Object Oriented Programming

Let's say we have a game, it has players as follows

```
const player1 = {
        name : 'John',
        battery : 20,
        charge() {
              ~~retu~~ player1. battery = 100
        }
}

const player2 = {
          name : 'Lisa',
          battery : 30,
          charge() {
                player2.battery = 100
          }
}
```

So if the game has n players, we're going to create n objects.

* ~~By~~ Above, we have bundled the data (name, battery) with the methods that operate on the object (charge) and this is nothing but encapsulation

However, we have to repeat code for every player even though they have the same charge method.

Soln: factory functions: Functions that create the object for us.

```
function createPlayer(name, battery){
        return {
            name: name,
            battery: battery,
            charge(){
                        this.battery =100
            }
        }
}
```

```
→ let player1 = createPlayer('John', 20)
  player1.charge()
→ let player2 = createPlayer('Lisa', 30)
```

Here createPlayer is a factory function that creates objects for us. Now we don't need to repeat any code.

However, the function charge is going to be same for all objects we create, and using createPlayer everytime creates the charge method for every object in the memory.

One possible soln

Extract out all common functions outside ~~to~~ like this

```
const createPlayerFuncs = {
            charge : function charge() {
                    this.battery = 100
            }
}

function createPlayer (name, battery) {
        return {
                name : name, battery : battery
                ~~charge~~: ~~charge~~
        }
}
```

→ let player1 = createPlayer('John',20)
player1.charge = createPlayerFuncs.charge
Player1 now has access to charge (A new copy is not created, it points to existing charge in memory)

→ player1.charge()   (To use it)

We can repeat the same process for any numbers of players. They will be all pointing to the same charge function in the memory.

But this seems like a lot of manual work, so in the next session we will look at another way to do it via object.create() and using prototypal inheritance principles