Homework 8: State Machine Due: Wed May 4, 2022

Your task is to implement a Finite State Automaton (FSA), using the State and Memento design patterns, and fluent design. Your code should be general enough to handle any FSA specified by transitions on certain events. The implemented FSA may be nondeterministic.

Your code will have a main class called FSA. It keeps track internally of the current state of the automaton, and the set of all its possible states. The FSA class has the following methods:

- nextState(e: string) takes as argument an event and returns *this*. This method changes the current state of the automaton by calling the nextState() method of the current state. It does nothing if the current state is `undefined`.
- createState(s: string, transitions: Transition[]): returns *this*. The arguments are a string with the name of the state to be created, and an array of transitions. A transition is an object with one property: type Transition = { [key: string]: string }, where the key is an event name, and the value is the name of the next state on receiving the event. Several transitions may exist for one event (the automaton is nondeterministic). If the FSA already has a state with the given name, it is replaced.
- addTransition(s: string, t: Transition): returns *this*: adds a transition to the state with name *s*, using the addTransition method of that state. Adding a transition (including with createState()) creates source and target states if they do not already exist.
- showState(): string: returns the name of the current state, or `undefined`.
- renameState(oldName, newName): returns *this*. If a state called oldName exists, renames it to newName, else does nothing. Assume no other state called newName exists.
- createMemento(): Memento: creates a memento object with the name of the current state.
- restoreMemento(m: Memento) takes a memento object and restores the FSA state to the state named in the memento object; it does nothing if no such state exists. It returns *this*.

The current state of the FSA is initially set to the first created state; prior to that it is `undefined`.

Your code should have a class called State that encodes the different states the FSA can be in. Class State has the following methods:

- constructor(name: string): creates a state with the given name
- getName(): string: returns the name of the state
- setName(s: string): returns *this*: changes the name of the state. Assume no other state with the given name exists. Changing the name of a state should not affect FSA behavior.
- addTransition(e: string, s: State) adds a transition that on event *e* moves to state *s*. It returns *this*.
- nextState(e: string): State: returns the next state as a result of event *e*. If several moves exist, it returns one at random; it should be possible to return any successor. If no move exists, return `undefined`.
- nextStates(e: string): State[] returns an array of all successor states as a result of event *e*.

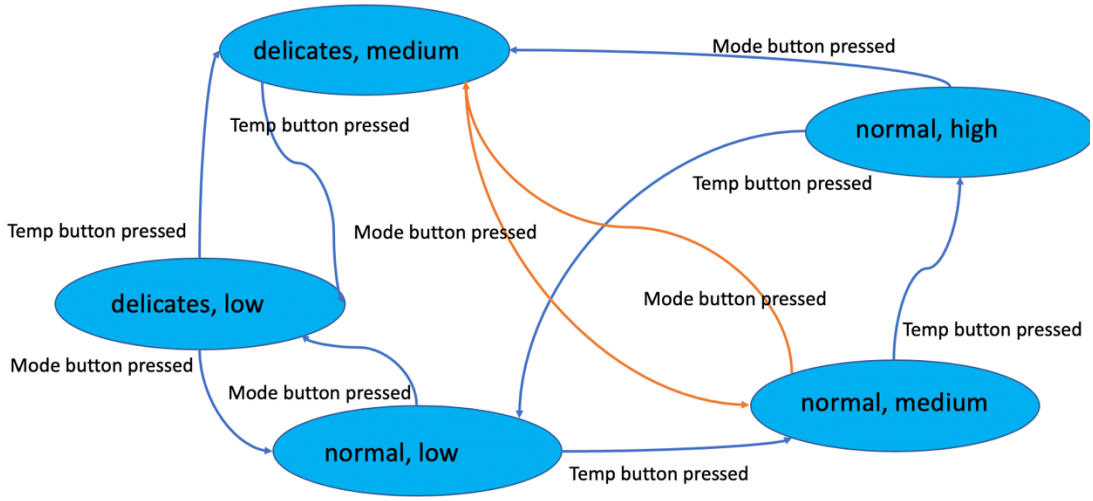
Your code should have a Memento class with the following methods:

- storeState(s: string): void, takes a state name as input, and saves it so it can be restored
- getState(): string, which returns the state name stored in the memento object

A client should only have access to the listed methods and not to any class internals.
State and Memento classes should be declared within the FSA class.

While the user specifies the destination of a transition by name, construct the logic of the FSA so it directly maps from an event to a next *state* rather than its name, avoiding a state lookup by name for every transition.

Here is an example test case for a washing machine.



This FSA could be implemented by code like the following:

```
let myMachine = new FSA()
        .createState("delicates, low", [{mode: "normal, low"}, {temp: "delicates, medium"}])
        .createState("normal, low", [{mode: "delicates, low"}, {temp: "normal, medium"}])
        .createState("delicates, medium", [{mode: "normal, medium"},
                                                {temp: "delicates, low"}]))
        .createState("normal, medium", [{mode: "delicates, medium"},
                                                {temp: "normal, high"}]))
        .createState("normal, high", [{mode: "delicates, medium"},{temp: "normal, low"}]))
```

We could then use this machine by executing

```
myMachine.nextState("temp")         // moves the machine to delicates, medium
                .nextState("mode")     // moves the machine to normal, medium
                .nextState("temp");    // moves the machine to normal, high
let restoreTo = myMachine.createMemento();   // creates memento from current state
console.log(restoreTo.getState());    // prints name of state in memento
myMachine.nextState("mode")         // moves the machine to delicates, medium
                .nextState("temp")     // moves the machine to delicates, low
                .restoreMemento(restoreTo)   // restores the machine to normal, high
```

Here is a sample **nondeterministic** FSA. The states of the machine are the positive divisors of 12 (1,2,3,4,6,12). The events are "D" or "M".

- on event "M", multiply the current state by either 2 or 3 to obtain the possible next states. If none of these are divisors of 12, the machine stays in the current state.
- on event "D", compute the possible next states dividing the current state number by either 2 or 3. If none of these are divisors, the machine stays in the current state.

To model that an event causes the machine to stay in the current state, add a transition to itself.