**DESIGN PATTERNS**

***Categories:***

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1. **Creational** | | | | Based on the concept of creating an object. | | | |
| ***Class*** | | | | | | | |
| *Factory Method* | | | | | This makes an instance of several derived classes based on interfaced data or events. | | |
| ***Object*** | | | | | | | |
| *Abstract Factory* | | | | | Creates an instance of several families of classes without detailing concrete classes. | | |
| *Builder* | | | | | Separates object construction from its representation, always creates the same type of object. | | |
| *Prototype* | | | | | A fully initialized instance used for copying or cloning. | | |
| *Singleton* | | | | | A class with only a single instance with global access points. | | |
|  |  |  |  |  |  |  |  |
| 1. **Structural** | | | | Based on the idea of building blocks of objects. | | | |
| ***Class*** | | | | | | | |
| *Adapter* | | | | | Match interfaces of different classes therefore classes can work together despite incompatible interfaces. | | |
| ***Object*** | | | | | | | |
| *Adapter* | | | | | Match interfaces of different classes therefore classes can work together despite incompatible interfaces. | | |
| *Bridge* | | | | | Separates an object's interface from its implementation so the two can vary independently. | | |
| *Composite* | | | | | A structure of simple and composite objects which makes the total object more than just the sum of its parts. | | |
| *Decorator* | | | | | Dynamically add alternate processing to objects. | | |
| *Facade* | | | | | A single class that hides the complexity of an entire subsystem. | | |
| *Flyweight* | | | | | A fine-grained instance used for efficient sharing of information that is contained elsewhere. | | |
| *Proxy* | | | | | A place holder object representing the true object. | | |
|  | | | | | | | |
| 1. **Behavioral** | | | | Based on the way objects play and work together. | | | |
| ***Class*** | | | | | | | |
| *Interpreter* | | | | | A way to include language elements in an application to match the grammar of the intended language. | | |
| *Template         Method* | | | | | Creates the shell of an algorithm in a method, then defer the exact steps to a subclass. | | |
| ***Object*** | | | | | | | |
| *Chain of        Responsibility* | | | | | A way of passing a request between a chain of objects to find the object that can handle the request. | | |
| *Command* | | | | | Encapsulate a command request as an object to enable, logging and/or queuing of requests, and provides error-handling for unhandled requests. | | |
| *Iterator* | | | | | Sequentially access the elements of a collection without knowing the inner workings of the collection. | | |
| *Mediator* | | | | | Defines simplified communication between classes to prevent a group of classes from referring explicitly to each other. | | |
| *Memento* | | | | | Capture an object's internal state to be able to restore it later. | | |
| *Observer* | | | | | A way of notifying change to a number of classes to ensure consistency between the classes. | | |
| *State* | | | | | Alter an object's behavior when its state changes. | | |
| *Strategy* | | | | | Encapsulates an algorithm inside a class separating the selection from the implementation. | | |
| *Visitor* | | | | | Adds a new operation to a class without changing the class. | | |

The Constructor Pattern

**Object Creation**

### var newObject = {};

### var newObject = Object.create( Object.prototype );

### var newObject = new Object();

**Assigning values**

**1. Dot syntax**

*// Set properties*

newObject.someKey = "Hello World";

*// Get properties*

var value = newObject.someKey;

**2. Square bracket syntax**

// Set properties

newObject["someKey"] = "Hello World";

// Get properties

var value = newObject["someKey"];

**3. Object.defineProperty**

var defineProp = function ( obj, key, value ){

  var config = {

    value: value,

    writable: true,

    enumerable: true,

    configurable: true

  };

  Object.defineProperty( obj, key, config );

};

// To use, we then create a new empty "person" object

var person = Object.create( Object.prototype );

// Populate the object with properties

defineProp( person, "car", "Delorean" );

defineProp( person, "dateOfBirth", "1981" );

defineProp( person, "hasBeard", false );

console.log(person);

// Outputs: Object {car: "Delorean", dateOfBirth: "1981", hasBeard: false}

**4. Object.defineProperties**

// Set properties

Object.defineProperties( newObject, {

  "someKey": {

    value: "Hello World",

    writable: true

  },

  "anotherKey": {

    value: "Foo bar",

    writable: false

  }

});

### **The Module Pattern**

Module pattern is used to further emulate the concept of classes in such a way that we're able to include both public/private methods and variables inside a single object, thus shielding particular parts from the global scope.

***Example 1:***

### var testModule = (function () {

### var counter = 0;

### return {

### incrementCounter: function () {

### return counter++;

### },

### resetCounter: function () {

### console.log( "counter value prior to reset: " + counter );

### counter = 0;

### }

### };

### })();

Now, *incrementCounter, resetCounter* are accessible but counter is not.

**Example 2:**

var basketModule = (function () {

  // privates

  var basket = [];

  function doSomethingPrivate() {

    //...

  }

  function doSomethingElsePrivate() {

    //...

  }

  // Return an object exposed to the public

  return {

    addItem: function( values ) {

      basket.push(values);

    },

    // Get the count of items in the basket

    getItemCount: function () {

      return basket.length;

    },

    // Public alias to a private function

    doSomething: doSomethingPrivate,

    // Get the total value of items in the basket

    getTotal: function () {

      var q = this.getItemCount(),

          p = 0;

      while (q--) {

        p += basket[q].price;

      }

      return p;

    }

  };

})();

#### **Disadvantage**: Changing the visibility not easy

## The Revealing Module Pattern

## Updated module pattern where we would simply define all of our functions and variables in the private scope and return an anonymous object with pointers to the private functionality we wished to reveal as public.

**Example:**

var myRevealingModule = (function () {

        var privateVar = "Ben Cherry",

        publicVar = "Hey there!";

        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,

            greeting: publicVar,

            getName: publicGetName

        };

    })();

myRevealingModule.setName( "Paul Kinlan" );

## The Singleton Pattern

## The Singleton pattern restricts instantiation of a class to a single object. Classically, the Singleton pattern can be implemented by creating a class with a method that creates a new instance of the class if one doesn't exist. In the event of an instance already existing, it simply returns a reference to that object.

## Example:

var SingletonTester = (function () {

  function Singleton( options ) {

    options = options || {};

    // set some properties for our singleton

    this.name = "SingletonTester";

    this.pointX = options.pointX || 6;

    this.pointY = options.pointY || 10;

  }

  // our instance holder

  var instance;

  // an emulation of static variables and methods

  var \_static = {

    name: "SingletonTester",

    getInstance: function( options ) {

      if( instance === undefined ) {

        instance = new Singleton( options );

      }

      return instance;

    }

  };

  return \_static;

})();

var singletonTest = SingletonTester.getInstance({

  pointX: 5

});

// Log the output of pointX just to verify it is correct

// Outputs: 5

console.log( singletonTest.pointX );