MODULE 3

JAVASCRIPT

Overview

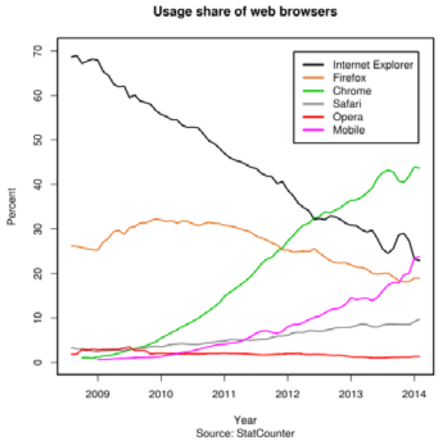
- Introducing the JavaScript Language
- Objects
- Arrays
- Strings
- Functions
- Variables

The JavaScript Language

- JavaScript is interpreted, not compiled.
- JavaScript was designed to work in web browsers.
- JavaScript is the standard technology for executing logic in web pages in web browsers.
- Server-side technologies generate clientside JavaScript to execute in a web browser.

JavaScript Engines

- Chrome's V8
- Firefox's IonMonkey
- Opera switched to V8 with the version 14


Usage share of web browsers

Source: StatCounter

JavaScript Life Cycle

- When a new web pages is loaded, the previous JavaScript variables and functions are cleared, they do not persist.
- For the new page, declarations of global variables and global-level functions, are initialized.
- Statements at global level executed. Functions can be called.
- User and system events can trigger function calls.

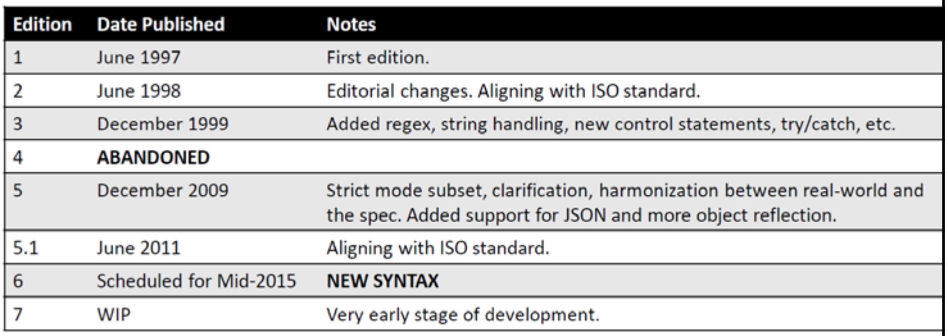
Prototype-based languages

- They instantiate a new object by copying (or cloning) an instance of another object.
- The structure of every new object is based on the dynamic structure of another object.
- JavaScript is NOT a pure prototype-based language, because the new operator cannot be applied to any object instance.

The JavaScript Language Is ECMAScript

- JavaScript is a registered trademark of Sun Microsystems, Inc.
- European Computer Manufacturers Association (ECMA)
- ECMA-262 - ECMAScript Language Specification

Versions



Edition	Date Published	Notes
1	June 1997	First edition.
2	June 1998	Editorial changes. Aligning with ISO standard.
3	December 1999	Added regex, string handling, new control statements, try/catch, etc.
4	**ABANDONED**	
5	December 2009	Strict mode subset, clarification, harmonization between real-world and the spec. Added support for JSON and more object reflection.
5.1	June 2011	Aligning with ISO standard.
6	Scheduled for Mid-2015	**NEW SYNTAX**
7	WIP	Very early stage of development.

- ES1 – 1997
- ES2 – 1998
- ES3 – 1999
- ES4 – RIP

- ES5 – 2009
- ES5.1 – 2011
- ES2015 - 2015 (ES6)
- ES2016 – 2016(ES7, JS.next)

Things to Remember

- Decide which versions of JavaScript your application supports.
- Be sure that any JavaScript features you use are supported by all environments where your application runs.
- Always test strict code in environments that perform the strict mode checks.
- Beware of concatenating scripts that differ in their expectations about strict mode.

UNDERSTAND JAVASCRIPT’S FLOATING-POINT NUMBERS

Numbers

- All numbers in JavaScript are double-precision floating-point numbers
- 64-bit encoding of numbers specified by IEEE 754
  - typeof 17; // "number"
  - typeof 98.6; // "number"
  - typeof -2.1; // "number“
- All of the integers from –9,007,199,254,740,992 (–2^{53}) to 9,007,199,254,740,992 (2^{53}) are valid doubles.

Arithmetic operators

- Most arithmetic operators work with integers, real numbers, or a combination of the two:
- 0.1 * 1.9 // 0.19
- -99 + 100; // 1
- 21 - 12.3; // 8.7
- 2.5 / 5; // 0.5
- 21 % 8; // 5

Arithmetic operators

> $(x + y) + z = x + (y + z)$

Arithmetic operators

> $(x + y) + z = x + (y + z)$

> But this is not always true of floating-point numbers:

Arithmetic operators

- (*x* + *y*) + *z* = *x* + (*y* + *z*)
- But this is not always true of floating-point numbers:

```
(0.1 + 0.2) + 0.3; // 0.6000000000000001
0.1 + (0.2 + 0.3); // 0.6
```

The bitwise arithmetic operators

- The bitwise arithmetic operators implicitly convert operands to 32-bit integers
  - 32-bit, big-endian, two’s complement integers
- 8 | 1; // 9
- (8).toString(2); // "1000“
- parseInt("1001", 2); // 9

Be careful

- 2-1.1 // 0.8999999999999999
- 100000000000000100
- x=x+50

Things to Remember

- JavaScript numbers are double-precision floating-point numbers.
- Integers in JavaScript are just a subset of doubles rather than a separate datatype.
- Bitwise operators treat numbers as if they were 32-bit signed integers.
- Be aware of limitations of precisions in floating-point arithmetic.

IMPLICIT COERCIONS

Beware of Implicit Coercions

- JavaScript can be surprisingly forgiving when it comes to type errors.
- Many languages consider an expression like

```
3 + true;
```

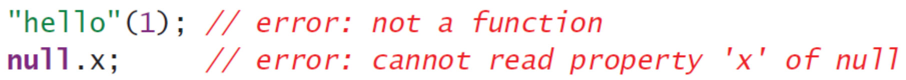
Beware of Implicit Coercions

- JavaScript can be surprisingly forgiving when it comes to type errors.
- Many languages consider an expression like

```
3 + true; // 4
```

Beware of Implicit Coercions

- There are a handful of cases in JavaScript where providing the wrong type produces an immediate error, such as calling a non-function or attempting to select a property of **`null`**:



```
"hello"(1); // error: not a function
null.x;     // error: cannot read property 'x' of null
```

Beware of Implicit Coercions

> But in many other cases, rather than raising an error, JavaScript *coerces* a value to the expected type by following various automatic conversion protocols.

```
"2" + 3; // "23"
2 + "3"; // "23"
1 + 2 + "3";
```

Beware of Implicit Coercions

> But in many other cases, rather than raising an error, JavaScript *coerces* a value to the expected type by following various automatic conversion protocols.

```
"2" + 3; // "23"
2 + "3"; // "23"
1 + 2 + "3";     // "33"
```

Beware of Implicit Coercions

> But in many other cases, rather than raising an error, JavaScript *coerces* a value to the expected type by following various automatic conversion protocols.

```
"2" + 3; // "23"
2 + "3"; // "23"
1 + 2 + "3";     // "33"
(1 + "2") + 3;
```

Beware of Implicit Coercions

> But in many other cases, rather than raising an error, JavaScript *coerces* a value to the expected type by following various automatic conversion protocols.

```
"2" + 3; // "23"
2 + "3"; // "23"
1 + 2 + "3";     // "33"
(1 + "2") + 3;
"17" * 3;
"8" | "1";
```

Beware of Implicit Coercions

> But in many other cases, rather than raising an error, JavaScript *coerces* a value to the expected type by following various automatic conversion protocols.

```
"2" + 3; // "23"
2 + "3"; // "23"
1 + 2 + "3";     // "33"
(1 + "2") + 3;
"17" * 3;  // 51
"8" | "1"; // 9
```

Beware of Implicit Coercions

> But in many other cases, rather than raising an error, JavaScript *coerces* a value to the expected type by following various automatic conversion protocols.

```
"2" + 3; // "23"
2 + "3"; // "23"
1 + 2 + "3";     // "33"
(1 + "2") + 3;
"17" * 3;  // 51
"8" | "1"; // 9
```

```
var x = NaN;
x === NaN;
```

Beware of Implicit Coercions

> But in many other cases, rather than raising an error, JavaScript *coerces* a value to the expected type by following various automatic conversion protocols.

```
"2" + 3; // "23"
2 + "3"; // "23"
1 + 2 + "3";     // "33"
(1 + "2") + 3;
"17" * 3;   // 51
"8" | "1"; // 9
var x = NaN;
x === NaN;     // false
```

Beware of Implicit Coercions

> But in many other cases, rather than raising an error, JavaScript *coerces* a value to the expected type by following various automatic conversion protocols.

```
"2" + 3; // "23"
2 + "3"; // "23"
1 + 2 + "3";     // "33"
(1 + "2") + 3;
"17" * 3;   // 51
"8" | "1"; // 9
var x = NaN;
x === NaN;     // false
isNaN(NaN); // true
```

Beware of Implicit Coercions

> Other values that are definitely not NaN, yet are nevertheless coercible to NaN, are indistinguishable to isNaN:

```
isNaN("foo");              // true
isNaN(undefined);          // true
isNaN({});                 // true
isNaN({ valueOf: "foo" }); // true
```

Beware of Implicit Coercions

> Since NaN is the only JavaScript value that is treated as unequal to itself, you can always test if a value is NaN by checking it for equality to itself:

```
var a = NaN;
a !== a;                    // true
var b = "foo";
b !== b;                    // false
var c = undefined;
c !== c;                    // false
var d = {};
d !== d;                    // false
var e = { valueOf: "foo" };
e !== e;                    // false
```

Beware of Implicit Coercions

> Abstract this pattern into a clearly named utility function:

```
function isReallyNaN(x) {
    return x !== x;
}
```

Beware of Implicit Coercions

> Objects can also be coerced to primitives.

> Most commonly used for converting to strings:

```
// "the Math object: [object Math]"
"the Math object: " + Math;
// "the JSON object: [object JSON]"
"the JSON object: " + JSON;
```

Beware of Implicit Coercions

- Objects are converted to strings by implicitly calling their **`toString`** method.
- Test by calling it yourself:

```
Math.toString(); // "[object Math]"
JSON.toString(); // "[object JSON]"
```

Beware of Implicit Coercions

- Objects are converted to strings by implicitly calling their **`toString`** method.
- Test by calling it yourself:

```
 Math.toString(); // "[object Math]"
 JSON.toString(); // "[object JSON]"
"J" + { toString: function() { return "S"; } };
2 * { valueOf: function() { return 3; } };
```

Beware of Implicit Coercions

- Objects are converted to strings by implicitly calling their **`toString`** method.
- Test by calling it yourself:

```
Math.toString(); // "[object Math]"
JSON.toString(); // "[object JSON]"
```

```
"J" + { toString: function() { return "S"; } }; // "JS"
2 * { valueOf: function() { return 3; } };       // 6
```

Beware of Implicit Coercions

```
var obj = {
    toString: function() {
        return "[object MyObject]";
    },
    valueOf: function() {
        return 17;
    }
};
"object: " + obj;
```

Beware of Implicit Coercions

```
var obj = {
    toString: function() {
        return "[object MyObject]";
    },
    valueOf: function() {
        return 17;
    }
};
"object: " + obj; // "object: 17"
```

Things to Remember

> Type errors can be silently hidden by implicit coercions.

> The + operator is overloaded to do addition or string concatenation depending on its argument types.

> Objects are coerced to numbers via **`valueOf`** and to strings via **`toString`**.

> Objects with **`valueOf`** methods should implement a **`toString`** method that provides a string representation of the number produced by **`valueOf`**.

> Use **`typeof`** or comparison to **`undefined`** rather than truthiness to test for undefined values.

UNDERSTAND JAVASCRIPT'S OBJECTS

JavaScript Objects

> A JavaScript object is an unordered collection of properties.

> Everything in a JavaScript program is an object.

> To make a new object:

- x=new Object();
    - Makes a clone of the default Built-In Object called Object.
    - Assigns it to a global variable x.
    - x is empty except for the built-in properties and methods.

New Instances of Built-In Objects

- Create a new empty object and place a reference to that object in the global variable x

```
x = new Object();
// alternatively: x = {};
```

- Create an instance of a Number object with value 1 and place a reference to that object in the global variable one

```
one = new Number(1);
// alternatively: one = 1;
```

- Create an instance of a String object with a value "HelloWorld" and place a reference to that object in the global variable x

```
s = new String("Hello World")
// alternatively: s = "Hello World";
```

Instance Variables Are Called Properties

```
x = new Object();
x.top = 5;
```

- The assignment algorithm works like this:
  - Check if object x has a property named top
  - If it does not:
    - Add a new property to x
    - Name the property top
  - Overwrite the value of the property top with the new value, in this example: 5.

Object Literal Notation (1/2)

`x = new Object();` is the same as `x = {};`

```
x = new Object();
x.top = 5;
x.left = 10;
x.y = new Object();
```

is the same as

```
x = {
   top : 5,
   left : 10,
   y : {}
};
```

Object Literal Notation (2/2)

```
x = new Object();
x.top = 5;
x.left = 10;
x.y = new Object();
x.y.color = “red”;
x.y.state = true;
```

is the same as

```
x = { top : 5,
      left : 10,
      y : { color: “red”, state: true }
};
```

Accessing Properties (1/2)

- Dot notation is equivalent to Associative Array syntax
- Given the example so far:

```
x = {top:5, left:10, y:{}};
```

`x.top` is the same as `x['top']`

`x.left` is the same as `x[“left”]`

`x.y` is the same as `x['y']`

Accessing Properties (2/2)

- A property name can be any value that can be evaluated as a string.
- For example, the following are legitimate properties in the JavaScript language.

```
x = new Object();
x[1] = "one";
x[1.1] = "one dot one";
x[-1] = false;
X['10']= 5;// Also accessed using x[10]
```

- **x** is not an not Array object.
  - The numbers are converted to strings, and used as property names.

Arrays

> Arrays extend the Object with an additional feature:
  - The ability to use an integer index to access member values.

Arrays – Basic Syntax

```
// creates an empty array object
a = new Array();
// the same a previous statement
a = new Array(0);
x = a.length; // assigns 0 to x
y = a[0]; // y gets value 'undefined'
// creates an array object with length 7 b = new
Array(7);
x = b.length; // assigns 7 to x
y = b[0]; // y gets 'undefined' value
...
y = b[6]; // y gets 'undefined' value
```

Length and Object Properties

> Arrays are objects with an additional feature:

  - The ability to use integer index to access members.

```
a = new Array();
//adds sting to 1st position in array
a[0] = "uno";
//adds an object to 2nd position in array
a[1] = {dos:2};
x = a.length; // assigns 2 to x
// puts a boolean value into PROPERTY
a['aim'] = true;
x = a.length; // assigns 2 to x
y = a.aim; // assigns true to y
```

Arrays – Object Literal Notation

> The following two array declarations are equivalent:

```
c = new Array("uno", "dos", "tres")
c = ["uno", "dos", "tres"];
```

> Square brackets indicates the declaration of an Array

Set Array Length Dynamically

```
// creates an array object with length 4
a = ["uno", "dos", "tres", “cuatro” ];
```

- Increasing the length adds positions with value undefined:

```
// adds 3 new positions: a[4],a[5],a[6]
a.length = 7;
```

- Decreasing the length deletes the extra positions:

```
// the array has 2 positions: a[0],a[1]
a.length = 2;
```

- Assigning a value to a new position sets the array's length to one less than that position:

```
a[50] = “cincuenta”;
y = a.length // y is assigned value 51.
```

Multi-Dimensional Arrays

- Multi-dimensional arrays are arrays of arrays:

```
a =
[
  ["uno", "dos", "tres"],
  ["un", "deux","trois"],
  ["eins", "zwei", "drei"],
];
// y is assigned value "deux"
y = a[1][1];
```

Strings Are Objects

- Strings extend an Object with a string value.

Basic Syntax

- The value is not a property of the String Object.

```
x = new String("123");
x.status = true;
y = x; // y gets “123”
z = x.status // z gets true
```

- Typically, do not use a String’s Object properties

Use Single or Double Quote Marks

- You create a string using the String object's constructor method or with a simple assignment.
- Equivalent initializations:

```
a = 'This is a String';
a = "This is a String";
a = new String('This is a String');
a = new String("This is a String");
```

Escaping Quote Marks

- To embed a single or double quote inside a string, use the other quotation marks, or precede the quote mark with a backslash.
- Equivalent:

```
b = "It's using single 'quotation'
marks.";
b = 'It\'s using single \'quotation\'
marks.';
```

The length Property

- The length property returns the number of characters in the string.
- For example:

```
c = "123";
d = c.length; // d gets value 3
```

- Assigning string length has no effect:

```
c = "123";
c.length = 2;
d = c.length; // d gets value 3
```

Concatenation

- The + operator is used to concatenate strings:

```
a = "one";
b = "two";
// c gets value “one and two”
c = a + " and " + b;
```

- Concatenation can cast Numbers to String
- Left-to-right operator precedence applies:

```
x = 2 + 3 + ","; // x gets value “5,“
```

PREFER PRIMITIVES TO OBJECT WRAPPERS

Prefer Primitives to Object Wrappers

> You can create a String object that wraps a string value:

```
var s = new String("hello");
typeof "hello"; // "string"
typeof s;       // "object"

var s1 = new String("hello");
var s2 = new String("hello");
s1 === s2; // false
```

Things to Remember

- Object wrappers for primitive types do not have the same behavior as their primitive values when compared for equality.
- Getting and setting properties on primitives implicitly creates object wrappers.

AVOID USING == WITH MIXED TYPES

Avoid using == with Mixed Types

> What would you expect to be the value of this expression?

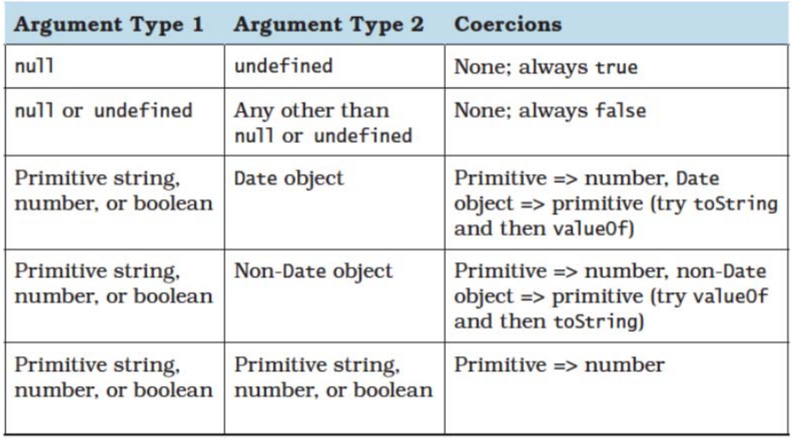
```
"1.0e0" == { valueOf: function() { return true; } };
```

Avoid using == with Mixed Types

> It's tempting to use these coercions for tasks like reading a field from a web form and comparing it with a number:

```
var today = new Date();

if (form.month.value == (today.getMonth() + 1) &&
    form.day.value == today.getDate()) {
    // happy birthday!
    // ...
}
```

Coercion Rules for the == Operator



Argument Type 1	**Argument Type 2**	**Coercions**
`null`	`undefined`	None; always `true`
`null` or `undefined`	Any other than `null` or `undefined`	None; always `false`
Primitive string, number, or boolean	Date object	Primitive => number, Date object => primitive (try `toString` and then `valueOf`)
Primitive string, number, or boolean	Non-Date object	Primitive => number, non-Date object => primitive (try `valueOf` and then `toString`)
Primitive string, number, or boolean	Primitive string, number, or boolean	Primitive => number

Things to Remember

> The == operator applies a confusing set of implicit coercions when its arguments are of different types.

> Use === to make it clear to your readers that your comparison does not involve any implicit coercions.

> Use your own explicit coercions when comparing values of different types to make your program's behavior clearer.

LEARN
THE LIMITS OF SEMICOLON INSERTION

Learn the Limits of Semicolon Insertion

- One of JavaScript's conveniences is the ability to leave off statement-terminating semicolons.
- Dropping semicolons results in a lightweight aesthetic:

```
function Point(x, y) {
    this.x = x || 0
    this.y = y || 0
}
Point.prototype.isOrigin = function() {
    return this.x === 0 && this.y === 0
}
```

- This works thanks to *automatic semicolon insertion,* a program parsing technique that infers omitted semicolons in certain contexts, effectively "inserting" the semicolon into the program for you automatically.

Learn the Limits of Semicolon Insertion

> The first rule of semicolon insertion is:

Semicolons are only ever inserted before a } token, after one or more newlines, or at the end of the program input.

```
function square(x) {
    var n = +x
    return n * n
}
function area(r) { r = +r; return Math.PI * r * r }
function add1(x) { return x + 1 }
```

> But this is error:

```
// error
function area(r) { r = +r return Math.PI * r * r }
```

Learn the Limits of Semicolon Insertion

> The second rule of semicolon insertion is:

Semicolons are only ever inserted when the next input token cannot be parsed.

```
a = b
(f());
```

parses just fine as a single statement, equivalent to:

```
a = b(f());
```

> B trast, this snippet

```
a = b
f();
```

is p o separate statements

```
a = b f(); // error
```

Learn the Limits of Semicolon Insertion

- Another common scenario is an array literal:

```
a = b
["r", "g", "b"].forEach(function(key) {
    background[key] = foreground[key] / 2;
});
```

- Looks like two statements: an assignment followed by a statement that calls a function on the strings "r", "g", and "b" in order.
- But because the statement begins with [, it parses as a single statement, equivalent to:

```
a = b["r", "g", "b"].forEach(function(key) {
    background[key] = foreground[key] / 2;
});
```

Learn the Limits of Semicolon Insertion

```
/Error/i.test(str) && fail();
```

- This statement tests a string with the case-insensitive regular expression /Error/i.

```
a = b
/Error/i.test(str) && fail();
```

- The code parses as a single statement equivalent to:

```
a = b / Error / i.test(str) && fail();
```

Learn the Limits of Semicolon Insertion

```
a = b     // semicolon inferred
var x     // semicolon inferred
(f())     // semicolon inferred
```

Learn the Limits of Semicolon Insertion

```
var x     // semicolon inferred
a = b     // no semicolon inferred
(f())     // semicolon inferred
```

Learn the Limits of Semicolon Insertion

```
return { };
```

- returns a new object

Learn the Limits of Semicolon Insertion

```
return
{ };
```

- parses as three separate statements, equivalent to:

```
return;
{ }
;
```

Learn the Limits of Semicolon Insertion

```
a
++
b
```

Learn the Limits of Semicolon Insertion

```
a
++
b
```

parses to

```
a; ++b;
```

Learn the Limits of Semicolon Insertion

```
for (var i = 0, total = 1 // parse error
     i < n
     i++) {
    total *= i
}

// parse error
function infiniteLoop() { while (true) }

function infiniteLoop() { while (true); }
```

Things to Remember

- Semicolons are only ever inferred before a }, at the end of a line, or at the end of a program.
- Semicolons are only ever inferred when the next token cannot be parsed.
- Never omit a semicolon before a statement beginning with (, [, +, -, or /.
- When concatenating scripts, insert semicolons explicitly between scripts.
- Never put a newline before the argument to return, throw, break, continue, ++, or --.
- Semicolons are never inferred as separators in the head of a for loop or as empty statements.

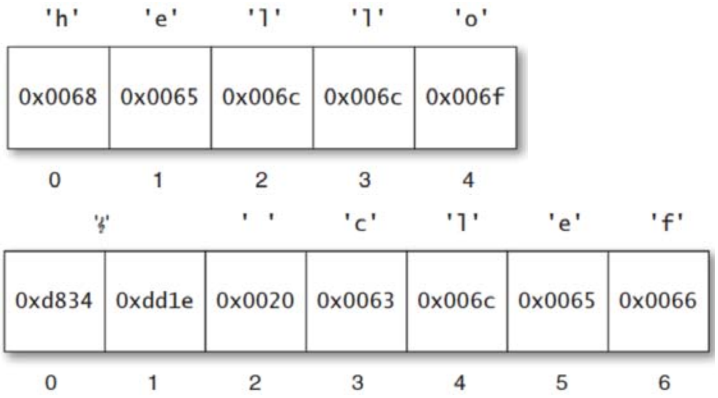
THINK OF STRINGS AS SEQUENCES OF 16-BIT CODE UNITS

Think of Strings As Sequences of 16-Bit Code Units

- Unicode has a reputation for being complicated
- Despite the ubiquity of strings, most programmers avoid learning about Unicode and hope for the best.
- The basics of Unicode are perfectly simple
  - Every unit of text of all the world’s writing systems is assigned a unique integer between 0 and 1,114,111
  - Known as a *code point* in Unicode terminology
- Hardly any different from any other text encoding, e.g. ASCII
  - While ASCII maps each index to a unique binary representation, Unicode allows multiple different binary encodings of code points.

Units

- Different encodings make trade-offs between the amount of storage required for a string and the speed of operations such as indexing into a string.
- There are multiple standard encodings of Unicode, the most popular are UTF-8, UTF-16, and UTF-32.



'h'	'e'	'l'	'l'	'o'
0x0068	0x0065	0x006c	0x006c	0x006f
0	1	2	3	4

'𝄞'		' '	'c'	'l'	'e'	'f'
0xd834	0xdd1e	0x0020	0x0063	0x006c	0x0065	0x0066
0	1	2	3	4	5	6

Think of Strings As Sequences of 16-Bit Code Units

- Internally, JavaScript engines may optimize the storage of string contents.
- As far as their properties and methods are concerned, strings behave like sequences of UTF-16 code units.
- Despite the fact that the string contains six code points, JavaScript reports its length as 7:

```
"𝄞 clef".length; // 7
"G clef".length; // 6
```

Units

> Extracting individual elements of the string produces code units rather than code points:

```
"𝄞 clef".charCodeAt(0);     // 55348 (0xd834)
"𝄞 clef".charCodeAt(1);     // 56606 (0xdd1e)
"𝄞 clef".charAt(1) === " "; // false
"𝄞 clef".charAt(2) === " "; // true
```

> Similarly, regular expressions operate at the level of code units.

> The single-character pattern (“.”) matches a single code unit:

```
/^.$/.test("𝄞");  // false
/^..$/.test("𝄞"); // true
```

Things to Remember

> JavaScript strings consist of 16-bit code units, not Unicode code points.

> Unicode code points 216 and above are represented in JavaScript by two code units, known as a surrogate pair.

> Surrogate pairs throw off string element counts, affecting length, charAt, charCodeAt, and regular expression patterns such as “.”.

> Use third-party libraries for writing code point-aware string manipulation.

> Whenever you are using a library that works with strings, consult the documentation to see how it handles the full range of code points.

MINIMIZE USE OF THE GLOBAL OBJECT

Minimize Use of the Global Object

```
var i, n, sum; // globals
function averageScore(players) {
    sum = 0;
    for (i = 0, n = players.length; i < n; i++) {
        sum += score(players[i]);
    }
    return sum / n;
}
var i, n, sum; // same globals as averageScore!
function score(player) {
    sum = 0;
    for (i = 0, n = player.levels.length; i < n; i++) {
        sum += player.levels[i].score;
    }
    return sum;
}
```

Minimize Use of the Global Object

```
function averageScore(players) {
    var i, n, sum;
    sum = 0;
    for (i = 0, n = players.length; i < n; i++) {
        sum += score(players[i]);
    }
    return sum / n;
}

function score(player) {
    var i, n, sum;
    sum = 0;
    for (i = 0, n = player.levels.length; i < n; i++) {
        sum += player.levels[i].score;
    }
    return sum;
}
```

Minimize Use of the Global Object

- JavaScript’s global namespace is also exposed as a *global object*
- In web browsers, the global object is also bound to the global **`window`** variable.
- Adding or modifying global variables automatically updates the global object:

```
this.foo; // undefined
foo = "global foo";
this.foo; // "global foo"
```

Minimize Use of the Global Object

- Updating the global object automatically updates the global namespace:

```
var foo = "global foo";
this.foo = "changed";
foo; // "changed"
```

Global Objects

- ES5 introduced a new global JSON object for reading and writing the JSON data format
- You can test the global object for its presence and provide an alternate implementation:

```
if (!this.JSON) {
    this.JSON = {
        parse: ...,
        stringify: ...
    };
}
```

Things to Remember

- Avoid declaring global variables.
- Declare variables as locally as possible.
- Avoid adding properties to the global object.
- Use the global object for platform feature detection.

ALWAYS DECLARE LOCAL VARIABLES

Always Declare Local Variables

> Forgetting to declare a local variable silently turns it into a global variable:

```
function swap(a, i, j) {
    temp = a[i]; // global
    a[i] = a[j];
    a[j] = temp;
}
```

Always Declare Local Variables

> A proper implementation declares temp with var:

```
function swap(a, i, j) {
    var temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

Things to Remember

- Always declare new local variables with var.
- Consider using lint tools to help check for unbound variables.

AVOID WITH

Avoid with

- There is probably no single more maligned feature in JavaScript

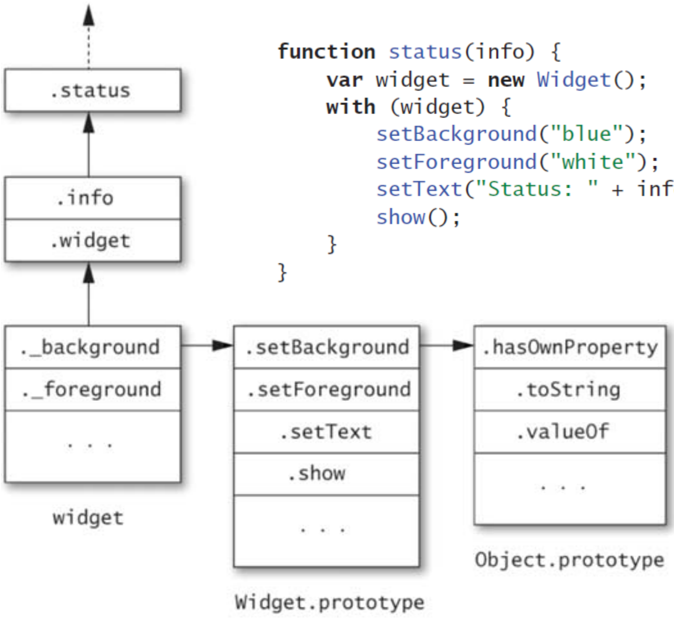
```
function status(info) {
    var widget = new Widget();
    with (widget) {
        setBackground("blue");
        setForeground("white");
        setText("Status: " + info); // ambiguous reference
        show();
    }
}
```

Avoid with

- It’s also tempting to use with to “import” variables from objects serving as modules:

```
function f(x, y) {
    with (Math) {
        return min(round(x), sqrt(y)); // ambiguous references
    }
}
```

Lexical environment



```
function status(info) {
    var widget = new Widget();
    with (widget) {
        setBackground("blue");
        setForeground("white");
        setText("Status: " + info);
        show();
    }
}
```

Avoid with

```
function f(x, y) {
    with (Math) {
        return min(round(x), sqrt(y));
    }
}
```

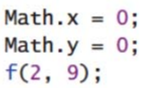
```
Math.x = 0;
Math.y = 0;
f(2, 9);
```

A better alternative

```
function status(info) {
    var w = new Widget();
    w.setBackground("blue");
    w.setForeground("white");
    w.addText("Status: " + info);
    w.show();
}

status("connecting"); // Status: connecting
Widget.prototype.info = "[[widget info]]";
status("connected");  // Status: connected
```

A better alternative

```
function f(x, y) {
    var min = Math.min, round = Math.round, sqrt = Math.sqrt;
    return min(round(x), sqrt(y));
}
```



```
Math.x = 0;
Math.y = 0;
f(2, 9);
```

Things to Remember

- Avoid using with statements.
- Use short variable names for repeated access to an object.
- Explicitly bind local variables to object properties instead of implicitly binding them with a with statement.

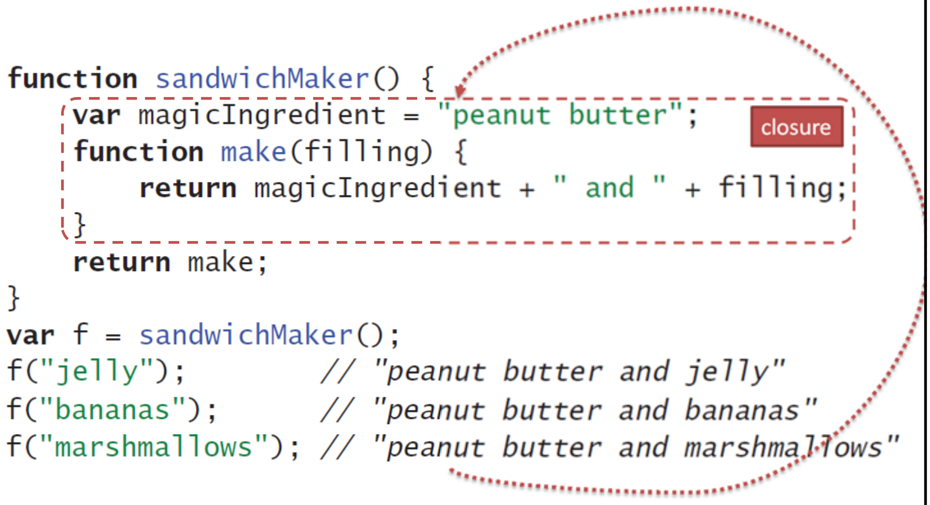
ITEM 11

GET COMFORTABLE WITH CLOSURES

Get Comfortable with Closures

```
function makeSandwich() {
    var magicIngredient = "peanut butter";
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make("jelly");
}
makeSandwich(); // "peanut butter and jelly"
```

Get Comfortable with Closures



```
function sandwichMaker() {
    var magicIngredient = "peanut butter";
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var f = sandwichMaker();
f("jelly");         // "peanut butter and jelly"
f("bananas");       // "peanut butter and bananas"
f("marshmallows"); // "peanut butter and marshmallows"
```

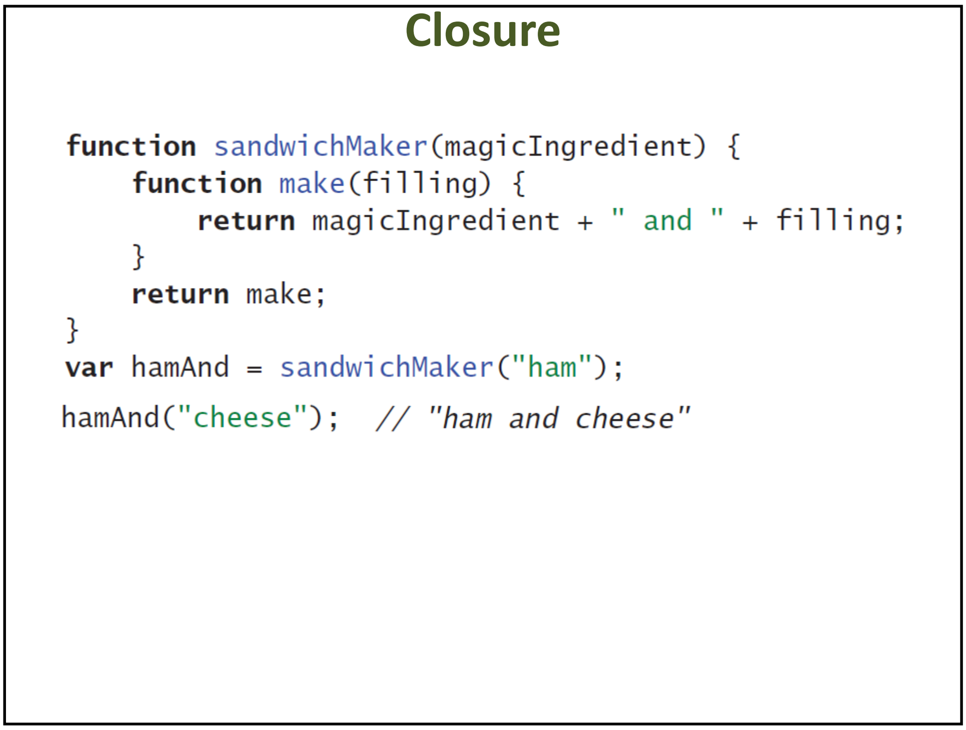
closure

Closure

```
function sandwichMaker(magicIngredient) {
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var hamAnd = sandwichMaker("ham");
```

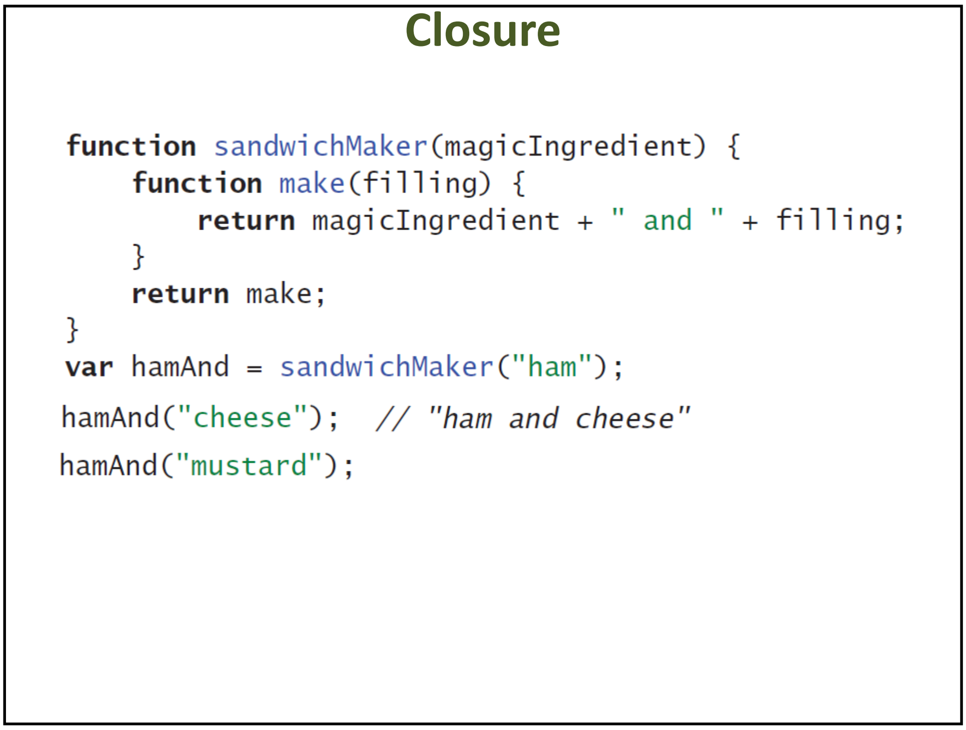
Closure

```
function sandwichMaker(magicIngredient) {
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var hamAnd = sandwichMaker("ham");
hamAnd("cheese");
```



Closure

```
function sandwichMaker(magicIngredient) {
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var hamAnd = sandwichMaker("ham");
hamAnd("cheese");   // "ham and cheese"
```



Closure

```
function sandwichMaker(magicIngredient) {
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var hamAnd = sandwichMaker("ham");
hamAnd("cheese");   // "ham and cheese"
hamAnd("mustard");
```

Closure

```
function sandwichMaker(magicIngredient) {
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var hamAnd = sandwichMaker("ham");
hamAnd("cheese");   // "ham and cheese"
hamAnd("mustard"); // "ham and mustard"
```

Closure

```
function sandwichMaker(magicIngredient) {
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var hamAnd = sandwichMaker("ham");
hamAnd("cheese");   // "ham and cheese"
hamAnd("mustard"); // "ham and mustard"
var turkeyAnd = sandwichMaker("turkey");
turkeyAnd("Swiss");
turkeyAnd("Provolone");
```

Closure

```
function sandwichMaker(magicIngredient) {
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var hamAnd = sandwichMaker("ham");
hamAnd("cheese");  // "ham and cheese"
hamAnd("mustard"); // "ham and mustard"
var turkeyAnd = sandwichMaker("turkey");
turkeyAnd("Swiss");      // "turkey and Swiss"
turkeyAnd("Provolone");  // "turkey and Provolone"
```

Function Expression

```
function sandwichMaker(magicIngredient) {
    return function(filling) {
        return magicIngredient + " and " + filling;
    };
}
```

Closure

```
function box() {
    var val = undefined;
    return {
        set: function(newVal) { val = newVal; },
        get: function() { return val; },
        type: function() { return typeof val; }
    };
}
var b = box();
```

Closure

```
function box() {
    var val = undefined;
    return {
        set: function(newVal) { val = newVal; },
        get: function() { return val; },
        type: function() { return typeof val; }
    };
}
var b = box();
b.type();
```

Closure

```
function box() {
    var val = undefined;
    return {
        set: function(newVal) { val = newVal; },
        get: function() { return val; },
        type: function() { return typeof val; }
    };
}
var b = box();
b.type(); // "undefined"
```

Closure

```
function box() {
    var val = undefined;
    return {
        set: function(newVal) { val = newVal; },
        get: function() { return val; },
        type: function() { return typeof val; }
    };
}
var b = box();
b.type(); // "undefined"
b.set(98.6);
b.get();
```

Closure

```
function box() {
    var val = undefined;
    return {
        set: function(newVal) { val = newVal; },
        get: function() { return val; },
        type: function() { return typeof val; }
    };
}
var b = box();
b.type(); // "undefined"
b.set(98.6);
b.get();  // 98.6
```

Closure

```
function box() {
    var val = undefined;
    return {
        set: function(newVal) { val = newVal; },
        get: function() { return val; },
        type: function() { return typeof val; }
    };
}
var b = box();
b.type(); // "undefined"
b.set(98.6);
b.get();
b.type();
```

Closure

```
function box() {
    var val = undefined;
    return {
        set: function(newVal) { val = newVal; },
        get: function() { return val; },
        type: function() { return typeof val; }
    };
}
var b = box();
b.type(); // "undefined"
b.set(98.6);
b.get();  // 98.6
b.type(); // "number"
```

Things to Remember

> Functions can refer to variables defined in outer scopes.

> Closures can outlive the function that creates them.

> Closures internally store references to their outer variables, and can both read and update their stored variables.

UNDERSTAND VARIABLE HOISTING

Understand Variable Hoisting

> JavaScript supports *lexical scoping*

> JavaScript does not support *block scoping*

```
function isWinner(player, others) {
    var highest = 0;
    for (var i = 0, n = others.length; i < n; i++) {
        var player = others[i];
        if (player.score > highest) {
            highest = player.score;
        }
    }
    return player.score > highest;
}
```

Understand Variable Hoisting

- JavaScript variable declarations consists of two parts
  - a declaration
  - an assignment
- JavaScript implicitly “hoists” the declaration part to the top of the enclosing function and leaves the assignment in place.
- The variable is in scope for the entire function, but it is only assigned at the point where the **`var`** statement appears.

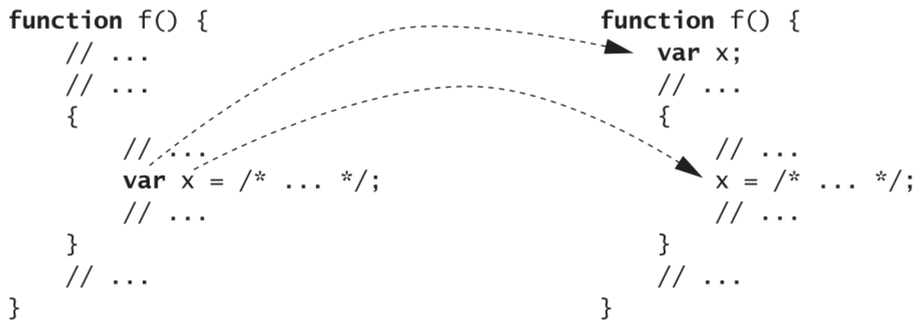
Understand Variable Hoisting

```
function trimSections(header, body, footer) {
  for (var i = 0, n = header.length; i < n; i++) {
    header[i] = header[i].trim();
  }
  for (var i = 0, n = body.length; i < n; i++) {
    body[i] = body[i].trim();
  }
  for (var i = 0, n = footer.length; i < n; i++) {
    footer[i] = footer[i].trim();
  }
}
```

Understand Variable Hoisting

```
function trimSections(header, body, footer) {
  var i, n;
  for (i = 0, n = header.length; i < n; i++) {
    header[i] = header[i].trim();
  }
  for (i = 0, n = body.length; i < n; i++) {
    body[i] = body[i].trim();
  }
  for (i = 0, n = footer.length; i < n; i++) {
    footer[i] = footer[i].trim();
  }
}
```

Understand Variable Hoisting



```
function f() {
    // ...
    // ...
    {
        // ...
        var x = /* ... */;
        // ...
    }
    // ...
}
```

```
function f() {
    var x;
    // ...
    {
        // ...
        x = /* ... */;
        // ...
    }
    // ...
}
```

Understand Variable Hoisting

```
function test() {
    var x = "var", result = [];
    result.push(x);
    try {
        throw "exception";
    } catch (x) {
        x = "catch";
    }
    result.push(x);
    return result;
}
test(); // ["var", "var"]
```

Things to Remember

> Variable declarations within a block are implicitly hoisted to the top of their enclosing function.

> Redeclarations of a variable are treated as a single variable.

> Consider manually hoisting local variable declarations to avoid confusion.

USE **IMMEDIATELY INVOKED FUNCTION EXPRESSIONS** TO CREATE LOCAL SCOPES

Use Immediately Invoked Function Expressions to Create Local Scopes

> What does this (buggy!) program compute?

```
function wrapElements(a) {
    var result = [], i, n;
    for (i = 0, n = a.length; i < n; i++) {
        result[i] = function() { return a[i]; };
    }
    return result;
}

var wrapped = wrapElements([10, 20, 30, 40, 50]);
var f = wrapped[0];
f(); // ?
```

Use Immediately Invoked Function Expressions to Create Local Scopes

> *Closures store their outer variables by reference, not by value.*

```
function wrapElements(a) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i++) {
        result[i] = function() { return a[i]; };
    }
    return result;
}

var wrapped = wrapElements([10, 20, 30, 40, 50]);
var f = wrapped[0];
f(); // undefined
```

Use Immediately Invoked Function Expressions to Create Local Scopes

> *Solution: immediately invoked function expression* (IIFE)

```
function wrapElements(a) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i++) {
       (function(j) {
           result[i] = function() { return a[j]; };
       })(i);
    }
    return result;
}
```

Things to Remember

- Understand the difference between binding and assignment.
- Closures capture their outer variables by reference, not by value.
- Use immediately invoked function expressions (IIFEs) to create local scopes.
- Be aware of the cases where wrapping a block in an IIFE can change its behavior.

BEWARE OF UNPORTABLE SCOPING OF NAMED FUNCTION EXPRESSIONS

Beware of Unportable Scoping of Named Function Expressions

> JavaScript functions may look the same wherever they go, but their meaning changes depending on the context.

```
function double(x) { return x * 2; }
```

> Depending on where it appears, this could be

  - a *function declaration*
  - a *named function expression*

Beware of Unportable Scoping of Named Function Expressions

```
var f = function double(x) { return x * 2; };

var f = function(x) { return x * 2; };
```

Beware of Unportable Scoping of Named Function Expressions

```
var f = function find(tree, key) {
    if (!tree) {
        return null;
    }
    if (tree.key === key) {
        return tree.value;
    }
    return find(tree.left, key) ||
           find(tree.right, key);
};

find(myTree, "foo");
```

Beware of Unportable Scoping of Named Function Expressions

```
var f = function find(tree, key) {
    if (!tree) {
        return null;
    }
    if (tree.key === key) {
        return tree.value;
    }
    return find(tree.left, key) ||
           find(tree.right, key);
};

find(myTree, "foo"); // error: find is not defined
```

Beware of Unportable Scoping of Named Function Expressions

```
var f = function(tree, key) {
    if (!tree) {
        return null;
    }
    if (tree.key === key) {
        return tree.value;
    }
    return f(tree.left, key) ||
           f(tree.right, key);
};
```

Beware of Unportable Scoping of Named Function Expressions

```
function find(tree, key) {
    if (!tree) {
        return null;
    }
    if (tree.key === key) {
        return tree.value;
    }
    return find(tree.left, key) ||
           find(tree.right, key);
}
var f = find;
```

Things to Remember

> Use named function expressions to improve stack traces in Error objects and debuggers.

> Beware of pollution of function expression scope with Object.prototype in ES3 and buggy JavaScript environments.

> Beware of hoisting and duplicate allocation of named function expressions in buggy JavaScript environments.

> Consider avoiding named function expressions or removing them before shipping.

> If you are shipping in properly implemented ES5 environments, you've got nothing to worry about.

BEWARE OF UNPORTABLE SCOPING OF BLOCK-LOCAL FUNCTION DECLARATIONS

Beware of Unportable Scoping of Block-Local Function Declarations

```
function f() { return "global"; }

function test(x) {
    function f() { return "local"; }

    var result = [];
    if (x) {
        result.push(f());
    }
    result.push(f());
    return result;
}
```

Beware of Unportable Scoping of Block-Local Function Declarations

```
function f() { return "global"; }

function test(x) {
    function f() { return "local"; }

    var result = [];
    if (x) {
        result.push(f());
    }
    result.push(f());
    return result;
}

test(true);
test(false);
```

Beware of Unportable Scoping of Block-Local Function Declarations

```
function f() { return "global"; }

function test(x) {
    function f() { return "local"; }

    var result = [];
    if (x) {
        result.push(f());
    }
    result.push(f());
    return result;
}
test(true);  // ["local", "local"]
test(false); // ["local"]
```

Beware of Unportable Scoping of Block-Local Function Declarations

```
function f() { return "global"; }

function test(x) {
    var result = [];
    if (x) {
        function f() { return "local"; } // block-local

        result.push(f());
    }
    result.push(f());
    return result;
}

test(true);  // ?
test(false); // ?
```

Beware of Unportable Scoping of Block-Local Function Declarations

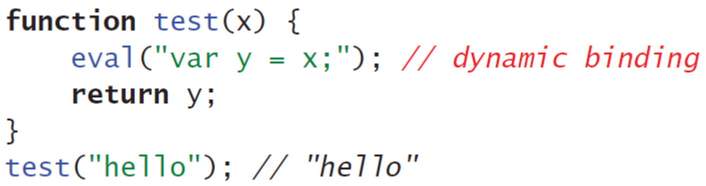
```
function f() { return "global"; }

function test(x) {
    var g = f, result = [];
    if (x) {
        g = function() { return "local"; }

        result.push(g());
    }
    result.push(g());
    return result;
}
```

Things to Remember

- Always keep function declarations at the outermost level of a program or a containing function to avoid unportable behavior.
- Use **`var`** declarations with conditional assignment instead of conditional function declarations.

AVOID
CREATING LOCAL VARIABLES
WITH EVAL

Avoid Creating Local Variables with eval



```
function test(x) {
    eval("var y = x;"); // dynamic binding
    return y;
}
test("hello"); // "hello"
```

Avoid Creating Local Variables with eval

```
var y = "global";
function test(x) {
    if (x) {
        eval("var y = 'local';"); // dynamic binding
    }
    return y;
}
test(true);  // "local"
test(false); // "global"
```

Basing scoping decisions on the dynamic behavior of a program is almost always a bad idea!

Avoid Creating Local Variables with eval

```
var y = "global";
function test(src) {
    (function() { eval(src); })();
    return y;
}
test("var y = 'local';");
```

Avoid Creating Local Variables with eval

```
var y = "global";
function test(src) {
    (function() { eval(src); })();
    return y;
}
test("var y = 'local';"); // "global"
```

Avoid Creating Local Variables with eval

```
var y = "global";
function test(src) {
    (function() { eval(src); })();
    return y;
}
test("var y = 'local';"); // "global"
test("var z = 'local';");
```

Avoid Creating Local Variables with eval

```
var y = "global";
function test(src) {
    (function() { eval(src); })();
    return y;
}

test("var y = 'local';"); // "global"
test("var z = 'local';"); // "global"
```

Things to Remember

> Avoid creating variables with eval that pollute the caller's scope.

> If eval code might create global variables, wrap the call in a nested function to prevent scope pollution.

PREFER INDIRECT EVAL
TO DIRECT EVAL

Prefer Indirect eval to Direct eval

> The **`eval`** function has a secret weapon: It’s more than just a function

> Most functions have access to the scope where they are defined

> **`eval`** has access to the full scope *at the point where it’s called.*

Prefer Indirect eval to Direct eval

- A function call involving the identifier **`eval`** is considered a “direct” call to **`eval`**:

```
var x = "global";
function test() {
    var x = "local";
    return eval("x"); // direct eval
}
test(); // "local"
```

Prefer Indirect eval to Direct eval

- The other kind of call to **`eval`** is considered “indirect,” and evaluates its argument in global scope.
- Binding the **`eval`** function to a different variable name and calling it through the alternate name causes the code to lose access to any local scope:

```
var x = "global";
function test() {
    var x = "local";
    var f = eval;
    return f("x"); // indirect eval
}
test(); // "global"
```

Things to Remember

> Wrap eval in a sequence expression with a useless literal to force the use of indirect eval.

> Prefer indirect eval to direct eval whenever possible.