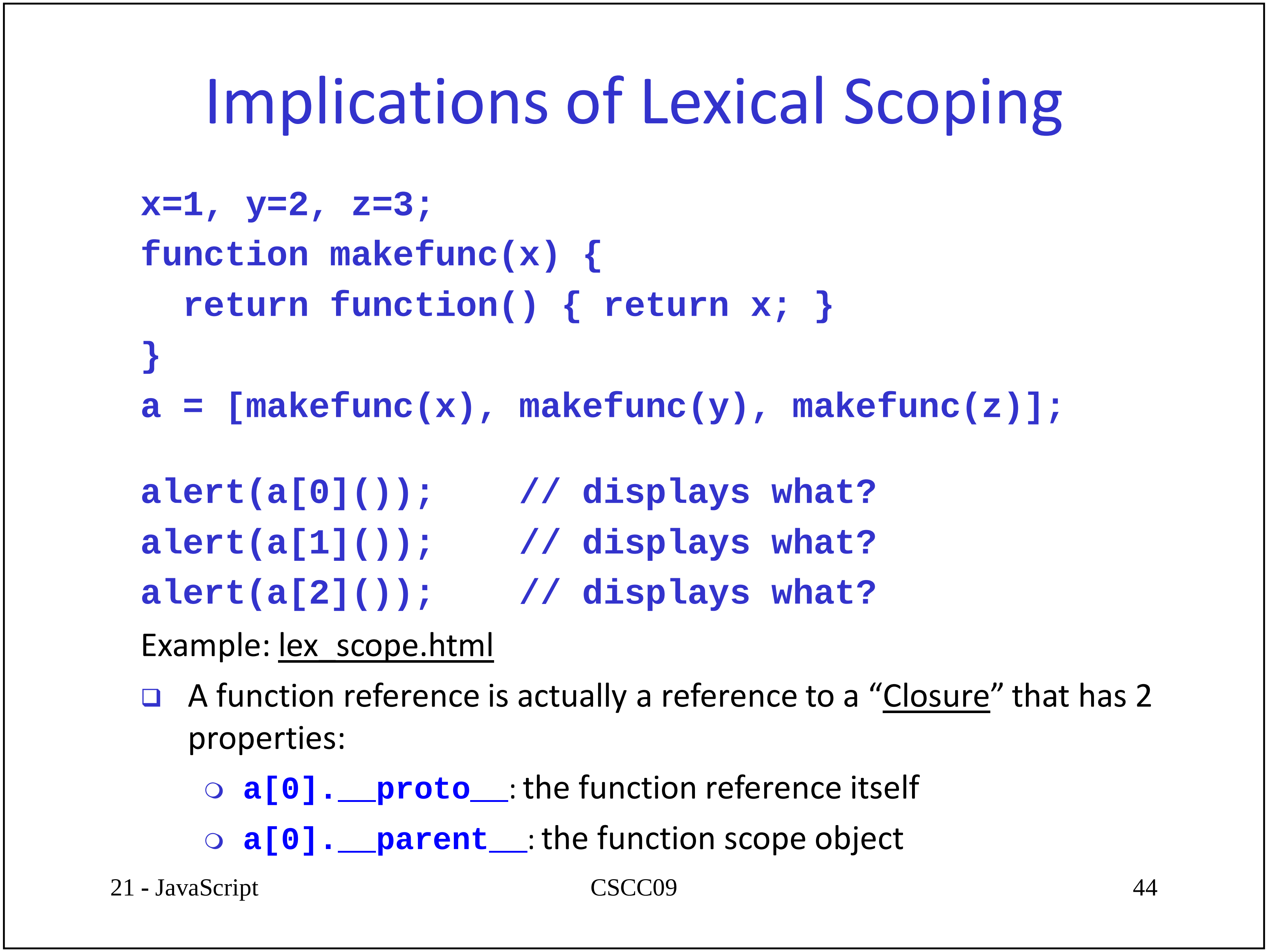


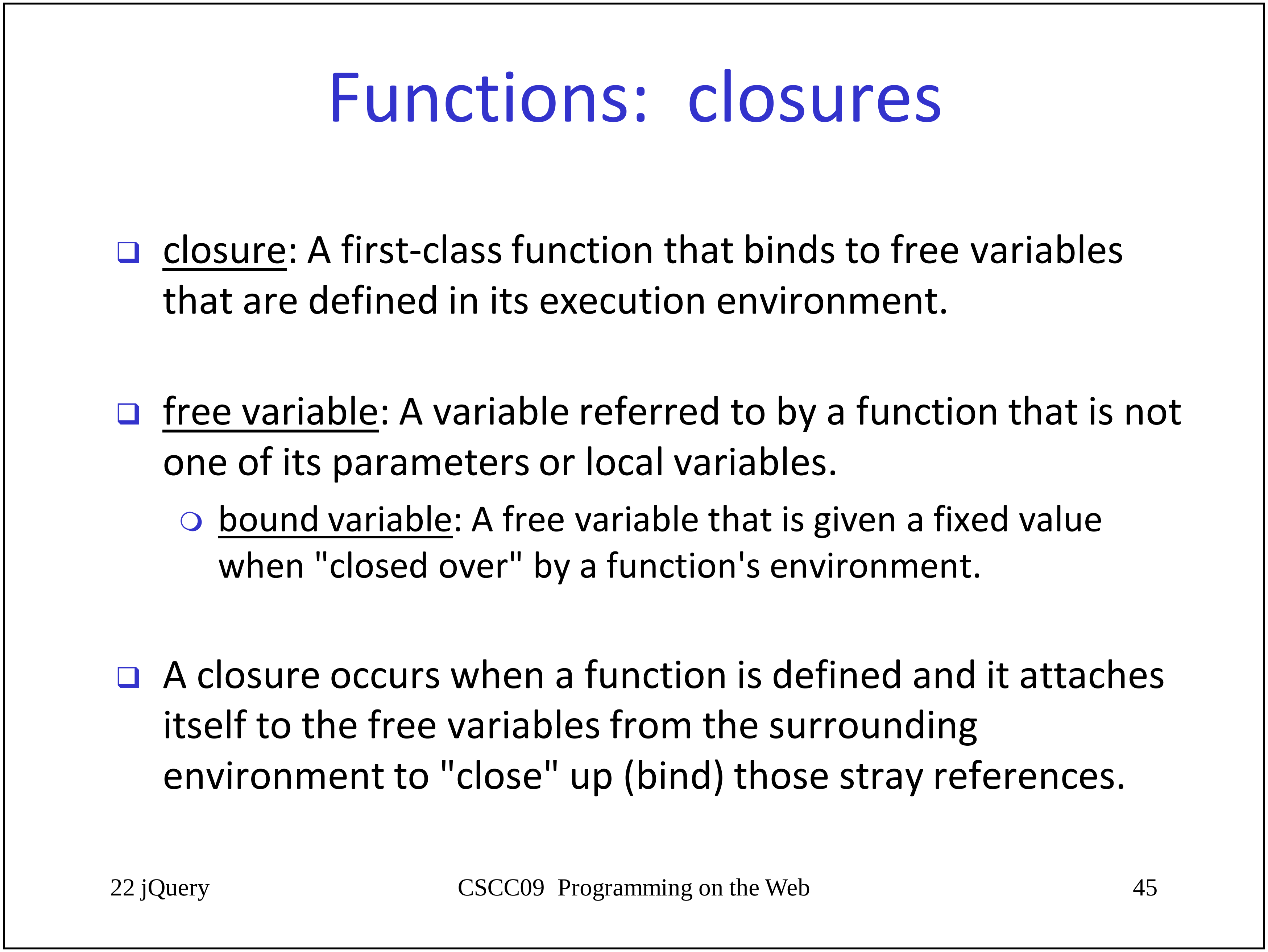
Implications of Lexical Scoping

```
x=1, y=2, z=3;
function makefunc(x) {
  return function() { return x; }
}
a = [makefunc(x), makefunc(y), makefunc(z)];

alert(a[0]());    // displays what?
alert(a[1]());    // displays what?
alert(a[2]());    // displays what?
```

Example: lex_scope.html

- A function reference is actually a reference to a “Closure” that has 2 properties:
  - `a[0].__proto__`: the function reference itself
  - `a[0].__parent__`: the function scope object

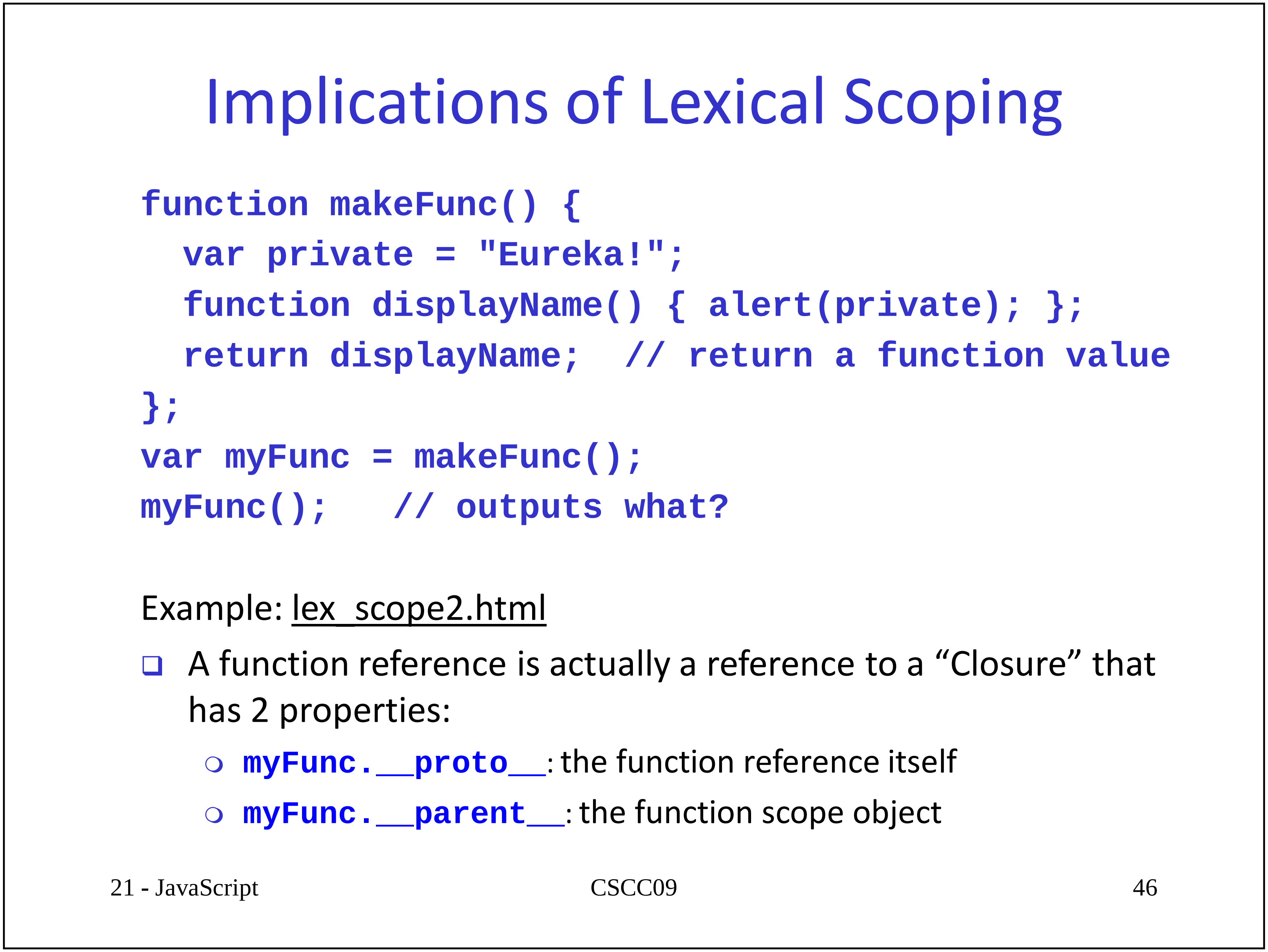


Functions: closures

- 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 (bind) those stray references.

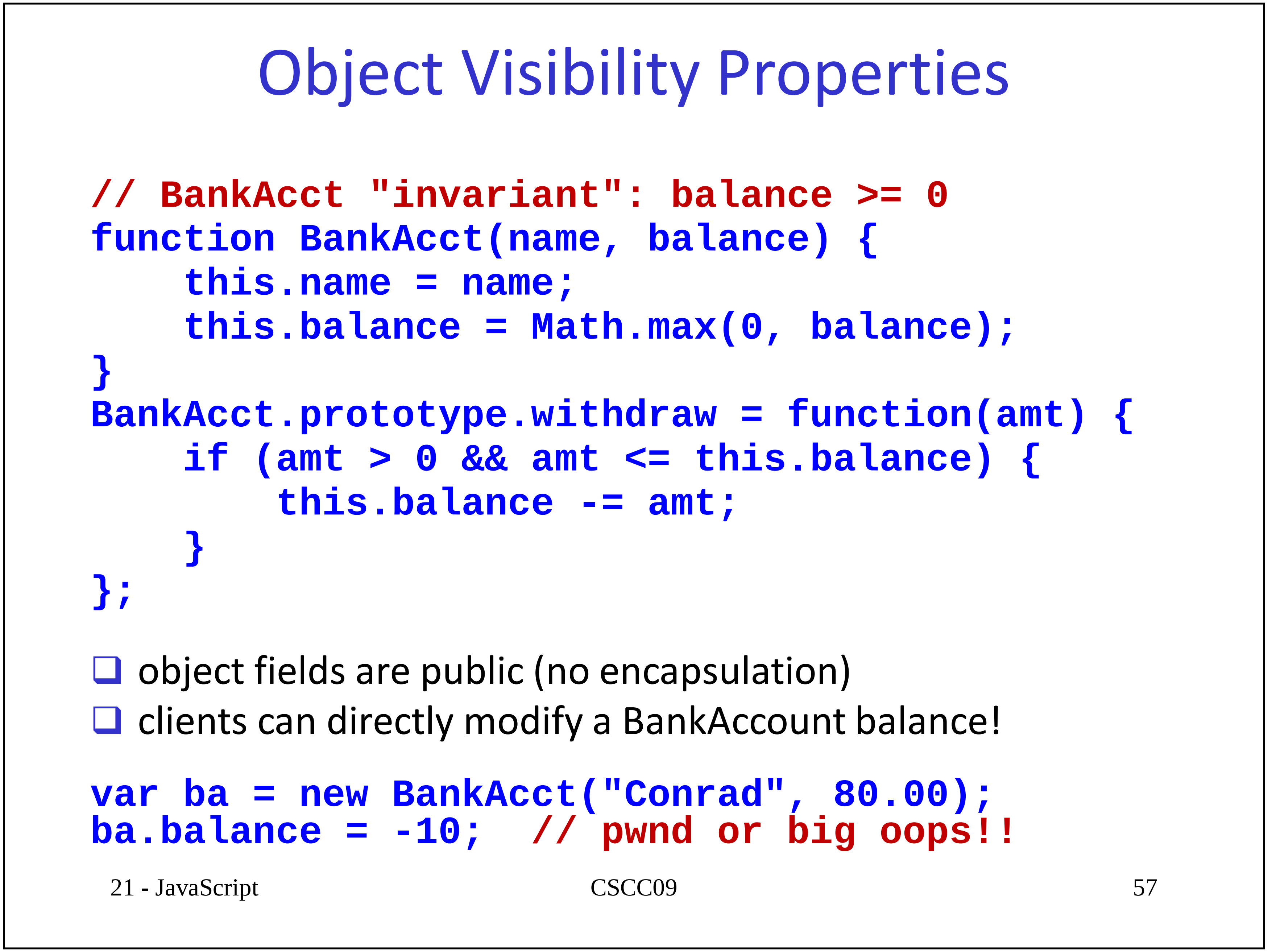


Implications of Lexical Scoping

```
function makeFunc() {
  var private = "Eureka!";
  function displayName() { alert(private); };
  return displayName;  // return a function value
};
var myFunc = makeFunc();
myFunc();   // outputs what?
```

Example: lex_scope2.html

- A function reference is actually a reference to a “Closure” that has 2 properties:
  - `myFunc.__proto__`: the function reference itself
  - `myFunc.__parent__`: the function scope object

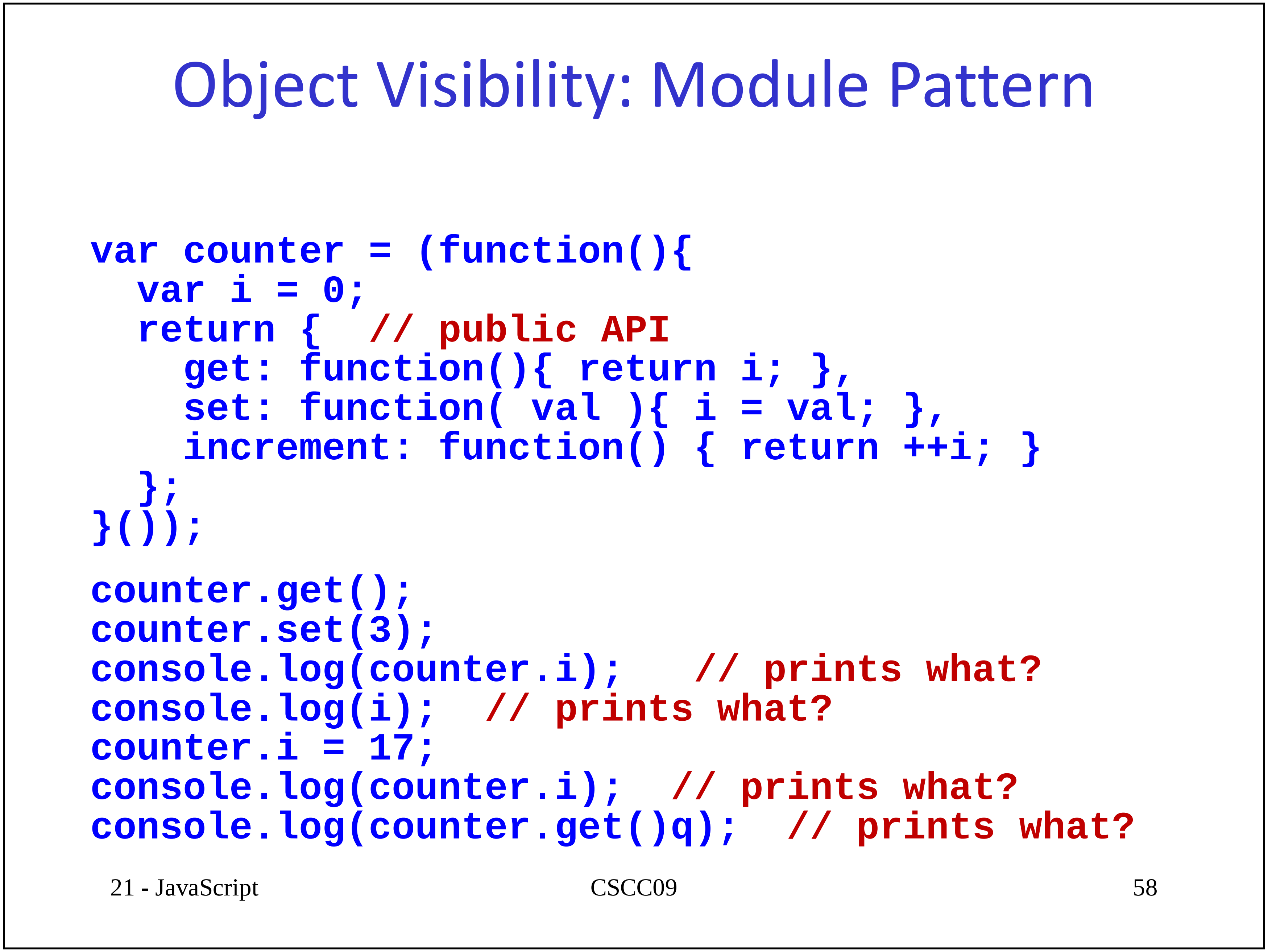


Object Visibility Properties

```
// BankAcct "invariant": balance >= 0
function BankAcct(name, balance) {
    this.name = name;
    this.balance = Math.max(0, balance);
}
BankAcct.prototype.withdraw = function(amt) {
    if (amt > 0 && amt <= this.balance) {
        this.balance -= amt;
    }
};
```

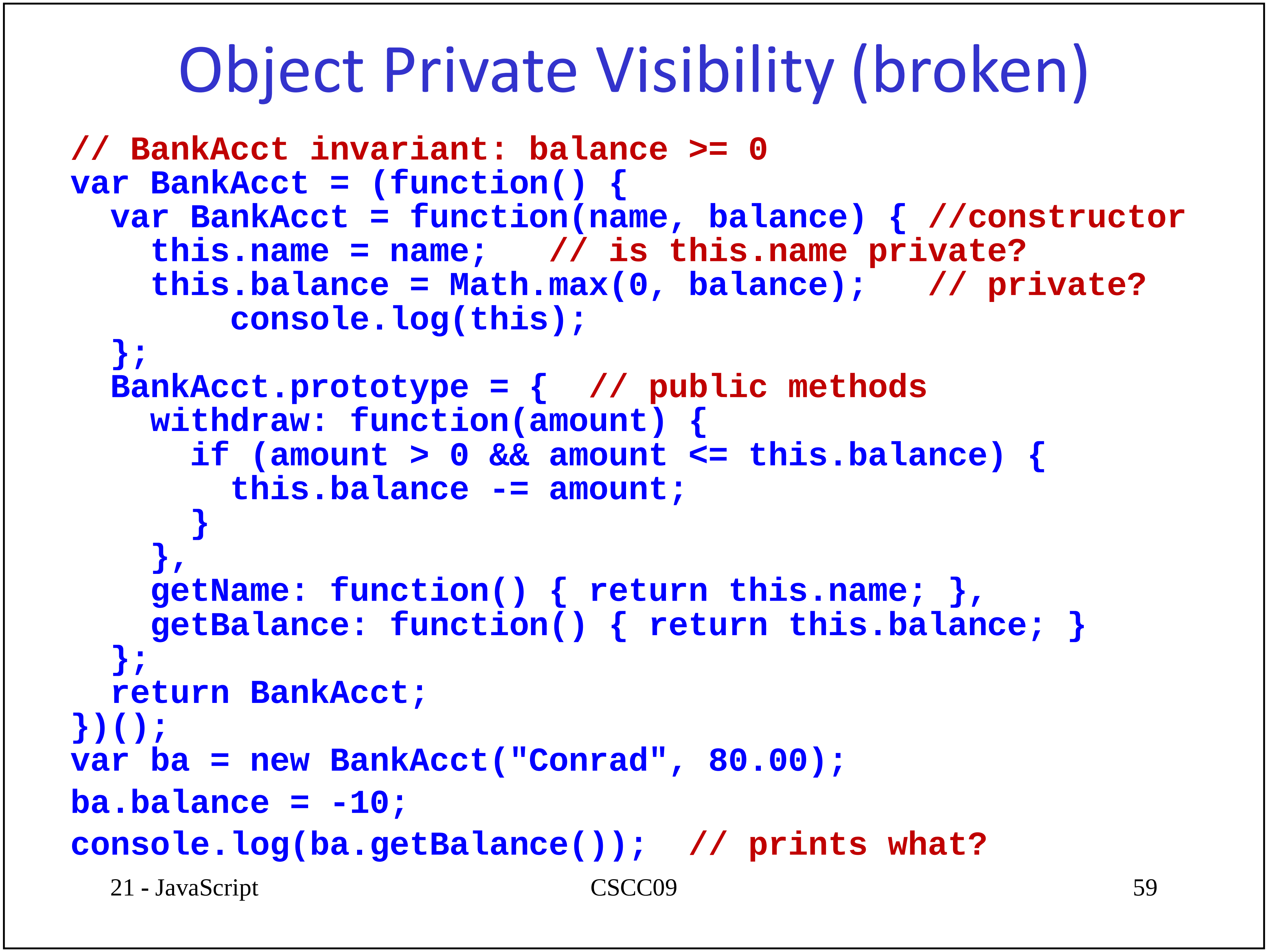
- ❑ object fields are public (no encapsulation)
- ❑ clients can directly modify a BankAccount balance!

```
var ba = new BankAcct("Conrad", 80.00);
ba.balance = -10;  // pwnd or big oops!!
```



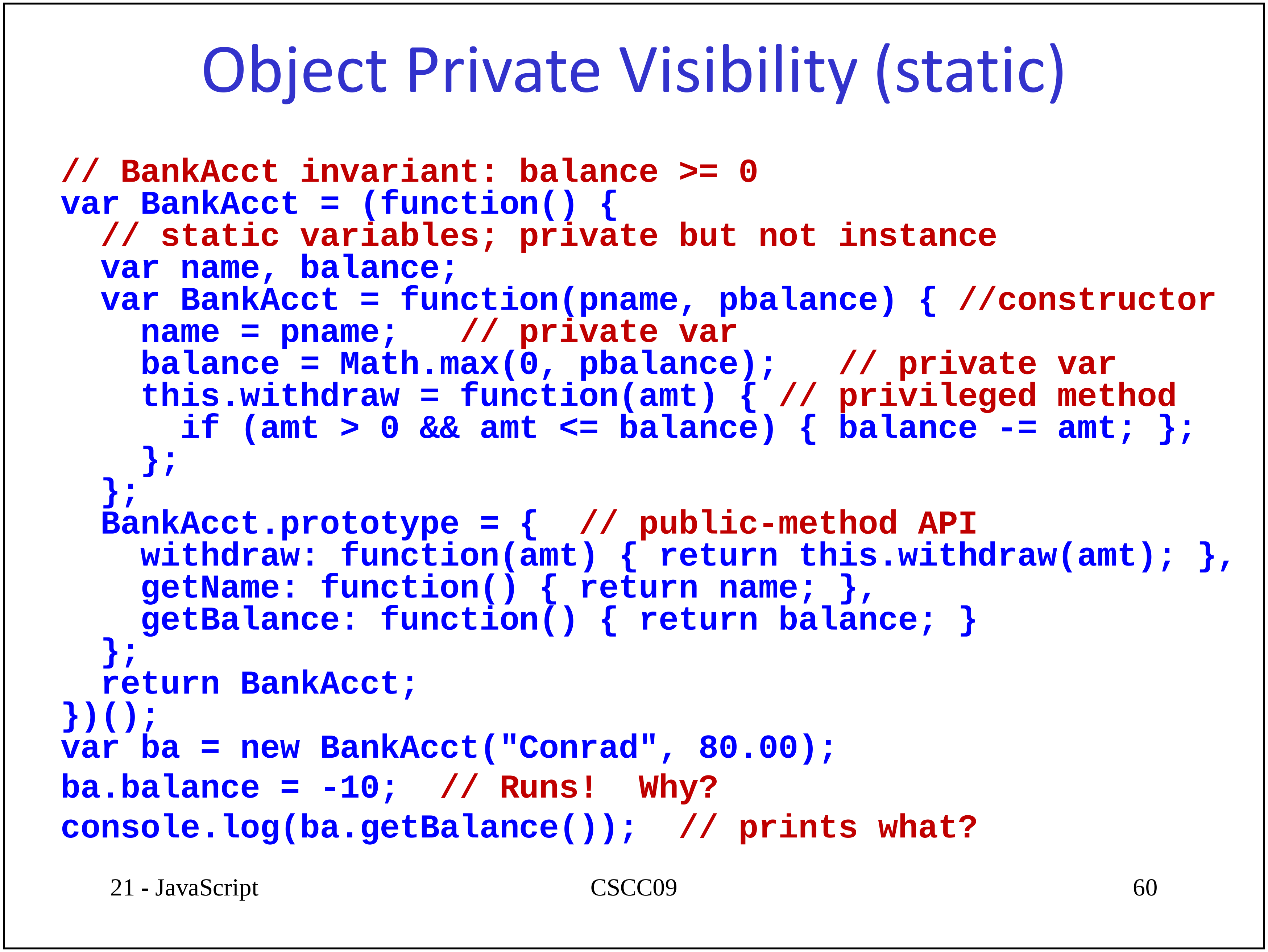
Object Visibility: Module Pattern

```
var counter = (function(){
  var i = 0;
  return {  // public API
    get: function(){ return i; },
    set: function( val ){ i = val; },
    increment: function() { return ++i; }
  };
}());
counter.get();
counter.set(3);
console.log(counter.i);   // prints what?
console.log(i);  // prints what?
counter.i = 17;
console.log(counter.i);  // prints what?
console.log(counter.get()q);  // prints what?
```



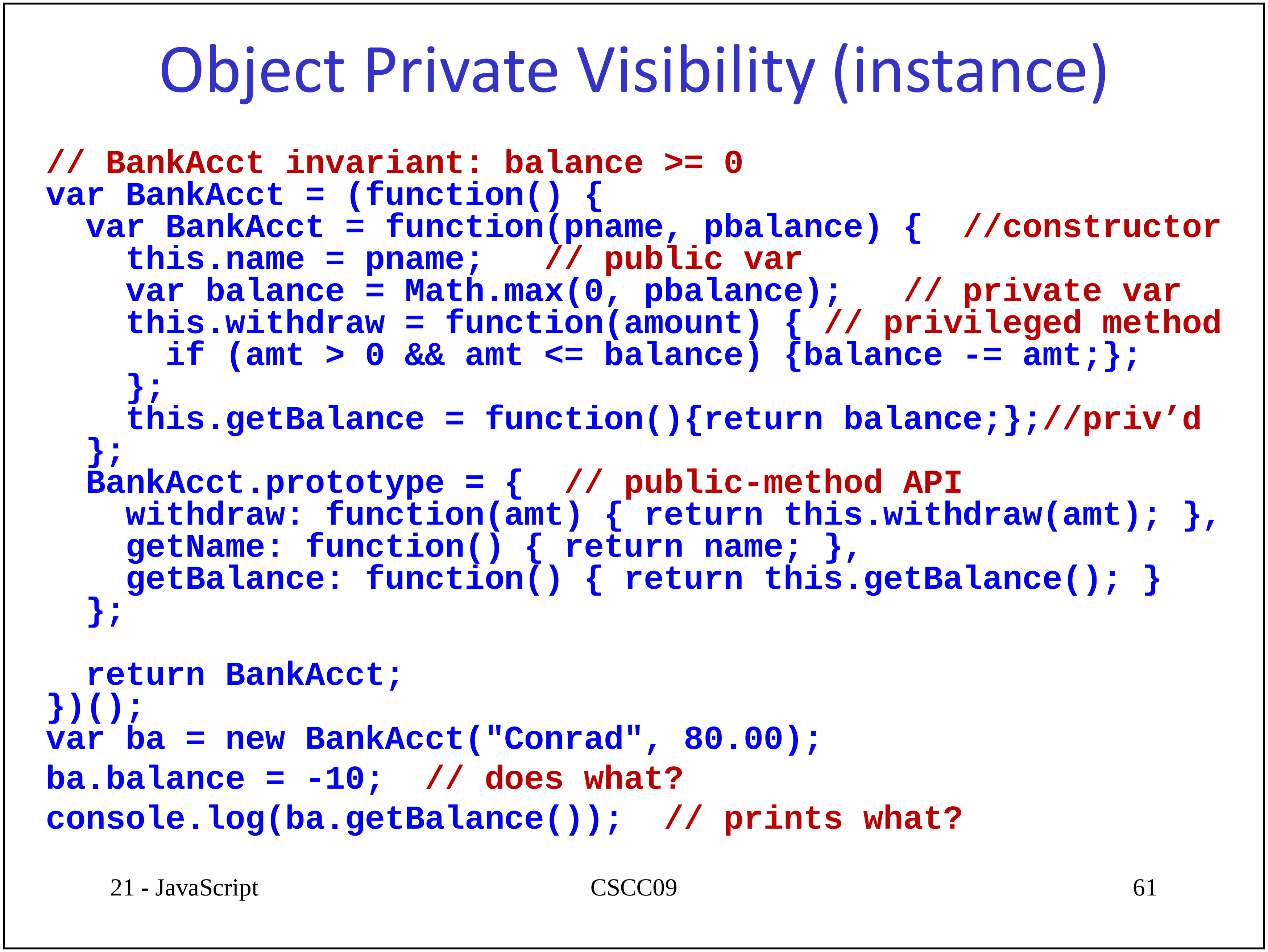
Object Private Visibility (broken)

```
// BankAcct invariant: balance >= 0
var BankAcct = (function() {
  var BankAcct = function(name, balance) { //constructor
    this.name = name;   // is this.name private?
    this.balance = Math.max(0, balance);   // private?
        console.log(this);
  };
  BankAcct.prototype = {  // public methods
    withdraw: function(amount) {
      if (amount > 0 && amount <= this.balance) {
        this.balance -= amount;
      }
    },
    getName: function() { return this.name; },
    getBalance: function() { return this.balance; }
  };
  return BankAcct;
})();
var ba = new BankAcct("Conrad", 80.00);
ba.balance = -10;
console.log(ba.getBalance());  // prints what?
```



Object Private Visibility (static)

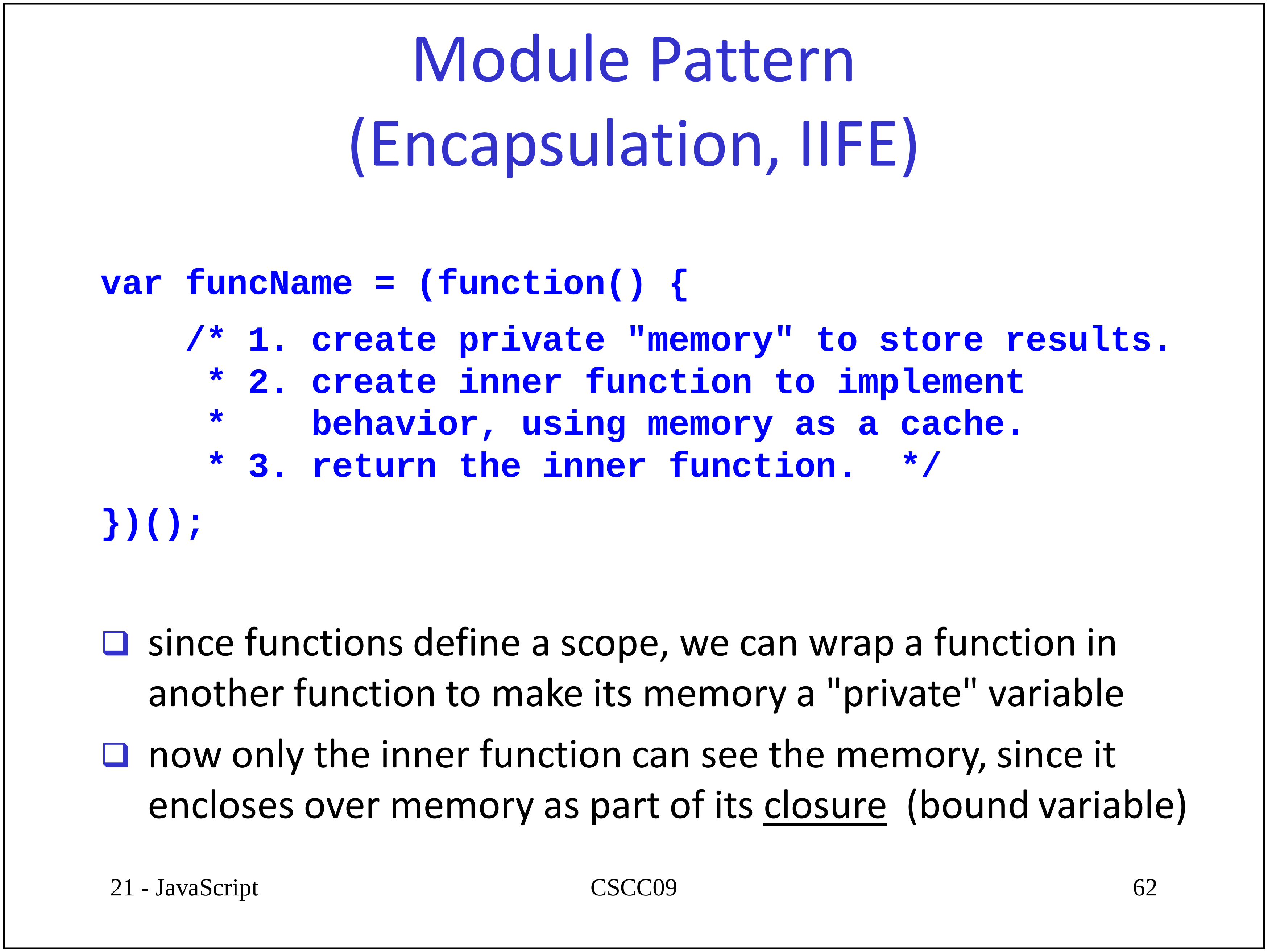
```
// BankAcct invariant: balance >= 0
var BankAcct = (function() {
  // static variables; private but not instance
  var name, balance;
  var BankAcct = function(pname, pbalance) { //constructor
    name = pname;   // private var
    balance = Math.max(0, pbalance);   // private var
    this.withdraw = function(amt) { // privileged method
      if (amt > 0 && amt <= balance) { balance -= amt; };
    };
  };
  BankAcct.prototype = {  // public-method API
    withdraw: function(amt) { return this.withdraw(amt); },
    getName: function() { return name; },
    getBalance: function() { return balance; }
  };
  return BankAcct;
})();
var ba = new BankAcct("Conrad", 80.00);
ba.balance = -10;  // Runs!  Why?
console.log(ba.getBalance());  // prints what?
```



Object Private Visibility (instance)

```
// BankAcct invariant: balance >= 0
var BankAcct = (function() {
  var BankAcct = function(pname, pbalance) {  //constructor
    this.name = pname;   // public var
    var balance = Math.max(0, pbalance);   // private var
    this.withdraw = function(amount) { // privileged method
      if (amt > 0 && amt <= balance) {balance -= amt;};
    };
    this.getBalance = function(){return balance;};//priv’d
  };
  BankAcct.prototype = {  // public-method API
    withdraw: function(amt) { return this.withdraw(amt); },
    getName: function() { return name; },
    getBalance: function() { return this.getBalance(); }
  };

  return BankAcct;
})();
var ba = new BankAcct("Conrad", 80.00);
ba.balance = -10;  // does what?
console.log(ba.getBalance());  // prints what?
```



Module Pattern (Encapsulation, IIFE)

```
var funcName = (function() {
    /* 1. create private "memory" to store results.
     * 2. create inner function to implement
     *    behavior, using memory as a cache.
     * 3. return the inner function.  */
})();
```

- ❑ since functions define a scope, we can wrap a function in another function to make its memory a "private" variable
- ❑ now only the inner function can see the memory, since it encloses over memory as part of its closure (bound variable)

Event Propagation

- When an event occurs on a DOM element (on a target element), that event first triggers any handlers registered for that target element
- That same event then fires on the target element's parent element,
  - and then on the parent's parent,
  - and so on up to the body element,
  - and finally the top-level document object to a default browser-event handler
    - e.g. click a link, default is to transition to the linked Web page.
- This is referred to as event propagation or bubbling

Event Propagation

- This is referred to as event propagation or bubbling
- Can you think of a situation in which this behavior might be useful?

- This propagation behavior can be suppressed, e.g. to avoid secondary handlers from activating, by calling
  - `event.stopPropagation()`to block propagation to parent event handlers and/or
  - `event.preventDefault()` to block execution of a browser-default event handler.