ADVANCED JAVASCRIPT

Agenda

- Look at some JavaScript pitfalls and best practices
- Understand how to simulate major Object Oriented concepts
- altJS

Automatic Initialization

- Like other modern programming languages, JavaScript supports automatic initialization
- The value of uninitialized variable is undefined
  - Not the same as null value

```
var num;

console.log(num == undefined);
```

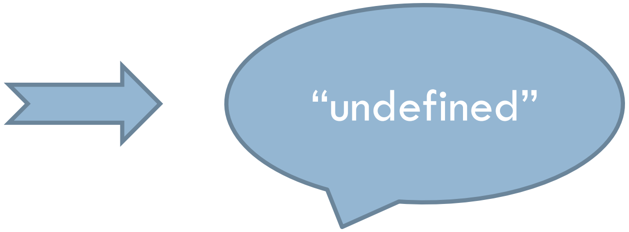
Undeclared Variable

- You cannot read a value of undeclared variable

```
try {
    if (xxx == 10) {
    }
}
catch (e) {
    console.log(e.message);
}
```

- You can ask for the typeof of an undeclared variable

```
console.log(typeof xxx);
```



“undefined”

Implicit Variable Declaration

- You can write into a variable even when this variable was not declared before
- Don’t do this !
- In this case a global variable is created

```
function () {
    global = 12;
    var local = ”abc”;
}

alert(local);
```

- Strict mode fixes that

Strict Mode

- Opt in to a restricted variant of JavaScript
- Makes several changes to “normal” JavaScript semantics
  - Less silent errors
  - Allows for better browser optimization
  - Prohibits future ECMAScript syntax
- Browsers not supporting strict mode will run code with different behavior

Applying Strict Mode

- Entire script (be aware of concatenation)
- Must come before any other statement

```
"use strict";

function one() { }

function two() { }
```

- Function scope (better)

```
function one() {
      "use strict";

      var x = 10;
}
```

- “use strict” effects declaration, not execution

```
(function strict() {
      "use strict";

      notStrict();
})();
```

```
function notStrict() {
      x = 12;
}
```

Strict Mode Changes

- □ Implicit variable declaration
- □ Invalid assignment throws an error

```
"use strict";

NaN = 10;
```

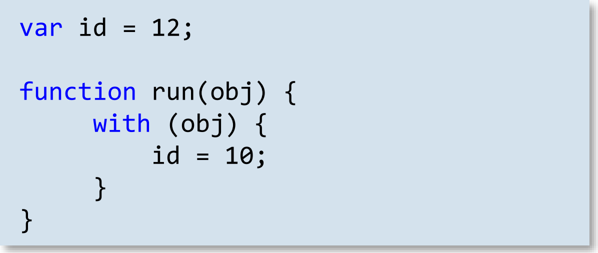
```
"use strict";

delete Object.prototype;
```

- □ Octal literals are gone

```
"use strict";

var num = 012;
```

with syntax kills optimization

- Consider the following



```
var id = 12;

function run(obj) {
     with (obj) {
         id = 10;
     }
}
```

- The JIT compiler cannot determine the location of the id variable
  - Can be a global one
  - Or, part of the obj parameter
- Therefore, strict mode prohibits the “with” syntax

eval

- □ Under strict mode cannot introduce new variables into the surrounding scope
- □ The following generates an error

```
"use strict";

eval("var x = 12;");

console.log(x);
```

- □ Below code is still supported

```
"use strict";

var x = 11;

eval("x = 12;");

console.log(x);
```

arguments, caller and callee

- □ Under non strict mode a function may access the arguments of another function

```
function g() {
    f();
}

function f() {
    console.log(g.arguments.length);
}

g(1, 2, 3);
```

- □ But this violates the concept of secured vs. privileged code
- □ Under strict mode a function can only access its own arguments

Window is the Global Scope

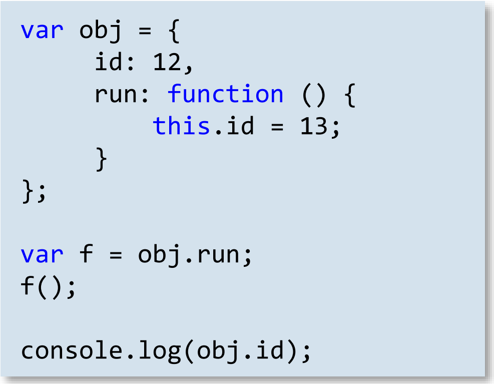
- □ Every global variable is a property of a global object named window

```
var num = 10;
console.log(window.num); //prints 10

window.num = 11;
console.log(num); // prints 11
```

- □ Objects in JavaScript are dynamic → Global scope is dynamic ☺
  - See next slides about objects

Global function and this

- □ Global function implicitly receives a reference to the global window object
- □ Might create surprising side effect

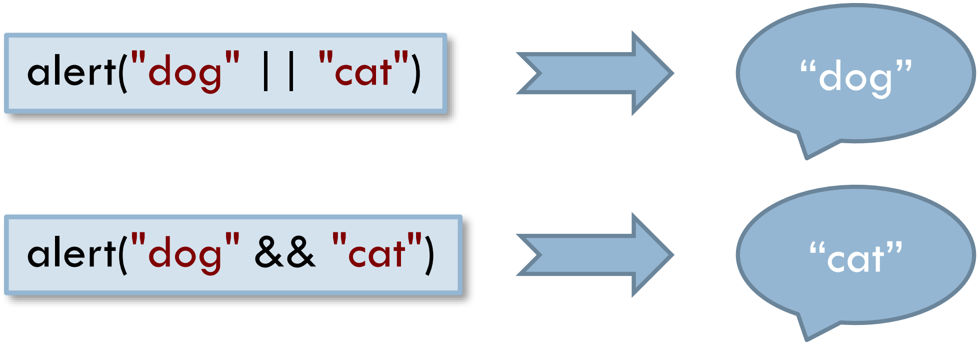


```
var obj = {
     id: 12,
     run: function () {
         this.id = 13;
     }
};

var f = obj.run;
f();

console.log(obj.id);
```

- □ Strict mode fixes that by setting this to undefined instead of the window object

Logical Operators

- Typically used with Boolean values
  - In that case, they return a Boolean value
  - Behavior is consistent with other static programming languages (C++/Java/C#)
- May be used with non Boolean values
  - In that case, they return a non-Boolean value



```
alert("dog" || "cat")
```

```
alert("dog" && "cat")
```

Where to declare variables ?

- A variable is accessible inside its surrounding function
- Even before point of declaration
- Therefore many JavaScript programmers declare all variables at the beginning of the method

```
var num = 11;

function doSomething() {
    console.log(num);
    var num = 10;
}

doSomething();
```

var is not block scoped

- A plain block (like for, if, else) does not create a scope
- Therefore, all below callbacks share the same i variable and print the same output

```
function runManyTasks() {
     for (var i = 0; i < 10; i++) {
          runTask(function () {
               console.log("Completed: " + i);
          });
     }
}

function runTask(completed) {
     setTimeout(completed, 1500); }

runManyTasks();
```

Overloading

- □ JavaScript does not support Overloading
- □ Last method wins
- □ You can simulate it

```
var ERR = "ERR";
var WRN = "WRN";
var MSG = "MSG";

function log(type, message) {
    if (message == undefined) {
        message = type;
        type = MSG;
    }

    console.log(type + " " + message);
}
```

```
log(ERR, "Internal Error");
log("Connecting to server");
```

Function inside an Object

- An object can contain functions

```
var obj = {
    id: 123,
    dump: function() {
        console.log("dumping: " + this.id);
    }
};

obj.dump();
```

- Feels like OOP
- The keyword this is used for accessing other properties (see next slide)

Function – Indirect Invocation

- A function can be invoked using special syntax

```
function f(name) {
    console.log("Hello " + name);
 }

f.call({}, "Ori");
f.apply({}, ["Ori"]);
```

- Although not intuitive, above syntax is quite common
- Mainly, when doing Object Oriented JavaScript
- Allows you to control the value of this

Function creates a Scope

- □ Function creates a new scope which is isolated from outer scope
- □ Outer scope cannot access local variables of a function

```
var num = 20;

function f() {
    var num = 10;

    console.log(num); // yields 10
}

f();

console.log(f.num); // yields undefined
```

Closure

- Inner function may access the local variables of the outer function
  - Even after outer function completes execution
- Allows us to simulate stateful function

```
function getCounter() {
    var num = 0;
    function f() {
        ++num;
        console.log("Num is " + num);
    }
    return f;
}
```

```
var counter = getCounter();
counter();
counter();
```

Self Executing Function

- □ A function can declared without a name
- □ Since no name exist no one can invoke it
- □ Except the code that declared it
- □ A.K.A self executing function

```
(function () {
    // External code has no access to these variables
    var url = "http://www.google.com";
    var productKey = "ABC";
})();
```

Sending Parameters

- Think about the $ sign
- Usually it points to jQuery global object
- But how can we ensure that?
  - There might be a case were additional 3rd party library overrides it

```
(function ($) {
    $.ajax({
        url: "www.google.com",
        type: "GET",
    });
})(jQuery);
```

Module

- □ Arrange your JavaScript code into modules
- □ Each module is surrounded with self executing function thus hiding all local variables and functions
- □ Peek the ones that should be public (sparsely)

```
var Server = (function () {
    var baseUrl = "http://www.google.com";

    function httpGet(relativeUrl) {
        $.ajax(...);
    }

    return {
        httpGet: httpGet,
    };
 })();
```

Summary

- JavaScript is quite ugly
- But is has some good parts
  - See Crockford’s book “JavaScript the good parts”
- Module pattern is the basic any king of serious JavaScript programming