Final Review

Returning a value

- A function can return a value back into the calling code as the result.
- The directive `return` can be in any place of the function.
  - When the execution reaches it, the function stops, and the value is returned to the calling code.
  - There may be many occurrences of `return` in a single function.
  - It is also possible to use `return` without a value. That causes the function to exit immediately.
- A function with an empty `return` or without it returns `undefined`

```
function oddEven(num){
  if (!num) return;
  if(num%2==0) return "Even";
  else return "Odd"
}
```

Most important

- The focus of the exam will be on programming with functions and arrays
- Also strings and basic objects
- There might be some questions involving string and array methods, but that will not be an emphasis
- Similarly there will be some on html, css, event handling, but not an emphasis
- Know all the RED topics

Key concept topics

- MAIN focus: programming with arrays, functions, strings, objects
- scope
- equality of objects and arrays
- push and pop
- multi d array
- value and reference
- recursion
- block vs inline
- DOM
- getelementbyid
- css id and class selectors
- reference vs function call and passing function vs calling fn

Local variables

- A variable declared inside a function is only visible inside that function.

```
function showMessage() {
  let message = "Hello, I'm JavaScript!"; // local variable
  alert( message );
}
showMessage(); // Hello, I'm JavaScript!
alert( message ); // <-- Error! The variable is local to the function
```

Scope revisited

- The scope of a variable determines how long and where a variable can be used.
- With let and const JavaScript has block scope
  - Parameters are local to a function.
- let and const → block scope.
- See example: *lecture_codes/lesson5/scopes.js*
  - which lines will cause errors, why?

Lexical scope in JavaScript (ES6+)

- From ES6, in JavaScript every block ({}) defines a scope
  - Via let and const

```
let x = 10;

function main() {
 let x;
 console.log("x1: " + x);
 if (x > 0) {
   let x = 30;
   console.log("x2: " + x);
 }

 x= 40;
 let f = function(x) { console.log("x3: " + x); }
 f(50);
}

main();
```

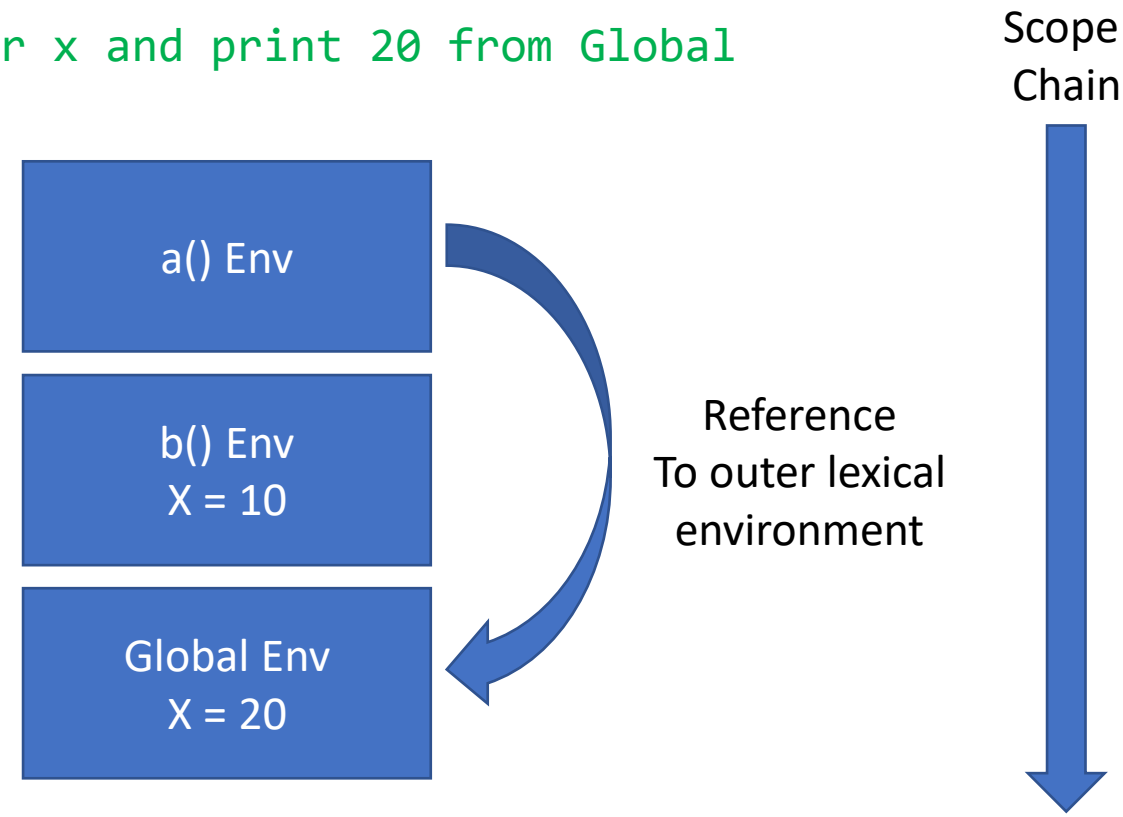
Global Scope

Block Scope

Block Scope

Scope Example

```
function a(){
        console.log(x); // consult Global for x and print 20 from Global
}

function b(){
        let x = 10;
        a(); // consult Global for a
        console.log(x);
}

let x = 20;
b();
```



Function expression & Anonymous Function

- The syntax that we used before is called a *Function Declaration*
- There is another syntax for creating a function that is called a *Function Expression*.
  - A function keyword can be used to define a function inside an expression

```
// function expression
let sayHi = function(){console.log("Hi");};
sayHi();
```

  - In JavaScript, a function is a value, so we can deal with it as a value.

- Function without a name is called *anonymous* function.

The execution context and stack

- The information about the process of execution of a running function is stored in its *execution context*.
- The execution context is an internal data structure that contains details about the execution of a function: most importantly the current variables the function is using.
- One function call has exactly one execution context associated with it.
- When a function makes call to another function, the following happens
  - The current function is paused
  - The execution context associated with it is remembered in a special data structure called *execution context stack*.
  - Called function executes
  - After it ends, the calling function is resumed with prior saved *execution context*.

Declaring an Array

- Using array literal syntax

```
const numbers = [];
const fruits = ["Apple", "Banana", "Mango"];
```

- Array being an object type can also be created using new keyword

```
const numbers = new Array(6);
```

- Almost all the time, literal syntax is use.

Using an Array

- Array elements are numbered, starting with zero.
- We can get an element by its number in square brackets:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

- We can replace an element:

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

- Or add a new one to the array:

```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

Size of an array

- In JavaScript, arrays have built-in property, `length`; which represents the current size of the array.
- The total count of the elements in the array is its `length`

```
let numbers = []
console.log(numbers.length); // 0
numbers = [1,2,3];
console.log(numbers.length) // 3
```

Looping through an array elements

- One of the oldest ways to cycle array items is the for loop over indexes:

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

- But for arrays there is another form of loop, `for..of`:

```
for (let fruit of fruits) {
  alert( fruit );
}
```

- The for..of doesn't give access to the index of the current element, just its value, but in most cases that's enough.
  - And it's shorter.
  - And avoids bugs that often occur from index errors at the end points
  - Favor for..of as default loop over arrays unless really need index

Array comparison

- Arrays are type Object
- When == or === operators are used on JavaScript objects, their references are compared

- **If array comparison is needed compare them item-by-item in a loop**.
  - Mocha has a very convenient assert.deepStrictEqual

Add/Remove elements To/From the end

- Array in JavaScript has inbuilt methods that allow you to add/remove elements to/from the end of the array.
  - `pop`: extracts the last element of the array and return it.

```
let fruits = ["Apple", "Orange", "Pear"];
console.log( fruits.pop() ); // remove "Pear" and log it
console.log( fruits ); // Apple, Orange
```

  - `push`: append element to the end of the array.

```
let fruits = ["Apple", "Orange"];
fruits.push("Pear");
console.log( fruits ); // Apple, Orange, Pear
```

  - The call `fruits.push(...)` is equal to `fruits[fruits.length] = ...`

splice

- The syntax is:

```
arr.splice(start[, deleteCount, elem1, ..., elemN])
```

- The `arr.splice` method is a swiss army knife for arrays.
  - It can remove/ replace array elements.
  - where to start
  - how many to delete
  - elements to insert
- insert elements without any removals.
  - set `deleteCount` to 0:
- Negative start means position from end of array
- See examples: *lecture_codes/arrays/splice.**

`arr.slice([start], [end])`

- It returns a new array copying all items from index `start` to end (not including end).
  - Both `start` and end can be negative, in that case position from array end is assumed.

- We can also call it without arguments: `arr.slice()` creates a copy of arr.
  - That's often used to obtain a copy for further transformations that should not affect the original array.

- See example: *lecture_codes/arrays/slice_demo*

Callback functions (revisited)

- A callback is a function passed as an argument to another function.

```
function myDisplayer(result) {
    console.log(`Result of the calculation is ${result}`);
}

function myCalculator(num1, num2, myCallback) {
    let sum = num1 + num2;
    myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
```

Multidimensional arrays

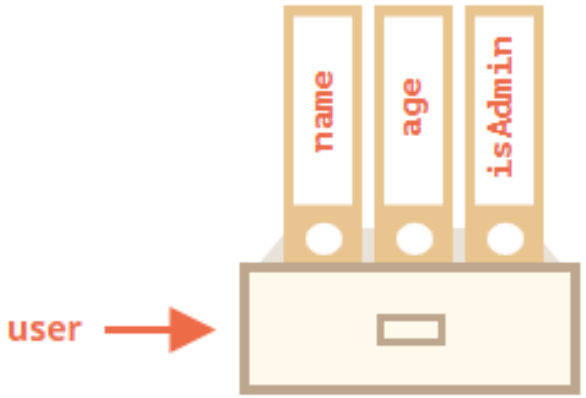
- Arrays can have items that are also arrays.
  - We can use it for multidimensional arrays, for example to store matrices:

```
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
  ];

  alert( matrix[1][1] ); // 5, the central element
```

Literals and properties

- We can immediately put some properties into `{...}` as “key: value” pairs:

```
let user = {           // an object
    name: "John",      // key "name", value "John"
    age: 30,           // key "age",  value 30
    isAdmin: true,     // key "isAdmin", true
};
```



Add and Remove properties

- In JavaScript, properties of an object can also be added and removed at the runtime.
  - Add syntax is like set syntax, except we use a new property name

```
user.id = 123; // adding a new property id in an exiting user object
```

  - To remove a property, we can use `delete` operator

```
delete user.age;
```

Value type vs Reference type

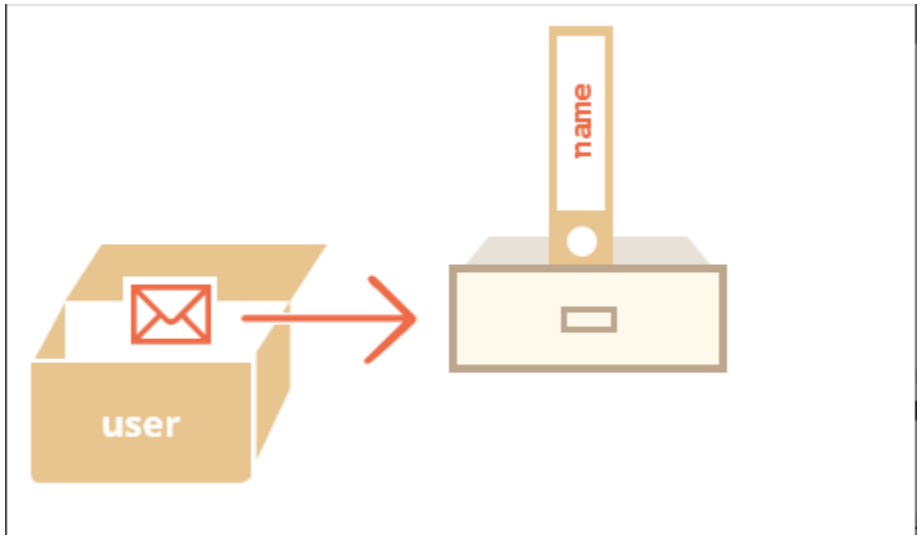
- fundamental difference of objects versus primitives
  - objects are stored and copied “by reference”,
  - primitive values: strings, numbers, booleans, etc. – always copied “as a whole value”.

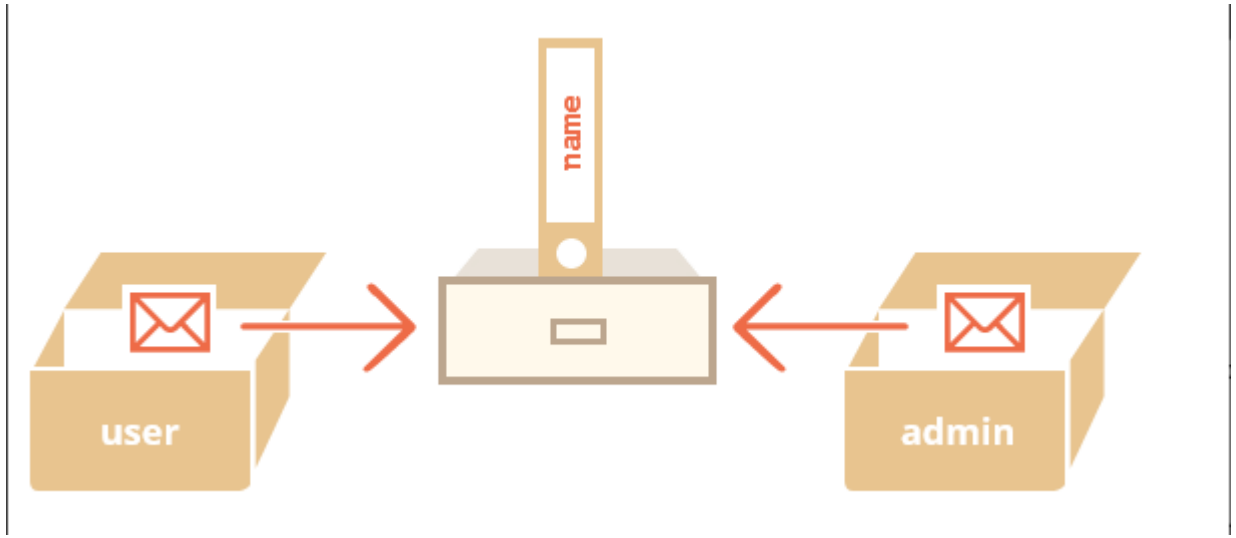
```
let message = "Hello!";
let phrase = message; // second copy of "Hello!";
message = "Hi!";
console.log(phrase); // Hi! or Hello! ??

let user = {id:123, name: "user"};
let admin = user; // there is still single copy of the object
admin.name= "admin";
console.log(user.name); // user or admin ??
```

Value type vs Reference type


```
let message = "Hello!";
let phrase = message;
```

```
let user = { name: 'John' }; let admin = user;
```





Comparison by reference

- Two objects are == / === only if they are the same object.

```
let a = {};
let b = a; // copy the reference

alert( a == b ); // true, both variables reference the same object
alert( a === b ); // true
```

- Two independent objects are not == / ===, even though they may look identical.

```
let a = {};
let b = {}; // two independent objects

console.log(a == b); // false
console.log(a === b); //false
```

Strings are immutable

- Strings can’t be changed in JavaScript.
  - It is impossible to change a character.

```
let str = 'Hi';
str[0] = 'h'; // doesn't work
console.log( str[0] ); // H
```

  - The usual workaround is to create a whole new string and assign it to the original variable.

```
let str = 'Hi';
str = 'h' + str[1]; // replace the string
alert( str ); // hi
```

String methods (APIs)

- JavaScript includes several useful string methods

```
let str = "Hello";

console.log(str.indexOf("l")); // 2
console.log(str.indexOf("ell")); // 1
console.log(str.toUpperCase()); // HELLO
console.log(str.toLowerCase()); // hello
console.log(str.startsWith("H")); // true
console.log(str.substr(1,3)); //ell
console.log(“llo”)); // true
console.log(“I am mighty”.split(“ ”)); // [“I”, “am”, “mighty”]
```

- https://www.w3schools.com/jsref/jsref_obj_string.asp

Iteration vs Recursion (two ways of thinking)

- Any problem that can be solved using recursion can also be solved using iteration, loops.

- In typical JavaScript implementations, recursive solutions are about three times slower than its iterative version.
  - But in some situations, recursive solutions are much more elegant (shorter, simpler and clearer) than the iterative ones.

The Base Case and Reduction Step

- To stop recursion going into an infinite recursion (and exceed the max recursion depth)
  - Reduction step: We need to make sure that each recursive call moves us closer to a base case
  - Base case: returns without calling itself

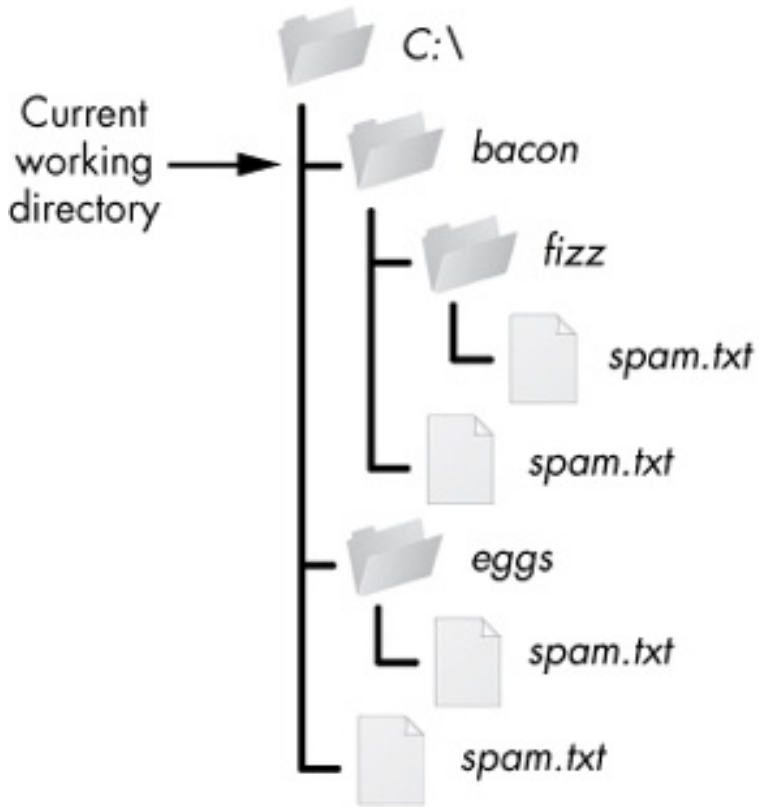
- Recursion creates stack frames on the call stack until the base case
  - Then it comes back down through the frames

Structure of an HTML5 page

- The **`header`** describes the page, and the **`body`** contains the page's contents
  - An HTML page is saved into a file ending with extension **`.html`**
- **`DOCTYPE`** html tag tells browser to interpret our page's code as HTML5.
- HTML is case insensitive, but we follow conventions.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title></title>
    </head>
    <body>
        page contents
    </body>
</html>
```

Relative vs absolute path

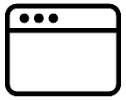


	Relative Paths	Absolute Paths
C:\	..\	C:\
bacon	.\	C:\bacon
fizz	.\fizz	C:\bacon\fizz
spam.txt	.\fizz\spam.txt	C:\bacon\fizz\spam.txt
spam.txt	.\spam.txt	C:\bacon\spam.txt
eggs	..\eggs	C:\eggs
spam.txt	..\eggs\spam.txt	C:\eggs\spam.txt
spam.txt	..\spam.txt	C:\spam.txt

Unordered list: `<ul>`, `<li>`

- `ul` represents a bulleted list of items (block)
- `li` represents a single item within the list (block)


```
<ul>
   <li>No shoes</li>
   <li>No shirt</li>
   <li>No problem!</li>
</ul>
```



- No shoes
- No shirt
- No problem!

Block vs Inline elements

- A block-level element always starts on a new line and takes up the full width available (stretches out to the left and right as far as it can). e.g., <p>, <h1>
- An inline element does not start on a new line and only takes up as much width as necessary. e.g., <input/>, <textarea>
  - Need to use line break **
** to move to the new line.

`<script>` element

- The `<script>` element is used to embed JavaScript codes.
- It can go anywhere in the HTML page, but by convention it is placed in the head section.

window

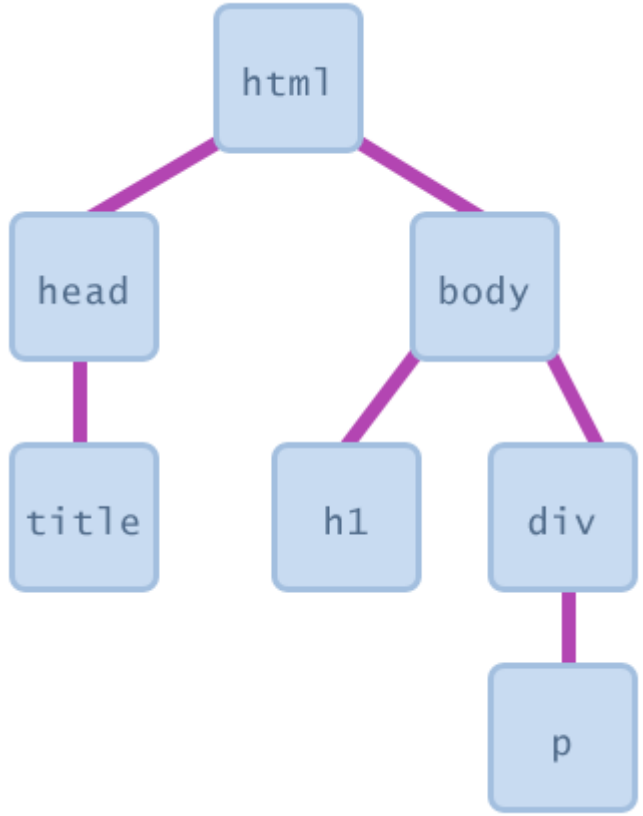
- When JavaScript runs on a browser, it runs inside the global environment called window (object).
  - `alert()` and `prompt()` are methods (functions) of window object for alerting output and displaying prompt for user input.

HTML Event Attributes

```
<button onclick = "doSomething()">Do it</button>


<script>
     function doSomething(){
          // code to do something.
     }
</script>
```

Document Object Model (DOM)

- All HTML elements are represented in browsers as objects
- All objects are nested together in one tree (DOM tree)
- Elements can have parents, siblings and children
- Most JS code manipulates elements (objects) on the DOM
  - it can change state (insert some new text into a span)
  - it can change styles (make a paragraph red)



HTML element's id attribute

- The id attribute specifies a unique identifier for a HTML element (the value must be unique within the HTML document)
- The id attribute is used to target elements in JavaScript (via. the HTML DOM)

Getting a DOM element using its id

- `document.getElementById("id")`
  - Get the element with the specified id.

- Example, Program to get first name and last name from the input fields and display full name inside span element on button click.

JavaScript in a separate file

- JS code can be placed directly in the HTML file's body or head
  - but this is not a good practice.
- script code should be stored in a separate .js file
  - script tag in HTML should be used to link the .js files

```
<script src="filename" type="text/javascript"></script>
```

- When more than one script file is included
  - interpreter treats them as a single file;
  - share global context.
  - order in which file files are loaded matters
    - interpreter executes code as soon as it hits the `<script>` tag.

Basic CSS rule syntax

- A **CSS** file consists of one or more rules
- A rule's selector specifies HTML element(s) and applies style properties
  - The * selector, selects all elements
  - To add a comment we use: /* */

```
selector {
  property: value;
  property: value;
  ...
}
```


```
p {
  font-family: sans-serif;
  color: red;
}
```

The HTML `class` and `id` attribute

- **`id` attribute** allows you to give a unique ID to any element on a page
  - Each ID must be unique; can only be used once in the page
- **`class` attribute** is used to group some elements and give a style to only that group
  - unlike an id, a class can be reused as much as you like on the page

Attach event handler to DOM element

```
// where element is a DOM element object
element.onevent = function; // syntax structure

<button id="ok">OK</button>

const okButton = document.getElementById("ok");
okButton.onclick = okayClick;
```

- good style to attach event handlers to DOM objects in your JavaScript code
  - notice that you do **not** put parentheses after the function's name

- Where should we put the above JS code?

Common unobtrusive JS errors

- many students mistakenly write () when attaching the handler

```
window.onload = pageLoad(); //what will happen?
window.onload = pageLoad;
okButton.onclick = okayClick();
okButton.onclick = okayClick;
```

  - IMPORTANT FUNDAMENTAL CONCEPT !!!
    - Function reference versus evaluation

- event names are all lowercase, not capitalized like most variables

```
window.onLoad = pageLoad;    //what will happen?
window.onload = pageLoad;
```

Common Timer Errors

```
function multiply(a, b) {
      alert(a * b);
}

setTimeout(hideBanner(), 5000); // what will happen?
setTimeout(hideBanner, 5000);

setTimeout(multiply(num1 , num2), 5000);
setTimeout(multiply, 5000, num1, num2);
```