

CSC 470 – Section 3

Topics in Computer Science: Advanced Browser Technologies

Mark F. Russo, Ph.D.

Spring 2016

Lecture 3

Eloquent JavaScript: Chapters 2 & 3

JavaScript Keywords/Reserved Words

break case catch class const continue
debugger default delete do else enum export
extends false finally for function if
implements import in instance of interface
let new null package private protected
public return static super switch this throw
true try type of var void while with yield

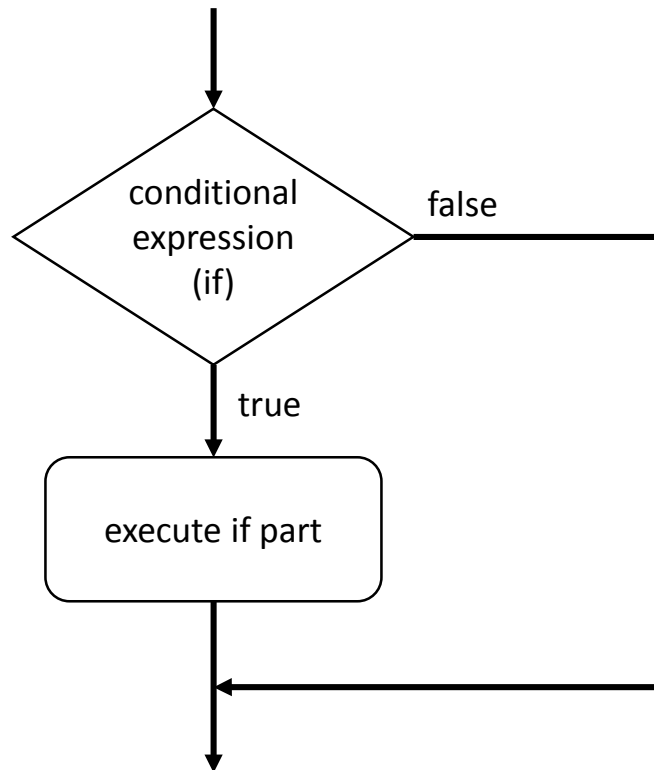
Not all reserved words are in use, currently

Control Flow

JavaScript has three conditional statements

1. if-statement
2. switch statement
3. conditional operator

if Statement

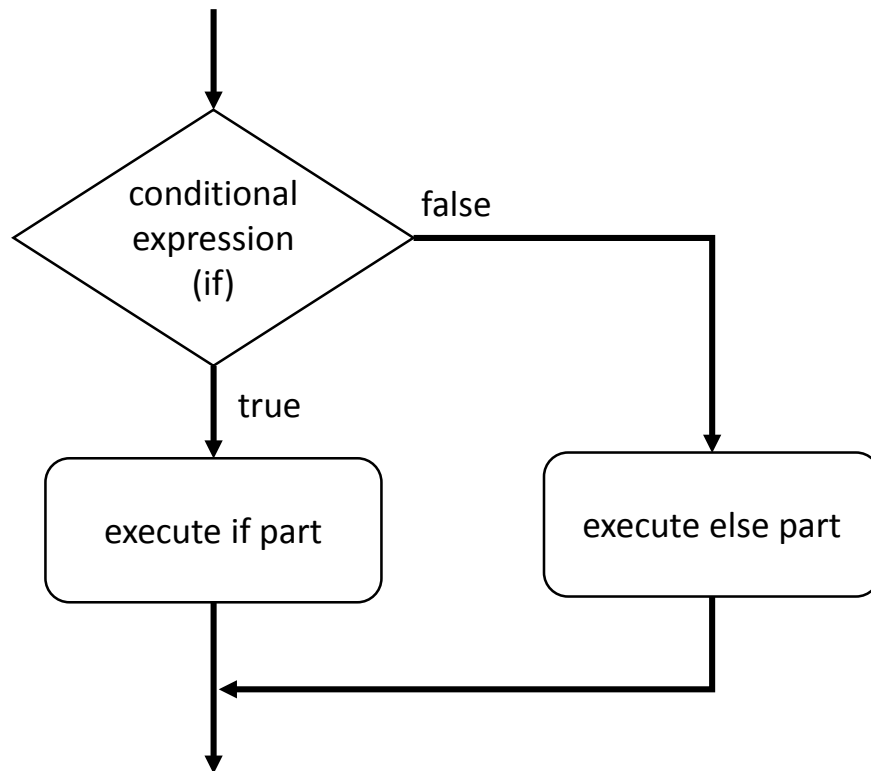


if Statement

```
if ( conditional-expression )  
{  
    // Statements  
}
```

```
var balance = 50;  
  
if ( balance < 1000 )  
{  
    console.log("too little");  
}
```

if/else Statement

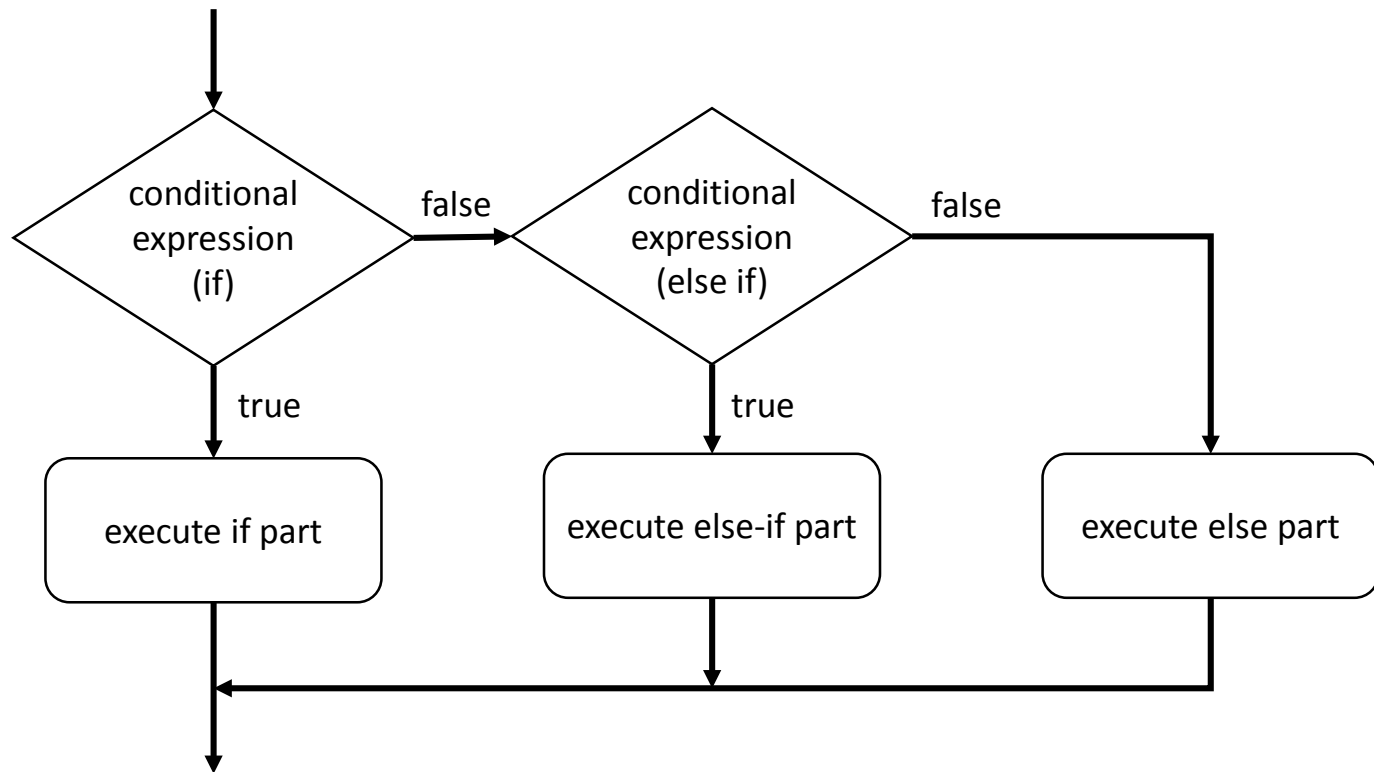


if/else Statement

```
if ( conditional-expression ) {  
    //statements executed when boolean_expression is true;  
} else {  
    //statements executed when boolean_expression is false;  
}
```

```
var balance = 1500;  
  
if ( balance < 1000 ) {  
    console.log("too little");  
} else {  
    console.log("Proceed with your purchase");  
}
```

if/else if Statement



if/else if Statement

```
if ( conditional-expression-1 ) {  
    //statements;  
} else if ( conditional-expression-2 ) {  
    //statements;  
} else {  
    //statements;  
}
```

```
var balance = 2500;  
  
if ( balance < 1000 ) {  
    console.log("too little");  
} else if ( balance > 2000 ) {  
    console.log("too much");  
} else {  
    console.log("Proceed with your purchase");  
}
```

switch Statement

```
switch (expression) {  
  
    case value1:  
        // statements  
        break;  
  
    case value2:  
        // statements  
        break;  
  
    default:  
        // statements  
}
```

- Evaluates an expression and determines if the evaluated value is equivalent to several provided value cases.
- Executes the statement block when a match is found.
- A `break` statement must end case. Otherwise, execution continues through next statement block.
- A `default` statement block may be provided. This will be executed when no case matches.
- Alternative to an if-statement.

switch Statement

```
var ans = "cheeseburger";

// Print selection based on integer entered
switch (ans) {
  case "doughnut":
    console.log("One doughnut coming up.");
    break;
  case "bacon":
    console.log("Sorry, we're all out of bacon.");
    break;
  case "cheeseburger":
    console.log("Grumpy cat? Is that you?");
    break;
  default:
    console.log("Invalid entry");
}
```

Conditional Operator (?:)

- A ternary operator (three operands) that executes in a way similar to an if-else statement
 - Operand 1: conditional-expression
 - Operand 2: evaluated if conditional-expression is true
 - Operand 3: evaluated if conditional-expression is false

```
conditional-express ? expression1 : expression2;
```

Examples

```
grade = (daysLate < 1) ? grade : 0.7*grade;  
  
var min = (num1 < num2) ? num1 : num2;
```

Iteration

JavaScript has four types of iteration:

1. `while loop`
2. `do while loop`
3. `for loop`
4. `for in loop`

while Statement (Loop)

- Executes a block of statements as long as conditional-expression continues to evaluate to true
- Referred to as Indefinite Iteration because the number of times the statements are repeated are not initially specified

```
while ( conditional-expression )  
{  
    // body statements  
}
```

```
var i = 0;  
  
while (i < 10) {  
    console.log(i);    // What is printed?  
    i++;  
}
```

while vs. if Statements

```
while ( conditional-expression )  
{  
    // Statements  
}
```

```
if ( conditional-expression )  
{  
    // Statements  
}
```

break **and** continue

- Use a `break` statement when the statement block is to be terminated early
- Use a `continue` statement when the statement block should be stopped and the continuation test re-evaluated immediately

break Statement

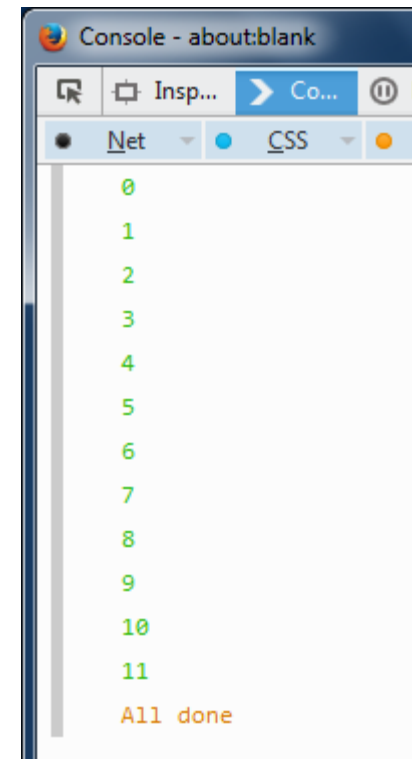
Loop 100 times, with a 1% chance of exiting the loop early

```
var i=0;
while ( i<100 ) {
    console.log(i);

    // 1% chance of exiting early
    if ( Math.random() <= 0.01 ) {
        break;
    }

    // Increment the counter
    i++;
}

console.log("All done");
```



continue Statement

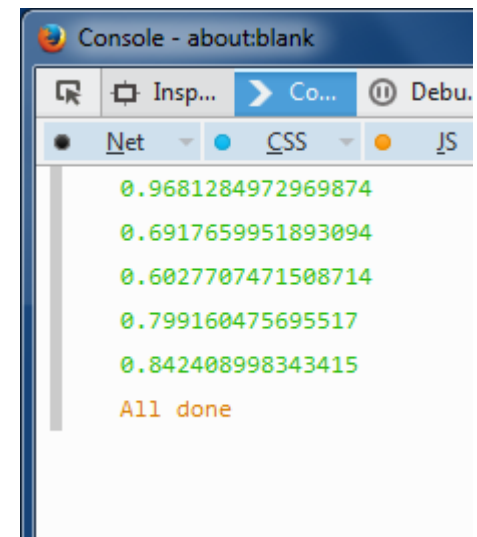
Skip random numbers < 0.5

```
// Loop 10 times
var i=0;
while ( i<10 ) {
    // Increment the counter
    i++;

    // Generate random number
    var n = Math.random();
    if (n < 0.5) {
        continue;
    }

    console.log(n);
}

console.log("All done");
```



for loop - Another Way to Iterate

Three statements between parentheses of a for-loop
Separated by semi-colons

1. Initializer

- Executed once before the for-loop begins

2. Continuation Test

- A boolean expression
- Evaluated **before** every loop iteration
- Iterations continue if the expression evaluates to true

3. Update

- Executed **after** each iteration

```
for ( initializer; continuation-test; update )  
{  
    // Statements  
}
```

for loop - while loop Similarities

- The for-loop statement is an alternative to the while-loop plus index counter pattern

1) Initializer

```
var name = "Jack";
```

```
var i = 0;
```

```
while (i < name.length) {  
    console.log( name.charAt(i) );  
    i++;  
}
```

2) Test for continuation

3) Update

- Note the repetition of multiple statements in two constructs

```
var name = "Jack";
```

```
for (var i = 0; i < name.length; i++) {  
    console.log(name.charAt(i) );  
}
```

for-in Loop - A 3rd Way to Iterate

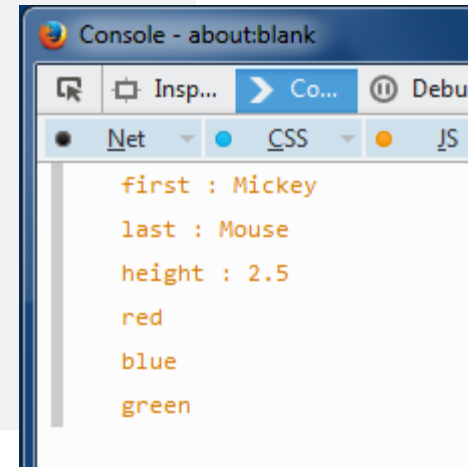
- Iterates over all keys in an object, and indexes in an array
- Must get values as a separate step

```
var employee = {first:'Mickey', last:'Mouse', height:2.5};

for (var key in employee) {
    var val = employee[key];
    console.log(key + ' : ' + val );
}

var colors = ['red', 'blue', 'green'];

for (var i in colors) {
    var color = colors[i];
    console.log(color);
}
```



do-while Loop - A 4th Way to Iterate

- Similar to while-loop, only conditional expression is after statement block

```
do {  
    // Statements  
} while ( conditional-expression );
```

- Statement block is always executed at least once

```
do {  
    var ans = prompt("What can I get you?");  
    ans = ans.trim();  
} while ( ans === '' );  
  
// Show selection based on value entered  
switch (ans) {  
    case "doughnut":  
        alert("One doughnut coming up.");  
        break;  
    case "bacon":  
        alert("Sorry, we're all out of bacon.");  
        break;  
    case "cheeseburger":  
        alert("Grumpy cat? Is that you?");  
        break;  
    default:  
        alert("Invalid entry");  
}
```

Which loop should I use?

`for-in` loop

- Most elegant for iterating over all properties of an object or array
- No automatic loop counter

`for` loop

- Convenient when the loop counter is required
- Useful when there is a fixed number of iterations known beforehand
- Useful when counting

`while` loop

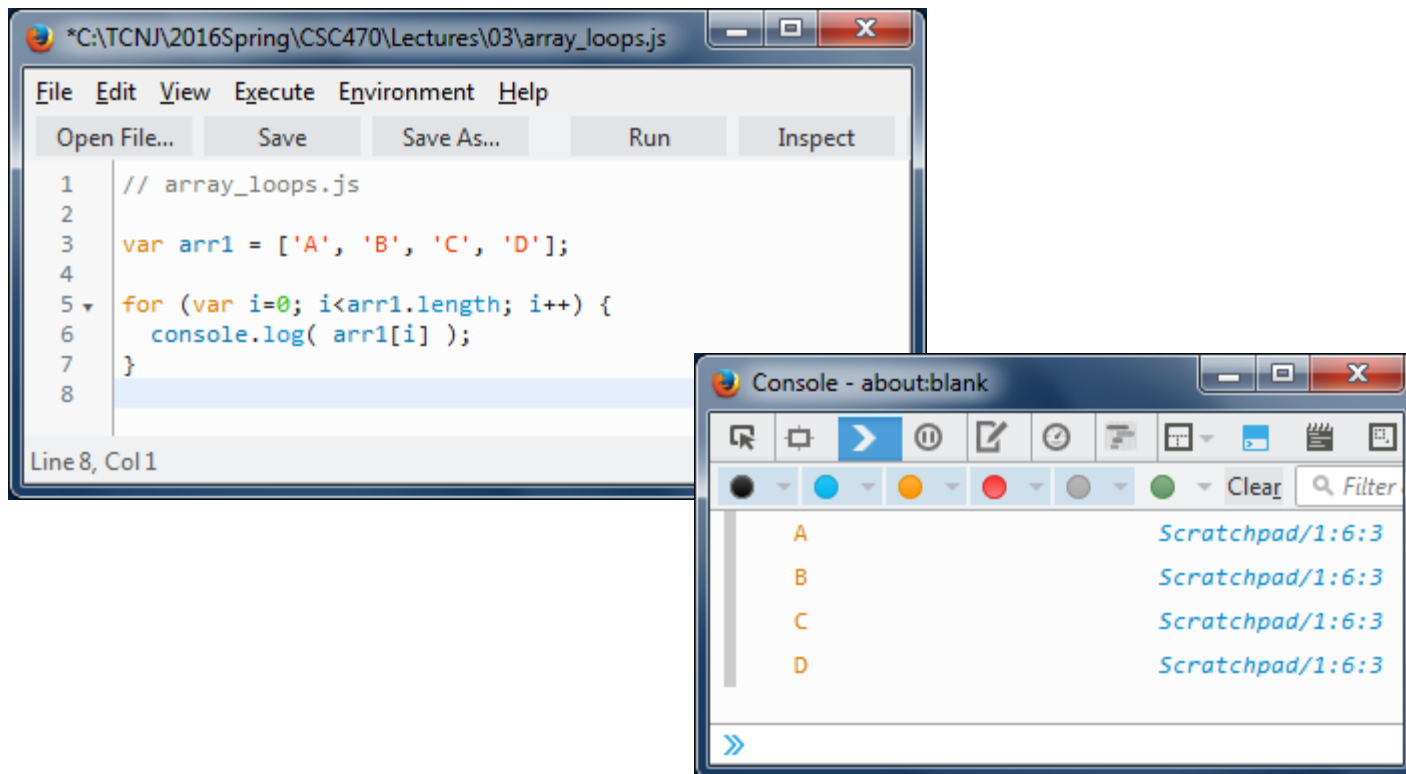
- Most flexible option
- May never execute statement block
- Useful when the number of iterations is not known beforehand

`do-while` loop

- Similar to `while`, but always executes loop statement block at least once

Looping over Arrays

Generate indexes and use to access array elements



The image shows a code editor window and a browser console window. The code editor window, titled `*C:\TCNJ\2016Spring\CSC470\Lectures\03\array_loops.js`, contains the following JavaScript code:

```
1 // array_loops.js
2
3 var arr1 = ['A', 'B', 'C', 'D'];
4
5 for (var i=0; i<arr1.length; i++) {
6   console.log( arr1[i] );
7 }
8
```

The status bar at the bottom of the code editor indicates `Line 8, Col 1`. The browser console window, titled `Console - about:blank`, shows the output of the code:

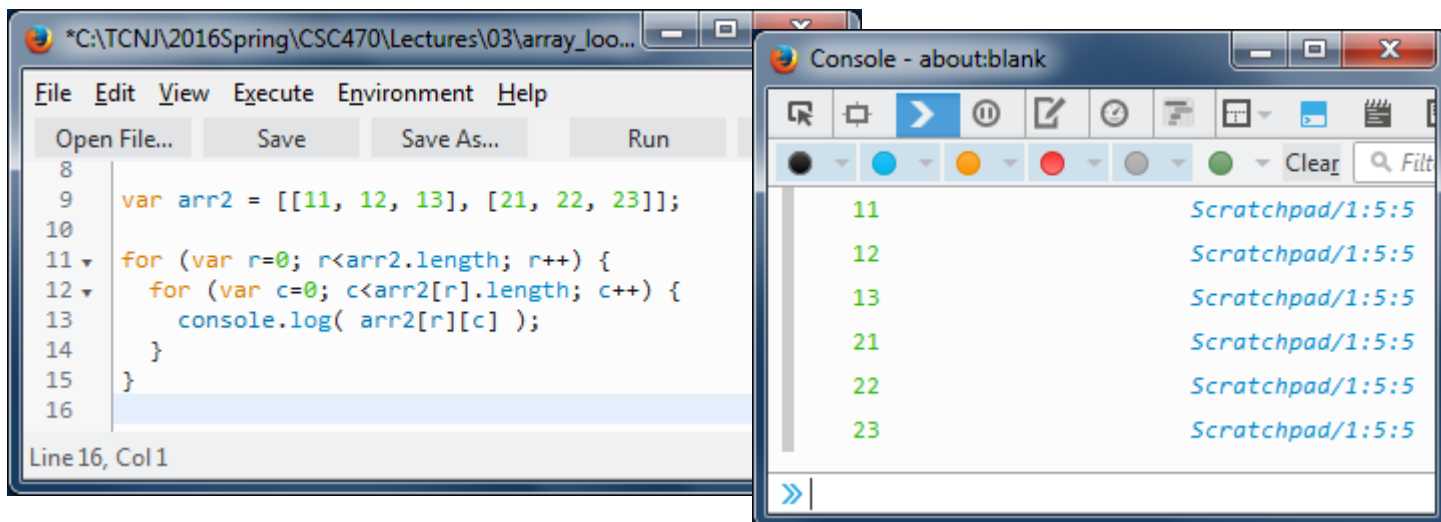
```
A Scratchpad/1:6:3
B Scratchpad/1:6:3
C Scratchpad/1:6:3
D Scratchpad/1:6:3
```


Arrays of Arrays and Nested Loops

Use multiple sets of `[]` to access inner values

```
var arr2 = [[11, 12, 13], [21, 22, 23]];

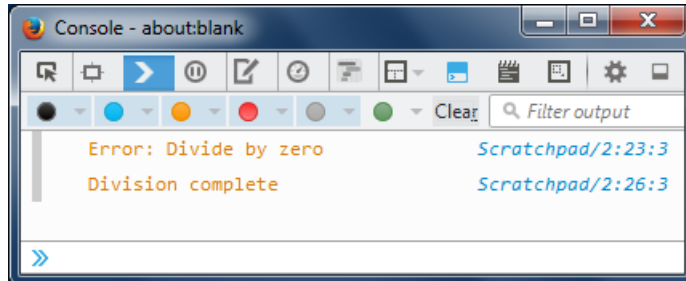
for (var r=0; r<arr2.length; r++) {
    for (var c=0; c<arr2[r].length; c++) {
        console.log( arr2[r][c] );
    }
}
```



Exception Handling and Raising

- JavaScript does not throw many exceptions, natively
- Nevertheless, it is possible to throw and catch exceptions
- Any object may be thrown, including built-in Error objects

```
try {  
    // statements  
}  
catch( error ) {  
    // handle error  
}  
finally {  
    // optional statements  
    // always executed  
}
```



Global Functions

The Global JavaScript object provides several built-in functions.

- **`isFinite()`** **Test is a number of not Infinity or -Infinity**
- **`isNaN()`** **Test if a value is not a number (NaN)**
- `parseFloat()` Convert a String into a floating point number
- `parseInt()` Convert a String into an integer number
- `encodeURIComponent()` Replace special URI chars with escaped values
- `decodeURIComponent()` Replace special URI component chars
- `decodeURI()` Replace escaped chars with unencoded values
- `decodeURIComponent()` Replace escaped component vals with unencoded
- `eval()` Evaluate a JavaScript string as code

Custom Functions

- Functions are first class objects.
- Two common ways to define a function.
- Both are used in the same manner.

function statement

```
function fname( params ) {  
    // statements  
    // optional return  
}
```

```
function multiply( x, y ) {  
    var z = x * y;  
    return z;  
}
```

Anonymous function assigned to a variable

```
var fname = function( params ) {  
    // statements  
    // optional return  
}
```

```
var multiply = function( x, y ) {  
    var z = x * y;  
    return z;  
}
```

```
var result = multiply( 2, 3 );  
console.log( result );
```

```
// 6
```

Functions as Objects

- Because functions are objects they may be used like any other object
 - Reference stored in a variable
 - Passed to another function
 - Declared within another function
 - Returned from a function
 - Stored in Arrays or Objects
- Function objects may have custom properties like any other object
 - These properties may hold values, and even other functions
 - This scenario is similar to class methods in Java.

Custom Functions

// --- Assigning a function to a variable

```
var multiply = function(x, y) {  
    return x * y;  
};
```

```
var result1 = multiply(3, 4);  
console.log( result1 );
```

// --- Passing a function to another function

```
var apply_func = function( x, y, f ) {  
    return f(x, y);  
};
```

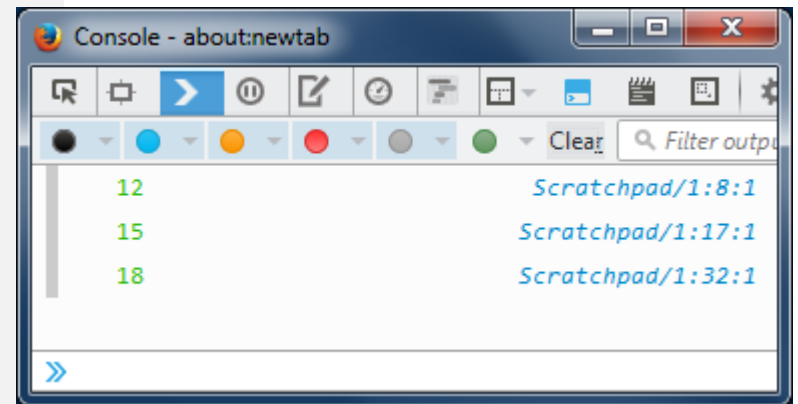
```
var result2 = apply_func(3, 5, multiply);  
console.log( result2 );
```

// --- Return a function from another function

```
var make_multiply_func = function( scale ) {  
    var f = function(y) {  
        return y*scale;  
    };  
    return f;  
};
```

```
// Make a new function and use it  
var multiply3 = make_multiply_func( 3 );
```

```
var result3 = multiply3( 6 );  
console.log( result3 );
```



Custom Functions

```
// --- Stored in an Array and/or Object
```

```
var func_arr = [  
  make_multiply_func(2), make_multiply_func(3), make_multiply_func(4)  
];  
  
for (var i=0; i<func_arr.length; i++) {  
  console.log( func_arr[i]( 2 ) );  
}  
  
var func_obj = {  
  func1 : make_multiply_func(2),  
  func2 : make_multiply_func(3),  
  func3 : make_multiply_func(4)  
};  
  
console.log( func_obj.func1( 2 ) );  
console.log( func_obj.func2( 2 ) );  
console.log( func_obj.func3( 2 ) );
```



```
// --- Functions may have custom properties including functions
```

```
multiply.created_at = 'TCNJ';  
  
multiply.get_name = function() {  
  return 'multiply_' + multiply.created_at;  
};  
  
console.log( multiply.get_name() );
```

Creating Functions

- A third method for creating functions is by using the Function constructor
- The Function Constructor takes a variable number of parameter names as strings, and ends with a string with function body statements.

```
new Function('arg1', 'arg2', ..., 'function body');
```

```
var multiply = new Function('x', 'y', 'return x*y;');
```

```
var result = multiply( 2, 3 );  
console.log( result );           // 6
```


Variable Scope

- JavaScript has function scope, but ...
- JavaScript does not have block scope (the way Java does)
 - Variables declared within a code block (`if`, `for`, ...) are equivalent to variables declared outside the block but in the scope of the function

JavaScript

```
var func1 = function( ) {  
  
    if (true) {  
        var x = 10;  
    }  
  
    console.log(x);  
};  
  
func1();
```

>> 10

No errors

Java

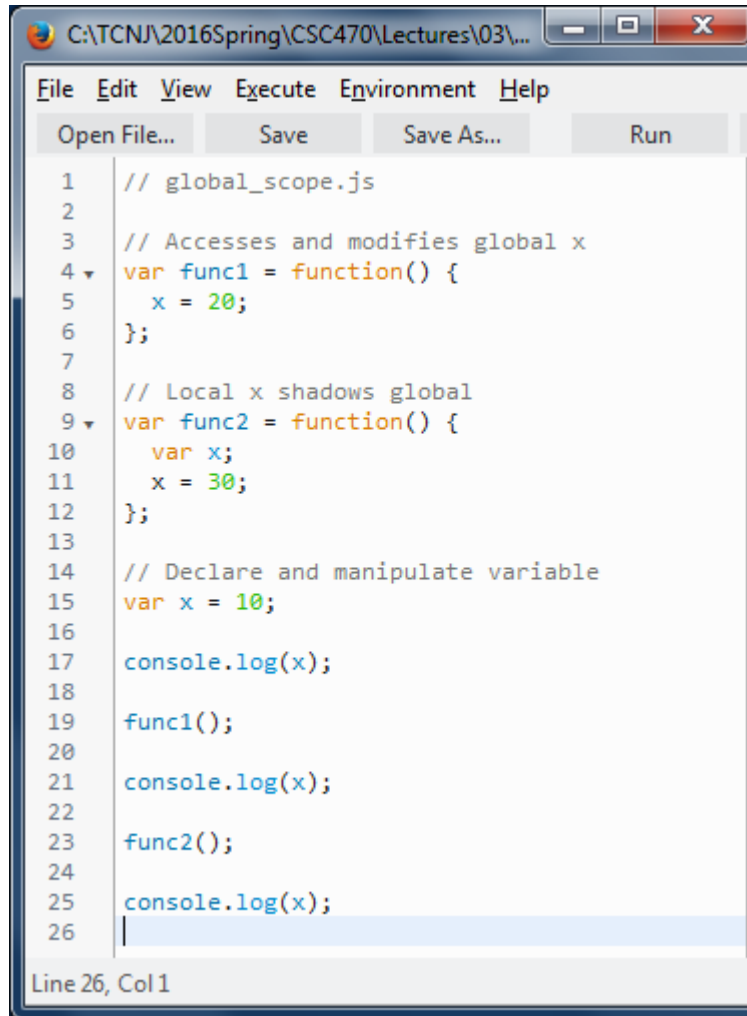
```
public class Test {  
    public static void main() {  
  
        if (true) {  
            int x = 10;  
        }  
  
        System.out.println(x);  
    }  
}
```

Compilation Error.

Cannot find symbol - variable x

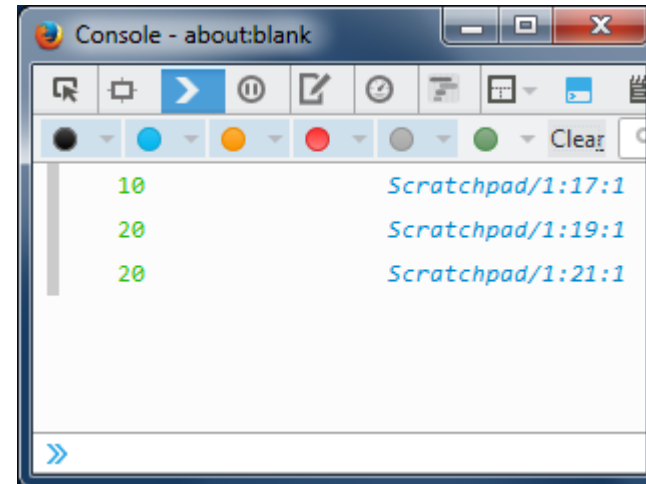
Variable Scope

- Variables declared in an outer scope is visible from an inner function scope
- Variables declared in an inner function scope may shadow global variables



```
1 // global_scope.js
2
3 // Accesses and modifies global x
4 var func1 = function() {
5     x = 20;
6 };
7
8 // Local x shadows global
9 var func2 = function() {
10     var x;
11     x = 30;
12 };
13
14 // Declare and manipulate variable
15 var x = 10;
16
17 console.log(x);
18
19 func1();
20
21 console.log(x);
22
23 func2();
24
25 console.log(x);
26
```

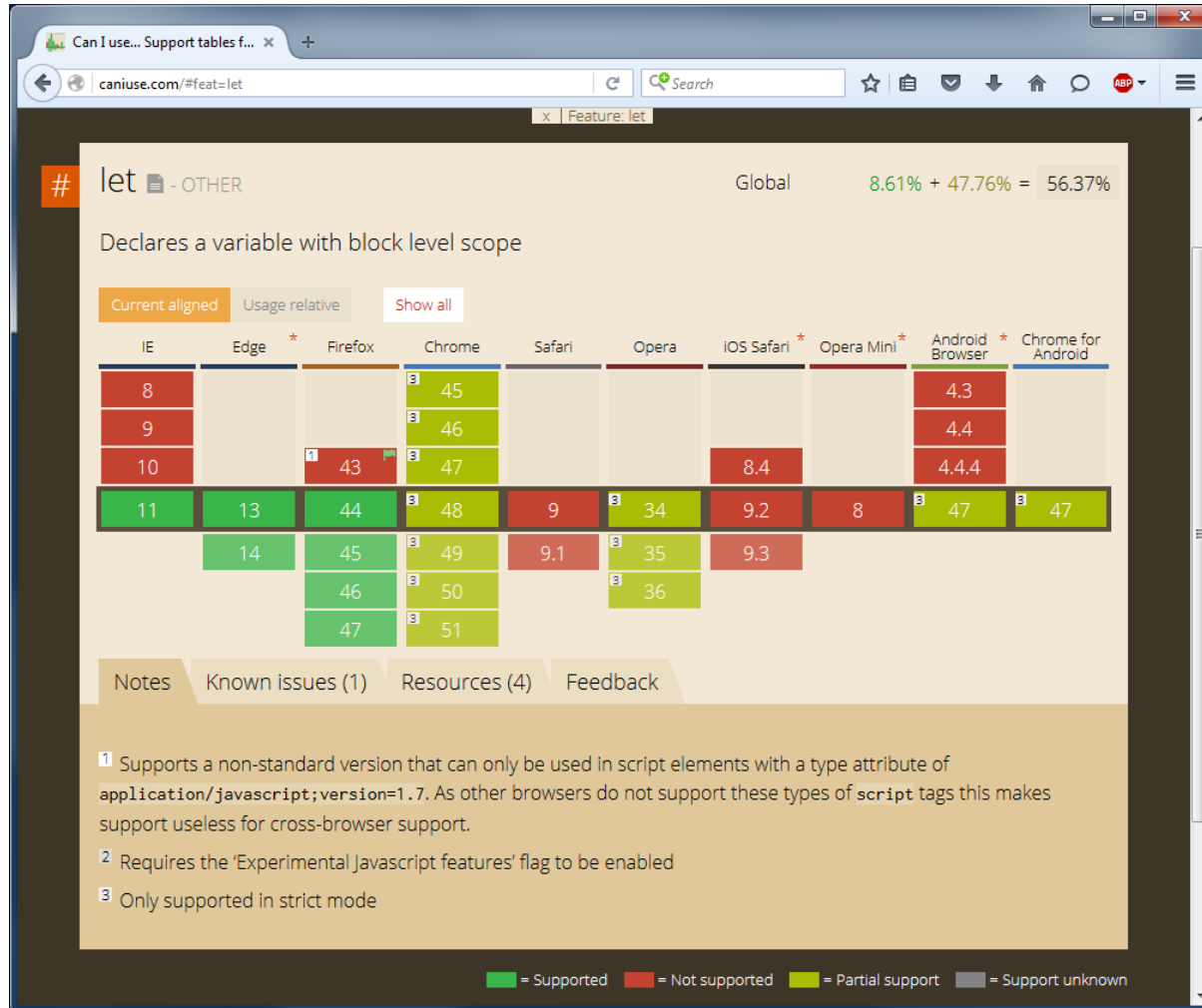
Line 26, Col 1



```
10 Scratchpad/1:17:1
20 Scratchpad/1:19:1
20 Scratchpad/1:21:1
```

Variable Scope in ES2015

In the ECMAScript 2015 standard, variables declared using the `let` keyword instead of `var` **WILL** have block scope



Variable Hoisting

- All variable declarations within a function are hoisted to the top of the function and declared immediately
- Variables are available to be used immediately, regardless of where they are declared inside a function
- Hoisted variables are not initialized – they start as `undefined`

JavaScript

```
var func2 = function() {  
    console.log(x);  
    var x = 10;  
    console.log(x);  
};  
func2();
```

```
>> undefined  
>> 10
```

Java

```
public class Test {  
    public static void main() {  
        System.out.println(x);  
        int x = 10;  
        System.out.println(x);  
    }  
}
```

Compilation Error.
Cannot find symbol - variable x

Function Hoisting

- Similar to the way variable declarations are hoisted to the top of a function, function declarations are hoisted to the top of a program

JavaScript

```
// function_hoisting.js
// Functions invoked before declared
// Must be declared using standard notation
func1();
func2();

function func1() {
  if (true) {
    var x = 10;
  }
  console.log(x);
};

function func2() {
  console.log(x);
  var x = 10;
  console.log(x);
};
```

```
>> 10
>> undefined
>> 10
```

Python

```
# function_hoising.py

# Functions invoked before declared
func1() ←
func2()

def func1():
    if True:
        x = 10
    print(x)

def func2():
    print(x)
    x = 10
    print(x)
```

```
Traceback (most recent call last):
  File "function_hoisting.py", line 3, in <module>
    func1()
NameError: name 'func1' is not defined
```

Arguments Objects

- The number of function arguments is flexible
 - Too many arguments passed? Extra arguments are ignored.
 - Too few arguments passed? Missing parameters are assigned `undefined`
- Arguments object
 - All arguments passed to a function are collected and made available within the scope of the function in an Arguments object named `arguments`
 - `arguments` is an array-like object that uses `[]` to access elements

`arguments.length`

- Reference to the number of arguments passed to the function.

`arguments[index]`

- Access elements in the arguments array.

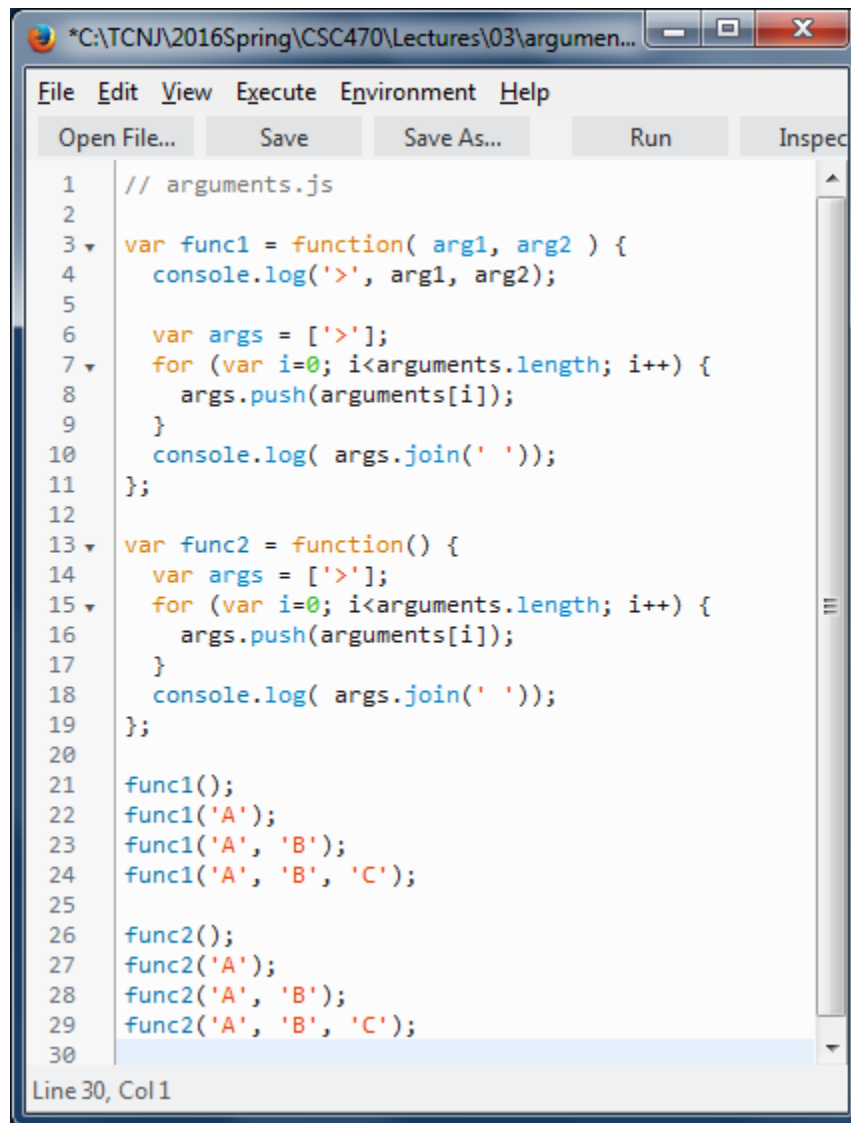
`arguments.callee`

- Reference to the currently executing function.

`arguments.caller`

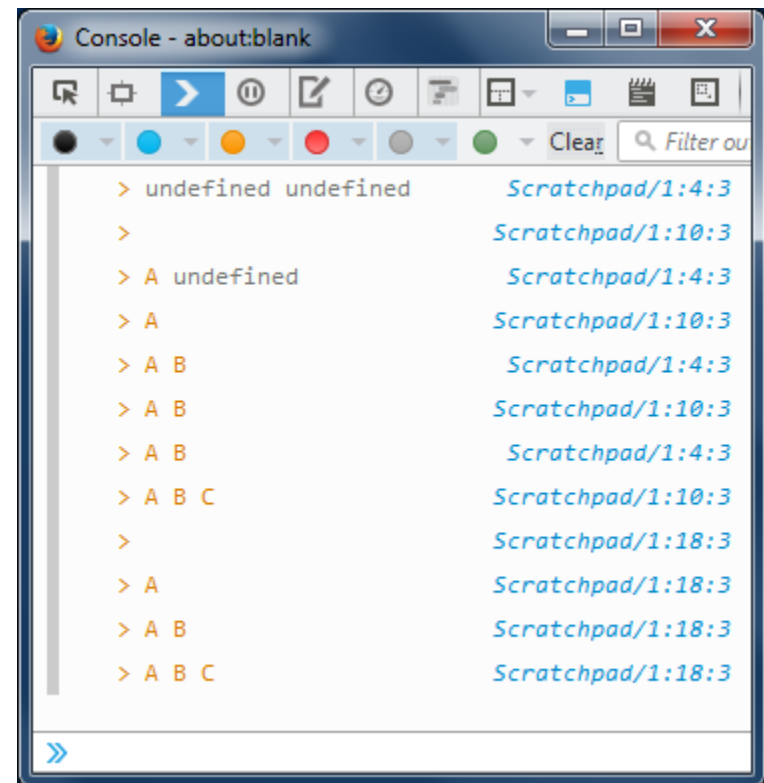
- Reference to the function that invoked the currently executing function.

Arguments Objects



```
1 // arguments.js
2
3 var func1 = function( arg1, arg2 ) {
4     console.log('>', arg1, arg2);
5
6     var args = ['>'];
7     for (var i=0; i<arguments.length; i++) {
8         args.push(arguments[i]);
9     }
10    console.log( args.join(' '));
11 };
12
13 var func2 = function() {
14     var args = ['>'];
15     for (var i=0; i<arguments.length; i++) {
16         args.push(arguments[i]);
17     }
18     console.log( args.join(' '));
19 };
20
21 func1();
22 func1('A');
23 func1('A', 'B');
24 func1('A', 'B', 'C');
25
26 func2();
27 func2('A');
28 func2('A', 'B');
29 func2('A', 'B', 'C');
30
```

Line 30, Col 1



```
> undefined undefined Scratchpad/1:4:3
> Scratchpad/1:10:3
> A undefined Scratchpad/1:4:3
> A Scratchpad/1:10:3
> A B Scratchpad/1:4:3
> A B Scratchpad/1:10:3
> A B Scratchpad/1:4:3
> A B C Scratchpad/1:10:3
> Scratchpad/1:18:3
> A Scratchpad/1:18:3
> A B Scratchpad/1:18:3
> A B C Scratchpad/1:18:3
```

Parameters defined in the function definition are for convenience.

Any number of arguments may be passed to any function.

Nested Functions

- Functions may be declared and used within an outer function

```
// inner_functions.js

var func = function()
{
    // Declare an inner function
    var multiply = function(x, y) {
        return x * y;
    };

    // Use the inner function
    var result = multiply( 5, 6 );
    console.log( result );
};

func();
```

>> 30

Nested Functions and Nested Scope

```
// nested_scope.js

var landscape = function () {
    var result = "";

    var flat = function ( size ) {
        for ( var count = 0; count < size ; count ++ )
            result += "_";
    };

    var mountain = function ( size ) {
        result += "/";

        for ( var count = 0; count < size ; count ++ ) {
            result += "' ' " ;
        }
        result += "\\\"";
    };

    flat (3) ;
    mountain(4) ;
    flat(6) ;
    mountain(1) ;
    flat(1) ;
    return result ;
};

console.log( landscape() ) ;
// ___/' ' '\____/' '\_
```

- Inner functions have access to variables in the scope of the outer function
- Eloquent JavaScript
 - Chapter 3

Context vs. Scope

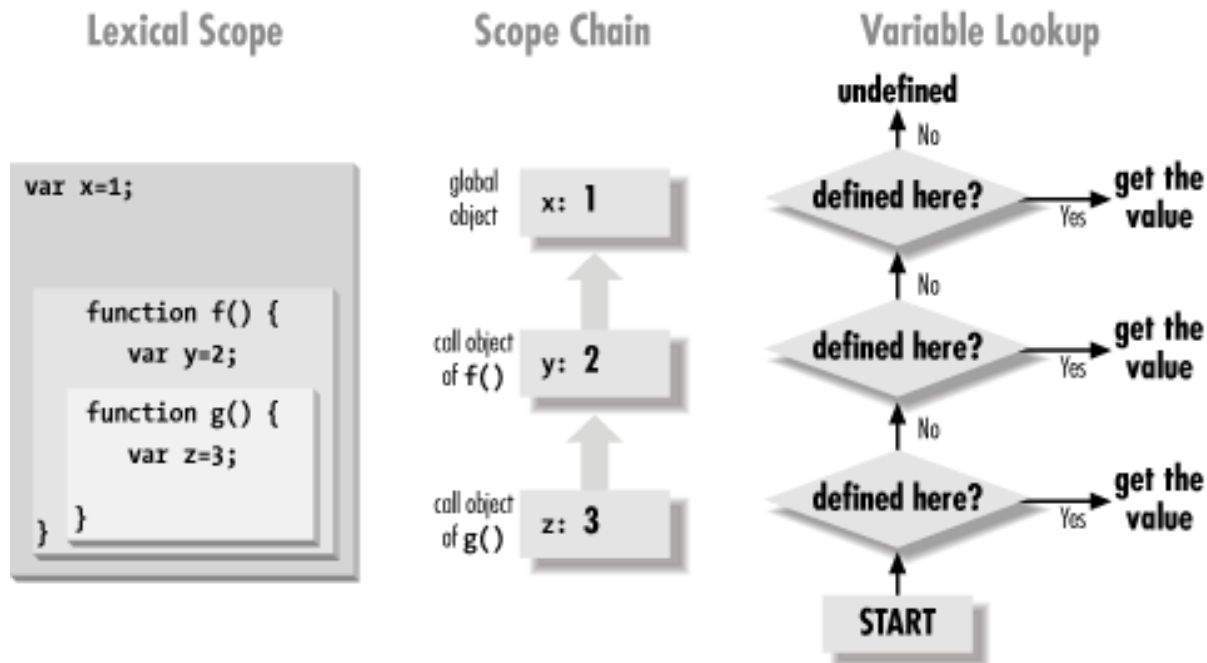
- Every function invocation has both a scope and a context associated with it
- **Scope** pertains to the variable access of a function when it is invoked and is unique to each invocation
 - Global scope – variables defined outside function body
 - Local scope – variables defined within function body
 - Block scope – variables defined with `let` keyword
- **Context** is always the value of the `this` keyword which is a reference to the object that “owns” the currently executing code
 - The object on which a method is invoked
 - The global object, when a function is invoked "unbound"

Execution Context and Scope Chain

- A JavaScript program always starts in the Global Execution Context
- Subsequent function invocations result in creation of NEW execution contexts
- Each time a new execution context is created it is appended to the top of the execution stack
- Once completed, it will be removed from the top of the stack
- For each execution context there is a scope chain coupled with it
- The scope chain contains the variable object for every execution context in the execution stack
- The scope chain is used for determining variable access and identifier resolution
 - Variable resolution starts with the innermost scope and works its way out
 - This is how variable shadowing comes about

Scope Chain

- When a function is defined, it stores the scope chain then in effect.
- When that function is invoked, it creates a new object to store its local variables, and adds that new object to the stored scope chain to create a new, longer, chain that represents the scope for that function invocation.



Scope Chain Example

```
// scope_chain.js

var x = 1;
var y = 2;
var z = 3;

var f1 = function() {
  var x = 4;
  console.log(x, y, z);

  var f2 = function() {
    var y = 5;
    console.log(x, y, z);

    var f3 = function() {
      var z = 6;
      console.log(x, y, z);
    };
    f3();
  };
  f2();
};

console.log(x, y, z);
f1();
```

What is the output from this program?
What does the scope chain look like?

Scope Chain Example

```
// scope_chain.js

var x = 1;
var y = 2;
var z = 3;

var f1 = function() {
  var x = 4;
  console.log(x, y, z);

  var f2 = function() {
    var y = 5;
    console.log(x, y, z);

    var f3 = function() {
      var z = 6;
      console.log(x, y, z);
    };
    f3();
  }
  f2();
};

console.log(x, y, z);
f1();
```

Output

1	2	3
4	2	3
4	5	3
4	5	6

Setting Context

- We can choose the context within which a function is executed using the `call()` and `apply()` methods

myFunction.apply(thisObj [, argsArray])

- Calls a function with a given `this` value and arguments provided as an array
- Set `thisObj` to `null` to execute in the context of the global object

myFunction.call(thisObj [, arg1[, arg2[, ...]]])

- Calls a function with a given `this` value and arguments provided individually
- Set `thisObj` to `null` to execute in the context of the global object

Context Example

```
// contexts.js

// A generic function that concatenates and returns named properties
var display = function() {
  var vals = [];
  for (var i=0; i<arguments.length; i++) {
    var prop = arguments[i];
    vals.push( this[prop] );
  }
  return vals.join(" ");
}

var obs = [
  {first:"Bart", last:"Simpson", grade:4},
  {first:"Lisa", last:"Simpson", grade:2}
];

// Apply display function to all object in array
for (var i=0; i<obs.length; i++) {
  var ob = obs[i];
  var result = display.apply( ob, ['first', 'last']);
  console.log(result);
}
```

Bart Simpson
Lisa Simpson

Function Closures

- What happens when we declare a function within another function and return it?
- When the constructed function is executed, what does the scope chain look like?
- Being able to reference a specific instance of local variables in an enclosing function—is called ***closure***.
- A function that “closes over” some local variables is called ***a closure***
- This behavior not only frees you from having to worry about lifetimes of variables but also allows for some creative use of function values.

Function Closures Example

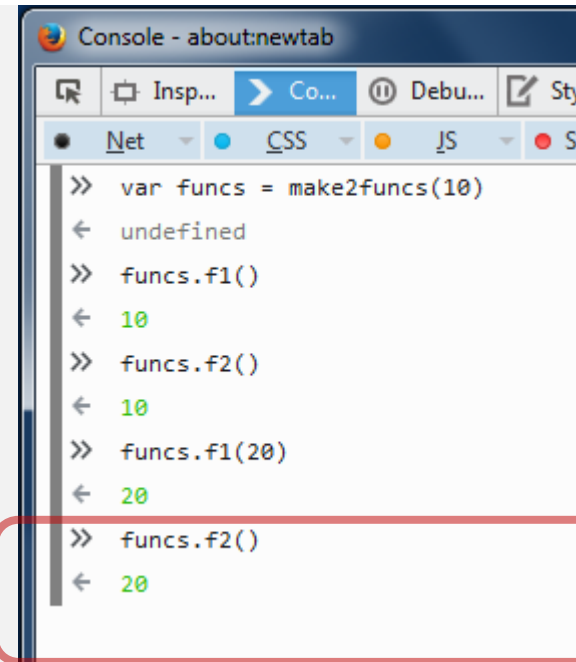
```
// closures.js

// Make and return two functions
function make2funcs(val) {

    // Inner function that redefines val,
    // if argument is provided
    var func1 = function(v1) {
        if (v1 !== undefined) {
            val = v1;
        }
        return val;
    }

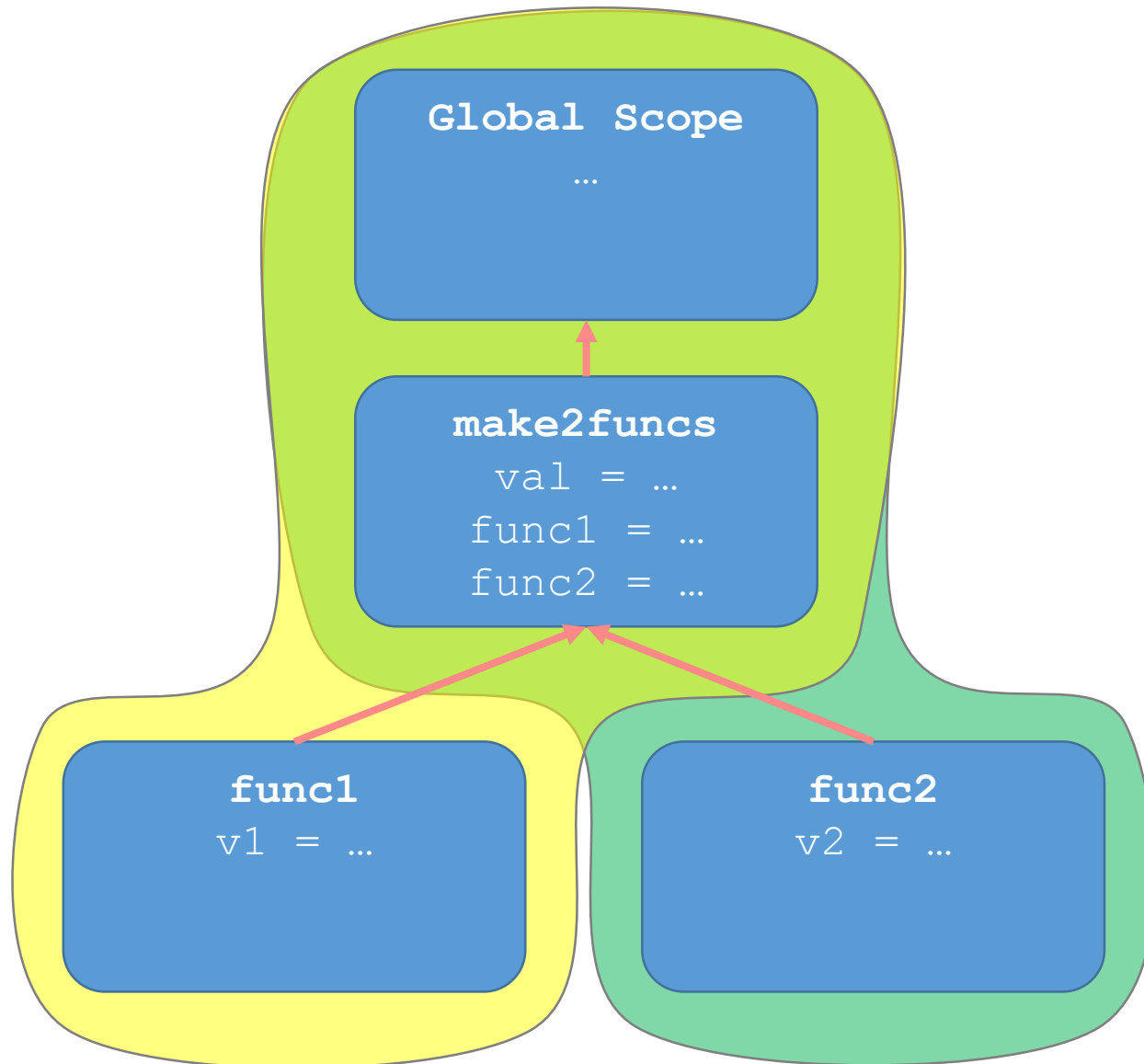
    // Another inner function that redefines val,
    // if argument is provided
    var func2 = function(v2) {
        if (v2 !== undefined) {
            val = v2;
        }
        return val;
    }

    // Return two inner functions in an object
    return {f1:func1, f2:func2};
}
```



func1 and func2 close over make2funcs()

Function Closures Example – Scope Chain



Immediately Invoked Function Expressions (IIFE)

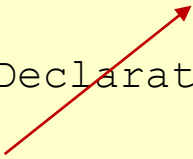
- Pollution of the global scope can be avoided by wrapping all calculations in a function.
- Function name declarations can be avoided by creating and immediately invoking an anonymous function
- This pattern is called and IFFE (iff-ee)
- Constructed as
 - Anonymous function in parentheses
 - Followed by a pair of parentheses that cause the function to be invoked immediately

```
(function() {  
  
    // Declarations and main program statements here  
  
}) ();
```

IIFE Variations

- Passing parameters into the IIFE

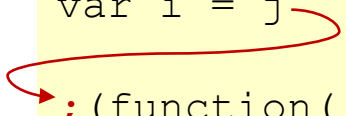
```
(function( param ) {  
    // Declarations and main program statements here  
})( val );
```



- Defensive semicolon

- In most cases semicolons in JavaScript are optional
- IIFE may be incorrectly evaluated if the statement before excludes a ;

```
var i = j  
;(function( param ) {  
    // Declarations and main program statements here  
})( val );
```



IIFE Example

```
// iife.js
// Execute a JavaScript program, leaving no trace in the global scope

// Calculate the first nvals Fibonacci Numbers
(function( nvals ) {

    // Declarations are confined to function scope

    // Cache calculated values
    var fibcache = [];

    var fibonacci = function(n) {
        if (n === 0) {
            fibcache[0] = 0;
            return 0;
        } else if (n === 1) {
            fibcache[1] = 1;
            return 1;
        } else {
            var val = fibcache[n-1] + fibcache[n-2];
            fibcache[n] = val;
            return val;
        }
    };

    // Main part of program that makes use of declarations
    for (var i = 0; i <= nvals; i++) {
        console.log( fibonacci(i) );
    }

})(100);
```

