Javascript

Javascript History

- First web scripting language
  - First appeared in 1995!
- Developed by Netscape
  - Initiated by Netscape and called LiveScript
  - Sun was developing Java and gained vast popularity
    - Netscape changed LiveScript name to Javascript hoping to gain a boost in popularity
- "NETSCAPE AND SUN ANNOUNCE JAVASCRIPT, THE OPEN, CROSS-PLATFORM OBJECT SCRIPTING LANGUAGE FOR ENTERPRISE NETWORKS AND THE INTERNET" – December 4, 1995

Introduction

- Client-side scripting enhances functionality and appearance
- **efficiency**: can modify a page without having to post back to the server (faster UI) ie form validation
- **customization**: can make small, quick changes to page based on user preferences and inputs
- **event-driven**: can respond to user actions like clicks and key presses
- Get information from user's machine ie. Browser type
- **Universality**: Browser has to have a built-in (JavaScript) interpreter
- Foundation for complex server side scripts

JavaScript: Object-Based Language

There are three object categories in JavaScript:

- **Native objects**: defined by JavaScript.
  - String, Number, Array, Image, Date, Math, etc.
- **Host objects** : supplied and always available to JavaScript by the browser environment.
  - window, document, forms, etc.
- **User-defined objects** : defined by the author/programmer

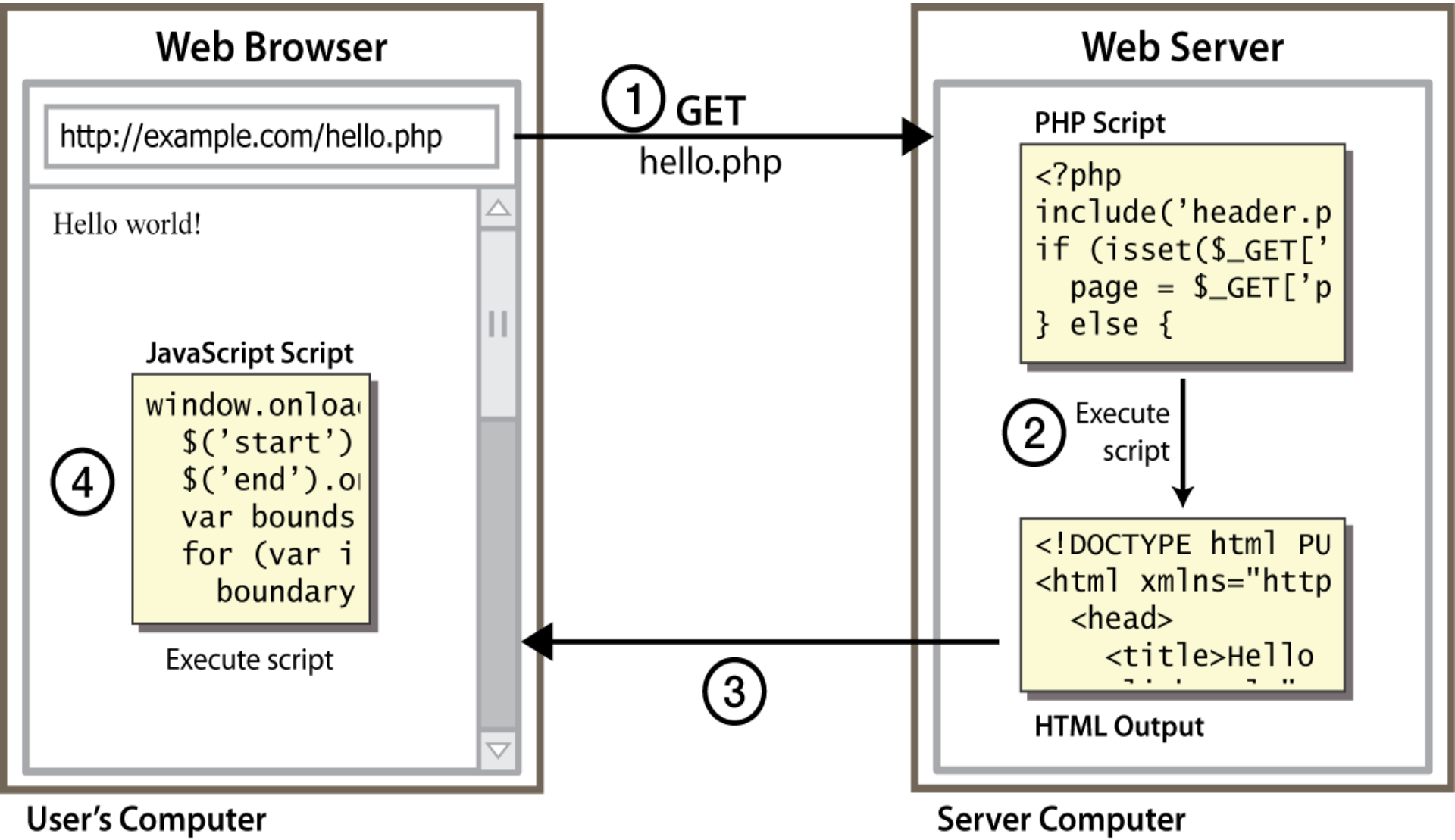
Javascript Characteristics

- **Universal Support**
  - All web browsers have built-in Javascript interpreters.
- **Dynamic typing** (untyped)
  - weakly or loosely typed.
- **Structured**
  - Supports if statements, while loops, switch statements, do while loops, etc.
- **Prototype-based**
  - Functions as object constructors
  - Functions as methods

Writing Javascript

- **Inline:**
  - Done inside HTML code itself.
  - <button onclick="**createParagraph()**">Click me!</button>
- **Internal:**
  - Inside **<script> … </script>** block
  - Can be written anywhere in the HTML file.
  - Can be done using multiple script blocks.
- **External**
  - Used to “import” Javascript files
  - Syntax: <script **src="myScript.js"**></script>
  - Can reference a URL:
    - <script src="https://www.someURL.com/js/myScript1.js"></script>

Client Side Scripting



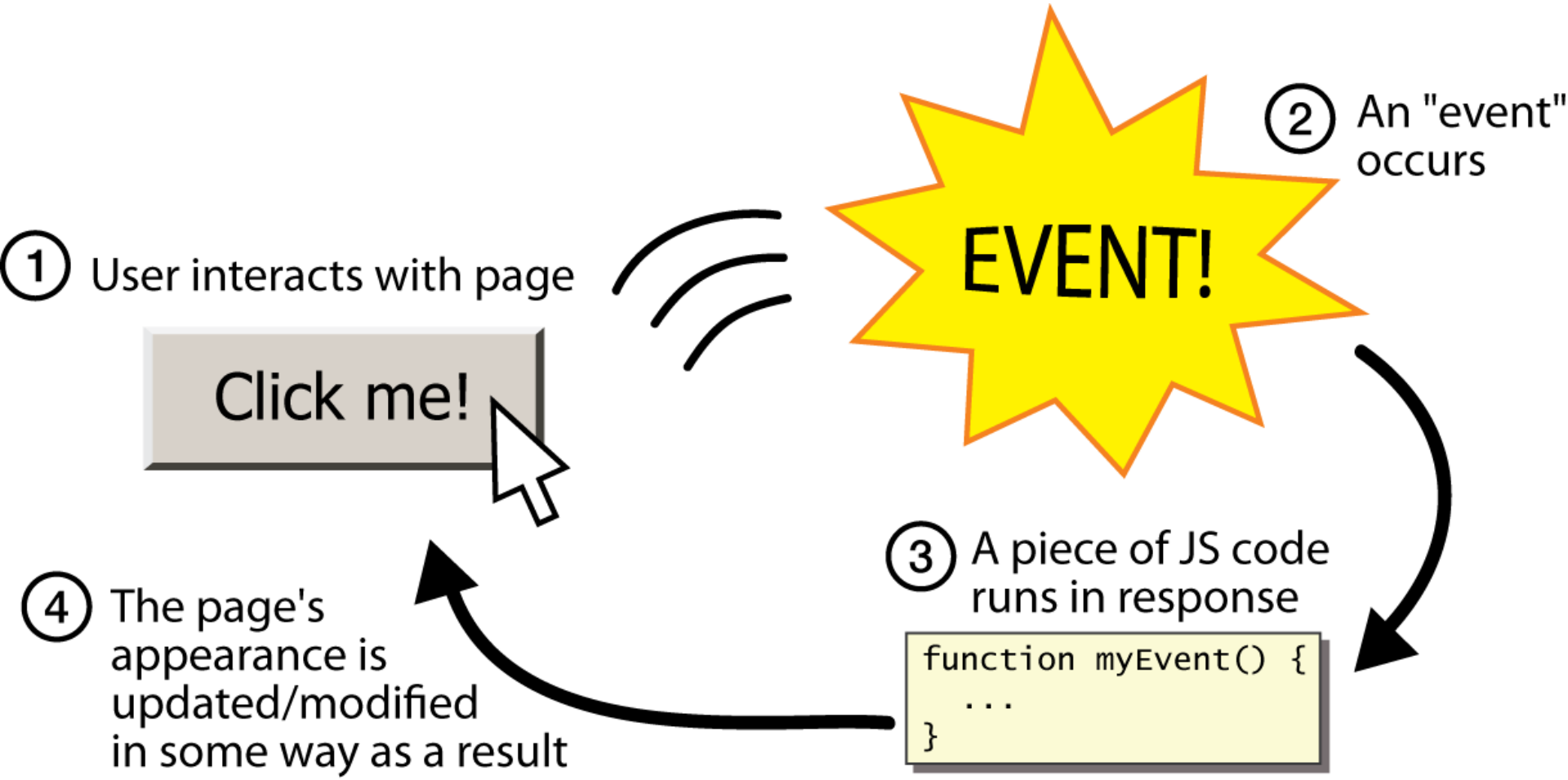
What is Javascript?

- a web standard (but not supported identically by all browsers)
- NOT related to Java other than by name and some syntactic similarities

Javascript vs Java

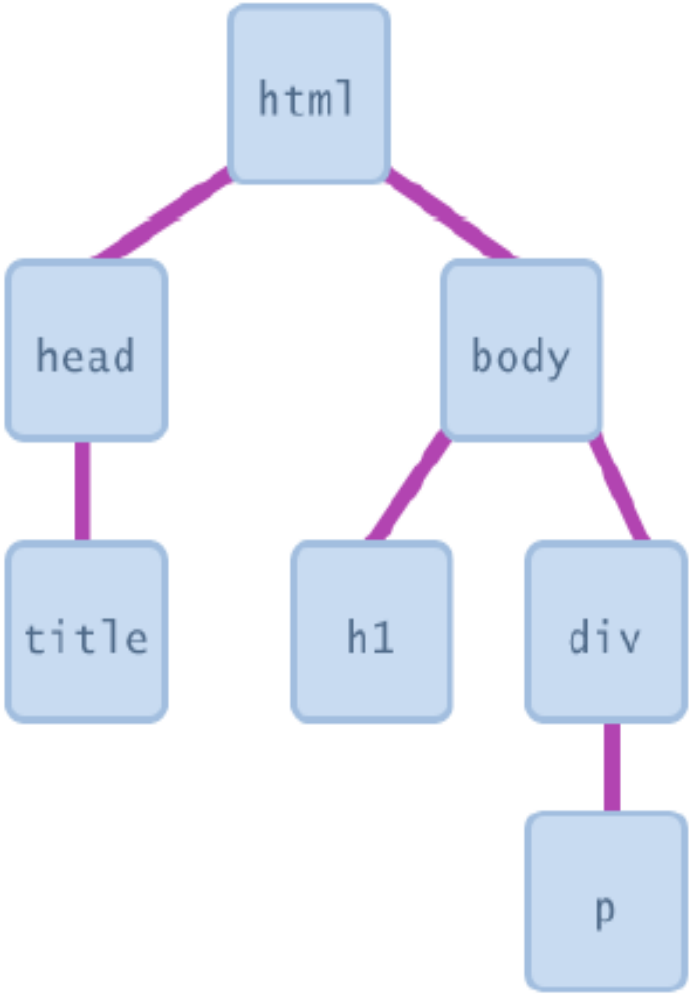
- interpreted, not compiled
- more relaxed syntax and rules
  - fewer and "looser" data types
  - variables don't need to be declared
  - errors often silent (few exceptions)
- key construct is the function rather than the class
  - "first-class" functions are used in many situations
- contained within a web page and integrates with its HTML/CSS content

Event-driven programming



Document Object Model (DOM)

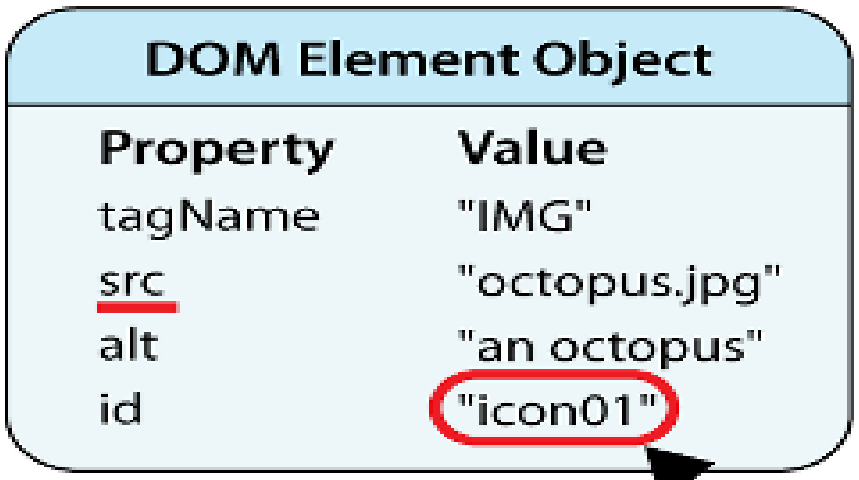
- most JS code manipulates elements on an HTML page
- we can examine elements' state
  - e.g. see whether a box is checked
- we can change state
  - e.g. insert some new text into a div
- we can change styles
  - e.g. make a paragraph red



DOM element objects

HTML

```
<p>
  Look at this octopus:
  <img src="octopus.jpg" alt="an octopus" id="icon01" />
  Cute, huh?
</p>
```



JavaScript

```
var icon = document.getElementById("icon01");
icon.src = "kitty.gif";
```

Accessing elements: Preetify


```
var name = document.getElementById("id");
```
JS

```
<button onclick="changeText();">Click me!</button>
<span id="output">replace me</span>
<input id="textbox" type="text" />
```
HTML

```
function changeText() {
        var span = document.getElementById("output");
        var textbox = document.getElementById("textbox");

         textbox.style.color = "red";

        // font styles added by JS:
        text.style.fontSize = "13pt";
        text.style.fontFamily = "Comic Sans MS";
        text.style.color = "red"; // or pink?

}
```
JS

Variables


```
var name = expression;
var a = {x:3, y:4, name:"Tommy"};
```

```
var clientName = "Connie Client";
var age = 32;
var weight = 127.4;
```

JS

- variables are declared with the var keyword (case sensitive)
- types are not specified, but JS does have types ("loosely typed")
  - `Number, Boolean, String, Array, Object, Function, Null, Undefined`
  - can find out a variable's type by calling `typeof`

Number type and comment


```
var enrollment = 99;
var medianGrade = 2.8;
var credits = 5 + 4 + (2 * 3);
```
JS

- integers and real numbers are the same type (no int vs. double)
- same operators: + - * / % ++ -- = += -= *= /= %=
- many operators auto-convert types: "2" * 3 is 6
- **Comment**

```
// single-line comment
/* multi-line comment */
```
JS

Math object


```
var rand1to10 = Math.floor(Math.random() * 10 + 1);
var three = Math.floor(Math.PI);
```

JS

- **methods:** abs, ceil, cos, floor, log, max, min, pow, random, round, sin, sqrt, tan
- **properties:** E, PI

Special values: null and undefined


```
var ned = null;
var benson = 9;
// at this point in the code,
// ned is null
// benson's 9
// caroline is undefined
```

JS

- `undefined` : has not been declared, does not exist
- `null` : exists, but was specifically assigned an empty or null value
- Why does JavaScript have both of these?

Logical operators

- > < >= <= && || ! == != === !==
- most logical operators automatically convert types:
  - 5 < "7" is true
  - 42 == 42.0 is true
  - "5.0" == 5 is true
- === and !== are strict equality tests; checks both type and value
  - "5.0" === 5 is false

if/else statement (same as Java)


```
if (condition) {
        statements;
} else if (condition) {
        statements;
} else {
        statements;
}
```

JS

JavaScript allows almost anything as a condition

Boolean type


```
var iLike190M = true;
var ieIsGood = "IE6" > 0; // false
if ("web devevelopment is great") { /* true */ }
if (0) { /* false */ }
```

JS

- any value can be used as a Boolean
  - "falsey" values: 0, 0.0, NaN, "", null, and undefined
  - "truthy" values: anything else
- converting a value into a Boolean explicitly:
  - `var boolValue = Boolean(otherValue);`
  - `var boolValue = !!(otherValue);`

Loops


```
var sum = 0;
for (var i = 0; i < 100; i++) {
        sum = sum + i;
}
```
JS

```
while (condition) {
        statements;
}
```
JS

```
do {
    statements;
} while (condition);
```
JS

Display possibilities


```
alert("message"); // message
confirm("message"); // returns true or false
prompt("message"); // returns user input string

document.write ("message");
console.log ("message");
document.getElementById("id").innerHTML = "message";
```

```
document.write ("<h1> hello world!</h1>");
document.write ("<h3> hello world!</h3>");
document.write ("<h3> hello world!</h3>");
document.write ("<h3> hello world!</h3>");
document.write ("<h3> hello world!</h3>");

document.getElementsByTagName("h3")[2].innerHTML = "<h6>hi
there</h6>";
```

Arrays


```
var name = []; // empty array
var name = [value, value, ..., value]; // pre-filled
name[index] = value; // store element
```

JS

```
var ducks = ["Huey", "Dewey", "Louie"];
var stooges = []; // stooges.length is 0
stooges[0] = "Larry"; // stooges.length is 1
stooges[1] = "Moe"; // stooges.length is 2
stooges[4] = "Curly"; // stooges.length is 5
stooges[4] = "Shemp"; // stooges.length is 5
```

JS

Array methods


```
var a = ["Stef", "Jason"]; // Stef, Jason
a.push("Brian"); // Stef, Jason, Brian
a.unshift("Kelly"); // Kelly, Stef, Jason, Brian
a.pop(); // Kelly, Stef, Jason
a.shift(); // Stef, Jason
a.sort(); // Jason, Stef
a.reverse(); //Stef, Jason
a.splice(0, 1, "Mike", "Jack");   //"Mike", "Jack", " Stef"
```

- Array serves as many data structures: list, queue, stack, ...
- **methods:** `concat, join, pop, push, reverse, shift, slice, sort, splice, toString, unshift`
  - push and pop add / remove from back
  - unshift and shift add / remove from front
  - shift and pop return the element that is removed

String type


```
var s = "Connie Client"; //double quotes
var fName = s.substring(0, s.indexOf(" ")); // "Connie"
var len = s.length; // 13
var s2 = 'Melvin Merchant'; //single quotes
```

JS

- methods: `charAt, charCodeAt, fromCharCode, indexOf, lastIndexOf, replace, split, substring, toLowerCase, toUpperCase`
- length property (not a method)
- charAt returns a one-letter String (there is no char type)
- Strings can be specified with single or double quote
- concatenation with + : "1" + 1 is "11"

More about `String`

- escape sequences: \' \" \& \n \t \\
- converting between numbers and Strings:

```
var count = 10;
var s1 = "" + count; // "10"
var s2 = count + " bananas, ah ah ah!"; // "10 bananas, ah
ah ah!"
var n1 = parseInt("42 is the answer"); // 42
var n2 = parseFloat("booyah"); // NaN
```
JS

- accessing the letters of a String:

```
var firstLetter = s[0]; // fails in IE
var firstLetter = s.charAt(0); // does work in IE
var lastLetter = s.charAt(s.length - 1);
```
JS

Splitting strings: split and join


```
var s = "the quick brown fox";
var a = s.split(" "); // ["the", "quick", "brown", "fox"]
a.reverse(); // ["fox", "brown", "quick", "the"]
s = a.join("!"); // "fox!brown!quick!the"
```

JS

- split breaks apart a string into an array using a delimiter
- join merges an array into a single string, placing a delimiter between them
- Can use regular expression for splitting the string.

```
var s = "It is a beautiful day.\nrekha\tsinghal";
var a = s.split(/[\s\t\n]/);
s = a.join("---");
```

JS

Objects

Objects in JS

```
var a = {x:3, y:4, name:"Jack"};
Var b = {};     //empty object
```

JS

```
a.x                              //accessing the field
a.x = 5;                         //setting the field
a['x'] = 5;
For (i in a) ....                //iterating through the fields of object
var a = {x:3, get y() {return "x prop is="+ x;}, set z(x) {this.x = x + 100;}
a.Y            //getter
a.z = 5;       //setter
a.y            //getter
delete a.x;  //deleting a field
```

Methods

- this is a lexical identifier
- Not declared but bound to an object
- **this** is the first implicit paramter to any function**.**

```
// this is the first implicit paramter to any function.
var dist = function(){
   return this.x + this.y;
}
```
JS

```
//syntactic variant
function dist(){
  return this.x + this.y;
}
```
JS

```
//installing a method in an object vrs. applying a function on an object
a.foo = dist;
a.foo();
dist.apply(a));
```

Functions

Function - first class object

Function

- is an instance of Object type
- can have properties
- can have methods
- can be stored in a variable
- can be passed as parameter
- can be returned from a function

```
//function is an object in Javascript
var f = function(x, y) {return x() + y;} //passing an function x as an argument
var g= function(){return 7;}
f(g, 3);   //calling function f with another function g and number 3
f.xyz = 10; //setting property of function f as it is an object
```

constructor

- Constructor in JS is a function that is used to populate the fields of an existing object
- Naming convention : start the name of constructor with capital letter
- Constructor is passed to the new operator which constructs new object and calls constructor to populate/initialize the object.
- Every object has constructor property set to the constructor of the object.

```
var Point = function(x, y){
   this.x = x;
  this.y =y;
}
var a = new Point(2, 3);
a.constructor === Point;
a.constructor.toString();
```

this

- When JS is run in browser, at top level ***this*** is bound to window object.
- When JS is run in a standalone interpreter like node, ***this*** is bound to a global object.
- When used as constructor, ***this*** is bound to newly created object
- When function is bound to an object, then ***this*** points to the binding object.
- When function is installed as an object method, ***this*** points to the host object.

Object binding


```
var a = new Point(3, 4);  //{x:3, y:4}
var b = new Point(-1, -4); //{x:-1, y:-4}
```

```
a.foo = dist;
a.foo(); //7
a.foo.apply(b); //-5 here a is ignored
dist.apply(a); //7

dist.apply(b); //-5
```

```
a.foo.bind(b)(); //-5, here a is ignored
dist.bind(a)(); //7
dist.bind(b)(); //-5
var b_dist = dist.bind(b); //new bound dist
b_dist(); //-5
b.y = -20;

b_dist(); //-21
```

//Can you guess the answer here?

```
b_dist().apply(a);
```

Changing the object context: *this*


```
var a = {x:2, y:3};
var dist2(b){return this.x + b;}
dist2.call(a, -1, -2);      //passing the actual args to function
dist2.apply(a, [2,3]); //arguments are packed in an array
```

```
var hello(){alert(”hello: ” + this);}
Hello.call(new Date());
Hello.call(window);
Hello();
Hello.call(”JavaScript”)
Hello.call([”foo”, ”bar”, 3])
```

Block Scoping: var and let

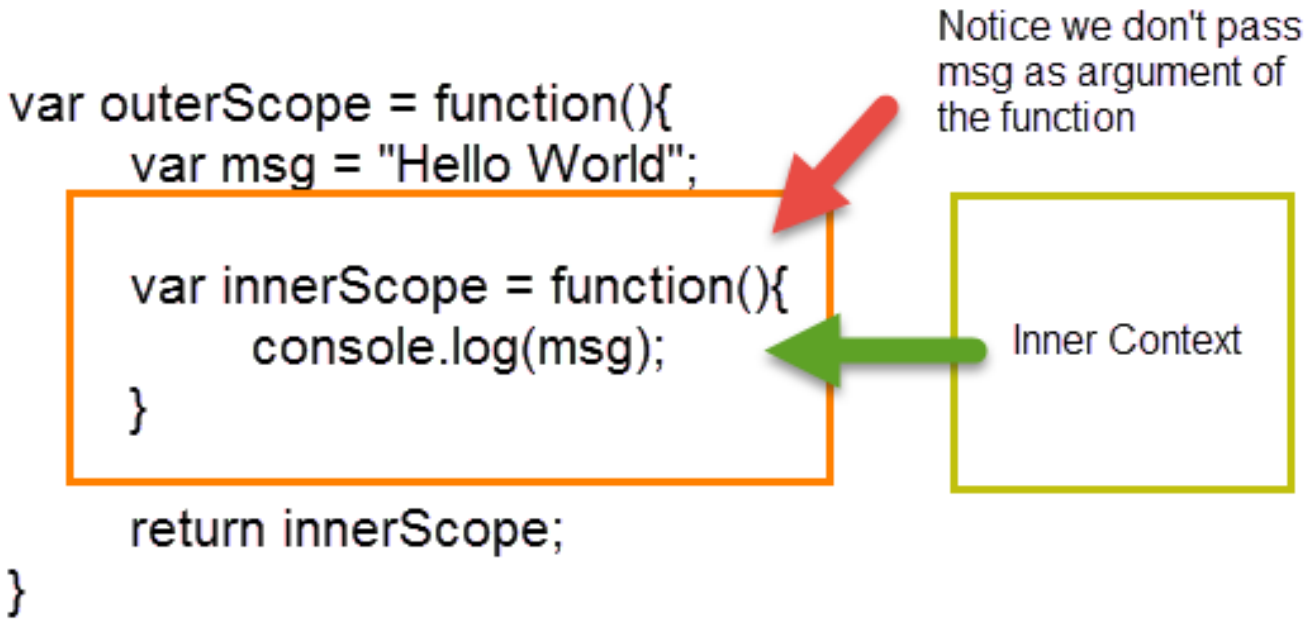

```
//Block scope
function checkLet()
{ let a = 0;
  { let a  = 1;
      { let a = 2;
       console.log(a);
      }
   console.log(a);
   }
 console.log(a);

}
```

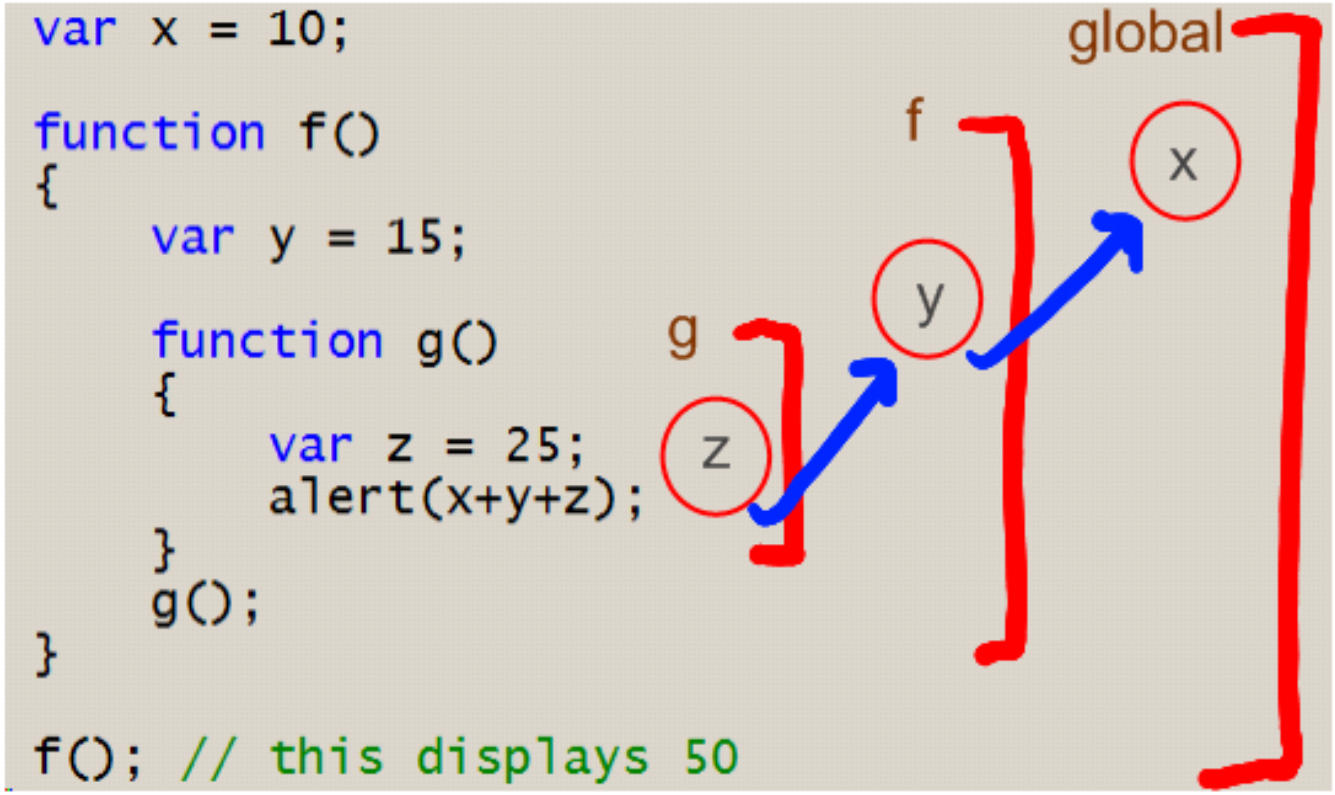
```
//function scope
function checkVar()
{ var a = 0;
  { var a  = 1;
      { var a = 2;
       console.log(a);
      }
   console.log(a);
   }
 console.log(a);

}
```

Closures


Notice we don't pass msg as argument of the function
var outerScope = function(){
var msg = "Hello World";
var innerScope = function(){
console.log(msg);
}
return innerScope;
}
Inner Context

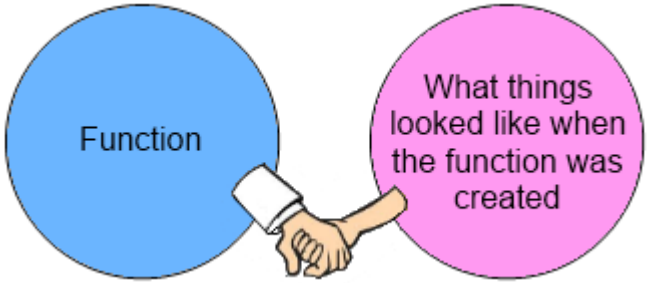
Scope of variables



A closure is a function having access to the parent scope, even after the parent function has closed.

Closures

- **closure:** A first-class function that binds to free variables that are defined in its execution environment
- **free variable:** A variable referred to by a function that is not one of its parameters or local variables.
  - **bound variable:** A free variable that is given a fixed value when "closed over" by a function's environment.
- A *closure* occurs when a function is defined and it attaches itself to the free variables from the surrounding environment to "close" up those stray references.



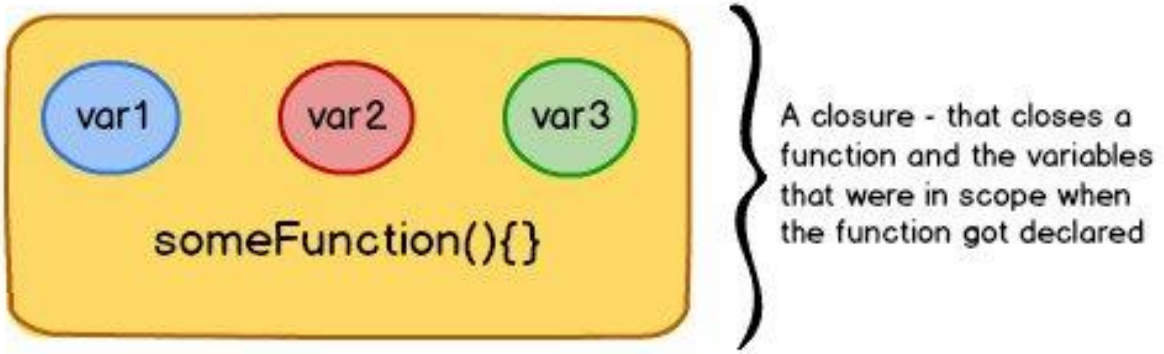

```
var x =1;
function f(){
    var y = 2;
   function g(){
          var z = 3;
         alert(x+y+z);
   }
  g();
}
{ //new variable x
  let x = -6;
   f();

}
```

```
var x =1;
function f(){
    var y = 2;
   function g(){
          var z = 3;
         alert(x+y+z);
   }
  g();
}
{//old x with new value
  var x = -6;
   f();

}
```

Closures



- Allows for the nesting of functions.
  - It grants the inner function full access to all the variables and functions defined inside the outer function.
  - The outer function does not have access to the variables and functions defined inside the inner function.
- A closure is created when the inner function is made available to any scope outside the outer function.
- Small pitfall:
  - If an enclosed function defines a variable with the same name as the name of a variable in the outer scope
    - There is no way to refer to the variable in the outer scope again!

Closure Example

```
var x = 1;
function f() {
    var y = 2;
    var ret =  function() {
        var z = 3;
        console.log (x + y + z);
    };
    y = 10;
    return ret;
}
var g = f();
g();            // 1+10+3 or 1+2+3
```

- a function closes over free variables as it is declared
  - grabs references to the names, not values (sees updates)

Module pattern - IIFE

```
(function(params) {
    statements;
})(params);
```

- declares and immediately invokes an anonymous function expression
  - used to create a new **scope** and **closure** around it
  - can help to avoid declaring global variables/functions
  - used by JavaScript libraries to keep global namespace clean

Module pattern (example)


```
var counter = 0;
function add() {
    counter += 1;
}
add();
add();
add();
```

```
function add() {
    var counter = 0;
    counter += 1;
}
add();
add();
add();
```

```
function add() {
    var counter = 0;
    function plus() {counter += 1;}
    plus();
    return counter;
}
```

```
var add = (function () {
    var counter = 0;
    return function () {return counter += 1;}
})();
add();
add();
add();

// the counter is now 3
```

Module example

```
// old: 3 globals

var count = 0;
function incr(n) {
  count += n;
}
function reset() {
  count = 0;
}
incr(4);  incr(2);
print(count);
```

```
// new: 0 globals!
(function() {
    var count = 0;
    function incr(n) {
        count += n;
    }
    function reset() {
        count = 0;
    }
    incr(4);  incr(2);
    print(count);
})();
```

- declare-and-call protects your code and avoids globals
  - avoids common problem with namespace/name collisions

Common closure bug

```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = function() { return i; };
}
```

```
> funcs[0]();
5

> funcs[1]();
5
```

Closures that bind a loop variable often have this bug.

- Why do all of the functions return 5?
- What will be the outcome if you replace var i with let i?

Fixing the closure bug

```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = (function(n) {
        return function() { return n; }
    })(i);
}
```

```
> funcs[0]();
1

> funcs[1]();
2
```