**Function declaration types**

function A(){};             // function declaration  
var B = function(){};       // function expression  
var C = (function(){});     // function expression with grouping operators  
var D = function foo(){};   // named function expression

Foo  is is only accessible within the function in 4th scenar  
var E = (function(){        // IIFE that returns a function  
 return function(){}  
})();  
var F = new Function();     // Function constructor  
var G = new function(){};   // special case: object constructor  
var H = x => x \* 2;         // ES6 arrow function

**Functions can be self-executed**

Var myfunction = myalert(stringarg)

{

alert(stringarg)

}(“alert string”);

// alerts “alert string”

var x = function (a, b) {return a \* b};

A **function expression** can be stored in a **variable**

The function above is actually an **anonymous function** (a function without a name).

Functions defined using **an expression are not hoisted.**

1. [**JavaScript Objects in Detail**](http://javascriptissexy.com/javascript-objects-in-detail/)

**primitive data types**: Number, String, Boolean, Undefined, and Null are immutable

**complex data type** :objects are mutable (can be changed).Anything that doesn’t belong to any of these five primitive types is considered an object.

An object is an unordered list of primitive data types (and sometimes reference data types) that is stored as a series of name-value pairs.

var ageGroup = {30: "Children", 100:"Very Old"};

console.log(ageGroup["30"]); // Children​

console.log(ageGroup.30) // This will throw an error​

primitive data saved-as-value

objects: save-as-reference

**Object Data Properties Have Attributes**

Each data property (object property that store data) has not only the name-value pair, but also 3 attributes (the three attributes are set to true by default):

—  **Configurable Attribute:** Specifies whether the property can be deleted or changed.

— **Enumerable**: Specifies whether the property can be returned in a for/in loop.

— **Writable**: Specifies whether the property can be changed.

Writable: If false, the value of the property can not be changed.

Configurable: If false, any attempts to delete the property or change its attributes (Writable, Configurable, or Enumerable) will fail.

**Creating Objects:**

1. **Object Literals**

​var mango =

{

color: "yellow",

​howSweetAmI: function () {

console.log("Hmm Hmm Good");

}

}

1. **Object Constructor**

var mango =  new Object ();

mango.howSweetAmI = function () {

console.log("Hmm Hmm Good");

}

**Constructor Pattern for Creating Objects**

function Fruit (theColor, theSweetness, theFruitName, theNativeToLand) {

​    this.color = theColor;

   this.showName = function () {

       console.log("This is a " + this.fruitName);

   }

}

var mangoFruit = new Fruit ("Yellow", 8, "Mango", ["South America", "Central America", "West Africa"]);

mangoFruit.showName(); // This is a Mango.​

**only constructor function can provide private members as well as public members (encapsulation).**

**Properties inherited from Object.prototype are not enumerable, so the for/in loop does not show them.**

**serialize an object means (converting it to a string);**

**Serialize to a STRING -- catchy**

**2.** [**JavaScript Prototype in Plain, Detailed Language**](http://javascriptissexy.com/javascript-prototype-in-plain-detailed-language/)

Every JavaScript function has a prototype property (this property is empty by default), and you **attach properties and methods on this prototype property when you want to implement inheritance**

This **prototype property** is **not enumerable;** that is, it isn’t accessible in a for/in loop. But Firefox and most versions of Safari and Chrome have a **\_\_proto\_\_ “pseudo” property** (an alternative syntax) that allows you to access an object’s prototype property.

The JavaScript prototype property allows you to add new properties to an existing prototype(new var/function to constructor)

[http://yehudakatz.com/\*2011/08/12/understanding-prototypes-in-javascript/](http://yehudakatz.com/*2011/08/12/understanding-prototypes-in-javascript/)

JavaScript objects also have one additional attribute: a pointer to *another* object. We call this pointer the object's *prototype*. If you try to look up a key on an object and it is not found, JavaScript will look for it in the prototype. It will follow the "prototype chain" until it sees a nullvalue. In that case, it returns undefined.

**the properties** that JavaScript includes on (props in) **Object.prototype are not enumerable**

<http://yehudakatz.com/2011/08/11/understanding-javascript-function-invocation-and-this/>

function Tree(name) {  
 this.myvar = name;

var localvar=”local”;

}

1) methods inside constructor **can use both  local variable** in the constructor and **this.myVar**

**var** localVar = {}

methods that does not require access to the local variable can be defined on the prototype.

prototypes can access **this.myvariable** = "value";

but prototypes  cannot access **var localVar** = "localValue";

**3.** [**JavaScript Variable Scope and Hoisting Explained**](http://javascriptissexy.com/javascript-variable-scope-and-hoisting-explained/)

***Scoping* is the ruleset used to lookup variable values.**

**C, and the rest of the C family, has block-level scope.**

**In JS:**

**Local Variables Have Priority Over Global Variables in Functions.**

All variables declared **outside a function are in the global scope**

In the **browser**, which is what we are concerned with as front-end developers, the **global scope is the window object** (or the entire HTML document).

---If a variable is initialized (assigned a value) without first being declared with the var keyword, it is automatically added to the global context and it is thus a global variable:

function showAge () {

// Age is a global variable because it was not declared with the var keyword inside this function​

age = 90;

console.log(age);// ​

}

​

showAge (); // 90​

​

​// Age is in the global context, so it is available here, too​

console.log(age); // 90

**for (var i = 1; i <= 10; i++) {}**

**Var i is global.**

// The use of the "this" object inside the setTimeout function refers to the Window object, not to myObj​

​

​var highValue = 200;

​var constantVal = 2;

​var myObj = {

highValue: 20,

constantVal: 5,

calculateIt: function () {

setTimeout (function () {

console.log(this.constantVal \* this.highValue);

}, 2000);

}

}

​

​// The "this" object in the setTimeout function used the global highValue and constantVal variables, because the reference to "this" in the setTimeout function refers to the global window object, not to the myObj object as we might expect.​

​

myObj.calculateIt(); // 400​

​// This is an important point to remember.

**OOP In JavaScript**

1. An instance is an implementation of a Function
2. (Encapsulation) is an Object Creation pattern
3. Constructor/Prototype Pattern -> encapsulation of data variables**(properties) and functions**
4. In js,every function has **a property called constructor**, and this property points to the **constructor** of the function.
5. **Disadvantage** of overwriting the prototype is that the **constructor property no longer points to the prototype**, so we have to set it manually.
6. **Parasitic Combination Inheritance Pattern**

function **inheritPrototype**(childObject, parentObject) {

   // As discussed above, we use the Crockford’s method to copy the properties and methods from the parentObject onto the childObject​

​// So the copyOfParent object now has everything the parentObject has ​

   var copyOfParent = Object.create(parentObject.prototype);

​

   //Then we set the constructor of this new object to point to the childObject.​

​// Whenever you overwrite an object’s prototype (object.prototype = someVal), you also overwrite the object’s constructor property. So setting it manually.

   copyOfParent.constructor = childObject;

​

   // Then we set the childObject prototype to copyOfParent, so that the childObject can in turn inherit everything from copyOfParent (from parentObject)​

  childObject.prototype = copyOfParent;

//initially child prototype will point the child constructor, we have to manually point it to parent }

function parent(p1, p2, p3)

{

this.somevar ="100";

this.somevar2 ="100";

this.someFunc = fucntion()

{

return somevar;

};

}

parent.**prototype**.somefunc2 = function () {

   return this.somevar2;

};

function child(p1, p2, p3)

{

**parent.call**(**this**, p1, p2, p3);//inheritance ->(this,param for parent)

//this refers the child object

}

​// inherit the methods and properties from parent

**​inheritPrototype(child, parent);**

**HOISTING**

Variable Hoisting for declaration **(declarations not assigned to var)** only.. Not for definition/initialization

function definition hoisting only occurs for function declarations, not function expressions

---It is important to know that only **variable declarations are hoisted** to the top, not variable initialization or assignments (when the variable is assigned a value)

*// Outputs: "Definition hoisted!"*  
definitionHoisted();  
  
*// TypeError: undefined is not a function*  
definitionNotHoisted();  
  
**function** definitionHoisted() {  
   console.log("Definition hoisted!");  
}  
  
**var** definitionNotHoisted **=** **function** () {  
   console.log("Definition not hoisted!");  
};

**function foo() {**

**bar();**

**var x = 1;**

**}**

**is actually interpreted like this:**

**function foo() {**

**var x;**

**bar();**

**x = 1;**

**}**

**Notice that the assignment portion of the declarations were not hoisted. Only the name is hoisted.**

**Example:**

**function test() {**

**foo(); // TypeError "foo is not a function"**

**bar(); // "this will run!"**

**var foo = function () { // function expression assigned to local variable 'foo'**

**alert("this won't run!");**

**}**

**function bar() { // function declaration, given the name 'bar'**

**alert("this will run!");**

**}**

**}**

**test();**

**Function Declaration Overrides Variable Declaration When Hoisted (when both has same name)**

// Both the variable and the function declaration(not assignments) are named myName​

​var myName;

​function myName () {

console.log ("Rich");

}

​

​// The function declaration overrides the variable name​

console.log(typeof myName); // function

// But in this example, the variable assignment overrides the function declaration​

​

var myName = "Richard"; // This variable assignment (initialization) that overrides the function declaration.​

​

​function myName () {

console.log ("Rich");

}

​

console.log(typeof myName); // string

**IMPORTANT**

**var a = 1;**

**function b() {**

**a = 10;**

**return;**

**function a() {}**

**}**

**b();**

**alert(a);**

**Using the code above, the browser will alert "1".**

**HOW?**

**function b() {**

**a = 10;**

**return;**

**function a() {}**

**}**

**will be rewritten by the interpreter to this**

**function b() {**

**function a() {}**

**a = 10;**

**return;**

**}**

**Weird, eh?**

**Also, in this instance,**

**function a() {}**

**behaved the same as**

**var a = function () {};**

**(Above is the reason for all confusion)**

**So, in essence, this is what the code is doing:**

**var a = 1;                 //defines "a" in global scope**

**function b() {**

**var a = function () {}; //defines "a" in local scope**

**a = 10;                 //overwrites local variable "a"**

**return;**

**}**

**b();**

**alert(a);                 //alerts global variable "a"**

**Play with:**

function b() {

alert("in1"+a);

    a = 10;//now becomes local scope because of hoisted function a() {} -> var a =function(){}

alert(a);//local

alert(“”win”+window.a);//global

return;

function a() {} //declared after return,but hoisted to top

}

b();

alert("out "+a);

**4.CLOSURES:**

**A closure is an inner function that has access to the outer (enclosing) function’s variables—scope chain.**

**these functions 'remember' the environment in which they were created.**

**Callback Functions Are Closures**

The closure has three scope chains:

1. it has access to its own scope (variables defined between its curly brackets)
2. it has access to the outer function’s variables,also to the outer function’s parameters
3. it has access to the global variables.

**Closures are also frequently used in jQuery and just about every piece of JavaScript code you read..**

**Closures’ Rules and Side Effects**

1. Closures have access to the outer function’s variable even after the outer function returns.So,you can call the inner function later in your program.
2. Closures store references to the outer function’s variables

function makeAdder(x) {  
 return function(y) {  
   return x + y;  
 };  
}  
  
var add5 = makeAdder(5);  
var add10 = makeAdder(10);  
  
console.log(add5(2));  // 7  
console.log(add10(2)); // 12

1. the closure (the anonymous function in this example) **has access to the outer function’s variables by reference**, not by value. **Creating closures in loops is a common mistake** . So,**Immediately Invoked Function Expression (IIFE)** can be used to avoid this problem

***Lexical Scoping* are *Static Scoping* (or) *Closure*.**

**jQuery, one of the most popular JS libraries, uses closures to maintain the integrity of its functions and variables. It also uses a self-executing function to further protect it's scope. Here are the first and last lines of the core library:**

(function( window, undefined ) {  
 // Define a local copy of jQuery  
 var jQuery = ...  
   
 // Expose jQuery to the global object  
 window.jQuery = window.$ = jQuery;  
 })(window);

**Promises:**

"callback hell" =  the callback nested into another callback.

**Promises come in to rescue**.

**Asynchronous (doesn't block)**

**Asynchronous work happens at an unknown or unpredictable time.**

**Synchronous (waits)**

**JavaScript is always synchronous and single-threaded.**

**(Multi threading would cause massive concurrency issues )**

**Ajax calls:**

**It won't interrupt any other code that's currently running.**

**the code will keep executing *until* the callback is fired. Unless, of course, you specify the async: false option, which as you said, is bad practice.**

**async functions like setTimeout and setInterval are pushed onto an queue known as the Event Loop.**

## I reach for promises when I see the need.

**That need typically arrises in one of the following scenarios:**

* **I need to wait for multiple async responses before continuing**
* **I want to cache a response and skip doing the work multiple times**
* **I want to chain methods together**
* **Working with async generators**

**PROMISES are like TRY CATCH wrap around code ,which will finish at unpredictable time,**

* **A promise can only succeed or fail once. It cannot succeed or fail twice, neither can it switch from success to failure or vice versa.**
* **If a promise has succeeded or failed and you later add a success/failure callback, the correct callback will be called, even though the event took place earlier.**

* + **fulfilled(resolve) - The action relating to the promise succeeded**
  + **rejected (reject)- The action relating to the promise failed**
  + **pending - Hasn't fulfilled or rejected yet**
  + **settled - Has fulfilled or rejected**

**The promise constructor takes one argument -> a callback with two parameters ,resolve and reject.**

it's useful to do **all your promise-related work** inside the promise constructor callback

**then()** called by **resolve()**

**catch() called** for error inside promise constructor callback (or) error in then

**For looping:**

// Start off with a promise that always resolves

var sequence = **Promise.resolve()**;

// Loop through our chapter urls

story.chapterUrls.forEach(function(chapterUrl) {

 // Add these actions to the end of the sequence

 sequence = sequence.then(function() {

   return getJSON(chapterUrl);

 }).then(function(chapter) {

   addHtmlToPage(chapter.html);

 });

})

This is the first time we've seen **Promise.resolve()**, which creates a promise that resolves to whatever value you give it. If you pass it an instance of Promise it'll simply return it (note: this is a change to the spec that some implementations don't yet follow). If you pass it something promise-like (has a then() method), it creates a genuine Promise that fulfills/rejects in the same way. If you pass in any other value, e.g., Promise.resolve('Hello'), it creates a promise that fulfills with that value. If you call it with no value, as above, it fulfills with "undefined".

There's also **Promise.reject(val)**, which creates a promise that rejects with the value you give it (or undefined).

**Promise.all** takes an **array of promises** and creates a promise that fulfills when **all** of them successfully complete. You get **an array of results** (whatever the promises fulfilled to) in the **same order** as the promises you passed in.

**ES6 also gives us** [**generators**](http://wiki.ecmascript.org/doku.php?id=harmony:generators)**, which allow functions to exit at a particular point, like "return", but later resume from the same point and state,**

**Returning a Promise from a then, will append it to the promise chain.**

function wait(millis) {

   return new Promise(resolve => setTimeout(resolve, millis));

}

const p = wait(5000).then(() => wait(4000)).then(() => wait(1000));

p.then(() => { /\* 10 seconds have passed \*/ });

The [Promise.race()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race) static method accepts an iterable of Promises and returns a new Promise which resolves or rejects as soon as the **first** of the promises in the iterable has resolved or rejected.

**Apply-call-and-bind-methods:**

* C[all](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call) invokes the function and allows you to pass in arguments one by one.
* [Apply](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply) invokes the function and allows you to pass in arguments as an array.
* [Bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) returns a new function, allowing you to pass in a this array and any number of arguments.

*Bind ()* method primarily to call a function with the ***this*** value set explicitly.

$ ("button").click (**user**.clickHandler.bind (**user**));

JavaScript’s Bind Allows Us to **Curry a Function**

Function Currying, also known as **partial function application,**  is the use of a function (that accept one or more arguments) that **returns a new function** with **some of the arguments already set**.

The function that is returned has access to the stored arguments and variables of the

function greet (gender, age, name)

{

// if a male, use Mr., else use Ms.​

var salutation = gender === "male" ? "Mr. " : "Ms. ";

​

if (age > 25) {

return "Hello, " + salutation + name + ".";

}

else {

return "Hey, " + name + ".";

}

}       var greetAnAdultMale = greet.bind (**null**, "male", 45);

//**sets second and third argument** and returns a new function

​      greetAnAdultMale ("**John Hartlove"**); // "Hello, Mr. John Hartlove."​ //male and 45 is already set

       var greetAYoungster = greet.bind (**null**, "", 16);

       greetAYoungster ("**Alex**"); // "Hey, Alex."​

       greetAYoungster ("**Emma Waterloo**"); // "Hey, Emma Waterloo."​

outer function.

**-------------------Apply and Call-----------------------------**

The apply and call methods are almost identical when setting the ***this*** value except that you pass the function **parameters to apply() as an array**, while you have to list the parameters **individually** to pass them to the **call()** method.

**C**all is for **c**omma (separated list) and **A**pply is for **A**rray.

First param is set to this, other params are used as arguments for the called function

**Borrowing Functions with Apply and Call (A Must Know)**

* **Array methods are generic (except *toString* and *toLocaleString*)**
* **generic -> not bound to specific var type**

An **array-like object** is an **object** that has its **keys** defined as **non-negative integers.**

var **anArrayLikeObj** = {0:"Martin", 1:78, 2:67, 3:["Letta", "Marieta"], length:4 };

               // First parameter sets the "this" value​

               var newArray = Array.prototype.slice.call (**anArrayLikeObj**, 0);

​

               console.log (newArray); // ["Martin", 78, 67, Array[2]]​

           Like the preceding example, we can also use *apply ()* and *call ()* to **borrow String methods**. Since **Strings are immutable**, only the non-manipulative arrays work on them, so you cannot use *reverse*, *pop* and the like.

**This keyword:**

<http://javascriptissexy.com/understand-javascripts-this-with-clarity-and-master-it/>

1. Fix this when used in a method passed as a **callback**

 $("button").click (user.clickHandler.**bind** (user));

1. Fix this **inside closure**

**Using this inside foreach will refer global window obj**

 // To capture the value of "this" when it refers to the user object, we have to set it to another variable here:​

   // We set the value of "this" to theUserObj variable, so we can use it later​

**var theUserObj = this;**

   this.data.forEach (function (person) {

   // Instead of using this , we now use theUserObj inside cloure

   console.log (person.name + " is playing at " + theUserObj.tournament);

   })

   }

1. Fix this when method is assigned to a variable

var global1 = {

firstname = “**g1fn**”;

Display: function(){console.log(this.firstname)};

}

var firstname = “**gfn**”;

Var caller = global1.display;

caller(); // logs **gfn**

Var caller2 = global1.display.**bind(global1);**

caller2(); // logs **g1fn**

1. Fix this when borrowing methods

problem:

gameController.avgScore = appController.avg();//returns this.someval i.e returns appController.someval

fix:

appController.avg.**apply** (gameController, gameController.scores);

**JS VS DEFFERED**

**https://developers.google.com/web/fundamentals/getting-started/primers/async-functions**

**webworkers api separate -threads**