Javascript Programming

Learn language features, Hands-on Lab,

Do some graphics ...

Data Types

- 3 main data types:
  - Booleans
  - Numbers
  - Strings
- 2 container types
  - Arrays
  - Objects (Bags / Tables)

- Boolean: true or false
  - Compare: ===, !==; logical operator: cond1&&cond2, cond1||cond2
- Numbers: integers, float, doubles
  - Operations: arithmetic: +, -, *, /, %, ++, --; logical compare: >, <, ===, !==,<=,>=
- Strings: “...”, ‘...’, `…`
  - Characters are found by its position/index: str[0], str[1], str[2], …
  - Concat: str1+str2; Compare: str1<str2 (lexically compare)
  - str=”abc”; str.length, str.toUpperCase(), str.toLowerCase(), …

    !!! Strings values cannot be changed, just create new ones !!!
  - Template string with embed data: a=123; `the final result is ${a}`

- Env: where do your code run
  - Browser,
  - Computer,…

- How your code interact with its env:
  - input: receive instructions from env/human
  - output: return result data to env/human

```
console.log(...string msg…)
console.log(“hello world!”)
```

Names

- Variables, Constants, Functions, …
- Constants: name some values which cannot change
  - const PI=3.1415
- Variables: name some values which can be changed
  - such as your intermediate calculation results
  - a=1; var a=123
  - !!! the correct way:
    - let a=123.3
    - let z=x+y

Control Flows

- Branch/decision, Loops
- Branch/decision:

```
if (<condition>) { //then
    doSomething
} else {
    doSomethingElse
}
```

More branches

```
if (<condition1> {
        ...
} else if(<condition2>) {
        …
} else if(...) {
        ...
} else {
        ...
}
```

Loop 1

```
while(<condition>) {
    doSomething
}

let i=0
while(i<10) {
    console.log(i)
    i=i+1
}
```

Arrays

- A group of values, grouped as a linear series, values can be of different types
- let a=[1,"we",3.1415,["hi","hao"]]
  - Empty array: let emptyArray=[]
- Each value in array will be found by an integer “index” - its position
  - Starting at 0, go up to array.length-1
  - a[0], a[1],..., a[a.length-1]
- Add/remove value at tail: a.push(v), a.pop()
  - Add/remove value at head: a.unshift(v), a.shift()
- a.reverse(), a.sort()

Loop 2

- let arry=[]
  for(i=0;i<10;i++) {
  arry.push(i)
  }

- for(i=0;i<arry.length;i++) {
  arry[i]=arry[i]+10
  }

Functions

- Package a group of instructions together as one unit
- Reuse: give it a name, “call” it, I.e. run the group of instructions again and again
- Perform a specific functionality: rotate-a-box, add a number, ...
- Call/invoke a function: give it inputs, receive output “result” from it

```
function add2and4( x ) {   // accept ”x” as input, called “arguments”
    let result=x+2
    result=result+4
    return result          // must use “return” keyword to return results to caller
}
```

Different Ways to Define Functions

- let addxyz = function(x,y,z) {

```
        let result=x+y
        result=result+z
        return result
    }
```

- let add2and4 = (x) => { // so called “lambda”

```
        return x+2+4
    }
```

Objects – another container

- Array – linear series of values, each value is found/indexed by a integer (its position in array)
  - An array provides a order of its values
- Objects – bag/aggregate of values, each value is found/indexed by its string name/key
  - An object provides no order
  - let obj1={key1:value1,key2:value2,…}; let emptyObject={}
    - let jason={name:”Jason”,age:17,weight:100}; jason.age+=1; jason[“age”]+=1
- Each name(key)/value pair in object called a property of object
  - Add a property to object: obj.key=value, obj[“key”]=value
  - Delete property: delete obj.key; delete basket[“apple”]

Enumeration/Iteration over containers

- Enumeration/Iteration over arrays:

```
let fruits=[“apple”,”orange”,”pear”]
for(let index=0;index<fruits.length;index++) {
    console.log(fruits[index])
}; //shorter-form:  for(let f of fruits) { console.log(f) }
```

- Enumeration/Iteration over objects/bags:

```
let basket={apple:3, orange:5, pear:2}
for(let key in basket) {
    console.log(“i got “+basket[key]+” “+key)
}
```

Functions: transformations or actions

- Functions as transformation on input/arguments
  - function xform(input/arguments) {
    … perform transformation …
    return result
    }
  - cook(rice,vegi,salt,oil,…) {
    … wash … cut … boil … fry …
    return meal
    }
  - function add(x,y) { z=x+y; return z }
  - Perform transformations(operations) on targets(operands)
  - operands receive transformation (effected on them) passively

Functions as Actions

- Subjects/actors perform actions actively: bird chirp, dog bark, people laugh ...
- Turn function into an action(method): add magic keyword “this”
  - function growOlder() {

    this.age+=1; this.height+=2; this.sigh()

    }
  - “this” - refer to subject of this action; used to access subject’s properties.
  - Note: “this” is not an input argument (magically appear)
- How to define a subject/actor? Use object
  - let jason={name:”Jason”,age:17,weight:100}
- Add action to object: jason.growUp=growOlder; jason.growUp()

Class

- Easy get wrong when defining object and its actions/methods separately
- Class allow defining object and methods together:

```
class Person {
    constructor(name,age) {  // create object by special method “constructor”
        this.name = name
        this.age = age
    }
    growUp() { this.age+=1 }  // method1
    doSAT(score) { this.SAT_score=score }  // method2 adds a new property
    talk() { console.log(“blah blah blah”) }
}
let jason=new Person(“Jason”,17); jason.growUp(); jason.doSAT(1530)
```

Base Classes, Abstractions

- Use base class (parent class) to build common abstractions shared by other classes (children classes)

```
class Shape {
    constructor(x,y) { this.x=x;this.y=y }
    move(dx,dy) { this.x+=dx;this.y+=dy }
}
```

- Children classes “extends” base class to inherit its properties and methods
  - In constructor, use super(…) to init/construct base class

```
class Circle extends Shape {
    constructor(x,y,r) { super(x,y); this.radius=r }
    scale(dr) { this.radius *= dr }
}
```

  - let c=new Circle(10,10,3); c.move(1,2); c.scale(10)

Function as value

- Function definition: bind a function “name” to a “function value”
  - let add = function(x,y) { return x+y }
  - “add” is a normal name/variable, can rebind to other values:
    - add=function(x,y) { return x-y }, or even add=”Hello”
  - before binding to a name (without a name), function values also called anonymous functions.
- Function values are used similar to other values
  - Stored to a object property: obj.add=function(x,y){return x+y}
  - Pass into other function as arguments:

    function doit(arg1,func1) { return func1(arg1) }; doit(12,function(x){return 2*x})
  - Returned from another function as result:

    function tellyou(x) { return function(msg) { return “I tell you,”+x+”,”+msg } };

    let tellJon=tellyou(“Jonathan”); telljon(“Keep learning”)

Events, Callback or Handler

- Callbacks: a common pattern to register functions for custom processing/handling when some events happen:

```
class Shape {
    constructor(x,y,name) { this.x=x; this.y=y; this.name=name }
    on(event, callback) { this[event]=callback } //register callbacks for events
    move(dx,dy) { this.x+=dx;this.y+=dy;this.moved(this) } //invoke callback after “move” event
}
let s=new Shape(1,2,”circle”);
s.on(“moved”,function(x){console.log(x.name+” moved!”)})
s.move(11,0)
```

- Other callback samples:

```
timeout(function(){console.log(“call me after 1 second”)},1000)
setInterval(function(){console.log(“call me once every second”)},1000)
function ff(x, func){ if (x>2) func() }; ff(3,function(){console.log(“call me if x>2”)})
```

Arrays Methods

- Methods which change array:
  - shift()/unshift() (at head), pop()/push() (at tail)
  - reverse(), sort(), fill(value,start_index,end_index)
- concat(4,5), concat([4,5]): add new elements; return new array; original array not touched; same as “+” operator
- Subarray: data.slice(head [, tail]), return subarray with values in range [head,tail): data[head], data[head+1],...,data[tail-1]
  - data.slice(2,4) //include data[2],data[3]; result.length=tail-head
  - data.slice(3) // same as data.slice(3,data.length)
  - can use negative index: count from tail of array
    - data.slice(-2), data.slice(1,-2)
- copyWithin(dest_start_index,src_start_index,src_end_index): copy a range of values in range [src_start,src_end) to [dest_start, …):
  - data=[1,2,3,4,5]; data.copyWithin(1,3,5) // data===[1,4,5,4,5]

Lambda: anonymous function revised

- Lambda (arrow notation, fat-arrow): using “=>”
  - omit keyword “function”
    - let add = (x,y)=>{ return x+y } //compare: let add=function(x,y){return x+y}
  - If function takes single argument, can omit parenthesis:
    - let add4 = x=>{ return x+4 }
  - If function body is a single expression, can omit curly braces and “return” keyword
    - let add = (x,y)=>x+y; let add4= x=>x+4
    - let hi = ()=>”hello!”
- Great for callbacks, event handlers:
  - data=[1,2,3,4,5]; data.sort((a,b)=>{return b-a}); data.forEach(v=>{console.log(v)})
  - “this” always points to the (correct) inner-most wrapping object; no need of “let that=this” trick; ball.on(“click”, evt=>{ setInterval(()=>{ … }, 1000) })

More Array Methods

- splice(index, num_values_delete, values_to_add…)
  - data=[1,2,3,4]; data.splice(1,2,12,13) // data===[1,12,13,4]
  - splice(…) return subarray deleted
  - super method to change array, simplify: just for delete: data.splice(index,3); just for insert values: data.splice(index,0,12,13)
- Search: find “target” value in array (return indx, or -1 if not found)
  - data.indexOf(target_val, start_index) //search from head to tail
  - data.lastIndexOf(target, start_index) //search from tail to head
- Lambda in array methods:
  - Invoke a lambda for each value: data.forEach(v=>{...})
  - use lambda to define rules for sorting: data.sort((a,b)=>{return b-a})
  - rules to define target of search: data.findIndex(v=>v>12); data.findLastIndexOf(…); Or return found val: data.find(v=>...)

Array Methods: Ultimate Powerful

- map: transformation - perform a operation on each value and collect its result into a new array to return as result
  - data=[1,2,3,4,5]; newdata=data.map(v=>2*v)
  - similar to: map(data, func) { res=[];for(let v of data){res.push(func(v))}; return res}
- filter: selection – select values which satisfy condition defined by a lambda, collect them into a new array to return as result
  - newdata=data.filter(v=>v>3); bigBalls=balls.filter(b=>b.radius>50)
- reduce: calculation – perform a calculation on each value “accumulatively” and return result (value, object, or another array)
  - Sum: data.reduce((accum,v)=>{return accum+=v},0)
    - Accumulator: 1st argument of lambda - “accum”, with initial value defined as 0
    - After calling lambda for a array value, its result is used as accum for next call for next value
    - After calling lambda for last array value, the result is returned as result of reduce()

Math Methods

- Constant properties:
  - Math.PI, Math.E, Math.SQRT2,
  - Math.LN2, Math.LN10 //natural logarithm of 2, 10
  - Math.LOG2E, Math.LOG10E //base 2/10 log of E
- Methods:
  - Math.abs(x); Math.min(a,b,...); Math.max(a,b,...);
  - Math.pow(x,y); Math.sqrt(x); Math.exp(x);
  - Math.ceil(x); Math.floor(x); Math.round(4.4);
  - Math.sin(x); .cos(x),tan(x),asin(x),acos(x),atan(x),
  - Math.random(); Math.log(x) //natural logarithm (base E) of x

String Methods

- Basic:
  - str.length; str.charAt(index) //same as str[index]
- Combining strings:
  - str.concat(str1,str2,...)
  - str+str1+str2+...
- Split strings: str="a b c"
  - str.split(" ") → ["a","b","c"]; str.split(" ",2)→["a","b"]; // str.split(",",num)
  - str.split()→["a b c"]; "abcd".split("")→["a","b","c","d"]
- Substrings:
  - str.substr(start,length); "hello".substr(1,3)→"ell"
  - str.substring(start,end); "hello".substring(1,3)→"el" // range [1,3)
  - str.slice(start,end); "hello".slice(1,3)→"el" // range [1,3), same as substring()

More String Methods

- Search/match with string:
  - return boolean:
    - str.endsWith(str1); str.startsWith(str1); str.includes(str1);
  - find index (return -1 if not found):
    - str.indexOf(str1); str.lastIndexOf(str1); str.search(str1)
  - compare:
    - let n = str.localeCompare(str1);
    - n - lexical order: same as “str<str1”
      - N<0: str<str1
      - N===0: str===str1
      - n>0: str>str1
- Case conversions:
  - str.toLowerCase(); str.toUpperCase();

Data Type Conversions

- Basic data types: Number, String, Boolean; how do we convert one data type to another !?
- Global conversion functions:
  - Number(x): Number("3.1415"), Number(" "), Number("")
    - parseInt(x), parseFloat(x)
  - String(x)
  - Boolean(x)
- Object methods : // obj.toString()
  - numbers to strings:

    123.toString()→"123"; 9.656.toExponential(2)→"9.66e+0"; 9.656.toFixed(2)→9.66;
    9.656.toPrecision(2)→"9.7"
  - boolean to strings: true.toString(), false.toString()
- Auto conversions:
  - at output, auto convert to strings (obj.toString() called):

    console.log(x,y);
  - "+" operator: when combined with other strings with "+":

    x+" "+y+" "+z; // x,y,z are converted to strings automatically
  - math operator "-","/","*": convert strings to number auto: "6"/"2", "2"*"3"

GUI Apps and Design with Markup Language

- Browsers, Games - GUI software (Graphical User Interface)
  - GUI software common design:
    - scene graph: a hierarchy/tree of graphical objects on screen
      - objects called diff names: nodes,widgets,elements,components
    - user interactions: how objects interact with user mouse/keyboard/touch events
- Scene graph normally created in two ways:
  - in programming languages: in your code drawing.js:
    - first create scene2d obj,
    - then create and add many balls and lines to scene2d
  - in markup languages, pure specification for non-programmers
    - use following markup to create a new ball in scene:

      ```
      <ball center="100 100" raidus="5" color="1 0 0">
      </ball>
      ```

    - think it as JS constructor: new ball(center,radius,color)

Key Parts of Markup Spec

- 3 key parts of markup spec for objects:

```
<ObjType id="obj" attr1="..." attr2="..." ...>
    ...content or children nodes...
</ObjType>
```

  - Consider above Markup as JS code:

```
let obj=new ObjType(attr1,attr2,...)
obj.addContent(...)
obj.addChildren(...)
```

- type (or tag): refer to object class/type/tag or shape, geometry

  <ball>,<box>,<button>,<header>,...

  - tags used in pair:

```
<text>  // opening tag
...content...
</text> // closing tag
```

    similar to in javascript parenthesis pair (...), {...}

- attributes: object-properties
  - how objects to be rendered: layout, color, style…
  - how user interact with the object: event-handlers/onclick,...
- id: object/variable name, so other code can refer to it

Containers and Examples

- container objects:
  - a inner-node/level of tree: for building up hierarchy
  - group behaviour: layout/style children objects together

    groups, lists, ...
- example: how to create a car object in markup

```
<group id="car" translation="0 0 0" scale="1 1 1" rotation="1 0 0 1.57">
    <body id="red-car-body" color="1 0 0">
    </body>
    <wheel id="wheel1" translation="-1 -1 0" color="black"></wheel>
    <wheel id="wheel1" translation="-1 1 0" color="black"></wheel>
    <wheel id="wheel1" translation="1 -1 0" color="black"></wheel>
    <wheel id="wheel1" translation="1 1 0" color="black"></wheel>
</group>
```

User Interactions

- devices: mouse, keyboard, touch-screen
- event types: MouseDown, KeyDown, TouchDown,...
- events targets: objects in scene graph
- event handlers/callbacks: javascript functions
- binding (eventType-target-handler):
  - in javascript:

    ball.onclick=clickFunc1
  - in markup, using markup attributes:

    <ball id=”ball1” onclick="clickFunc1()">

    </ball>

X3D - Markup for Web 3D Graphics

- X3D alias VRML - Virtual Reality Markup Language
- Scene graph:
  - root node (root of scene graph tree) - <scene>:

```
<x3d>
    <scene>
        ......
    </scene>
</x3d>
```

- Graphics objects:
  - use <shape> to wrap geometry and appearance

```
<shape>
    <appearance>
        <material diffuseColor="1 0 0"></material>
    </appearance>
    <box></box>
</shape>
```

  - primitive geometries:

    <cone>,<box>,<sphere>

  - <shape> objects created default at origin "0 0 0"

    wrap it with <transform> to move to diff location

W3D Markup Continued

- Container nodes
  - group: <group>...</group>
  - transform:

```
<transform translation="2 0 0" scale="1 1 1" rotation="0 1 0 1.57">
    ...children <shape>...
</transform>
```

  - all children nodes will be transformed together
- Coordinate system: right-hand coordinate system
  - Origin: “0 0 0” on screen surface center
  - X: point to right
  - Y: point to up
  - Z: point out of screen
- Default user viewpoint position "0 0 10", looking into screen at origin “0 0 0”

X3D Markup Continued

- Event handling:
  - Bind event type and javascript handler using attributes (onclick,onmousedown,...)

```
<shape onclick="rotateCone()">
    <cone></cone>
</shape>
```

- How to refer and change a X3D object in Javascript
  - name object with "id" in markup:

```
<shape id="mybox">
    <box></box>
</shape>
```

  - find and use the object with id in javascript:

```
let mybox=document.getElementById("mybox")
mybox.getAttribute('size'); mybox.setAttribute('size',…)
mybox.getFieldValue('size'); mybox.setFieldValue('size',...)
```