

```
CharactersToGiveIf: [Int:String] = [
                                        1:"sword"
]
```

The sword will now be given to the inventory upon the condition in **CharItemFlag** being met.

3.3: Characters that block paths.

The engine also contains conditions for characters to block paths. This is not a function exclusive to characters, rather the function was adapted to characters as an extension from a variable designed to block paths with like… locked gates and stuff… So this will basically be a repeat of section 1.2.

This is handled with 5 Libraries called

```
PathBlockedText
NorthIsBlockedOf
EastIsBlockedOf
SouthIsBlockedOf
WestIsBlockedOf
```

To block a path for say north of room one we will add PathBlockedText with a message that will show that something is blocking the path. ie:

```
var PathBlockedText: [Int:String] = [
        1:"A shady figure blocks the path to the north."
]
```

Then in the appropriate library, in this case it's **NorthIsBlockedOf** we will place the room number followed what item is needed to in order to pass, be it a key, a passport… or… How about… The brutalPickOfFuckingDestiny!? Yeah… let's go with that…

```
var NorthIsBlockedOf: [Int:String] = [
                                        1:"brutalPickOfFuckingDestiny"
]
```

This will declare that we need the brutalPickOfFuckingDestiny to get passed the shady man! Of course players may want to utilize the *talk* command to talk to these characters blocking their path, it is recommend that you of course program in hints into **OriginChar**.

4: Creating a battle:

At this time in the code there is no defined battle system. so all battles must still be hard coded. However, this is still some internal battle code. You can place a battle on the map for when entering a room using using the variable **battleMap.**

```
var battleMap: [Int:Bool] = [2:true]
```

By placing **“2:true”** in the dictionary you are saying that there is an undefeated monster in room 2. The Bool will be changed to false whenever the monster is defeated.

4.1: Setting the battle function:

Battles are handled in by the **function doBattle(roomNum: Int)** The core code will read from dictionary **battleMap** and pass the current room number into the doBattle function. Your battle code will look need to look something like this.

```
func doBattle(roomNum: Int)
{
        if(roomNum==2)
        {
                print(“Let the battle begin!”)
        }
}
```