Code Academy

- If you have a problem completing assignment 1:
  - Create Google e-mail account
  - Using this account sign into: http://www.codecademy.com/learn
  - Complete all 8 JavaScript modules at http://www.codecademy.com/tracks/javascript
  - Demonstrate your score to the instructor during the next lab, and answer several questions about your solutions
- This gives you 80% of lab 1 and the first assignment and a chance to perform well in this course

Reminder: literal and constructor (class)-based object declaration

- *Literal*: named variable (see: definition)
- *Constructor*: function which prescribes how an object should be created

Reminder: literal object declaration

```
//declare variable (hash table) with fields.
var university = {
    name:””,
    address:””,
    …
};
```

Reminder: literal object declaration

```
//declare variable (hash table) with fields. Each field has a key, and a value.
var university = {
    name:””,
    address:””,
    …
};
```

Reminder: literal object declaration

```
//declare variable (hash table) with fields. Each field has a key, and a value.
//The value can be a literal or a function
var university = {
    name:””,
    address:””,
    students: [],
    startUniversity: function (name, address)
    {
        this.name = name;
        this.address=address;
    },
    addStudent: function (newStudent)
    {
        var count = this.students.length;
        this.students [count ] = newStudent;
    }
};
```

Reminder: constructor-based object declaration

```
//define how an object of type Student should be created
function Student (name, bDay, bMonth, bYear)
{
        this.name = name;
        this.birthDate = new Date(bYear, bMonth, bDay);
        var today = new Date();

        this.age = today.getFullYear() -  bYear;
        this.toString = function ()   { return this.name + ": " + this.age;};
}
```

Reminder: constructor-based object declaration

```
//define how an object of type Student should be created. Parameters are
passed during object creation and are assigned to object’s fields
function Student (name, bDay, bMonth, bYear)
{
        this.name = name;
        this.birthDate = new Date(bYear, bMonth, bDay);
        var today = new Date();

        this.age = today.getFullYear() -  bYear;
        this.toString = function ()  { return this.name + ": " + this.age;};
}
```

Reminder: constructor-based object declaration

//define how an object of type Student should be created. Parameters are passed during object creation and are assigned to object’s fields. One field is storing **function** definition

```
function Student (name, bDay, bMonth, bYear)
{
        this.name = name;
        this.birthDate = new Date(bYear, bMonth, bDay);
        var today = new Date();

        this.age = today.getFullYear() -  bYear;
        this.toString = function ()  { return this.name + ": " + this.age;};
}
```

Reminder: constructor-based object declaration

//Creating a **new** Student object. Constructor function is executed. **Each object contains its own definition of toString method.**

```
function Student (name, bDay, bMonth, bYear)
{
        this.name = name;
        this.birthDate = new Date(bYear, bMonth, bDay);
        var today = new Date();

        this.age = today.getFullYear() -  bYear;
        this.toString = function ()   { return this.name + ": " + this.age;};
}

var student1 =  new Student ("Bob", 1,1,1991);
```

Class-level function definition: **prototype**

//Now toString function is not copied into each new object. It is **stored inside class definition** itself and is accessed when called

```
function Student (name, bDay, bMonth, bYear)
{
        this.name = name;
        this.birthDate = new Date(bYear, bMonth, bDay);
        var today = new Date();

        this.age = today.getFullYear() -  bYear;
        Student.prototype.toString = function ()
                { return this.name + ": " + this.age;};
}
```

Reminder: constructor-based object declaration

```
//Initializing university.
university.startUniversity (“VIU”, “Nanaimo”);

//Adding students to the University
var student1 =  new Student ("Bob", 1,1,1991);
var student2 =  new Student ("Margaret", 31,3,1989);

university. addStudent (student2);
university. addStudent (student1);

//Now printing
console.log (university.students);   //does not print as expected. This is
because console.log prints values of different types, not necessarily strings
//to force it to print strings – concatenate “”.
console.log (""+ university.students);
```

Anonymous functions (with no names)

//Passing a **comparison function** as a parameter to a sorting routine of an array. Function is defined in place.

```
university.students.sort
                    ( function (a, b) { return a.age - b.age;}  );
```

What problem do you see with using anonymous function declaration in this case?

The complete sample code:

html

JS

JavaScript and DOM

Lecture 3

Separate responsibilities

- HTML – structure
- CSS – style
- JavaScript - action

HTML document structure

```
<!DOCTYPE html>
<html>
      <head>
      </head>

      <body>
      </body>
</html>
```

Version of HTML: HTML5

HTML document structure

```
<!DOCTYPE html>
<html>
      <head>
      </head>

      <body>
      </body>
</html>
```

Metadata of a page: charset, tittle

HTML document structure

```
<!DOCTYPE html>
<html>
        <head>
        </head>

        <body>
        </body>
</html>
```

Displayed content

Reminder: HTML tags

<!-- -->	comments
<a href=“irule.html”> </a>	anchor for a hyperlink
<form> </form>	contains fields where user enters a data
<input type=“button” value=“Click me”/>	defines input control to a form
<p>	a new paragraph
<ul> </ul>	unordered list (bulleted)
<ol> </ol>	ordered list (numbered)
<li> </li>	list item inside both lists
<select> </select>	dropdown list
<option value=“1”> One </option>	option item inside select

DOM: in-memory object which represents HTML page

- HTML elements are nested inside each other

Closing tag

Opening tag

DOM: in-memory object which represents HTML page

- HTML elements are nested inside each other
- Nested elements are represented as child nodes of an enclosing element
- On top is the **document** object

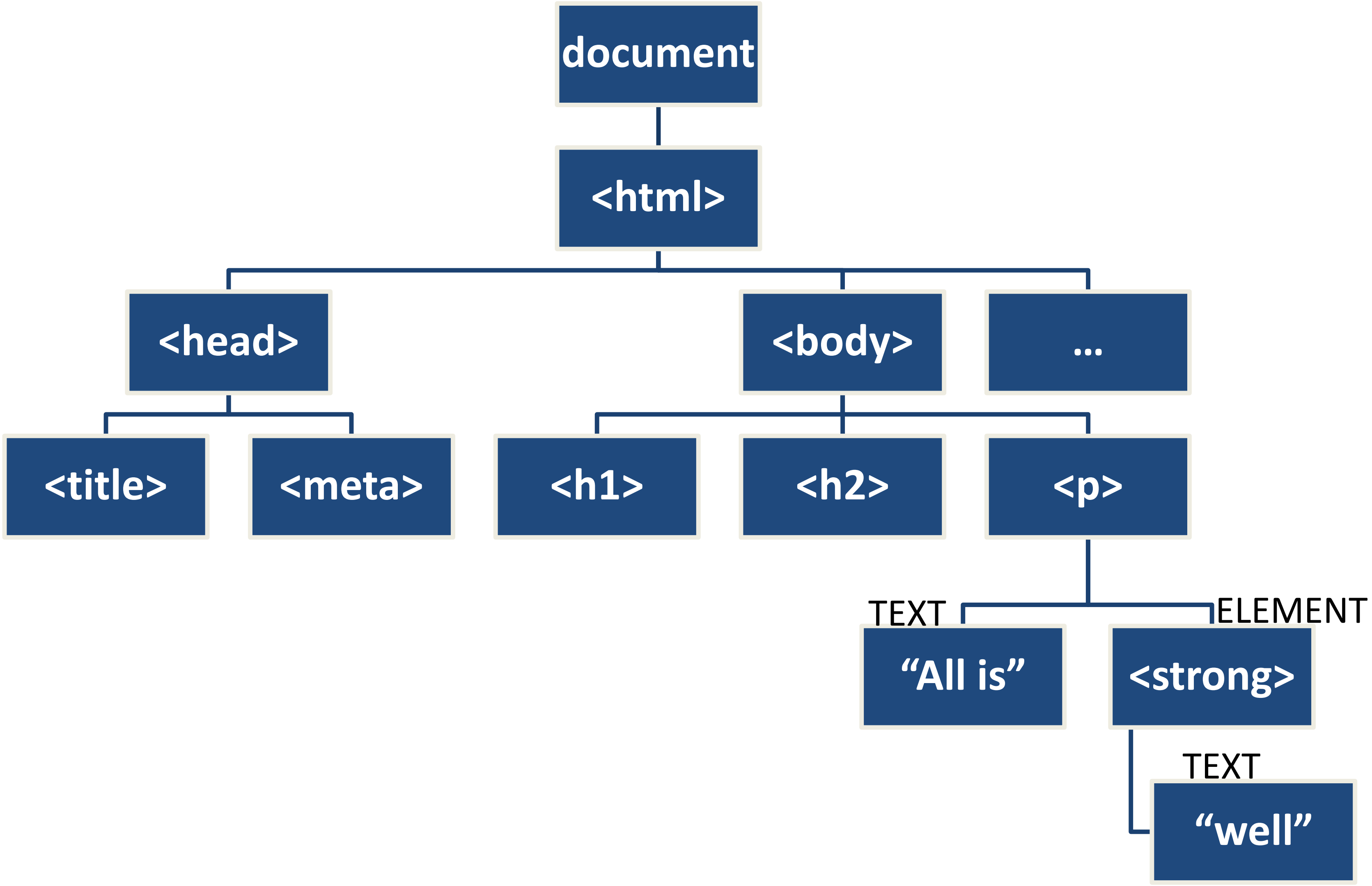
Creating DOM from HTML document

```
<!doctype html>
<html>
  <head>
        <title>Planets</title>
        <meta charset="utf-8">
  </head>
  <body>
        <h1>Planets diary </h1>
        <h2>Green Planet</h2>
        <p id="greenplanet">All is <strong> well </strong></p>

        <h2>Red Planet</h2>
        <p id="redplanet"><em>Nothing</em> to report</p>

        <h1>Blue Planet</h1>
        <p id="blueplanet">All systems A-OK</p>
  </body>
</html>
```

Sample DOM tree



DOM tree

- To traverse elements in JavaScript
- To understand Cascading Style Sheets:
  - style cascades down the tree until it is stopped by the declaration in a child element which overrides previous style,
  - and now this style cascades to all children of this node

The simplest way to change HTML element: getElementById

```
<script>
    var planet = document.getElementById("greenplanet");
    planet.innerHTML = “<strong>Red Alert</strong>: hit by phaser fire!";
</script>

<p id="greenplanet">All is <strong> well </strong></p>
```

Sample file: html

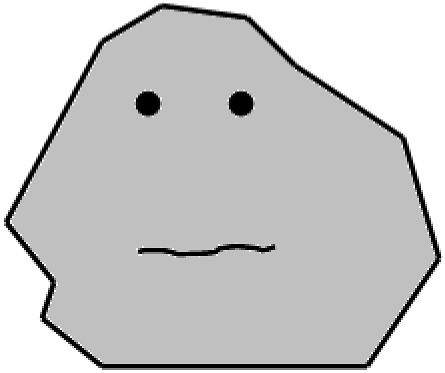
It does not work

Uncaught TypeError: Cannot set property 'innerHTML' of null

Corrected example: html

Reminder: Events

Simple two-way communication between the user and an artificial pet: iRock



img
innerHTML childElementCount firstChild
onclick ondblclick …
appendChild insertBefore setAttribute getAttribute

Img on click

```
<img id="rockImg" src="rock.png" alt="iRock" onclick="touchRock();" />

function touchRock() {
    userName = prompt("What is your name?",
                                                        "Enter your name here.");
    if (userName) {
        alert("It is good to meet you, " + userName + ".");
    }
    document.getElementById("rockImg").src ="rock_happy.png";
}
```

iRock V1

The iRock 1 is unrealistic

- He is in a consistent state of happiness
- The iRock 2 gets lonely if not touched for **10** seconds:

```
setTimeout("document.getElementById('rockImg').src = 'rock.png';", 10 *  1000);
```

iRock V2

Timed events

- The iRock 2 gets lonely if not touched for 10 seconds:

```
setTimeout("document.getElementById('rockImg').src = 'rock.png';", 10 *  1000);
```

iRock V2

1000 ms = 1 second

Timer events:

setTimeout (code,millisec);

setInterval (code,millisec);

The iRock 2 has a short memory

- When the browser is closed, all JavaScript variables are erased
- We want iRock to remember user’s name

Persistent data with Cookies

- When the browser is closed, all JavaScript variables are erased
- A Cookie is a piece of data stored on client computer: a persistent JavaScript variable

- Cookies are stored without involving a server

Cookies

- Each cookie is a name –value pair, plus an expiration date
- If an expiration date is not specified, cookie behaves like an ordinary JavaScript variable and gets destroyed when page is reloaded
- Cookies are stored as one long string associated with each server domain
- Each cookie is separated by semicolon

Writing a cookie

```
function writeCookie(name, value, days) {
  // By default, there is no expiration so the cookie is temporary
  var expires = "";

  // Specifying a number of days makes the cookie persistent
  if (days) {
    var date = new Date();
    date.setTime(date.getTime() + (days * 24 * 60 * 60 * 1000));
    expires = "; expires=" + date.toGMTString();
  }

  // Set the cookie to the name, value, and expiration date
  document.cookie = name + "=" + value + expires + "; path=/";
}
```

Reading a cookie

```
function readCookie(name) {
    // Find the specified cookie and return its value
    var searchName = name + "=";

    var cookies = document.cookie.split(';');

    for(var i=0; i < cookies.length; i++) {
        var c = cookies[i];
        while (c.charAt(0) == ' ')
            c = c.substring(1, c.length);
        if (c.indexOf(searchName) == 0)
            return c.substring(searchName.length, c.length);
    }

    return null;
}
```

What we have learned

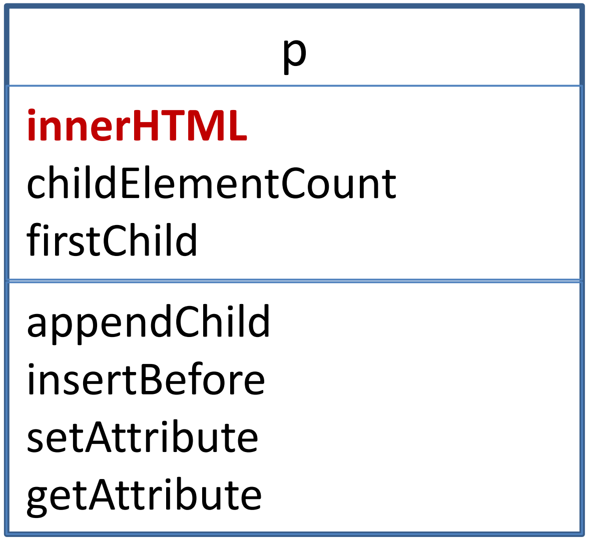
- How to enable timed events
- How to store values of JavaScript variables on client computer inside cookies

Final version with timer and cookies:

iRock V3

cookie.js

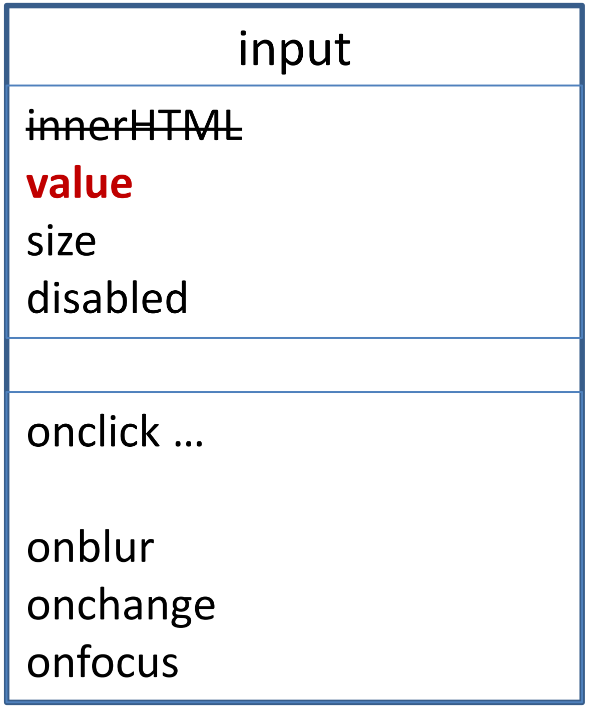
Markup elements vs. Form controls



p

innerHTML
childElementCount
firstChild

appendChild
insertBefore
setAttribute
getAttribute



input

~~innerHTML~~
value
size
disabled

onclick …

onblur
onchange
onfocus

Many ways of accessing form elements

- document.forms[0].elements[0];
- document.myForm.foo;
- document.getElementById('foo');
- document.getElementById('myForm').foo;

W3C recommended

Example of working with form controls

Full sample code: link

What we have learned:

- How to get access to form elements and read their values
- Set up event listener
- Change value of a form control from JavaScript

How to bake your very own DOM with JavaScript

- Changing elements
- Adding elements
- Removing elements

Our first "WEB application"

Playlist manager

Plan

1. Create HTML page: input field to write a song name, and a button "add song"
2. Set up a handler to handle user's click
3. Write the handler
4. Create a new element to hold a song
5. Add new element to the page's DOM

1. HTML page

```
<form>
 <label for="songTextInput">Song name</label>
 <input type="text" id="songTextInput" size="40">
 <input type="button" id="addButton" value="Add Song">
</form>

<ul id="playlist">
</ul>
```

2. Handler for “addButton”

```
var button = document.getElementById("addButton");
button.addEventListener ('click', addSong ,false);
```

3. Handler function: addSong

```
function addSong(){
   var textInput = document.getElementById("songTextInput");
   var songName = textInput.value;


…
}
```

4. Create element: list item “li”

```
if (songName){
        var li = document.createElement("li");
        li.innerHTML = songName;
 ….
}
```

5. Add element to DOM

```
<ul id="playlist">
</ul>
```

```
if (songName){
        var li = document.createElement("li");
        li.innerHTML = songName;
        var ul = document.getElementById("playlist");
        ul.appendChild(li);
}
```

Sample code: link

Adding persistence to Song list

- HTML5 Web storage API gets you a local storage of up to 5-10 MB (for each domain), instead of 4K in Cookies
- Your app can store data in the browser, reducing communication with the server

- Local storage is a set of key-value pairs (both in the form of strings)
- Local storage is available through the ***localStorage*** object

HTML5 web storage API

```
function save(item) {
    var localStorageList = localStorage.getItem("playlist");
    var playlistArray =  JSON.parse (localStorageList );
    playlistArray.push(item);
    localStorage.setItem("playlist", JSON.stringify(playlistArray));
}
```

Persistent song list: link

What we have learned

- Interaction with DOM
- Local storage