Friday, January 27, 2017
11:07 AM

```
function Cat() {
   this.name = 'fluffy'
   this.color ='White'
}
var cat = new Cat();

>console.log(cat.name);
VM662:1 fluffy

>console.log(cat.color);
VM677:1 White


function Cat(name,color) {
   this.name = name
   this.color =color
}

var cat = new Cat('blacky','black');


var cat = Object.create(Object.prototype,
{
  name:[
   {value:'Flaffy',
   enumerable:true,
   writable:true,
   configurable:true}],
   color:[
   {value:'white',
   enumerable:true,
   writable:true,
   configurable:true}
   ]
});

var Cat = new  cat('whity','brown');

===
'use strict'

class Cat
{
  constructor(name,color)
  {
    this.name=name;
    this.color=color;
  }
```

```
 speak()
 {
   return "Meewo";
 }
};

var obj = new  Cat('whity','brown');

console.log(obj);
console.log(obj.speak());
===
```

Bracket notation for a properties

```
var obj = new  Cat('whity','brown');
obj['eyeColor']='green';

console.log(obj,obj['color']);
console.log(obj.speak());

===

 Object.defineProperty(obj,'name',{writable:false});
 obj['name'] ='white';
 console.log(Object.getOwnPropertyDescriptor(obj,'name'));
===
 Object.defineProperty(obj,'name',{writable:false});
 Object.freeze(obj.name);
 obj['name'].firstname ='browny';  -- can be changed but freeze will stop override the value...
 console.log(Object.getOwnPropertyDescriptor(obj,'name'));


==

Object.defineProperty(obj,'name',{enumerable:false});
 for (var propertyName in obj){
   console.log(propertyName +":" + obj[propertyName]);
 }

console.log(JSON.stringify(obj));

=====
```

Object.defineProperty

Friday, January 27, 2017
2:41 PM

enumerable ,
writable
configurable

writable

```
Object.defineProperty(obj,'name',{writable:false});
obj['name'] ='white';
console.log(Object.getOwnPropertyDescriptor(obj,'name'));
```

===

```
Object.defineProperty(obj,'name',{writable:false});
Object.freeze(obj.name);
obj['name'].firstname ='browny';  -- can be changed but freeze will stop override the value...
console.log(Object.getOwnPropertyDescriptor(obj,'name'));
```

==

enumerable

```
Object.defineProperty(obj,'name',{enumerable:false});
for (var propertyName in obj){
  console.log(propertyName +":" + obj[propertyName]);
}

console.log(JSON.stringify(obj));
```

configurable

```
Object.defineProperty(obj,'name',{configurable:false});

Object.defineProperty(obj,'name',{writable:true});

delete  obj.name
```

Getter & Setter

Friday, January 27, 2017
3:29 PM

```
var cat = {
  name :{firstname:'Fluffy', last:'Kat'},
  color:'White'
}

Object.defineProperty(cat,'fullName',{
    get: function(){
        return this.name.firstname + ' ' + this.name.last;
    },
    set: function(value){
      var nameparts = value.split(' ')
      this.name.firstname = nameparts[0]
      this.name.last = nameparts[1]
    }
})

cat.fullName= "Zip Zap"
console.log(cat.name.firstname);
console.log(cat.name.last);
```

Prototype

Friday, January 27, 2017
4:03 PM

```
var arr = ['a','b','c'];

Object.defineProperty(Array.prototype,'last',{get: function(){
  return this[this.length-1]
}});

var last = arr.last;

console.log(last);

var nexta = ['1','3','4'];

var slast = nexta.last;

console.log(slast);
```

Object prototype : obj.__proto__

Function prototype : A function prototype is the object instance that will become the prototype for all objects created using this function as a constructor.

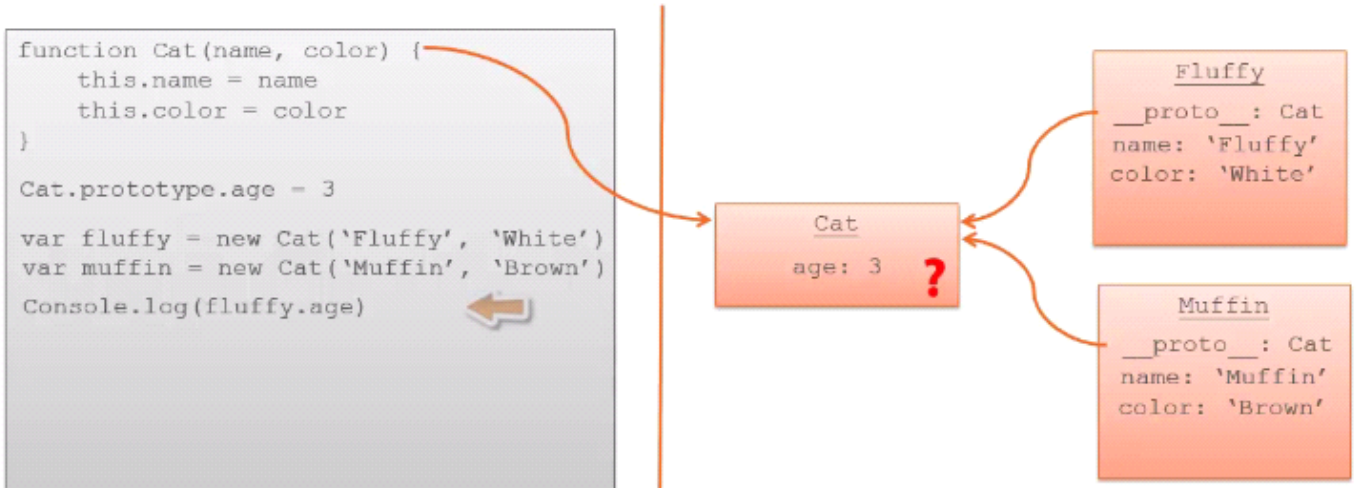
An Object's prototype: An object's prototype is the object instance from which the object is inherited.

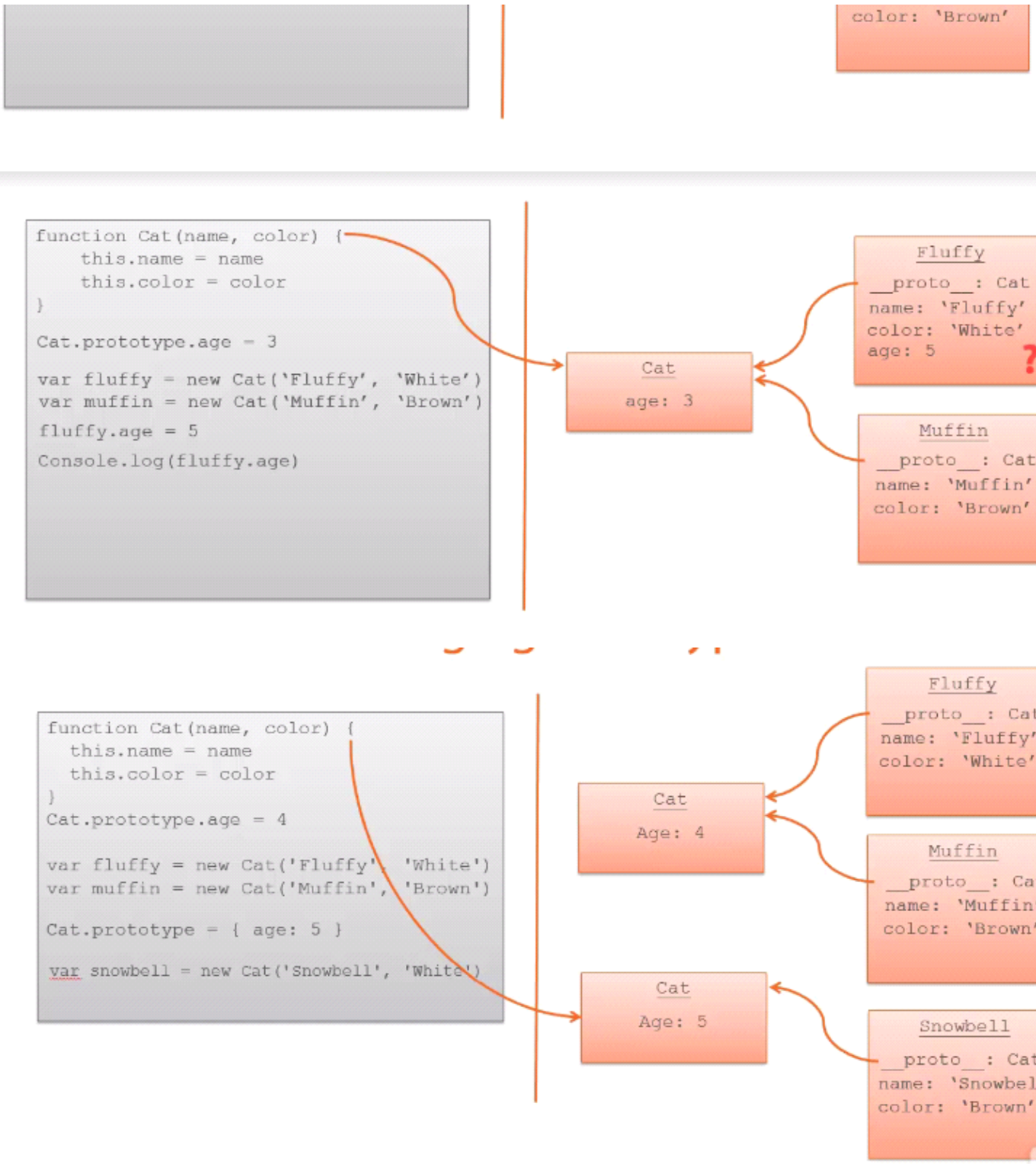
```
'use strict'
var cat =['a','b','c'];

console.log(cat.__proto__);

var funk = function(){
  return "hello";
}

console.log(funk.prototype);
```




color: 'Brown'
function Cat(name, color) {
this.name = name
this.color = color
}
Cat.prototype.age = 3
var fluffy = new Cat('Fluffy', 'White')
var muffin = new Cat('Muffin', 'Brown')
fluffy.age = 5
Console.log(fluffy.age)
Cat
age: 3
Fluffy
__proto__: Cat
name: 'Fluffy'
color: 'White'
age: 5
?
Muffin
__proto__: Cat
name: 'Muffin'
color: 'Brown'
function Cat(name, color) {
this.name = name
this.color = color
}
Cat.prototype.age = 4
var fluffy = new Cat('Fluffy', 'White')
var muffin = new Cat('Muffin', 'Brown')
Cat.prototype = { age: 5 }
var snowbell = new Cat('Snowbell', 'White')
Cat
Age: 4
Cat
Age: 5
Fluffy
__proto__: Cat
name: 'Fluffy'
color: 'White'
Muffin
__proto__: Cat
name: 'Muffin'
color: 'Brown'
Snowbell
__proto__: Cat
name: 'Snowbell'
color: 'Brown'

```
'use strict';

function Cat(name, color) {
  this.name = name
  this.color = color
}
Cat.prototype.age = 4

var fluffy = new Cat('Fluffy', 'White')

display(fluffy.__proto__)
display(fluffy.__proto__.__proto__)
display(fluffy.__proto__.__proto__.__proto__)
```

```
Cat {
  age: 4
}

Object {
}

null
```

```
'use strict'

function animal(){
 this.type='pet';
}

 animal.prototype.speak = function(){
  return 'Grunt';
 }

function cat(name,color)
{
 animal.call(this);
 this.name = { first:name, last:'Kat'};
 this.color = color;
}

cat.prototype = Object.create(animal.prototype);

var fluffy = new cat('Kit','Black');

console.log(fluffy);
```

Design Patterns

Saturday, January 28, 2017
6:45 PM

Creational Patterns	
Abstract Factory	Creates an instance of several families of classes (encapsulate)
Builder	Separates object construction from its representation
Factory Method	Creates an instance of several derived classes
Prototype	A fully initialized instance to be copied or cloned
Singleton	A class of which only a single instance can exist

Structural Patterns	
Adapter	Match interfaces of different classes
Bridge	Separates an object's interface from its implementation
Composite	A tree structure of simple and composite objects
Decorator	Add responsibilities to objects dynamically
Facade	A single class that represents an entire subsystem
Flyweight	A fine-grained instance used for efficient sharing
Proxy	An object representing another object

Behavioral Patterns	
Chain of Resp.	A way of passing a request between a chain of objects
Command	Encapsulate a command request as an object
Interpreter	A way to include language elements in a program
Iterator	Sequentially access the elements of a collection
Mediator	Defines simplified communication between classes
Memento	Capture and restore an object's internal state
Observer	A way of notifying change to a number of classes
State	Alter an object's behavior when its state changes
Strategy	Encapsulates an algorithm inside a class
Template Method	Defer the exact steps of an algorithm to a subclass
Visitor	Defines a new operation to a class without change

Inheritance

var nextobj = Object.create(parentObj);

Objects in Java Scripts:

var obj ={};

var nextobj = Object.create(Object.prototype); //inheritance

var lastobj = new Object();

```
var obj = {};
obj.param = 'new value';
console.log(obj.param);
```

```
var obj ={}
var value='param;
Obj['param']='new value';
Console.log(obj[value] (or) obj.param);
```

=========

defineProperty

```
Object.defineProperty(obj, 'name', {
value: 'my name',
writable: true,
enumerable: true,
configurable: true
})
```

==========

Create Design Patterns
Used to construct new object
Adapting creation to the situation

Constructor Patterns - DP

Wednesday, February 01, 2017
5:02 PM

1) Constructor Patterns

Use to create new objects with their own scope

When a 'new' keyword is used
- Creates a brand new object
- Links to an object prototype
- Binds 'this' to the new object scope
- Implicitly returns this

Node

task.js

```
var Task = function(name)
{
 this.name=name;
 this.completed = false;

};

 Task.prototype.save = function(){
  console.log('saving Task' + this.name);
 };

 Task.prototype.update = function(){
  console.log( 'update ' + this.name);
 };

Task.prototype.update = function(){
 console.log( 'Proto update ' + this.name);
};

module.exports = Task;
```

main.js

======

```
var Task = require('./task');

var task1 = new Task('Constructor');
var task2 = new Task('modules');
var task3 = new Task('singleton');
var task4 = new Task('prototype');

task1.complete();
```

```
task2.update();
task3.save();
task4.save();
```

Module Pattern DP

Wednesday, February 01, 2017

Sample way to Encapsulate a methods

taskrepo.js

```
var repo = (function() {
   var sum = 0;
    var get = function() {
        console.log('from Get function');
          sum = sum +1;
        return sum;
    };
   var save= function() {
         console.log('save function');
         sum = sum -1 ;
         return sum;
       };

   var update= function() {
         console.log('update function');
         sum = 0;
         return sum;
       };

   return {
      get:get,
      save:save,
      update:update
   }

}());

module.exports=repo;
```

Main.js

```
var Repo = require('./taskrepo');

Repo.get();
Repo.save();
Repo.update();
```

Factory Pattern

Thursday, February 02, 2017
4:14 PM

A pattern used to simplify object creation.

Simplifies object creation
Creating different object based on the need.
Repository Creation

Structural design Pattern

Monday, February 06, 2017
11:45 AM

Concerned with how objects are made up and simplify relationships between objects.

1) Extend functionality
2) Simplify functionality

Decorator Pattern

Used to add new functionality to an existing object, without being obtrusive.

More complete inheritance
Wrap an object
Protects existing object
Allows extended functionality

Façade Pattern

Used to provide a simplified interface to a complicated system.

Think about the front of a building
Façade hides the chaos from us
Simplifies the interface
E.g. Jquery

Flyweight Pattern

Flyweight Pattern shares data across objects. Resulting in a small memory footprint. But only if you have large number of the objects. Else it has its own overheads.

Behavioural Design Pattern

Wednesday, February 08, 2017
2:38 PM

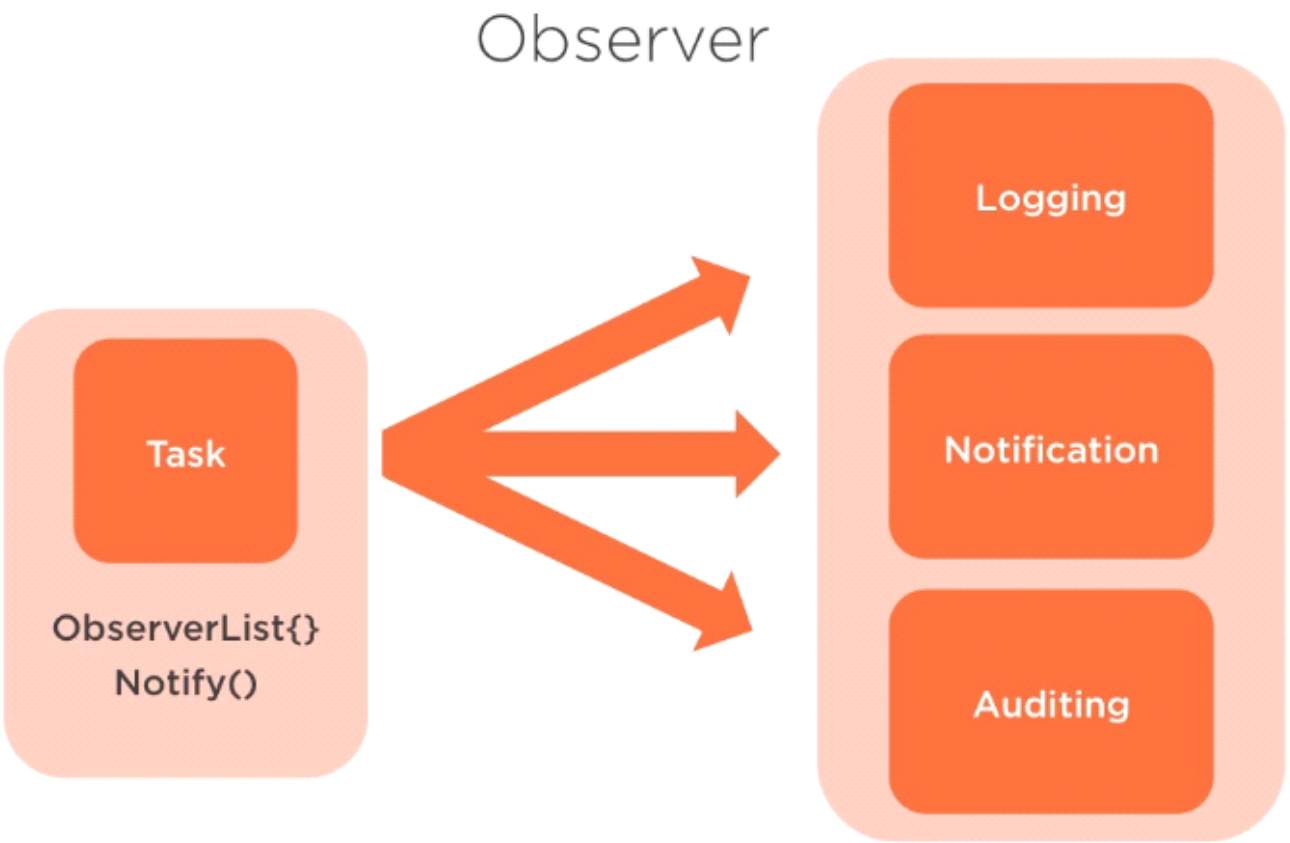
Concerned with the assignment of responsibilities between objects and how they communicate.

Deals with the responsibilities of objects
Help objects cooperate
Assigns clear hierarchy
Can encapsulate requests

Observer Pattern

Allows for loosely coupled system
One object is the focal point
Group of objects watch for changes

- Observers
- Subject
- Notification

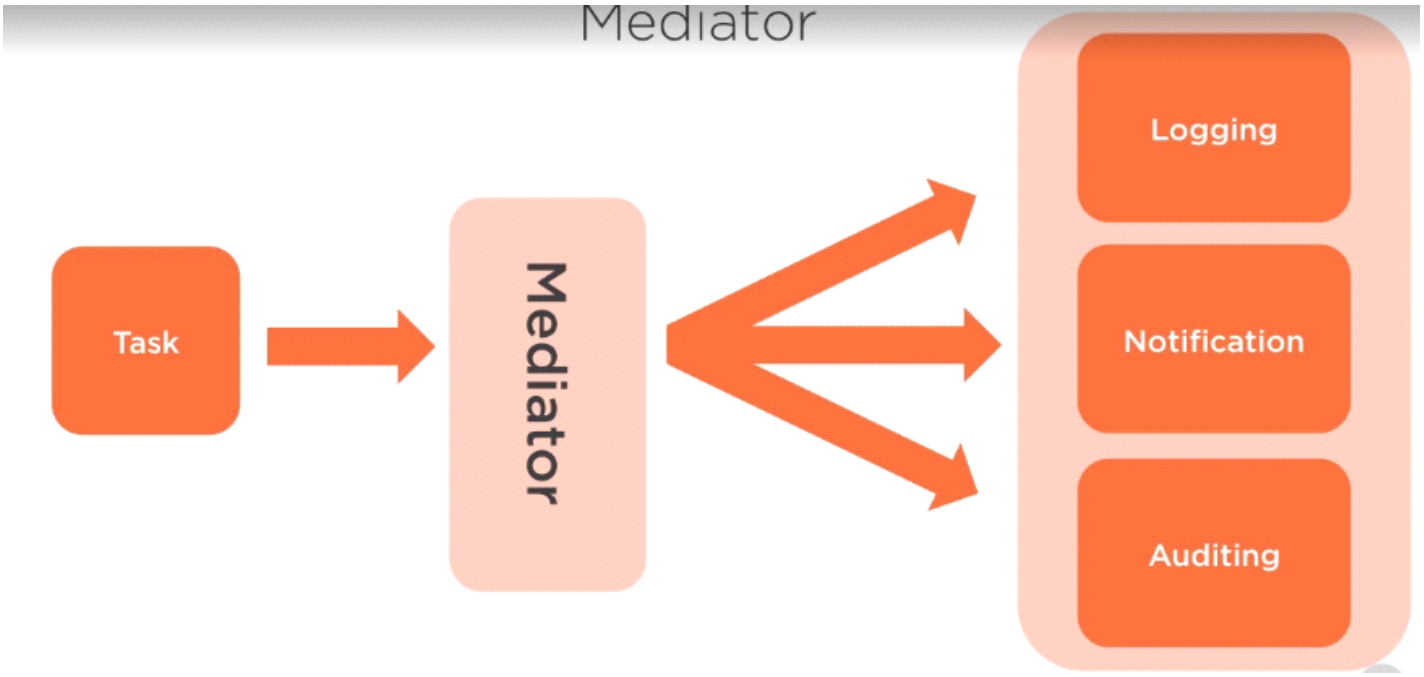


Mediator Pattern

Thursday, February 09, 2017
3:18 PM

Controls communication between objects so neither object has to be coupled to the others.

Allows for loosely coupled system
One object managers all communication
Many to many relationship



Command Pattern

Thursday, February 09, 2017
4:17 PM

Encapsulates the calling of a method as an object

Fully decouples the execution from the implemenation.

Allows for less fragile implementations
Supports undo operations
Supports auditing and logging of operations