```javascript
// Example of nested function

function circumference(radius) {
    function double(number) {       // nested function declaration
      return 2 * number;
    }

    return 3.14 * double(radius);   // call the nested function
  }

console.log(circumference(5));

// Global scope, no functions thus one single scope
var name = 'Julian';
console.log(name);

for (var i = 0; i < 3; i += 1) {
  console.log(name);
}

console.log(name);

// Function scope, adding a function
// When invoking the greet() function, it can access the name variable since code within a
function inherits
// access to all variables in all surrounding scopes
var name = 'Julian';

function greet() {
  console.log(name);
}

greet();

// Nestd functions work the same way
var name = 'Julian';

function greet() {
  function say() {
    console.log(name);
  }

  say();
}

// Creating a closure: when a function retains access to the var scope currently in effect
// Closure retains access to everything in scope when closure is created, retains references
for as long as
// the closure exists, so function can access references when we invoke the function
// when variable value changes after creating a closure that invludes the variable, closure
sees new value
var count = 1;

function logCount() {  // create a closure
  console.log(count);
}

logCount();             // logs: 1

count += 1;             // reassign count
logCount();             // closure sees new value for count; logs: 2

// JavaScript uses Lexical Scoping to resolve variables;
// it uses the structure of the source code to determine the variable's scope.
// That is, the source code defines the scope. At any point in a JavaScript program,
// there is a hierarchy of scopes from the local scope of the code up to the program's global scope.
// When JavaScript tries to find a variable, it searches this hierarchy from the bottom to the top.
// It stops and returns the first variable it finds with a matching name.
// This means that variables in a lower scope can shadow, or hide, a variable with the same name in a higher scope.

// Most mainstream programming languages use lexical scoping rules (also called "static scoping").
// Some languages use "dynamic scoping" instead, or make dynamic scoping a choice.

// Adding variables to current scope
// 1. using var keyword
function lunch() {
  var food = 'taco';
}
// 2. using arguments passed to function
function eat(food) {
  console.log('I am eating ' + food);
}
// 3. function declaration itself creates a variable with the same name as the function
function drink() {
  console.log('I am drinking a glass of water');
}

// variable scoping rules apply to assignment and referencing equally
var country = 'Spain';
function update() {
  country = 'Liechtenstein';
  // checks current scope and each higher scope, looking for var
  // with name country. JS sets first country var it finds to
  // 'Liechtenstein'
}

console.log(country);
update();
console.log(country);

// if JS can't find matching var, it creates new global var
function assign() {
  var country1 = 'Kiechtenstein';
  country2 = 'Spain';
}

assign();
console.log(country2); // Spain
console.log(country1); // gets ReferenceError

// example to demo the effect of updating a function
// why does the last log print out what's in the function?
var country = 'Spain'
function update() {
  country = 'Not Spain';
}

console.log(country); // Spain
update();
console.log(country); // Not Spain

// when no variable country existed before the function
function update() {
  country = 'Not Spain';
}

```

```javascript
update();
console.log(country); // Not Spain

// variable shadowing
// q: what's the difference between having a var and not
// having a var declared within a function?
// in this case none, name = 'Logan' would produce the same
// because within the greet() function, only access inner name
var name = 'Julian';
function greet() {
  var name = 'Logan';
  console.log(name);
}
greet();

// if function definition has parameter with same name as var
// from an outer scope? parameter shadows outer variable
// so the local parameters shadows the outer var
var name = 'Julian';
function greet(name) {
  console.log(name);
}

greet('Sam');

// Some scoping rules:
// 1. every function declaration creates a new var scope
// 2. all vars in the same or surrounding scopes are available to code


```