# Functions

- Functions are “first-class” objects in JavaScript, meaning that they can be used in most contexts where other types of values are allowed, e.g.:
  - parameter values, return values, variable values, object and array values
- Functions can have methods, and in this case behave more like classes in Object-Oriented (OO) languages
- Function “objects” inherit methods from Function.prototype (more on prototype later)

# Functions

- Duck Typing – functions are not declared with explicit signatures (no input/output type declarations)
- Overloading – a given function name may be applied to different input types and varying numbers of parameters resulting in different behaviors.
  - Use the “typeof” operator if you need to perform different tasks depending on parameter type

# Functions: Declarations

- Function statement (declaration):

```
function foo() { … };  // code “hoisting” applies
```

- Function use:

```
foo();
```

- Will this work? How/Why?

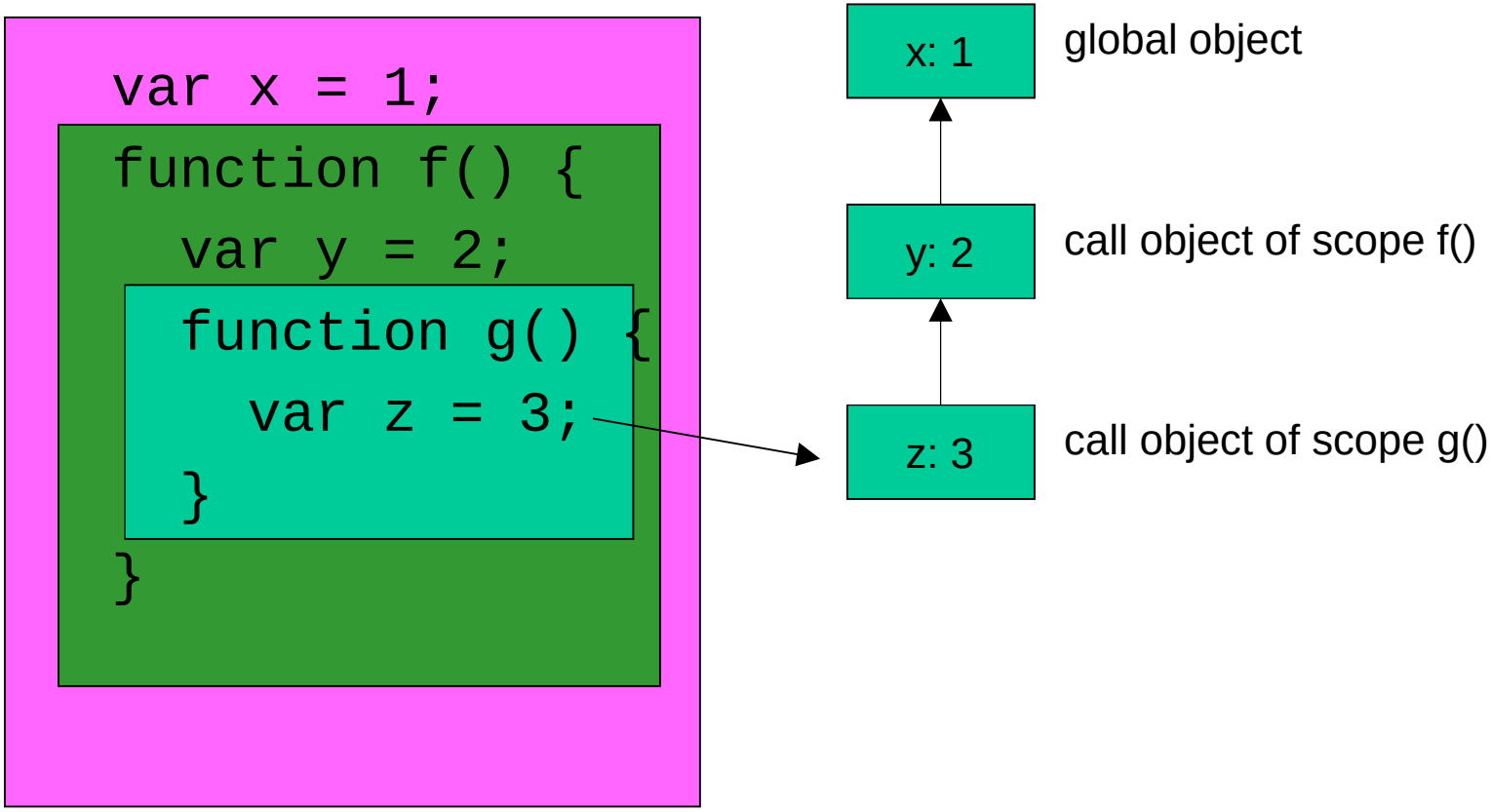
```
foo();
function foo() { … }
```

- Function (anonymous) expressions; :

```
var foo = function() { … }
```

Note that code-hoisting does not apply here; variable assignment occurs within normal execution flow

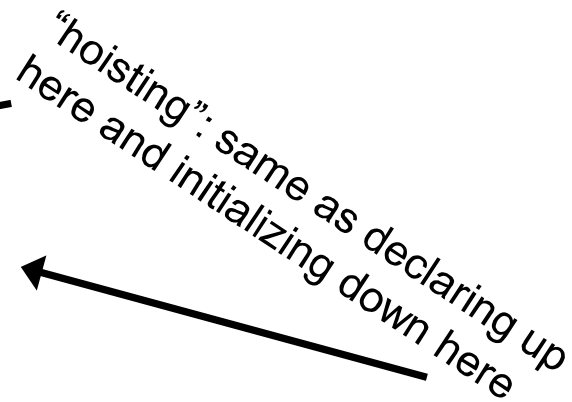
# Scope Chain


var x = 1;
function f() {
var y = 2;
function g() {
var z = 3;
}
}
x: 1
global object
y: 2
call object of scope f()
z: 3
call object of scope g()

# Functions: variable-scoping issues

- What will function f( ) display when called? Example global.html

```
var scope = "global"
function f() {
    alert(scope);
    var scope = "local";
    alert(scope);
}
f();
```




- Beware of references to variables that have not yet been assigned values! JavaScript programs can fail "silently"

# Functions: `return` statement

- `return expn;  // no line-break allowed before expn!`
- `return;  // return value is "undefined"`

- If return statement is omitted, the function's value is undefined

- Default return value for constructors is **`this`** - the newly created object (more on **`this`** later)

# Functions: parameters

- No checking on type or number of actuals parameters passed by caller!
- If you care about type-checking, must perform it yourself using the **`typeof()`** operator
- If extra parameters are passed they are ignored (but still available in arguments object – see next slide)
- If too few parameters are passed, missing ones are assigned value undefined

# Functions: `arguments` array

- Pseudo-parameter passed along with the parameters themselves
- Contains all the arguments in an array-like object
- Useful for functions that take a variable number of parameters, e.g. a function to sum value-lists.
- **`arguments.length`** gives number of arguments passed
- Example merge.html
- Should be treated as a readonly variable, else weird and bad things may happen (e.g. changing it can cause parameters themselves to change)

# Functions: `this`

- A pseudo parameter to functions (need not be explicitly passed)
- Enables a single-instance of a function object to operate across multiple object instances or prototypes by giving the function a way to know which instance object invoked it
- Mechanism used for prototype-inheritance operation

# Function Invocation

- **`thisObject.methodName(args){ … this … }`**
  - **`this`** refers to thisObject (object that contains the method)
- **`functionObject(args) { … this … }`**
  - **`this`** refers to the global object, since there is no "parent" object to associate with this
  - recent versions of the JS standard (5+/strict) say that **`this`** has value undefined in this scenario
- **`new FunctionObj(args){…this…} // constructor`**
  - **`this`** refers to the newly created object
- **`functionObject.apply(thisObject, args)`**
  - Allows you to specify the value of **`this`** as the 1st parameter

# Functions: `this`

- Just when you think you understand **`this`**, it jumps out and bites you.

```
Foo.method =
 function() {
   function test() {
    // refers to what?
    … this …
   }
   test();
}
```

- The fix? Create a local method-variable, **`self`**, to keep a reference to **`this`**:

```
Foo.method =
 function() {
   var self = this;
   function test() {
     // Use self to
     // refer to Foo
     … self …
   }
   test();
}
```

(example object.html)

# Objects

- The language has no class construct per se
  - Objects are similar to associative arrays or dictionaries (example object.html)

```
var point = new Object();
point.x = 2;  // equivalent to point['x'] = 2
point['y'] = 3;  // equivalent to point.y = 3
alert(point.x); alert(point.y);
for (var i in point) console.log(i + ": " +
  point[i]);
```

- Note, a JavaScript object's definition is determined at run time. Unlike C++ or Java, JavaScript allows properties/methods to be dynamically added/changed at runtime!

# Object Constructors

Example:
constructor.html

```
function Rectangle_area() {
    return this.width * this.height;
} /* "this" refers to the current object */

function Rectangle(w,h) {
    this.width = w;
    this.height = h;
    this.area = Rectangle_area;   // note no ()
}

var rec = new Rectangle(2,4);
alert(rec.area());   // outputs what?
var rec2 = Rectangle(2,4);
alert(rec2.area());   // outputs what?
```

# Object Constructors

Example: constructor2.html

- Forgetting the **new** operator can have other nasty consequences; what does this code alert? Why?

```
function Person(first, last){
    this.name = first + ", " + last;
}

name = "Simpson";
var person = Person("Bart", name);

alert(name);
```