Mini Lab 3

In this last mini-lab you will build an implementation of the Singleton and Strategy design patterns discussed in the slides.

Part 1 - Singleton Design Pattern

For this part, let's suspend the multiverse theory and assume a single universe. You are to create a universe singleton class that will be instantiated only once.

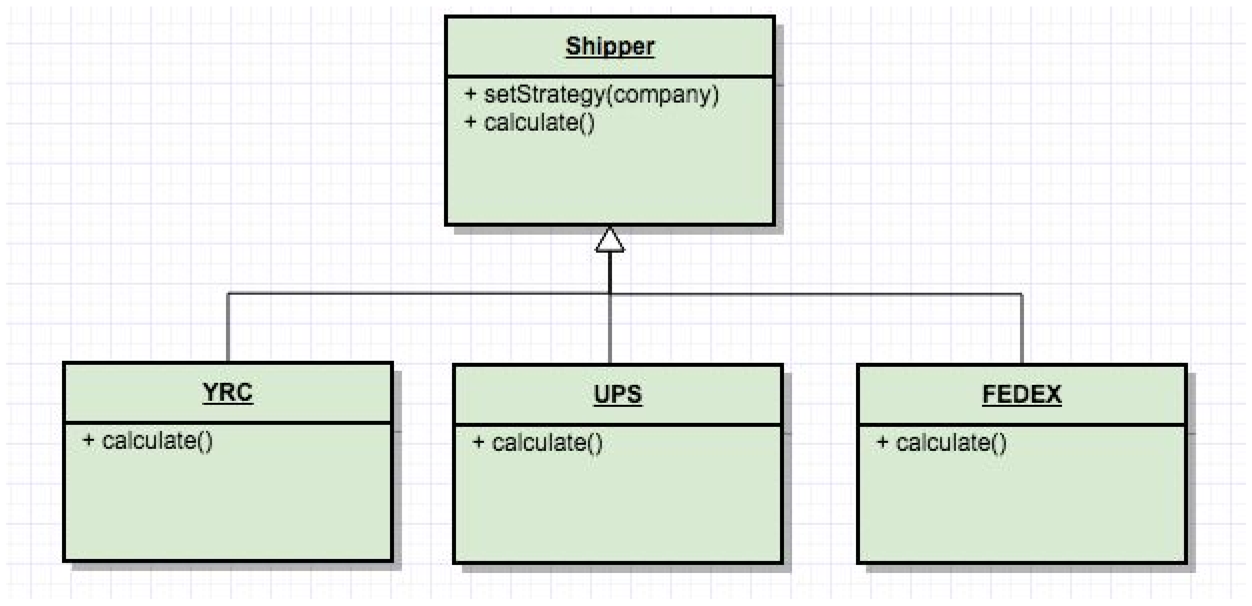
The returned universe instance should contain a number property called **secondsSinceBigBang** which is initiated to 0, and two methods:

1) *bigBang()* - when called, should output "Bang!" to the console and start incrementing the **secondsSinceBigBang** property every 1 second. To do this, use the *setInterval(function, delay)* JavaScript function which accepts a function which will be called every interval, and a delay number (in milliseconds) which tells the thread how long the interval should be.
2) *howOldIsTheUniverse()* - prints the **secondsSinceBigBang** number to the console.

After getting a new instance of the universe, call the *bigBang()* method. Then put the thread to sleep for 10 seconds using *setTimout(function, timeout)*. Pass in a function that, after the thread wakes up, calls the *howOldIsTheUniverse()* method. Verify the output. Try creating a new instance of universe and printing out *howOldIsTheUniverse()*. It should return the same instance and not create a new one.

Part 2 - Strategy Design Pattern

In this part, assume a scenario in which we have a product order that needs to be shipped from a warehouse to a customer. Different shipping companies are evaluated to determine the best price. Use the strategy pattern to implement this by creating a Shipping context and 3 different shipping companies.



Step 1: Create a Shipping class that will act as the context for the pattern. Attach two methods to the prototype: 1) *setStrategy(company)* and 2) *calculate(parcel)*.

Step 2: Create 3 different shipping classes (YRC, UPS, Fedex). Each needs a calculate function that accepts a parcel, and returns a price based on the parcel's weight.

Step 3: Create a **costCalculator** object to keep track of the total cost from a shipping strategy. Have a **totalCost** property and an *add(cost)* method that accepts a number and adds it to the total cost.

Step 4: Create two different parcel objects, each with a weight member that differs.

Step 5: Finally, instantiate the Shipper class, and a single instance of each of the three shipper strategy classes. For each shipper strategy, set the strategy on the shipper context and calculate the cost for each parcel. Have the **costCalculator** keep track of the cost for each package and output the total cost for each shipper strategy.