Exercize 1

The object model of a component is an object with the following fields and methods: a Constructor method, that as a default value takes an empty function, a render method which is also is also a function and it as default value a function that returns an empty sting, all the other fields provided as input they will become field of the component, and finally all the generate fields generated by the invocation of the Constructor methods defined before.

```
var React = (function() {
    var Component = function(obj) {
        if (typeof obj.constructor !== "function")
            obj.constructor = function() {};
        if (typeof obj.render !== "function")
            obj.render = function() {
                return '';
            };
        return obj;
    };
    var generateHTML = function(node) {
        if (typeof node !== "object") {
            return node;
        } else if (node instanceof Array) {
            var elem = "";
            for (var i = 0; i < node.length; i++) {
                elem += generateHTML(node[i]);
            }
            return elem;
        } else {
            var e = '';
            e += "<" + node.tag + " ";
            for (var key in node.attrs) {
                e += key + "='" + node.attrs[key] + "'";
            }
            e += ">";
            e += generateHTML(node.children);
            e += "</" + node.tag + ">";
            return e;
        }
    };
    return {
        class: function(obj) {
            comp = Component(obj);
            comp.constructor();
            return comp;
        },
        render: function renderer(component, DOM) {
            var virtualdom = component.render();
            DOM.innerHTML = generateHTML(virtualdom);
        }
    };
})();
```

Listing 1: Simplest definition of react component and a basic render method

Notice that in this basic render, we didn't include the connection between function and methods. Indeed, we have just print out the content of the corresponding function.

Exercise 2

In order to optimize the rendering function we need to add a new field, in the component, to keep track of the last rendered VDOM. The latter is lightweight structure that has the same structure of a DOM. Therefore, we will have:

```
var Component = function(obj) {
    if (typeof obj.constructor !== "function")
        obj.constructor = function() {};
    if (typeof obj.render !== "function")
        obj.render = function() {
            return '';
        };
    obj.old = null;
    return obj;
};
```

Listing 2: Keep the old VDOM in the component

Therefore we are going to change the render method, of the class React.

```
render: function renderer(component, DOM) {

    var virtualdom = component.render();
    };
    if (component.old === null) {
        DOM.innerHTML = generateHTML(virtualdom);
    } else {
        if(diff(component.old, virtualdom)!==undefined){
          patch(virtualdom, DOM.children);
        }
    }
    component.old = JSON.parse(JSON.stringify(virtualdom));
}
```

Listing 3: New render method

The first time we render a component, we simply call *generateHTML* in all the VDOM. The second time, instead of re-render all the VDOM, we make use of the old VDOM (saved in the component and update at each render) to modify just the part of the DOM that has been really change. Therefore, we employed two functions: *diff* and *patch*. The first go trough the old VDOM and the current VDOM (with the updated values), to mark whether a node should be re-render or not. Furthermore, *diff* is going to flag the node with different marker, using an ”enum” (conceptually), based on the kind of operation has been done in the VDOM. In the following we have the code:

```
var fl = {
    MARKER: 3,
    LEAF: 1,
    NEWNODE: 2,
    TAG: 4,
    ATT: 5
};
var diff = function(oldVdom, newVdom) {
  switch (type(oldVdom, newVdom)) {
      case "array":
          for (var i = 0; i < newVdom.length; i++) {
              var val = diff(oldVdom[i], newVdom[i]);
              if (val) {
                  newVdom.flag = fl.MARKER;
                  return val;
              }
          }
          break;
      case "object":
          if (oldVdom.tag !== newVdom.tag) {
              newVdom.flag = fl.TAG;
              return fl.MARKER;
          } else {
              for (var key in newVdom.attrs) {
                  if (key !== "onClick" && newVdom.attrs[key] !== oldVdom.attrs[key]) {
                      newVdom.flag = fl.ATT;
                      return fl.MARKER;
                  }
```

```
                }

            }
            var childVal = diff(oldVdom.children, newVdom.children);
            if (childVal === fl.LEAF || childVal === fl.NEWNODE || childVal === fl.MARKER)
                {
                newVdom.flag = childVal;
                return fl.MARKER;
            }
            break;
        case "literal":
            if (oldVdom !== newVdom) {
                return fl.LEAF;
            }
            break;
        case null:
            return fl.NEWNODE;

    }
};

var type = function(oldVdom, newVdom) {
  if (oldVdom instanceof Array && newVdom instanceof Array)
      return "array";
  else if (typeof oldVdom === "object" && typeof newVdom === "object")
      return "object";
  else if (typeof oldVdom === "string" && typeof newVdom === "string" ||
      typeof oldVdom === "number" && typeof newVdom === "number")
      return "literal";
  else {
      return null;
  }
};
```

Listing 4: *diff* function

After the *diff* function, that works in the VDOM, we will know whether we need to modify the real DOM or not, and exactly where. Indeed if the function return undefined, it means that nothing has been change and we don't need to patch the DOM. In the case the return of the *diff* is a number we call the patch procedure. The patch function will go through the labelled VDOM and the DOM. Notice that it will not go through all the tree, but just in the part that has been labelled. Here the code of the function.

```
var patch = function(newVdom, DOM) {
    if (typeof newVdom !== "object") {
        return newVdom;
    } else if (newVdom instanceof Array) {
        for (var i = 0; i < newVdom.length; i++) {
            patch(newVdom[i], DOM[i]);
        }
    } else {
        switch (newVdom.flag) {
            case fl.LEAF:
                DOM.innerHTML = newVdom.children;
                break;
            case fl.TAG:
                DOM.outerHTML = generateHTML(newVdom);
                break;
            case fl.ATT:
                for (var key in newVdom.attrs) {
                    DOM.setAttribute(key, newVdom.attrs[key]);
                }
                break;
            case fl.NEWNODE:
                var node = document.createElement(newVdom.tag);
                node.innerHTML = generateHTML(newVdom.children[newVdom.children.length -
                    1], f);
                DOM.appendChild(node);
                break;
            case fl.MARKER:
                if (typeof newVdom.children === "object" &&
                    !(newVdom.children instanceof Array)) {
                    patch([newVdom.children], DOM.children);
                } else {
                    patch(newVdom.children, DOM.children);
                }
                break;
        }

    }
};
```

Listing 5: *patch* function

Notice that, we need to add some special cases, because the DOM treat the child nodes always as an Array, instead our VDOM (node) it can have single node or an arrays of node.

Exercise 3

To connect DOM events to the functions in the Component, we will modify two parts of the previous code: Class and Generate HTML. In the first, we will add an identifier for each function, this will be used to uniquely identify the function of the component. In the second, we connect the the respectively attribute value with the function. In practise, to handle DOM event with the functions, we add the function to Window and we use the bind() method to bind that function to the right component. Here the code:

```
class: function(obj) {
    comp = Component(obj);
    for (var funName in comp) {
        if (funName !== "constructor" &&
            funName !== "render" &&
            funName !== "old") {
            var funId = 'fun' + index++;
            comp[funName] = {
                name: funId,
                fun: comp[funName]
            };
        }
    }
    comp.constructor();
    return comp;
}
```

Listing 6: New class

```
var generateHTML = function(node, funTemplate) {
    if (typeof node !== "object") {
        return node;
    } else if (node instanceof Array) {
        var elem = "";
        for (var i = 0; i < node.length; i++) {
            elem += generateHTML(node[i], funTemplate);
        }
        return elem;
    } else {
        var e = '';
        e += "<" + node.tag;
        for (var key in node.attrs) {
            var funobj = node.attrs[key];
            (function(funobj) {
                window[funobj.name] = function(e) {
                    funTemplate(funobj.fun, e);
                };
            })(funobj);
            if (!funobj.name) {
                e += " " + key + "='" + node.attrs[key] + "'";
            } else {
                e += " " + key + "='" + funobj.name + "(event)'";
            }
        }
        e += ">";
        e += generateHTML(node.children, funTemplate);
        e += "</" + node.tag + ">";
        return e;
    }
};
```

Listing 7: New generateHTML

Therefore, each time we call a generateHTML we will provide the VDOM and a funTemplate. For examples:

```
var f = function(f, e) {
    f.bind(component)(e);
    renderer(component, DOM);
};
generateHTML(virtualdom, f);
```

Listing 8: Example

To see the "counter" example in action, click the following link to JSfiddle page: JSFIDDLE