Table of Contents

js-handbook

Repo for book https://www.gitbook.com

Design patterns

JavaScript Design Patterns

Resources:

- Addy Osmani's book in GitHub
- http://www.leanpanda.com/blog/2015/06/28/amd-requirejs-commonjs-browserify/

Anti-Pattern

is a bad design that is worthy of documenting

Categories Of Design Pattern

Creational Design Patterns

focus on handling object creation mechanisms where objects are created in a manner suitable for the situation we're working in. Patterns: Constructor, Factory, Abstract, Prototype, Singleton and Builder.

Structural Design Patterns

are concerned with object composition and typically identify simple ways to realize relationships between different objects. Patterns: Decorator, Facade, Flyweight, Adapter and Proxy

Behavioral Design Patterns

focus on improving or streamlining the communication between disparate objects in a system. Patterns: Iterator, Mediator, Observer and Visitor.

Patterns

Pattern	Description
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.
-	-
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.
-	-
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

```
function Car( model, year, miles ) {
  this.model = model;
}

// Note here that we are using Object.prototype.newMethod rather than
// Object.prototype so as to avoid redefining the prototype object
Car.prototype.toString = function () {
  return "Model is: " + this.model;
};

// Usage:
var civic = new Car("Honda Civic");
```

The Module Pattern

Modules

- The Module pattern
- Object literal notation

- AMD modules
- CommonJS modules
- ES6 modules

Object Literals

```
var myModule = {
  myConfig: {
    useCaching: true,
    language: "en"
  },

  saySomething: function () {
    console.log( "Where in the world is Paul Irish today?" );
  }
};

// usage
myModule.saySomething();
```

The Module Pattern

```
var testModule = (function () {
  var private = 0;
  function privateMethod() {/* ... */}
  return {
    public: function () {
      return counter++;
    }
  };
})();

// Usage:
testModule.public();
```

The Singleton Pattern

```
var mySingleton = (function () {
  // Instance stores a reference to the Singleton
  var instance;

  function init() {
    // Private methods and variables
    function privateMethod(){
        console.log( "I am private" );
    }
    return {
      // Public methods and variables
      publicMethod: function () {
        console.log( "The public can see me!" );
      },
      publicProperty: "I am also public"

    };
  };

  return {
    // Get the Singleton instance if one exists
    // or create one if it doesn't
    getInstance: function () {
      if ( !instance ) {
        instance = init();
      }
      return instance;
    }
  };
})();

// Usage:
var singleA = mySingleton.getInstance();
var singleB = mySingleton.getInstance();
console.log( singleA.getRandomNumber() === singleB.getRandomNumber() ); // true
```

The Observer Pattern. Ref

The Observer is a design pattern where an object (known as a subject) maintains a list of objects depending on it (observers), automatically notifying them of any changes to state.

When a subject needs to notify observers about something interesting happening, it broadcasts a notification to the observers (which can include specific data related to the topic of the notification).

When we no longer wish for a particular observer to be notified of changes by the subject they are registered with, the subject can remove them from the list of observers.

Differences Between The Observer And Publish/Subscribe Pattern

The Observer pattern requires that the observer (or object) wishing to receive topic notifications must subscribe this interest to the object firing the event (the subject).

The Publish/Subscribe pattern however uses a topic/event channel which sits between the objects wishing to receive notifications (subscribers) and the object firing the event (the publisher). This event system allows code to define application specific events which can pass custom arguments containing values needed by the subscriber. The idea here is to avoid dependencies between the subscriber and publisher.

This differs from the Observer pattern as it allows any subscriber implementing an appropriate event handler to register for and receive topic notifications broadcast by the publisher.

```
// Publish
// jQuery: $(obj).trigger("channel", [arg1, arg2, arg3]);
$( el ).trigger( "/login", [{username:"test", userData:"test"}] );

// Subscribe
// jQuery: $(obj).on( "channel", [data], fn );
$( el ).on( "/login", function( event ){...} );

// Unsubscribe
// jQuery: $(obj).off( "channel" );
$( el ).off( "/login" );
```

The Mediator Pattern

A real-world analogy could be a typical airport traffic control system. A tower (Mediator) handles what planes can take off and land because all communications (notifications being listened out for or broadcast) are done from the planes to the control tower, rather than from plane-to-plane. A centralized controller is key to the success of this system and that's really the role a Mediator plays in software design.

A Mediator is an object that coordinates interactions (logic and behavior) between multiple objects. It makes decisions on when to call which objects, based on the actions (or inaction) of other objects and input.

Events

Both the event aggregator and mediator use events, in the above examples.

Mediator Vs. Facade

The Mediator centralizes communication between modules where it's explicitly referenced by these modules. In a sense this is multidirectional. The Facade however just defines a simpler interface to a module or system but doesn't add any additional functionality. Other modules in the system aren't directly aware of the concept of a facade and could be considered unidirectional.

The Prototype Pattern

The GoF refer to the prototype pattern as one which creates objects based on a template of an existing object through cloning.

We can think of the prototype pattern as being based on prototypal inheritance where we create objects which act as prototypes for other objects. The prototype object itself is effectively used as a blueprint for each object the constructor creates. If the prototype of the constructor function used contains a property called name for example (as per the code sample lower down), then each object created by that same constructor will also have this same property.

```
var vehicle = {
  getModel: function () {
    console.log( "The model of this vehicle is.." + this.model );
  }
};

var car = Object.create(vehicle, {
  "id": {
    value: MY_GLOBAL.nextId(),
    // writable:false, configurable:false by default
    enumerable: true
  },
  "model": {
    value: "Ford",
    enumerable: true
  }
});
```

The Command Pattern, ref

The Command pattern aims to encapsulate method invocation, requests or operations into a single object and gives us the ability to both parameterize and pass method calls around that can be executed at our discretion. In addition, it enables us to decouple objects invoking the

action from the objects which implement them, giving us a greater degree of overall flexibility in swapping out concrete classes (objects).

The Facade Pattern

When we put up a facade, we present an outward appearance to the world which may conceal a very different reality. This was the inspiration for the name behind the next pattern we're going to review - the Facade pattern. This pattern provides a convenient higher-level interface to a larger body of code, hiding its true underlying complexity. Think of it as simplifying the API being presented to other developers, something which almost always improves usability.

```
var addMyEvent = function( el,ev,fn ){
   if( el.addEventListener ){
            el.addEventListener( ev,fn, false );
      }else if(el.attachEvent){
            el.attachEvent( "on" + ev, fn );
      } else{
           el["on" + ev] = fn;
    }
};
```

The Factory Pattern

The Factory pattern is another creational pattern concerned with the notion of creating objects. Where it differs from the other patterns in its category is that it doesn't explicitly require us use a constructor. Instead, a Factory can provide a generic interface for creating objects, where we can specify the type of factory object we wish to be created.

The Mixin Pattern

Sub-classing

```
// a new instance of Person can then easily be created as follows:
var clark = new Person( "Clark", "Kent" );

// Define a subclass constructor for for "Superhero":
var Superhero = function( firstName, lastName, powers ){

    // Invoke the superclass constructor on the new object
    // then use .call() to invoke the constructor as a method of
    // the object to be initialized.

    Person.call( this, firstName, lastName );

    // Finally, store their powers, a new array of traits not found in a normal "Perso
n"
    this.powers = powers;
};

Superhero.prototype = Object.create( Person.prototype );
var superman = new Superhero( "Clark", "Kent", ["flight","heat-vision"] );
console.log( superman );

// Outputs Person attributes as well as powers
```

Mixins

```
// Extend both constructors with our Mixin
_.extend( CarAnimator.prototype, myMixins );
_.extend( PersonAnimator.prototype, myMixins );
```

The Decorator Pattern

Classically, Decorators offered the ability to add behaviour to existing classes in a system dynamically. The idea was that the decoration itself wasn't essential to the base functionality of the class, otherwise it would be baked into the superclass itself.

They can be used to modify existing systems where we wish to add additional features to objects without the need to heavily modify the underlying code using them. A common reason why developers use them is their applications may contain features requiring a large quantity of distinct types of object.

```
// A vehicle constructor
function Vehicle( vehicleType ){
    // some sane defaults
    this.vehicleType = vehicleType || "car";
    this.model = "default";
    this.license = "00000-000";
}

// Lets create a new instance of vehicle, to be decorated
var truck = new Vehicle( "truck" );

// New functionality we're decorating vehicle with
truck.setModel = function( modelName ){
    this.model = modelName;
};

truck.setColor = function( color ){
    this.color = color;
};
```

JavaScript MV* Patterns

MVC

MVC is an architectural design pattern that encourages improved application organization through a separation of concerns. It enforces the isolation of business data (Models) from user interfaces (Views), with a third component (Controllers) traditionally managing logic and user-input.

Models

Models manage the data for an application. They are concerned with neither the user-interface nor presentation layers but instead represent unique forms of data that an application may require. When a model changes (e.g when it is updated), it will typically notify its observers (e.g views, a concept we will cover shortly) that a change has occurred so that they may react accordingly.

Views

Views are a visual representation of models that present a filtered view of their current state. A view typically observes a model and is notified when the model changes, allowing the view to update itself accordingly. Design pattern literature commonly refers to views as "dumb" given that their knowledge of models and controllers in an application is limited.

Controllers

Controllers are an intermediary between models and views which are classically responsible for updating the model when the user manipulates the view.

MVP

Model-view-presenter (MVP) is a derivative of the MVC design pattern which focuses on improving presentation logic.

The P in MVP stands for presenter. It's a component which contains the user-interface business logic for the view. Unlike MVC, invocations from the view are delegated to the presenter, which are decoupled from the view and instead talk to it through an interface. This allows for all kinds of useful things such as being able to mock views in unit tests.

MVVM

MVVM (Model View ViewModel) is an architectural pattern based on MVC and MVP, which attempts to more clearly separate the development of user-interfaces (UI) from that of the business logic and behavior in an application. To this end, many implementations of this pattern make use of declarative data bindings to allow a separation of work on Views from other layers.

The ViewModel can be considered a specialized Controller that acts as a data converter. It changes Model information into View information, passing commands from the View to the Model.

Modern Modular JavaScript Design Patterns

AMD

The overall goal for the AMD (Asynchronous Module Definition) format is to provide a solution for modular JavaScript that developers can use today. The AMD module format itself is a proposal for defining modules where both the module and dependencies can be asynchronously loaded.

Loading AMD Modules Using RequireJS

```
require(["app/myModule"],
    function( myModule ){
        // start the main module which in-turn
        // loads other modules
        var module = new myModule();
        module.doStuff();
});
```

CommonJS

A Module Format Optimized For The Server. The CommonJS module proposal specifies a simple API for declaring modules server-side and unlike AMD attempts to cover a broader set of concerns such as io, file-system, promises and more.

```
// package/lib is a dependency we require
var lib = require( "package/lib" );

// behaviour for our module
function foo(){
    lib.log( "hello world!" );
}

// export (expose) foo to other modules
exports.foo = foo;
```

AMD-equivalent Of The First CommonJS Example

```
define(function(require){
   var lib = require( "package/lib" );

    // some behaviour for our module
    function foo(){
        lib.log( "hello world!" );
    }

    // export (expose) foo for other modules
    return {
        foobar: foo
    };
});
```

UMD

For developers wishing to create modules that can work in both browser and server-side environments, existing solutions could be considered little lacking. To help alleviate this, James Burke, I and a number of other developers created UMD (Universal Module Definition) https://github.com/umdjs/umd.

ES6 Modules

tbd

Namespacing Patterns, ref

maybe, one day, later...

Short guide about principles of OOP in JS

Types

Primitive Types

- Boolean
- Number
- String
- Null
- Undefined

Identifying Primitives:

```
typeof "text"; // "string"
typeof undefined; // "undefined"

var n = 2;
n.constructor.name; // "Number"
n.constructor === Number; // true

typeof null; // "object" - an error in JS
// to check if null use
value === null
```

Reference Types

- Array
- Date
- Error
- Function
- Object
- RexExp

Dereferencing objects:

```
var o = new Object();
o = null; // dereference - let garbage collectot free up memory
```

Identifying References:

```
typeof myFun; // "function"
myFun instanceof Function; // true
listArr instanceof Array; // true
listArr instanceof Object; // true because Array was inherited from Object
Array.isArray(listArr); // true
```

Functions

arguments

```
Array.isArray(arguments); // false

function myFn(par1) {
    arguments;
}

myFn.length; // 1 - function's arity - number of params that it's expecting
```

```
var fn = function() {
    console.log(arguments.constructor.name); // Object
    var argsArr = Array.prototype.slice.call(arguments);
    console.log(argsArr.constructor.name); // Array
}
```

Objects

Detecting properties

```
var o = {prop: false};

if (o.prop) {}; // incorrect
"prop" in o; // true, check both for own and prototype properties
o.hasOwnProperty("someMethod"); // checks only for own methods
```

Types of Properties - accessors:

```
var person = {
  _name: "Jack",

  get name() {
    console.log("Getting name");
    return this._name;
  },

  set name(value) {
    console.log("Setting name to %s", value);
    this._name = value;
  }
};

person.name
// Getting name
//"Jack"

person.name = "John"
// Setting name to John
// "John"
```

multiple properties:

```
var person = {};
Object.defineProperties(person, {
    _name: {
        value: "Jack",
        enumerable: true,
        configurable: true,
        writable: true
    },

    name: {
        get: function() {},
        set: function() {},
        enumerable: true,
        configurable: true
    }
});
```

Preventing Obj Modification

```
var person = { name: "Jack "};

// preventExtensions
Object.preventExtensions(person);
Object.isExtensible(person); //false
person.lastName = "Smith";
person; // Object {name: "Jack "}

// seal
Object.seal(person); // nonextendable, nonconfigurable
delete person.name
person; // Object {name: "Jack "}
person.name = 'John'
person; // Object {name: "John"}

// freeze
Object.freeze(person); // snapshot of object, cannot be changed anything
```

Constructors and Prototypes

Prototypes and Constructors

```
function Person(name) {
    this.name = name;
}

// 1. in case of overwriting prototype
Person.prototype = {
    // 2. important to keep connection
    constructor: Person,

    sayName: function() {},
    toString: function() {}
};

var p1 = new Person("Jack");
p1 instanceOf Person; // true
p1.constructor === Person
```

Inheritance

Methods from `Object.prototype`

```
valueOf(); // gets called whenever an operator is used on an object
var now = new Date(),
    earlier = new Date(2010, 1, 1);
now > earlier; // true, because 'getTime()' method was used as 'valueOf()'

toString(); // is called whenever valueOf() returns a reference value instead of primi
tive
```

Object Inheritance

`Object.create()`

```
var person1 = {
  name: "Jack",
  sayHi: function() { console.log("Hi, " + this.name); }
}

var person2 = Object.create(person1, {
  name: 'John'
});
var person2 = Object.create(person1, {
  name: {
    value: "John"
  }
});

person2.sayHi(); // Hi, John
person1.isPrototypeOf(person2); // true
```

Constructor Inheritance

```
// parent
function Parent() { this.p = 'parent'; }
Parent.prototype.parentMethod = function() { return 'parent method'; }

// child
function Child() { this.c = 'child'; }
Child.prototype.childMethod = function() { return 'child method'; }

// inheritance
Child.prototype = new Parent();
Child.prototype.constructor = Child;

// inherited object
var c = new Child();
// Child {c: "child", p: "parent", constructor: function, parentMethod: function}

// child prototype was overwritten and childMethod dissapeared

Child.prototype.constructor.name; // "Child"
```

similar with `Object.create()`

```
Child.prototype = Object.create(Parent.prototype, {
  constructor: {
    value: Child
  }
});

var c = new Child();
// Child {c: "child", parentMethod: function}
// only prototype was inherited - parent's p was not
```

Stealing constructor - `call` and `apply`

```
// parent
function Parent() { this.p = 'parent'; }
Parent.prototype.parentMethod = function() { return 'parent method'; }

// child
function Child() { this.c = 'child'; Parent.call(this) }

// inheritance
Child.prototype = new Parent();
Child.prototype.constructor = Child;

var c = new Child();
// Child {c: "child", p: "parent", constructor: function, parentMethod: function}
// c has own values from Parent
```

`getPrototypeOf`

```
var personPrototype = {name: ''};
var ben = Object.create(personPrototype); // inherited obj

Object.getPrototypeOf(ben); // personPrototype
personPrototype.isPrototypeOf(ben); // true
```

Classical vs Prototypal Inheritance

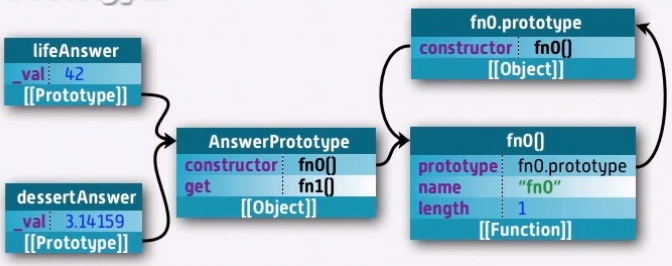
letscodejavascript.com

```
var AnswerPrototype = {
  constructor: function fn0(value) {
    this._val = value;
  },
  get: function fn1() {
    return this._val;
  }
};

var lifeAnswer = Object.create(AnswerPrototype);
lifeAnswer.constructor(42);
lifeAnswer.get();  // →42

var dessertAnswer = Object.create(AnswerPrototy…
dessertAnswer.constructor(3.14159);
dessertAnswer.get();  // →3.14159
```

Prototypal



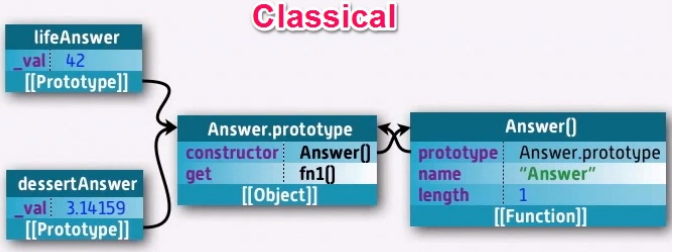

```
function Answer(value) {
  this._val = value;
}

Answer.prototype.get = function fn1() {
  return this._val;
};

var lifeAnswer = new Answer(42);
lifeAnswer.get();  // →42

var dessertAnswer = new Answer(3.14159);
dessertAnswer.get();  // →3.14159
```

Classical



letscodejavascript.com

Prototypal Model

```
var AnswerPrototype = {
  constructor: function fn0(value) {
    this._val = value;
  },
  get: function fn1() {
    return this._val;
  }
};

var lifeAnswer = Object.create(AnswerPrototype);
lifeAnswer.constructor(42);
lifeAnswer.get();  // →42

var dessertAnswer = Object.create(AnswerPrototype);
dessertAnswer.constructor(3.14159);
dessertAnswer.get();  // →3.14159


var FirmAnswerPrototype = ↵
                    Object.create(AnswerPrototype);

FirmAnswerPrototype.get = function fn2() {
  return AnswerPrototype.get.call(this) + "!!";
};

var luckyAnswer = Object.create(FirmAnswerPrototype);
luckyAnswer.constructor(7);
luckyAnswer.get();  // →"7!!"

var magicAnswer = Object.create(FirmAnswerPrototype);
magicAnswer.constructor(3);
magicAnswer.get();  // →"3!!"
```

Classical Model

```
function Answer(value) {
  this._val = value;
}
Answer.prototype.get = function fn1() {
  return this._val;
};

var lifeAnswer = new Answer(42);
lifeAnswer.get();  // →42

var dessertAnswer = new Answer(3.14159);
dessertAnswer.get();  // →3.14159


function FirmAnswer(value) {
  Answer.call(this, value);
}
FirmAnswer.prototype = ↵
                Object.create(Answer.prototype);
FirmAnswer.prototype.constructor = FirmAnswer;

FirmAnswer.prototype.get = function fn2() {
  return Answer.prototype.get.call(this) + "!!";
};

var luckyAnswer = new FirmAnswer(7);
luckyAnswer.get();  // →"7!!"

var magicAnswer = new FirmAnswer(3);
magicAnswer.get();  // →"3!!"
```

letscodejavascript.com

Classical Model

```
function Answer(value) {
  this._val = value;
}
Answer.prototype.get = function fn1() {
  return this._val;
};

var lifeAnswer = new Answer(42);
lifeAnswer.get();  // →42

var dessertAnswer = new Answer(3.14159);
dessertAnswer.get();  // →3.14159


function FirmAnswer(value) {
  Answer.call(this, value);
}
FirmAnswer.prototype = ↵
                Object.create(Answer.prototype);
FirmAnswer.prototype.constructor = FirmAnswer;

FirmAnswer.prototype.get = function fn2() {
  return Answer.prototype.get.call(this) + "!!";
};
```

ES6 Syntax

```
class Answer {
  constructor(value) {
    this._val = value;
  }

  get() {
    return this._val;
  }
}

var lifeAnswer = new Answer(42);
lifeAnswer.get();  // →42

var dessertAnswer = new Answer(3.14159);
dessertAnswer.get();  // →3.14159


class FirmAnswer extends Answer {
  constructor(value) {
    super(value);
  }

  get() {
    return super() + "!!";
  }
}
```

Resources

1. http://www.objectplayground.com/ (https://www.youtube.com/watch?v=PMfcsYzj-9M)
2. Principles of Object-Oriented JavaScript by Nicholas C. Zakas (http://www.nostarch.com/oojs)

Closures and Event Listeners

Resource:

- https://www.udacity.com/course/javascript-design-patterns--ud989

The problem:

Let's say we're making an element for every item in an array. When each is clicked, it should alert its number. The simple approach would be to use a for loop to iterate over the list elements, and when the click happens, alert the value of `num` as we iterate over each item of the array. Here's an example:

```
// clear the screen for testing
document.body.innerHTML = '';
document.body.style.background="white";

var nums = [1,2,3];

// Let's loop over the numbers in our array
for (var i = 0; i < nums.length; i++) {
    // This is the number we're on...
    var num = nums[i];
    // We're creating a DOM element for the number
    var elem = document.createElement('div');
    elem.textContent = num;
    // ... and when we click, alert the value of `num`
    elem.addEventListener('click', function() {
        alert(num);
    });
    // finally, let's add this element to the document
    document.body.appendChild(elem);
};
```

If you run this code on any website, it will clear everything and add a bunch of numbers to the page. Try it! Open a new page, open the console, and run the above code. Then click on the numbers and see what gets alerted. Reading the code, we'd expect the numbers to alert their values when we click on them.

But when we test it, all the elements alert the same thing: the last number. But why?

What's actually happening

Let's cut out the irrelevant code so we can see what's going on. The comments below have changed, and explain what is actually happening.

```
var nums = [1, 2, 3];
for (var i = 0; i < nums.length; i++) {
    // This variable keeps changing every time we iterate!
    //  It's first value is 1, then 2, then finally 3.
    var num = nums[i];
    // On click...
    elem.addEventListener('click', function () {
        // ... alert num's value at the moment of the click!
        alert(num);
        // Specifically, we're alerting the num variable
        // that's defined outside of this inner function.
        // Each of these inner functions are pointing to the
        // same `num` variable... the one that changes on
        // each iteration, and which equals 3 at the end of
        // the for loop.  Whenever the anonymous function is
        // called on the click event, the function will
        //  reference the same `num` (which now equals 3).
    });
};
```

That's why regardless of which number we click on, they all alert the last value of `num`.

How do we fix it?

The solution involves utilizing closures. We're going to create an inner scope to hold the value of `num` at the exact moment we add the event listener. There are a number of ways to do this -- here's a good one.

Let's simplify the code to just the lines where we add the event listener.

```
var num = nums[i];
elem.addEventListener('click', function() {
    alert(num);
});
```

The `num` variable changes, so we have to somehow connect it to our event listener function. Here's one way of doing it. First take a look at this code, then I'll explain how it works.

```
elem.addEventListener('click', (function(numCopy) {
    return function() {
        alert(numCopy)
    };
})(num));
```

The bolded part is the outer function. We immediately invoke it by wrapping it in parentheses and calling it right away, passing in num. This method of wrapping an anonymous function in parentheses and calling it right away is called an IFFE (Immediately-Invoked Function Expression, pronounced like "iffy"). This is where the "magical" part happens.

We're passing the value of num into our outer function. Inside that outer function, the value is known as numCopy -- aptly named, since it's a copy of num in that instant. Now it doesn't matter that num changes later down the line. We stored the value of num in numCopy inside our outer function.

The Final Version

Here's our original code, but fixed up with our closure trick. Test it out!

```
// clear the screen for testing
document.body.innerHTML = '';
var nums = [1,2,3];
// Let's loop over the numbers in our array
for (var i = 0; i < nums.length; i++) {
    // This is the number we're on...
    var num = nums[i];
    // We're creating a DOM element for the number
    var elem = document.createElement('div');
    elem.textContent = num;
    // ... and when we click, alert the value of `num`
    elem.addEventListener('click', (function(numCopy) {
        return function() {
            alert(numCopy);
        };
    })(num));
    document.body.appendChild(elem);
};
```

Event Loop

Resource

- What the heck is the event loop anyway? | JSConf EU 2014 - https://www.youtube.com/watch?v=8aGhZQkoFbQ - http://latentflip.com/loupe

Resources

- http://www.sitepoint.com/introduction-functional-javascript/
- http://scott.sauyet.com/Javascript/Talk/FunctionalProgramming/#slide-0
- https://www.youtube.com/watch?v=L7b7AW14rYE (slides)
- https://www.youtube.com/playlist?list=PL0zVEGEvSaeEd9hlmCXrk5yUyqUag-n84
- https://en.wikipedia.org/wiki/Functional_programming
- https://drboolean.gitbooks.io/mostly-adequate-guide

Functional Programming (from wiki)

functional programming is a programming paradigm — a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result f(x) each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods -- Wikipedia https://en.wikipedia.org/wiki/Object-oriented_programming

Concepts

First-class and higher-order functions

Higher-order functions are functions that can either take other functions as arguments or return them as results.

Higher-order functions are closely related to first-class functions in that higher-order functions and first-class functions both allow functions as arguments and results of other functions. The distinction between the two is subtle: "higher-order" describes a mathematical concept of functions that operate on other functions, while "first-class" is a computer science

term that describes programming language entities that have no restriction on their use (thus first-class functions can appear anywhere in the program that other first-class entities like numbers can, including as arguments to other functions and as their return values).

Higher-order functions enable **currying**, a technique in which a function is applied to its arguments one at a time, with each application returning a new function that accepts the next argument.

Recursion

Recursive functions invoke themselves, allowing an operation to be performed over and over until the base case is reached

Pure functions

Purely functional functions (or expressions) have no side effects (memory or I/O).

- If a pure function is called with arguments that cause no side-effects, the result is constant with respect to that argument list (sometimes called referential transparency), i.e. if the pure function is again called with the same arguments, the same result will be returned.

Functional JavaScript

First Class Functions

in JS functions can be in variable, obj value ...

Pure Functions

A pure function is a function that, given the same input, will always return the same output and does not have any observable side effect.

for every input it has to produce the same output. Mathces one to one or many to one, but not many to many.

- Does not depend on external state
- Does not depend in IO
- Does not cause side effects
-

Currying

Taking a function that takes multiple arguments and transforming it to a chain of functions that accept a single argument

```
function add(x) {
    return function(y) {
        return x + y;
    }
}

add(1)(2);
```

```
var add = function(x) {
  return function(y) {
    return x + y;
  };
};

var increment = add(1);
increment(2);
// 3
```

Compositions

Functions which consume the return value of the function that follows. It's when we string together functions in the linear sequence, the first fn goes to the input of the second function and so on.

```
a() b() c()
//    Composition
a(b(c(x))) === compose(a,b,c)(x)
```

The composition of two functions returns a new function. This makes perfect sense: composing two units of some type (in this case function) should yield a new unit of that very type.

```
var compose = function(f,g) {
  return function(x) {
    return f(g(x));
  };
};

var toUpperCase = function(x) { return x.toUpperCase(); };
var exclaim = function(x) { return x + '!'; };
var shout = compose(exclaim, toUpperCase);

shout("send in the clowns");
//=> "SEND IN THE CLOWNS!"
```

Higher-order functions

Functions that take other function as an argument.

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.

Immutable JS

Resources:

- https://www.youtube.com/watch?v=wA98Coal4jk

Data Visualization and D3.js

Resources:

- Udacity - Data Visualization and D3.js
- https://www.dashingd3js.com/table-of-contents
- https://square.github.io/intro-to-d3/
- https://www.youtube.com/watch?v=8jvoTV54nXw - good video, examples
  - continuing here
- https://www.youtube.com/watch?v=DTjLcLytNt8

Interesting Interview Questions

Resources:

- https://medium.com/javascript-scene/10-interview-questions-every-javascript-developer-should-know-6fa6bdf5ad95#.dktyo9hf5
-

Imperative vs Declarative

Resources:

- http://latentflip.com/imperative-vs-declarative/
- https://www.netguru.co/blog/imperative-vs-declarative
- http://codenugget.co/2015/03/05/declarative-vs-imperative-programming-web.html
- http://www.sitepoint.com/quick-tip-stop-writing-loops-start-thinking-with-maps/

Imperative programming

That's basically what imperative programming is all about — describing a program in terms of instructions which change its state.

Declarative programming

In contrast to the imperative one, declarative programming is about describing what you're trying to achieve, without instructing how to do it.

Monkey patch

A monkey patch is a way for a program to extend or modify supporting system software locally (affecting only the running instance of the program). This process has also been termed duck punching, based on the term "duck typing".

Resources:

- http://me.dt.in.th/page/JavaScript-override/
- https://davidwalsh.name/monkey-patching
-

Scrum, Kanban, Agile

Resources:

- http://www.agileweboperations.com/scrum-vs-kanban
- http://scrumtrainingseries.com/
-