Javascript Design Patterns

JavaScript Design Patterns

- Since JS is a prototype based OO language as against a class based OO language which C++ and Java are, a certain set of design patterns are implemented slightly differently

- GoF patterns are implementable with Javascript

Constructor Pattern

- Ways to create an object

```
var v1 = {};

//Inherit from the prototype of the passed in object
var v2 = Object.create( Object.prototype );

//Inherit from the prototype of the constructor function
var v3 = new Object();
```

- Ways to access an object

```
//Dot syntax
newObject.someKey = "Hello World";

var value = newObject.someKey;

//Square bracket syntax
newObject["someKey"] = "Hello World";

var value = newObject["someKey"];
```

Module Pattern

- Object literal modules

```
var myModule = {

    myProperty: "someValue",

    // object literals can contain properties and methods.
    // e.g we can define a further object for module configuration:
    myConfig: {
       useCaching: true,
       language: "en"
    },

    // a very basic method
    saySomething: function () {
        console.log( “A day is a great day when I learn something new!” );
    },

    // output a value based on the current configuration
    reportMyConfig: function () {
        console.log( "Caching is: " + ( this.myConfig.useCaching  ? "enabled" : "disabled") );
    },

    // override the current configuration
    updateMyConfig: function( newConfig ) {

        if ( typeof newConfig === "object" ) {
            this.myConfig = newConfig;
              console.log( this.myConfig.language );
        }
    }
};
```

Module Pattern

- Modules can contain scoped variables
- Scoped objects
- Constructor functions (Classes)
- regular functions

Providing Encapsulation

- Encapsulated module state

```
var testModule = (function () {
  var counter = 0;
  return {
    incrementCounter: function () {
      return counter++;
    },
    resetCounter: function () {
      console.log( "counter value prior to reset: " + counter );
      counter = 0;
    }
  };
})();

// Usage:
// Increment our counter
testModule.incrementCounter();

// Check the counter value and reset
// Outputs: counter value prior to reset: 1
testModule.resetCounter();
```

Modules look like classes

```
var myNamespace = (function () {
  var myPrivateVar, myPrivateMethod;
  // A private counter variable
  myPrivateVar = 0;
  // A private function which logs any arguments
  myPrivateMethod = function( foo ) {
      console.log( foo );
  };
  return {
    // A public variable
    myPublicVar: "foo",
    // A public function utilizing privates
    myPublicFunction: function( bar ) {
      // Increment our private counter
      myPrivateVar++;
      // Call our private method using bar
      myPrivateMethod( bar );
    }
  };
})();
```

Lets Try It

- Convert the Collections classes we created into a module

Christian Heilmann's Revealing Module pattern

- Declare everything private and return an object with references to whatever needs to be made public

```
var myRevealingModule = (function () {
    var privateVar = “I am secret",
        publicVar  = “I am famous!";
    function privateFunction() {
        console.log( "Name:" + privateVar );
    }
    function publicSetName( strName ) {
        privateVar = strName;
    }
    function publicGetName() {
        privateFunction();
    }
    // Reveal public pointers to
    // private functions and properties
    return {
        setName: publicSetName,
        pubvar: publicVar,
        getName: publicGetName
    };
})();

myRevealingModule.setName( “Mahesh Singh" );
```

Proxy Pattern

- Add additional intercepting functionality around existing functionality for certain methods

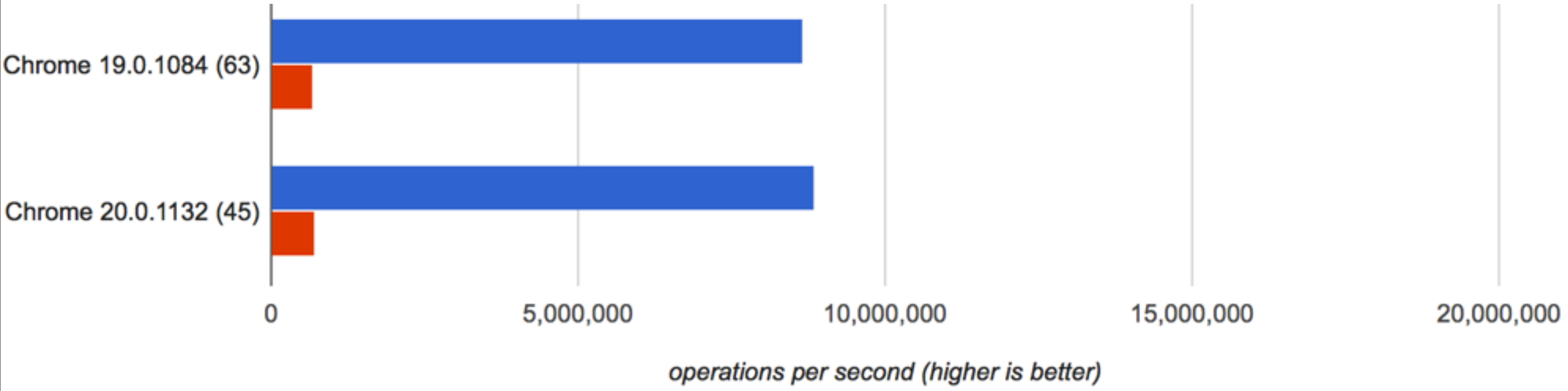
```
function GeoCoder() {
    this.getLatLng = function(address) {
        if (address === "Amsterdam") {
            return "52.3700° N, 4.8900° E";
        } else if (address === "London") {
            return "51.5171° N, 0.1062° W";
        } else if (address === "Paris") {
            return "48.8742° N, 2.3470° E";
        } else if (address === "Berlin") {
            return "52.5233° N, 13.4127° E";
        } else {
            return "";
        }
    };
}
function GeoProxy() {
    var geocoder = new GeoCoder();
    var geocache = {};
    return {
        getLatLng : function(address) {
            if (!geocache[address]) {
                geocache[address] = geocoder.getLatLng(address);
            }
            log.add(address + ": " + geocache[address]);
            return geocache[address];
        }
    };
};
```

Facade

- When we put up a facade, we present an outward appearance to the world which may conceal a very different reality.
- They can also be integrated with other patterns such as the Module pattern.
- JQuery is an example of facade $(“#elemId”) hides the getElementById(“elemId”)

Facades Can Bring Performance Penalties

- JQuery performance for getElementById and $ (“#id”)



Observer Pattern

- One or more observers are interested in the state of a subject and register their interest with the subject by attaching themselves.
- When something changes in our subject, a notify message is sent which calls a method in each observer.
- When the observer is no longer interested in the subject's state, they can simply detach themselves

Observer Example

```
function Subject(){
  this.observers = new ObserverList();
}

Subject.prototype.addObserver = function( observer ){
  this.observers.add( observer );
};

Subject.prototype.removeObserver = function( observer ){
  this.observers.removeAt( this.observers.indexOf( observer, 0 ) );
};

Subject.prototype.notify = function( context ){
  var observerCount = this.observers.count();
  for(var i=0; i < observerCount; i++){
    this.observers.get(i).update( context );
  }
};
```

JQuery Observers

- We can listen to events beyond the standard events on elements. And somewhere in code we can emit the events.

- This enables us to listen to higher level events on UI controls

```
$("#elemId").trigger("created",{name: 'Ranjan', age:22});

$("#elemId").on(eventName, handler);

dialog.on("created", function(obj){
   console.log("User created: "+obj.name);
});
```

Lets Try It

- Modify The Dialog to have “Created”, “Hidden” events. Put listeners on it and ensure it fire. (Hint: attach the event to the enclosing div of the dialog. Pass the div as the data to the handler)

Singleton Pattern

- Allows us to create exactly one instance using a given constructor function

- Useful for situations like service interface or modal dialogs

    - Ex: There can be only one instance of a service proxy used to communicate to the server

    - Ex: There can only be one modal dialog on a page at a time

Singletons With Closures

```
var mySingleton = (function() {
    var instance;
    function init() {
        function privateMethod() {
            console.log("I am private");
        }
        var privateVariable = "Im also private";
        var privateRandomNumber = Math.random();
        return {
            publicMethod : function() {
                console.log("The public can see me!");
            },
            publicProperty : "I am also public",
            getRandomNumber : function() {
                return privateRandomNumber;
            }
        };
    };

    return {
        getInstance : function() {
            if (!instance) {
                instance = init();
            }
            return instance;
        }
    };
})();
```

Lets Try It

- Make our modal dialog a singleton. So that we can access it from anywhere and hide/enable it.

Decorator Pattern

- Decorator is another way to add functionality (Other than inheritance).
- Decorator does composition instead of inheritance
- ObjectB(ObjectA) - An object wraps another object to add more functionality to it

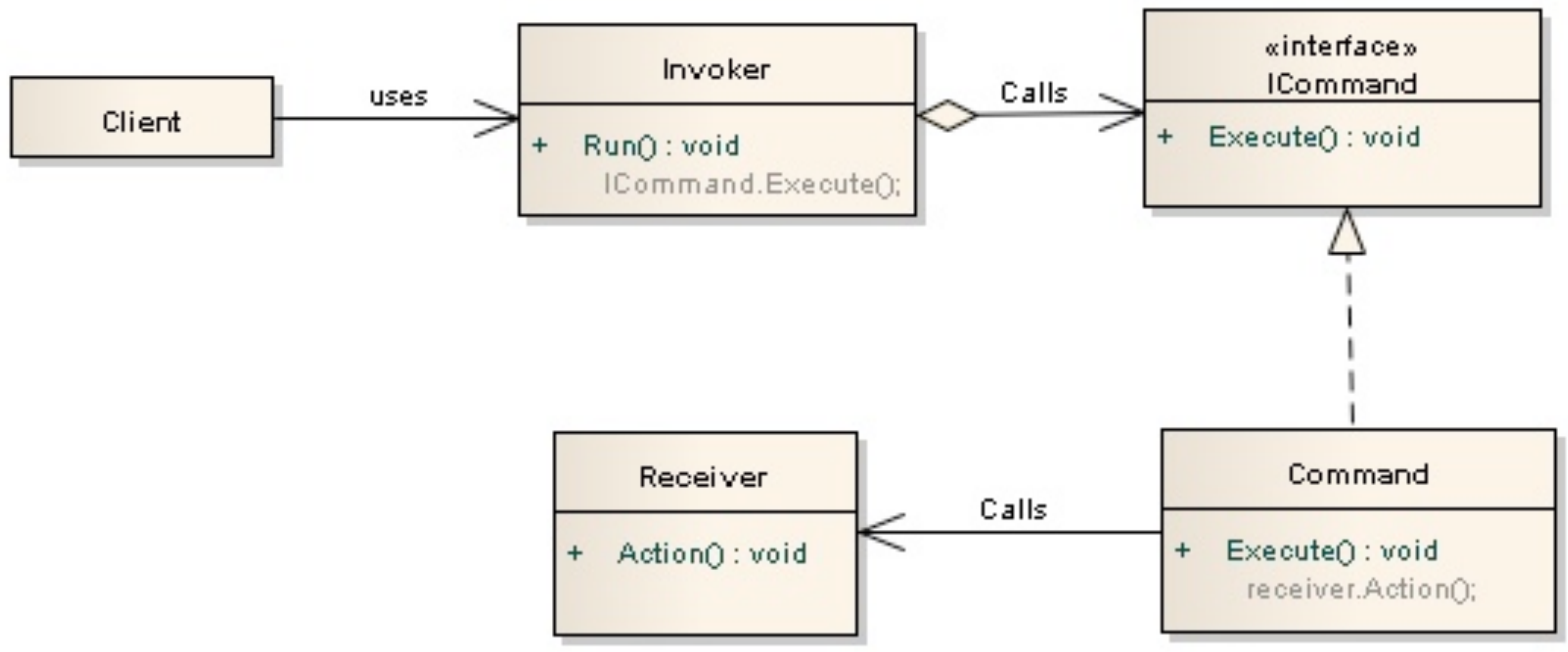
Lets Try It

- Create a new class DecoratedIgnoreDialog that does the role of a dialog with ignore button by decorating the existing Dialog class

Command Pattern

- The Command pattern aims to encapsulate method invocation into a single object
- Then this object can be passed around for execution in any context
- A good pattern to implement the Open-Close principle

Command Pattern



Lets Try It

- Create a text area and an input box. When we type into the input box, the text area should show the text.
- Provide an undo button that removes the text that was typed in reverse order