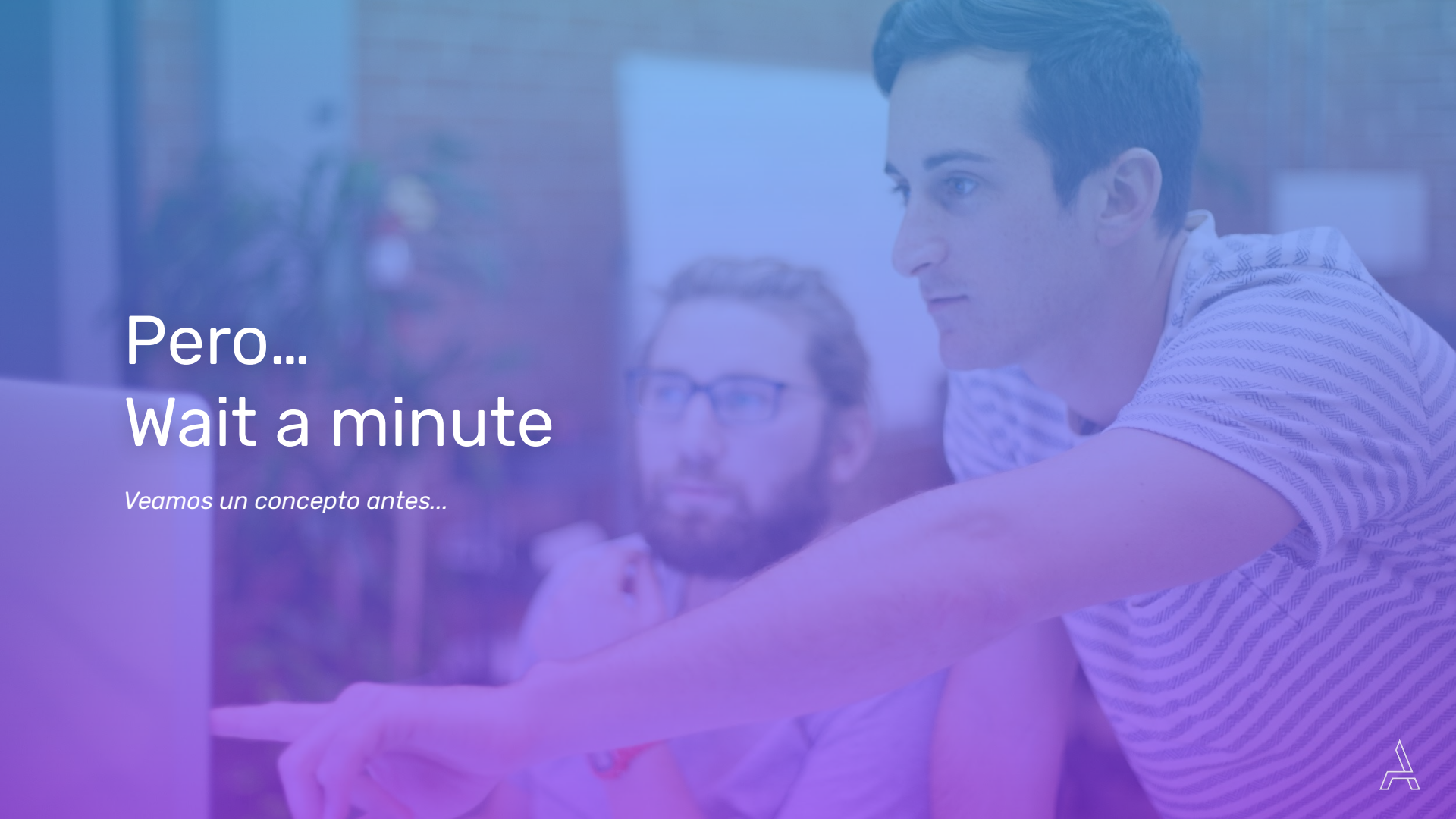
DESARROLLO WEB FULL STACK

Programación orientada a objetos

Empezando a modelar el software como objetos de la vida real

DWFS COR



Pero…
Wait a minute

Veamos un concepto antes…

Objeto this

En general, el valor de **this** está determinado por cómo se invoca a la función. No puede ser establecida mediante una asignación en tiempo de ejecución, y puede ser diferente cada vez que la función es invocada. ES5 introdujo el método **bind()** para establecer el valor de la función this independientemente de como es llamada, y **ES2015** introdujo las **funciones flecha** que nos proporcionan su propio "binding" de **this** (se mantiene el valor de this del contexto léxico que envuelve a la función)

This & funciones comunes

Usando funciones comunes, **this** es el objeto desde el cual la función es llamada (scope dinámico).

Como una regla nemotécnica se puede decir que **this** es el primer objeto que se encuentra a la izquierda del "." que finalmente invoca el método donde se usa el **this**.

Ej:
objeto1.objeto2.verThis();

this === objeto2

```
var persona = {
    nombre: "Pepito",
    pasos: 0,
    caminar: function () {
        this.pasos++;
    },
    pasosCaminados: function () {
        return this.pasos;
    } ,
};
```

This & funciones flecha

Las funciones flechas siempre refieren a su scope lexicográfico, por lo que en este caso:

This === window

```
var persona = {
    nombre: "Pepito",
    pasos: 0,
    caminar: () => {
        this.pasos++; // undefined
    },
    pasosCaminados: () => {
        return this.pasos; // undefined
    },
};
```

Scope léxico

Las funciones arrow toman el scope léxico más próximo.

Las funciones siempre crean scope léxico.

```
var persona = {
    nombre: "Pepito",
    pasos: 0,
    caminar: function () {
        window.setTimeout(() => {
            console.log(this); // persona
        }, 1000);
    },
    pasosCaminados: function () {
        window.setTimeout(() => {
            console.log(this); // persona
        }, 1000);
    },
};
```

Programación orientada a ¿Objetos?

Namespace

En JavaScript un **espacio de nombres** es un **objeto** que permite a métodos, propiedades y objetos asociarse.

La idea de crear espacios de nombres en JavaScript es simple: Crear un único objeto global para las variables, métodos, funciones convirtiendolos en propiedades de ese objeto.

El uso de los namespace permite minimizar el conflicto de nombres con otros objetos haciéndolos únicos dentro de nuestra aplicación.

Namespace

Se crea un namespace, se pueden agregar sub-namespaces y se le agregan métodos.

```
// namespace global
var MIAPLICACION = MIAPLICACION || {};

// sub-namespace
MIAPLICACION.evento = {} ;

MIAPLICACION.numero = {
  esValido: function(valor){
    // se fija si el caracter ingresado es
realmente un número
  },
  esMayorQue: function(valor, comparador){
    // se fija si el número ingresado es mayor
que un número dado
  },
}
```

Objetos básicos

JavaScript tiene varios objetos incluidos en su núcleo, como Math, Object, Array y String.

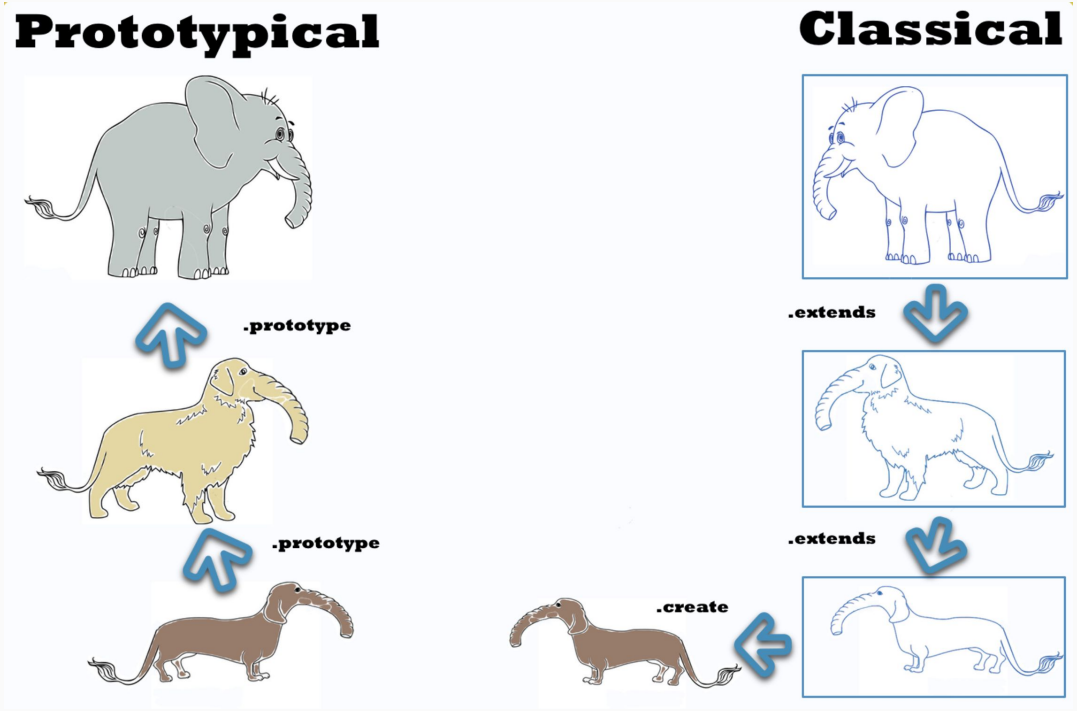
El siguiente ejemplo muestra cómo utilizar el objeto Math para obtener un número al azar mediante el uso de su método random()

```
alert (Math.random ());
```

Programación orientada a prototipos

Programación basada en **prototipos**, es un estilo de programación orientada a objetos en el cual los objetos no son creados mediante la instanciación de clases, sino, mediante la clonación de otros objetos.

De esta forma los objetos ya existentes pueden servir de prototipos para los que el programador necesite crear.



La Clase

JavaScript es un lenguaje **basado en prototipos** que no contiene ninguna declaración de **clase**, como se encuentra, por ejemplo, en C + + o Java.

Esto es a veces confuso para los programadores acostumbrados a los lenguajes con una declaración de clase.

En su lugar, JavaScript utiliza **funciones** como **clases**. Definir una clase es tan fácil como definir una función. En el ejemplo siguiente se define una nueva clase llamada Persona.

Definición de clase

Las funciones en JavaScript son clases.

```
function Persona() { }
```

Instanciación de clase (creación de un objeto)

Para crear una instancia de una clase y crear un objeto que deriva de esa clase, usamos la declaración **new**, asignando el resultado (que es de tipo objeto) a una variable.

```
function Persona() { }

var persona1 = new Persona();
var persona2 = new Persona();
```

El constructor

El **constructor** es llamado en el momento de la creación de la **instancia** (el momento en que se crea el objeto). El constructor es un método de la clase.

En JavaScript, **la función sirve como el constructor del objeto**, por lo tanto, no hay necesidad de definir explícitamente un método constructor. Cada acción declarada en la clase es ejecutada en el momento de la creación de la instancia.

El constructor se usa para establecer las propiedades del objeto o para llamar a los métodos para preparar el objeto para su uso.

Instanciación de clase (creación de un objeto)

Para crear una instancia de una clase y crear un objeto que deriva de esa clase, usamos la declaración **new**, asignando el resultado (que es de tipo objeto) a una variable.

```
function Persona() {
    console.log('Una instancia de Persona');
}

var persona1 = new Persona();
var persona2 = new Persona();
```

La Propiedad (atributo del objeto)

Las **propiedades** son **variables** contenidas en la clase, cada instancia del objeto tiene dichas propiedades.

Las propiedades deben establecerse a la propiedad prototipo de la clase (función), para que la herencia funcione correctamente.

Para trabajar con propiedades dentro de la clase se utiliza la palabra reservada **this**, que se refiere al objeto actual.

El acceso (lectura o escritura) a una propiedad desde fuera de la clase se hace con la sintaxis: **NombreDeLaInstancia.Propiedad**. (Desde dentro de la clase la sintaxis es **this.Propiedad** que se utiliza para obtener o establecer el valor de la propiedad).

Instanciación de clase (creación de un objeto)

En el siguiente ejemplo definimos la propiedad primerNombre de la clase Persona y la definimos en la creación de la instancia.

```
function Persona(primerNombre) {
     this.primerNombre = primerNombre;
     console.log('Una instancia de Persona');
}

var persona1 = new Persona('Alicia');
var persona2 = new Persona('Sebastián');

console.log('persona1 es ' + persona1.primerNombre);
console.log('persona2 es ' + persona2.primerNombre);
```

Los métodos

Los **métodos** siguen la misma lógica que las propiedades, la diferencia es que **son funciones** y se definen como funciones.

Llamar a un método es similar a acceder a una propiedad, pero se agrega () al final del nombre del método, posiblemente con argumentos.

Los métodos

En el siguiente ejemplo se define y utiliza el método diHola() para la clase Persona.

```
function Persona(primerNombre) {
     this.primerNombre = primerNombre;
     console.log('Una instancia de Persona');
}

Persona.prototype.diHola = function () {
     console.log('Hola, Soy ' + this.primerNombre);
};

var persona1 = new Persona('Alicia');
var persona2 = new Persona('Sebastián');

persona1.diHola();
persona2.diHola();
```

Los métodos

En el ejemplo se muestran todas las referencias que tenemos de la función diHola (una de ellas es persona1, otra en Persona.prototype, en la variable funcionSaludar, etc.) todas se refieren a la misma función. El valor durante una llamada a la función depende de como realizamos esa llamada.

```
function Persona(primerNombre) {
     this.primerNombre = primerNombre;
     console.log('Una instancia de Persona');
}

Persona.prototype.diHola = function () {
     console.log('Hola, Soy ' + this.primerNombre);
};

var persona1 = new Persona('Alicia');
var persona2 = new Persona('Sebastián');
var funcionSaludar = persona1.diHola;

persona1.diHola();
persona2.diHola();
funcionSaludar(); // 'Hola Soy undefined'
```

Los métodos

En el ejemplo se muestran todas las referencias que tenemos de la función diHola (una de ellas es persona1, otra en Persona.prototype, en la variable funcionSaludar, etc.) todas se refieren a la misma función. El valor durante una llamada a la función depende de como realizamos esa llamada.

```
var persona1 = new Persona('Alicia');
var persona2 = new Persona('Sebastián');
var funcionSaludar = persona1.diHola;

persona1.diHola();
persona2.diHola();
funcionSaludar(); // Error

// True
console.log(funcionSaludar === persona1.diHola);
// True
console.log(funcionSaludar === Persona.prototype.diHola);


funcionSaludar.call(persona1);
```

Ejercicio:
Modelar una clase animal implementando propiedades y métodos que creas necesarios.

Herencia

La **herencia** es una manera de **crear una clase** como una versión **especializada** de una o más clases, JavaScript **sólo permite herencia simple**.

La clase especializada comúnmente se llama hija o secundaria, y la otra clase se le llama padre o primaria.

En JavaScript la herencia se logra mediante la asignación de una instancia de la clase primaria a la clase secundaria, y luego se hace la especialización.

https://ingenieria.udistrital.edu.co/pluginfile.php/39190/mod_resource/content/1/Herencia%20simple.pdf

Herencia

Creamos la clase Persona con dos métodos.

Algo bastante útil a saber es que podemos declarar métodos fuera de la cadena de prototipos en el constructor de la clase simulando un método "**privado**".

```
function Persona(primerNombre) {
     this.primerNombre = primerNombre;

     // método "privado" de Persona
     this.getNombre = function () {
          return this.primerNombre;
     };
}

Persona.prototype.diHola = function () {
     console.log('Hola, Soy ' + this.primerNombre);
};

Persona.prototype.caminar = function () {
     console.log('Estoy caminando');
};
```

Herencia

Definimos la clase Estudiante como una clase especializada de Persona.

Luego redefinimos el método diHola() y agregamos el método diAdios().

```
function Estudiante(primerNombre, matricula) {
    Persona.call(this, primerNombre);

    this.matricula = matricula;
}

Estudiante.prototype = Object.create(Persona.prototype);
Estudiante.prototype.constructor = Estudiante;

Estudiante.prototype.diHola = function () {
    console.log('Hola, Soy ' + this.primerNombre + ' Mi
matricula es ' + this.matricula);
};

Estudiante.prototype.diAdios = function () {
    console.log('¡Adios!');
```

Herencia

En este caso **diHola** está definido en dos clases, pero por como funciona la cadena de prototipos **la primer ocurrencia en la cadena es ejecutada**, por lo tanto en este caso se ejecuta el **diHola de Estudiante**.

Con la palabra instanceof podemos saber si un objeto es una instancia particular de una clase o superclase.

```
var estudiante = new Estudiante('Alicia', 131414);


estudiante.diHola();
estudiante.caminar();
estudiante.diAdios();

// True
console.log(estudiante instanceof Estudiante);
// True
console.log(estudiante instanceof Persona);
```

Ejercicio:
Sobre la clase animal modelada anteriormente crear una clase especializada mamífero con las propiedades y métodos que creas necesarios.