**Notes: September 12, 2013**

* There is no document.onload, only window.onload (fires when all resources and objects have loaded)
* In JQuery, document.ready fires when DOM is loading
* Implicit global rule, “undefined” assignation to variables that are assigned some value, but not explicitly instantiated (aka no var myVar, just myVar = 8;). Error if trying to use one otherwise.
* Static functions that can be shared can be made using a prototype. [explained later]

SCOPE:

* JS – Global, function, and no “for loop” or brackets scope, so anything defined in one of those brackets would be globally accessible

CLOSURE:

* What is a closure? Whenever you see a function inside another function, the inner function can access the variables in the outer function.
* Things get interesting when the inner function is returned. [see ppt for week 3]

OVERLOADING:

* NO FUNCTION OVERLOADING: last function defined will be the one that is used

PROTOTYPE:

* No classes, so no class inheritance. Instead, each object has a prototype object linked to it.
* All objects created with literals have the same prototype object associated with them, called Object.prototype
* every object type has a prototype property: ie Number.prototype, Array.prototype, String.prototype, YourConstructor.prototype
* How is the link created? The lookup only happens during retrieval [see slides]

THIS:

* behaves differently: When a function is called as a method of an object, its this is set to the object the method is called on; When a function is used as a constructor (with the *new*) … [see slides!]

ADDING BEHAVIOR TO OBJECTS:

* can modify prototype to reflect a change in all the “child” objects

DEBUGGING:

* it is hard in JS, but there are tools in Chrome (console.log, breakpoints, stepping through code)

**Notes: September 19, 2013**

Name Lookup

* obj.name – try to look up ‘name’ in the lookup table and bring it back

But, there is a catch:

* if the name is not found in obj, the lookup goes to the prototype: obj.\_\_proto\_\_ recursively till it returns a value or if it doesn’t find it, it returns ‘undefined’

What about this.name?

* you are always executing some function, and every function has a hidden parameter that you access with the keyword ‘this’.
* this.name means go to that hidden parameter and look up the value stored under that name

!!! this parameter is set differently depending on invocation

Function objects

* in addition to the normal lookup table, you can think that the function has a private lookup table to store local variables that the function code can access only, then things are dynamically linked so that outer scopes’ variables are accessible
* \*this.A would point to not the local A, but within the this lookup cell for A, then up the proto chain

Closures

* Lookup chain for variables is established dynamically when the object is created and there is a hidden link to the function that created that function, and once established, that link stays. Period. These lookup tables are a lot like stack frames, unlike globals; they are actually on the heap, where they live until they are garbage collected. That’s why closures work. So closures have access to the locals they could access when they were first created until those locals are no longer available.

Unqualified Names (**see slides!!!**)

* if something can’t be found through the lookup chains, it will think “maybe it’s meant to be this.name”, if it’s not found there, it’ll check the globals, which are shorthand for window, essentially, which then returns ‘undefined’

<Canvas>

* allows you do low level drawing: lines, paths, texts, images, affine transformations, gradients, colors, etc
* Basically full output/drawing API
* **See Example Sites in ppt! 🡪 2D and 3D drawing**

**(3D won't be talked about here, though)**

* accessing canvas: get context
* you draw using the context, not the canvas itself. You can move it, define paths and stroke or fill those paths, draw images or text, etc. You don’t draw anything till you call fill() or stroke()
* you can set attributes of you context (pen) – color, thickness, cap style, etc
* Context has all these setting for your pen:

save() – saves current state of context

restore() – restores context to last saved state

* Transformations: translate(x,y), rotate(x,y), setTransform. – Changes your frame of reference (origin and orientation of axis) for all subsequent drawing (don't think of the canvas moving underneath, think about the context), transformations also can be saved and restored
* Be SURE to call beginPath() before!

**Notes: September 26, 2013**

* You create this canvas in HTML ,canvas id=”” style=”…”></canvas>
* There is one default element in canvas called context, like a magic pen
* Draw in JS - use context property in canvas
* var canvas = document.getElementById(“myCanvas”);
* var context = canvas.getContext(“2d”); // from sept. 19 slides

TODAY:

* Name resolution (for variables and properties)
* Prototypes and Inheritance
* P2 (makes you canvas expert! ☺ )

Name Resolution

* How JS looks up and finds identifiers

Variables use “var identifier”, Properties belong to an object

Variables – NR based on scope

* two scopes: Global level scopes – not in fxns [Window is global obj]
* function scope – new scope created on every function call
* Name Resolution works here by going up through enclosing scopes till name is found, or if not, returns error

Property Name Resolution

* Everything is an obj, and every obj had \_\_proto\_\_ property
* Every fxn has a prototype property – so fxn has both \_\_proto\_\_ and prototype property
* Chain: first look at obj properties, if property not found, repeat search in \_\_proto\_\_, if nothing is found then “undefined” is returned

**[See slides for example]**

Preferred way of setting \_\_proto\_\_

* Create object using constructors. When created this way the new obj’s \_\_proto\_\_ is automatically set to MyConstructor.prototype 🡪 think of the constructor as the type, set prototype to something in constructor
* **[See slides for(preferred) prototype example]**

Assign Statements work differently

* Assign value to non-existing property 🡪that property gets created in the current object
* Doesn't’ modify/check property in \_\_proto\_\_, it just creates one if it doesn’t exist in the obj already
* Assign value to non-existing var 🡪 crawls up scopes to find where it is. If nothing is found, a var is created in the top level(global) – the implicit global rule

“THIS” 🡨 important, not like JAVA

* In most cases the value of “this” is decided by how a function is called 🡪 decided during runtime

4 General Rules

* “this” in global context (outside of a fxn) 🡪 refers to the global object (window)
* “this” in global function 🡪 refers to the global object (window)
* “this” in obj’s method 🡪 refers to the obj which the object is called on
* “this” in function 🡪 refers to the obj which the object is called on ???
* **[See slides for rule examples]**

Introducing call(…)

* functionName.call(obj, attrs …);
* It invokes the function ‘functionName’ and replace all “this” in the function is passed in 🡪 **[See slide for example]**

Inheritance

* It’s good for sharing beviors and reusing code

Objs in JS are different than in JAVA

* JS doesn’t explicitly specify that one class can inherit from another
* however we still want the benefits of inheritance
* **[See link in slide]**

[3 slides of review of what we know 🡪 what if we want to define a more specific type of animal?]

Super()

* Solution: use function.call(obj, params);
* Calls function, passing in obj as the ‘this’ object to the function, and the params as the params to the object (params can be comma-separated list)

Set a parent obj shared by all children **[see 3 slides!! + other slides]**

**CHECK OUT: [INHERITANCE RESOURCES SLIDE!!]**

**Notes: October 3, 2013**

* There are two types of events in JS: User and System

Event Handlers

* Functions that are called when something should be done when something happens
* Functions that aren’t called immediately – event callbacks or callback

Be careful of Syntax

* When you want to pass a function to another obj, use only functionName(without the parentheses) 🡪 see code on slide

Handling input in JS

* 3 elems: event, target, and action
* Event: what you want to capture
* Target: where this event should happen
* Action: what you want to happen

2 ways to capture events

* using event handlers in JS file – myDiv.addEventListner(“click”, myFunc); 🡪 cn add multiple handlers to single event
* Using HTML inline properties <div onClick=”myFunc()”>Click Me!</div>

Add Event Listener in JS

* Getting the target(s) using 🡪 document.getElementById(…) returns an element -🡪 document.getElementByClassName(…) returns an array of elements
* element.addEventListener(“click”, foo); and element.removeEventListener(“click”, foo);

Add listeners in JS

* you can add multiple handlers to an event:

element.addEventListener(“click”, foo);

element.addEventListener(“click”, foo2);

element.addEventListener(“click”, foo3);

[See slides for resources]

Inline HTML events (see slide for resources)

* can’t assign multiple handlers

**// the result will only be ‘hello’ when theDiv is clicked**

theDiv.onClick = function(){alert(‘hi’););

theDiv.onClick = function(){alert(‘hello’););

Event properties (see slide for resource)

* Your event handler is passed a parameter which gives details about the event 🡪 eg target, type, etc

console.log(event.type);

console.log(event.type.id);

Types of Input events

* click, mouseover, keydown, onLoad

“this” in event handlers

* The target element 🡪 ex. in onClick: the element that is clicked
* Sometimes, closure is useful when you want to save “this” in other context and use in your event handler

**function myObj(a){**

**this.a = a;**

**var self = this;**

**window.setTimout(function() {self.a = 100;}, 3000);**

**}**

Event Orders

* B/c HTML elements are often nested, one user action could trigger multiple elements’ events

2 Phases

* Capturing: outer element, tunnel down
* Bubbling: target up tree to parents
* According to W3C event model, the event is first captured (tunnels) until it reaches target element, then bubbles back up

Event orders

* There is an optional 3rd parameter called *useCapture* in addEventListener to let you decide which phase you want to register your event in
* By default this value is false – always fires in bubbling phase

stopPropogation

* Use “stopPropogation” to stop bubbling the event 🡪 see slides for more info

Project 2 – Problems encountered

* An event handler is used called image.onload
* An image starts to load when you assign a source to it
* Onload firs when the image finished loading
* Problem occurs when some variables used in the onload function get modifiec elsewhere 🡪 see slide for code snippet
* It is always a good practice to load function in a constructor instead of it’s draw method, so the image Is only loaded once 🡪 draw function only does draw
* Then all we need to do is make sure the draw function is called AFTER the image is loaded

Possible Solution 1

* Create a prop in DoodleImage called “loaded”/ It is false originally and set to true when the image finishes loading
* In draw function in Doodle (main canvas), only start to draw when all children (including children in the main containers) finish loading
* You can write a recursive function to loop through children
* If any image is not loaded yet, wait for 10 ms and check again
* If everything is loaded, start drawing

Possible Solution 2

* Create a prop in DoodleImage called “loaded”/ It is false originally and set to true when the image finishes loading. Create another prop called “drawnBeforeLoaded”. default is false. 🡪 **see slides**

**Notes: October 17, 2013**

* AJAX – asynch JS and XML – invoke code from servers without refreshing the page
* XMLHttpRequest

open(method, url) – initialized a request, specify a method (GET, POST,

etc), and a URL to which you send the request

,send() – send the request(opt data arg)

[see slides]