



Pontificia Universidad Católica de Chile  
Escuela de Ingeniería  
Departamento de Ciencia de la Computación

# JAVASCRIPT

## MÁS CONCEPTOS

Raúl Montes T.

# Hoisting

---

Declaraciones con `var` y `function` se interpretan primero

Equivale a siempre declarar las variables al principio del scope

```
console.log(foo,
f());
var foo;
function f() { }

// "equivalente" a
var foo;
function f() { }
console.log(foo);
```

# Functions

---

```
// ambas son equivalentes casi siempre
function ejemplo() {
    alert("Hola!");
}
var ejemplo = function() {
    alert("Hola!");
}
```

La diferencia es el efecto del **hoisting**

En el segundo caso sólo la declaración se interpreta primero. La creación y asignación de la función al ejecutar el programa

# Métodos

---

```
// los métodos son sólo funciones asignadas
// como una de las propiedades de un objeto
foo.metodo = function() {
    alert("Método de instancia: " + this.uno);
}
// lo ejecutamos
// this será una referencia al objeto en el cual se llama
foo.metodo();
```

# Otra forma de llamar funciones...

---

```
const f = function() {  
  // cuerpo de la función  
};  
  
// "call" llama a la función y permite elegir el "this" que usará  
// y además entregar los params como en una llamada normal  
f.call(referenciaAThis, param1, param2, ...);  
// "apply" es lo mismo, pero los params se entregan en un Array  
f.apply(referenciaAThis, paramsArray);
```

# Scope

---

`var` y `function` tienen scope de función

`let` y `const` tienen scope de bloque (el “normal”)

```
// scope de función, no de bloque
function f() {
  var v1 = 1;
  if (v1 > 0) {
    var v2 = 2;
  }
  console.log(v2);
}
f(); // muestra 2, funciona
```

```
// el scope incluye el scope externo
function f1() {
  var v1 = 1;
  function f2() {
    console.log(v1);
  }
  f2();
}
f1(); // muestra 1
```

# Closure

---

El **closure** de una función contiene todas las **variables existentes en el scope** en el cual fue declarada la función

```
function f1() {  
  const v1 = 1;  
  const f2 = function() {  
    console.log(v1);  
  };  
  return f2;  
}  
var unaFuncion = f1();  
unaFuncion(); -> muestra 1
```

La variable local **v1** forma parte del closure de **f2**

# Constructores

---

Cualquier función puede ser usada como **Constructor** de objetos. Tendrá a **this** como referencia al objeto que se está construyendo. Se crea el objeto con **new**

```
function Perro(nombre, raza) {  
    this.nombre = nombre;  
    this.raza = raza;  
}  
var perro = new Perro("Snoopy", "Beagle");
```



# Construir objetos con métodos

---

```
function Perro(nombre, raza) {  
  this.nombre = nombre;  
  this.raza = raza;  
  this.habla = function(veces) {  
    while (veces-- > 0) {  
      console.log("Guau!");  
    }  
  }  
}  
  
const perro = new Perro("Snoopy", "Beagle");  
perro.habla(5);
```

# Y esto?

---

```
const perro1 = new Perro("Pluto", "QuienSabe");  
const perro2 = new Perro("Snoopy", "Beagle");
```

```
perro1.habla === perro2.habla -> false
```

```
// no son el mismo método... estamos creando la función cada vez  
// un poco tonto o no?
```

# Prototype

---

Todos los objetos (salvo el objeto base) tienen un **prototipo**.

El **prototipo** es... un objeto... que tiene prototipo...

Las **propiedades** de un objeto se buscarán en el objeto mismo y, si no están ahí, en su **prototipo**, y **prototipo** del **prototipo** y... => **herencia!**

# Prototype

---

Todas las funciones, además de su prototipo como objeto, tienen un prototype para asignárselo a los objetos que construyen

```
const perro1 = new Perro("Pluto", "QuienSabe");  
// para obtener el prototipo de un objeto  
// sólo la segunda es 100% segura en browsers "no modernos" (sí, IE... :P)  
perro.__proto__, perro.constructor.prototype, Object.getPrototypeOf(perro)  
// para obtener el prototipo que una función asigna a sus objetos:  
Perro.prototype  
// este mismo prototipo es el que se asigna a los objetos  
Perro.prototype === perro1.__proto__ -> true
```

# Prototype

---

Entonces, si **modificamos** el **prototype** que se le asigna a los objetos...

```
Perro.prototype.habla = function(veces) {  
  while(veces-- > 0) {  
    console.log(this.nombre);  
  }  
}  
perro1.habla === perro2.habla -> true
```

El **objeto función es el mismo**, pero dependiendo desde donde se llame, **this** será un diferente objeto :-)

# Herencia con prototipos

```
function Mamifero(nombre) {  
  this.nombre = nombre;  
}  
Mamifero.prototype.habla = function(veces) { console.log("..."); }  
// subclase de Mamifero  
function Perro(nombre, raza) {  
  Mamifero.call(this, nombre); // llamamos al constructor de la super "clase"  
  // (como llamar a un "super()")  
  this.raza = raza;  
}  
// hacemos que el prototipo que asigne este constructor sea un objeto de la superclase  
// así los Perros tendrán todos los métodos de los Mamíferos y además podremos  
// agregar/sobreescribir otras propiedades  
// Además asignamos Perro como el constructor del prototipo  
Perro.prototype = new Mamifero();  
Perro.prototype.constructor = Perro;  
const perro = new Perro("Odie", "Teckel");  
perro.habla === perro1.habla -> true  
PerroEducado.prototype.habla = function(veces) {  
  while(veces-- > 0) {  
    console.log(this.nombre);  
  }  
};  
perro.habla === mamifero.habla -> false
```

# Atributos/Métodos privados, privilegiados y públicos

```
const Perro = function(nombre, raza) {  
  // variables locales como nombre o raza, independiente  
  // de si almacenan una referencia a un objeto cualquiera  
  // o una función, serán propiedades de visibilidad privada, sólo  
  // accesibles por quien tenga estas variables en su scope/closure  
  
  // funciones definidas dentro del constructor pero  
  // asignadas a una propiedad del objeto resultante (this),  
  // serán de visibilidad "privilegiada" (de acceso público,  
  // pero con acceso a propiedades privadas)  
  this.getNombre = function() {return nombre;}  
  
}  
  
// propiedades definidas al objeto o a su prototipo pero fuera de la  
// función constructora, serán de visibilidad pública, pero no tendrán  
// acceso a propiedades privadas (pero sí a privilegiadas)  
Perro.prototype.ladra = function(veces) {  
  while(veces-- > 0) {  
    console.log(this.getNombre() + ": Guau!");  
  }  
}
```

# class

---

- EcmaScript 2015 agregó “clases”
- Pero son sólo syntax sugar para funciones constructoras
- La implementación es igualmente a través de prototipos
- Algo declarado como “clase” será, de hecho, una función



# class

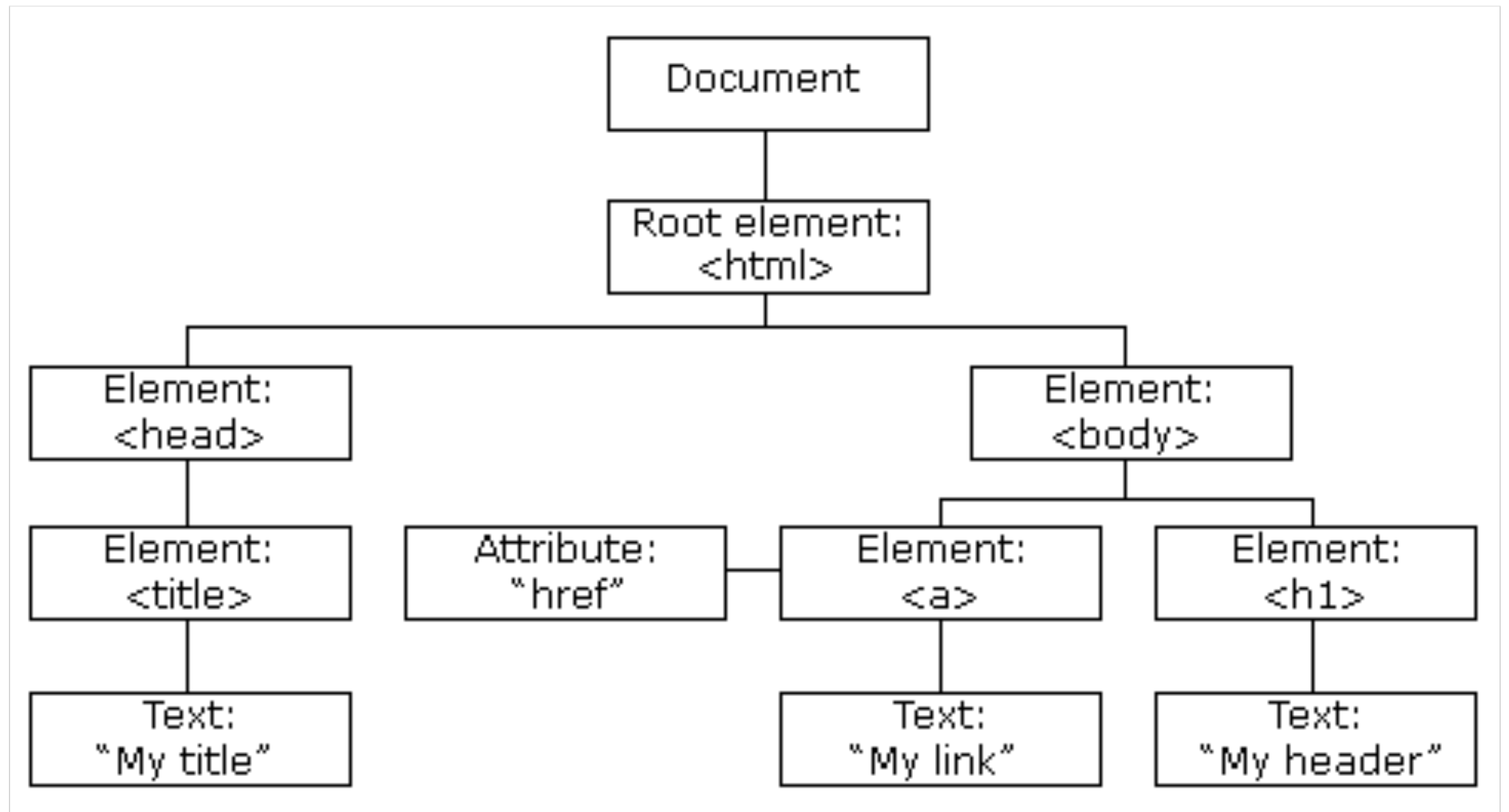
---

```
class Perro extends Mamifero {  
  constructor(name) {  
    super(name);  
  }  
  habla() {  
    console.log(` ${this.name}: woof!` );  
  }  
}
```

¿Y qué hacemos con JavaScript en el cliente?

Su **interacción** con el **browser** la realiza principalmente gracias al...

# Document Object Model



... y sus **eventos**...

¿Y cómo hacemos interactuar JavaScript con  
un documento HTML?

Esta **NO** es la forma de usar JavaScript con HTML

```
<a href="ejemplo2.html" id="button"  
onclick="alert('Ehh! me hiciste click!!!')">Click me!!!!</a>
```

Así como evitamos mezclar estilos con HTML,  
también debemos evitar mezclar comportamiento  
con la estructura/contenido

Esto se conoce como **unobtrusive** JavaScript

Incluimos el **script** en el documento **HTML**, dentro de **<head>** o al final de **<body>**

```
<script type="text/javascript" src="ejemplo.js"></script>
```

Y en ejemplo.js

```
// escuchamos el evento de carga del DOM
// sino, podría no existir el link en el DOM aún
document.addEventListener('DOMContentLoaded', function() {
  var link = document.getElementById('button');
  // al handler del evento se le entrega el evento gatillado
  link.addEventListener('click', function(e) {
    alert("Ehh! me hiciste click!");
    // impedimos que se siga con la acción normal del evento
    // que en este caso es ir al href del anchor
    e.preventDefault();
  });
});
```

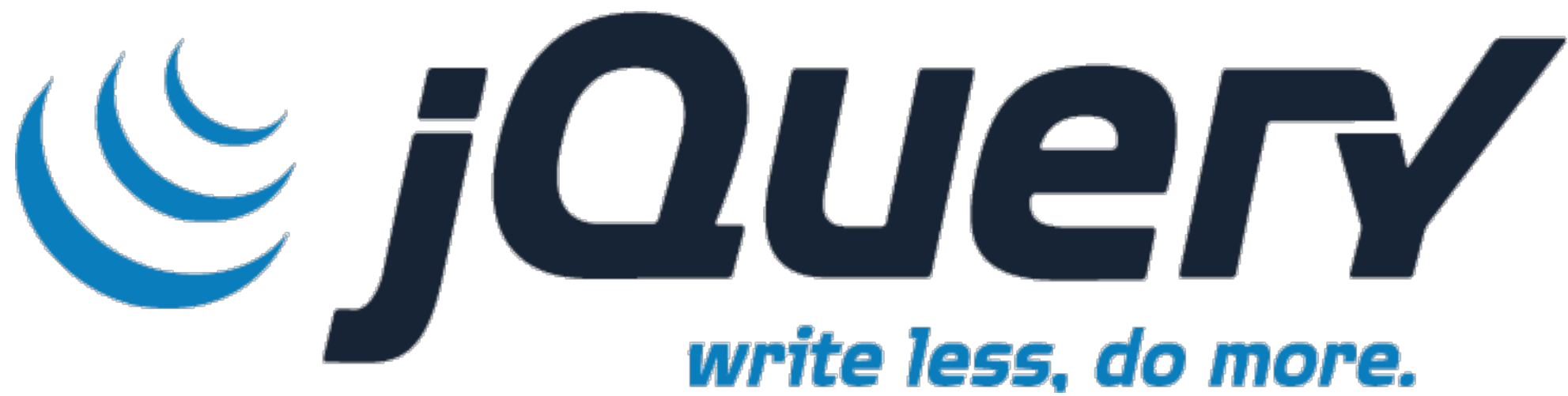
# Mediante DOM se puede realizar cualquier manipulación y en todo el documento

```
document // referencia al documento
document.childNodes // nodos hijos, de cualquier tipo
document.childNodes[0] // Doctype
document.childNodes[1] // nodo <html>
// nodos hijos de <html> pero sólo de tipo Element
document.childNodes[1].children
// para encontrar un elemento por su atributo id
var elem = document.getElementById('idDelElemento')
var elem = document.getElementsByClassName('claseDelElemento') // o clase
var elem = document.querySelectorAll('a.boton') // o selector CSS
// propiedades y métodos de un nodo
elem.nodeName // nombre del nodo. Si es un elemento -> DIV, A, etc.
elem.nodeType // número que representa el tipo (1 -> Elemento, 3 -> Texto...)
elem.parentNode // nodo padre
elem.nextSibling, elem.previousSibling // nodos hermano
document.createElement() // para crear nuevos nodos de tipo elemento
document.createTextNode() // crea nodos texto
elem.appendChild(child) // agrega un nodo al final de los hijos
elem.cloneNode() // crea una copia de un elemento
elem.removeChild(child) // elimina el hijo indicado
elem.set/get/removeAttribute() // cambia/crea/obtiene/elimina atributos
// se puede acceder a todas las propiedades de estilo
elem.style.etc // guiones se cambian por camelCase
```



Pero hoy en día se suelen usar **librerías** para, además de extender el lenguaje, lidiar con las diferencias entre diferentes browsers.

la más usada es...



## Sólo agrega una función...

# jQuery (o \$, para los amigos)

```
$(funcion); // ejecuta la función cuando se carga el DOM (DOMContentLoaded)
$(selectorCSSConEsteroides); // entrega un objeto jQuery
var h1s = $('h1'); -> [primerH1, segundoH1, ...]
// el objeto jQuery tiene muuuuchos métodos útiles
h1s.hide();
h1s.append("contenido a agregar");
// ... y muuuuuuuuchos más... ver documentación
// además, casi siempre devuelven el mismo objeto jQuery, para chaining pattern
h1s.show().addClass('nuevaClase');
```

Así, lo anterior nos queda...

```
$(function() {  
    $('#button').on('click', function(event) {  
        alert('Ehhh!! me hiciste click!!!');  
        event.preventDefault();  
    });  
});
```

# Un uso común de JavaScript es client side validation

Por ejemplo...

```
var $form = $('#form');
$form.on('submit', function(e) {
    // revisar los valores de cada campo
    var $field = $form.find('#first_name');
    var value = $field.val();
    // realizar las validaciones
    // si hay un error, mostrarlo mediante manipulación de DOM
    // y además impedir el envío del formulario
    if (error) {
        e.preventDefault();
    }
});
```

¿Otro uso común? R: lo que, de hecho, popularizó enormemente este lenguaje... **AJAX**

## **Asynchronous JavaScript and XML**

... aunque hoy en día XML está presente, generalmente, sólo en el nombre...

# AJAX

---

- `HTML` y `CSS` para la vista
- `DOM` para interactuar con la vista
- `XML`, `JSON`, `HTML` o `JS` para intercambio de info
- El `XMLHttpRequest` para requests asíncronos
  - y además `fetch` en browsers modernos
- `JavaScript` para unirlos a todos... ~~y atarlos en las tinieblas~~

# AJAX

---

En lugar de cambiar el estado completo de la app con un request normal, hacemos un **request especial** de manera **asíncrona**, y cuando la respuesta llega, actualizamos sólo lo que debiera cambiar en la vista.

La actualización de la vista es **más rápida** y no se necesita esperar por el request (**la app no se bloquea**). Las apps se sienten más “**responsivas**”