

Prototipos

Prototipos

¿Por qué tan mala fama?

¡Es un mecanismo muy sencillo!

Distinto a otros lenguajes

Prototipos

Un objeto:

```
var obj = {uno: 1, dos: 2};
```

qué pasa si hacemos:

```
obj.uno; //1
```

Prototipos

```
var obj = {uno: 1, dos: 2};
```

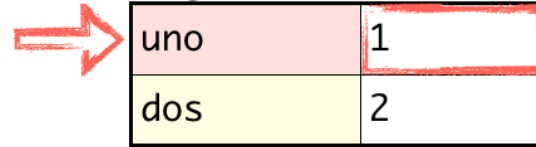
obj

uno	1
dos	2

```
obj.uno; // 1
```

obj

uno	1
dos	2



Prototipos

```
var obj = {uno: 1, dos: 2};
```

obj

uno	1
dos	2

Si hacemos:

```
obj.tres; // undefined
```

obj

uno	1
dos	2
Not found!	undefined



Prototipos

```
var obj = {uno: 1, dos: 2};
```

obj

uno	1
dos	2

¿De dónde sale?

```
obj.toString(); // '[object Object]'
```

obj

uno	1
dos	2
Not found!	undefined



¿?

Prototipos

```
obj.toString(); // '[object Object]'
```

obj

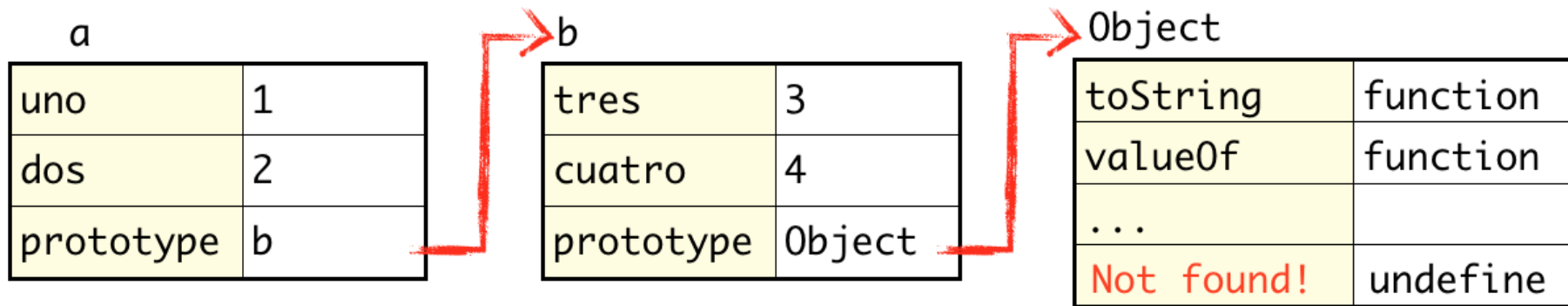
uno	1
dos	2
prototype	Object

Object

toString	function
valueOf	function
...	
Not found!	undefined

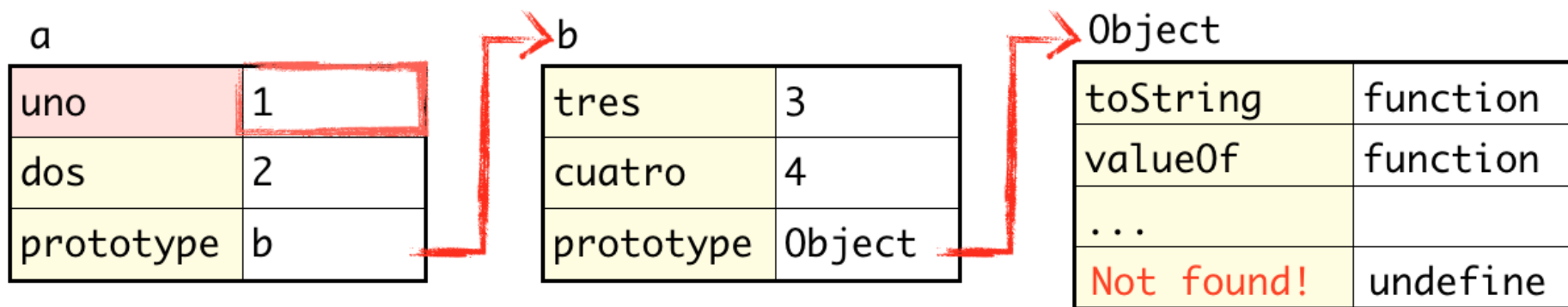


Prototipos



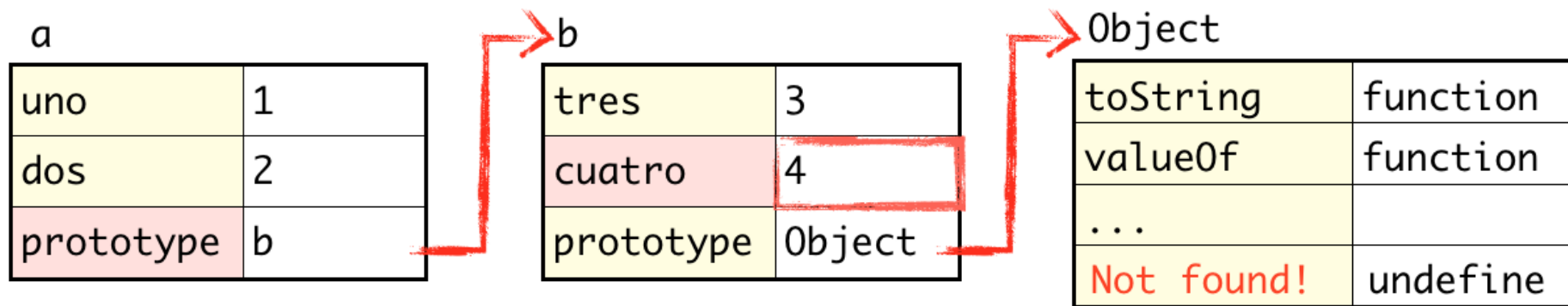
Prototipos

`a.uno; // 1`



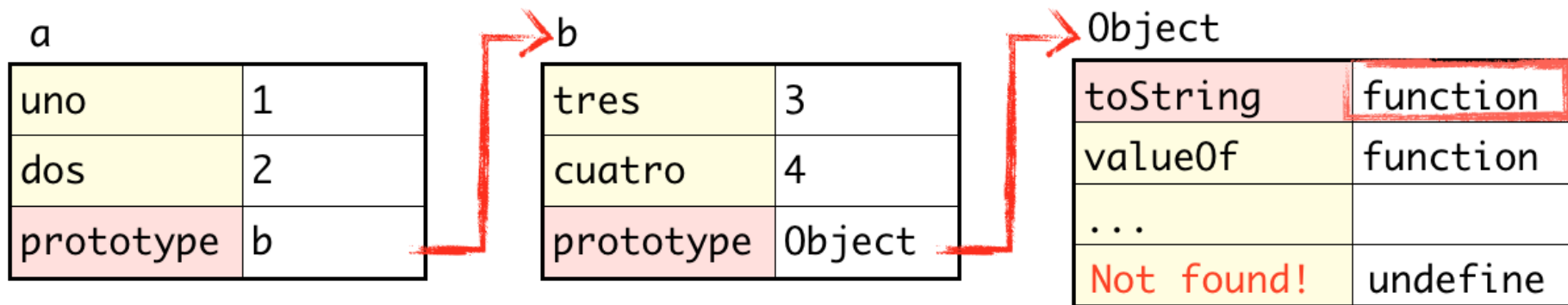
Prototipos

a.cuatro; // 4



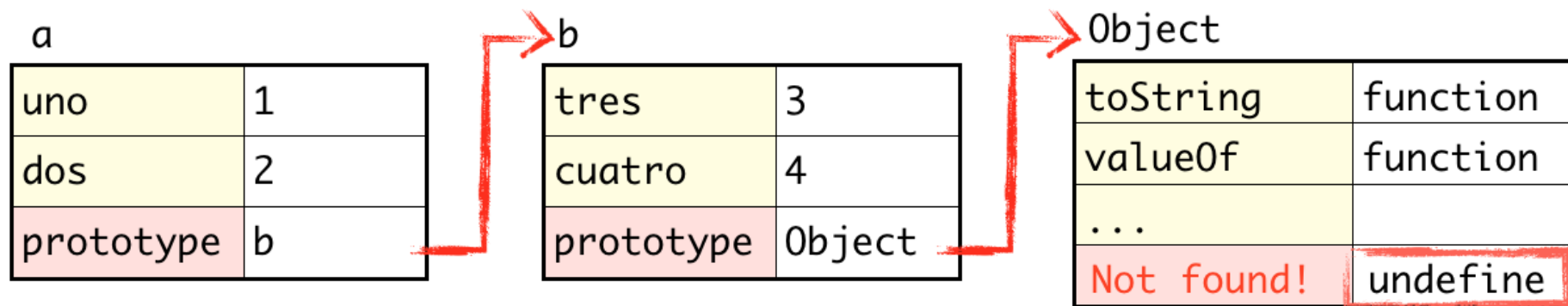
Prototipos

```
a.toString; // [object Object]
```



Prototipos

`a.noExiste; // undefined`



Prototipos

Pero... ¿Cómo establezco el prototipo de un objeto?

No se puede hacer directamente

No se puede modificar el prototipo de objetos literales

Solo objetos generados (con new)

Constructores!

Prototipos

Una función se puede ejecutar de 4 maneras:

- Invocando directamente la función

- Enviando un mensaje a un objeto (método)

```
(function() { alert("Hey!"); })();
```

```
objeto.metodo();
```

```
new MiConstructor();
```

```
fn.call({}, "param");
```

Prototipos

```
new MiConstructor();
```

Constructores

Funciones

Invocación precedida por new

Su contexto es un objeto recién generado

return implícito

La única manera de manipular prototipos

Constructores

```
function Constructor(param) {  
    // this tiene otro significado!  
    this.propiedad = "una propiedad!";  
    this.cena = param;  
}  
  
var instancia = new Constructor("Pollo asado");  
instancia.propiedad; // una propiedad!  
instancia.cena; // "Pollo asado"
```

Constructores

Tres pasos:

1. Crear un nuevo objeto
1. Prototipo del objeto = propiedad prototype del constructor
1. El nuevo objeto es el contexto del constructor

Constructores

```
var b = {uno: 1, dos: 2};
```

```
function A() {  
    this.tres = 3;  
    this.cuatro = 4;  
}
```

```
A.prototype = b;
```

```
var instancia = new A();  
instancia.tres; // 3  
instancia.uno;  // 1
```

Constructores

```
var b = {uno: 1, dos: 2};
```

```
function A() {  
  this.tres = 3;  
  this.cuatro = 4;  
}
```

```
A.prototype = b;
```

```
var instancia = new A();  
instancia.tres; // 3  
instancia.uno;  // 1
```

instancia

tres	3
cuatro	4
proto	b



b

uno	1
dos	2
proto	Object

Constructores

`.hasOwnProperty(name)`

Distinguir las propiedades heredadas de las propias
true solo si la propiedad es del objeto

```
instancia.hasOwnProperty("tres"); // true  
instancia.hasOwnProperty("uno"); // false
```

Constructores

```
var comun = { empresa: "ACME" };
```

```
function Empleado(nombre) {  
  this.nombre = nombre;  
}
```

```
Empleado.prototype = comun;
```

```
var pepe = new Empleado("Pepe");
```

```
pepe.nombre; // "Pepe"
```

```
pepe.empresa; // ???
```

Constructores

```
var comun = { empresa: "ACME" };
```

```
function Empleado(nombre) {  
    this.nombre = nombre;  
}
```

```
Empleado.prototype = comun;
```

```
var pepe = new Empleado("Pepe");
```

```
comun.empresa = "Redradix";
```

```
var antonio = new Empleado("Antonio");  
antonio.empresa; // ???
```

Constructores

```
var comun = { empresa: "ACME" };
```

```
function Empleado(nombre) {  
    this.nombre = nombre;  
}
```

```
Empleado.prototype = comun;
```

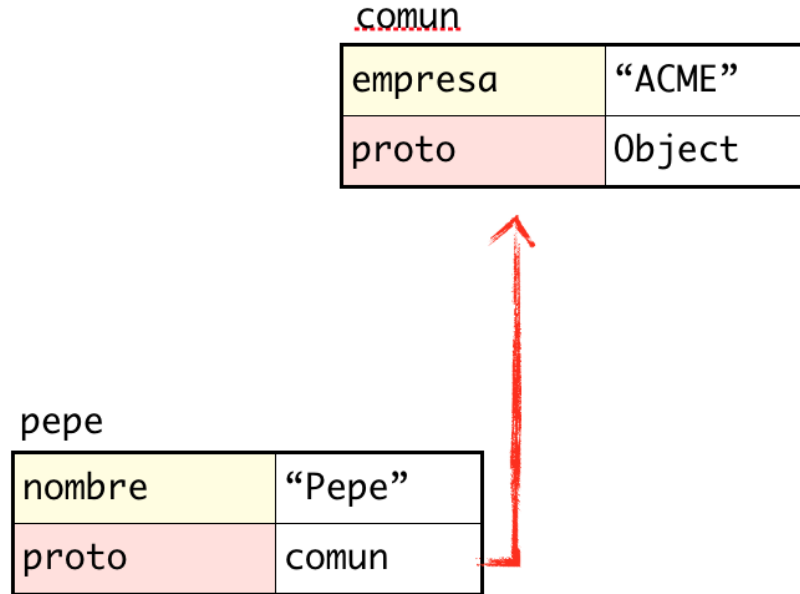
```
var pepe = new Empleado("Pepe");
```

```
comun.empresa = "Redradix";
```

```
var antonio = new Empleado("Antonio");
```

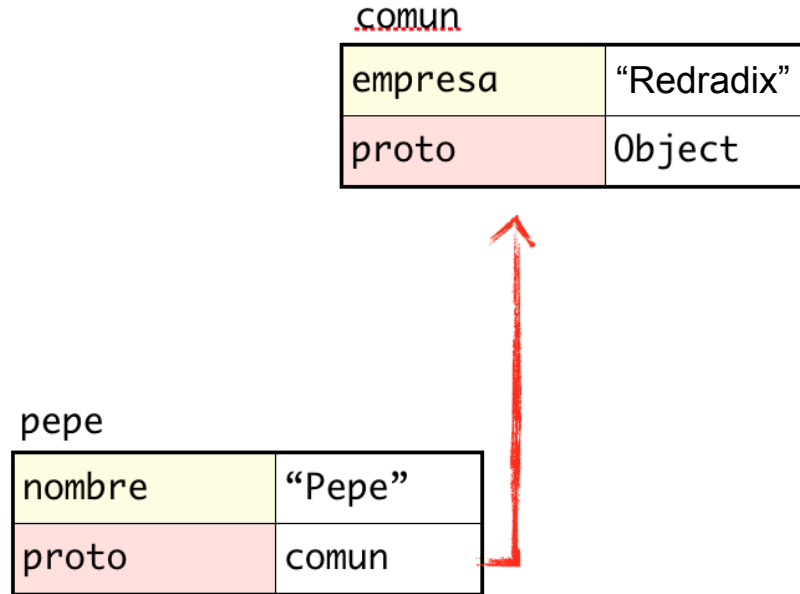
```
pepe.empresa; // ???
```


Constructores



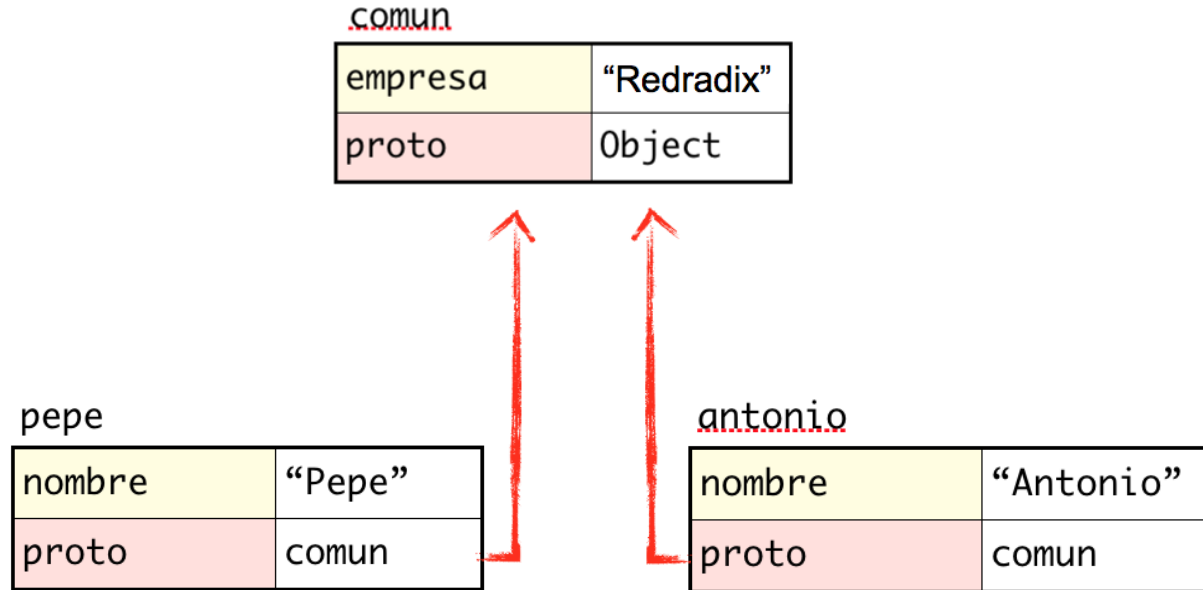
```
var pepe = new Empleado("Pepe");
```

Constructores



`comun.empresa = "Redradix"`

Constructores



```
var antonio = new Empleado("Antonio");
```

Constructores

Es decir:

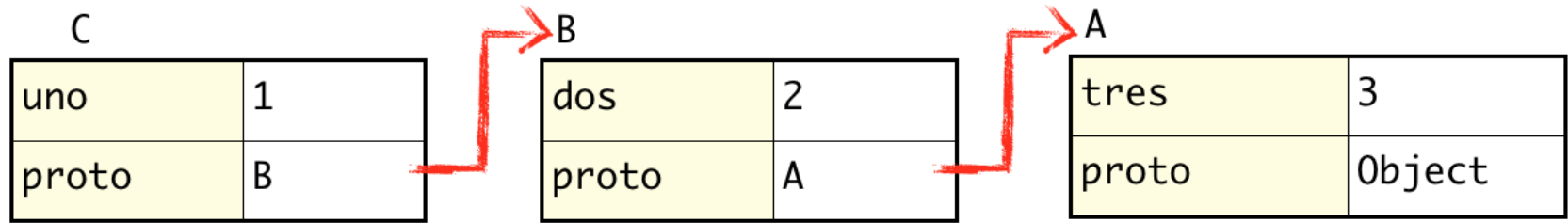
Las propiedades de los prototipos se comparten!

Se resuelven dinámicamente

Modificar un prototipo afecta a todas las instancias anteriores (y futuras)!

Constructores

¿Cómo hacer que C herede de B que hereda de A?



```
var instancia = new C();  
instancia.tres; // 3
```

Constructor

```
function C() {  
    this.uno = 1;  
}
```

```
var instancia = new C();  
instancia.tres; // 3
```

Constructor

```
var B = {dos: 2}
```

```
function C() {  
  this.uno = 1;  
}
```

```
C.prototype = B;
```

```
var instancia = new C();  
instancia.tres; // 3  
instancia.dos; // 2
```

Constructor

```
var A = {tres: 3};  
function B() {  
    this.dos = 2;  
}  
B.prototype = A;  
function C() {  
    this.uno = 1;  
}  
C.prototype = B;  
  
var instancia = new C();  
instancia.tres; // 3
```


Constructor

```
var A = {tres: 3};  
function B() {  
    this.dos = 2;  
}  
B.prototype = A;
```

```
function C() {  
    this.uno = 1;  
}  
C.prototype = B;  
var instancia = new C();  
instancia.dos; // ???
```

Constructor

```
var A = {tres: 3};  
function B() {  
    this.dos = 2;  
}  
B.prototype = A;  
function C() {  
    this.uno = 1;  
}  
C.prototype = B;  
  
typeof C.prototype;  
C.prototype.dos; // ???
```

Constructor

```
var A = {tres: 3};  
function B() {  
    this.dos = 2;  
}  
B.prototype = A;  
function C() {  
    this.uno = 1;  
}  
C.prototype = new B();  
  
var instancia = new C();  
instancia.tres; // ???
```

Constructor

```
var A = {tres: 3};  
function B() {  
    this.dos = 2;  
}  
B.prototype = new A();  
function C() {  
    this.uno = 1;  
}  
C.prototype = new B();  
  
var instancia = new C();  
instancia.tres; // ???
```

Cadena de prototipos

La herencia en varios niveles necesita:

Encadenar prototipos

La propiedad prototype del “sub constructor” ha de ser siempre new Padre()

Es la única manera de mantener el “padre del padre” en la cadena!

Asignación de prototipo

Con `Object.create(proto [, properties])`:

```
Object.create(null);
```

```
Object.create(Array.prototype)
```

```
Object.create(Object.prototype, {  
  unaPropiedad: { writable:true, value: "unaPropiedad" },  
  otraPropiedad: {  
    configurable: false,  
    get: function() { return 10 },  
    set: function(value) {  
      console.log("Setting `unaPropiedad` to", value); this.unaPropiedad  
        = value}  
  }});
```

Atributos de propiedades

Enumerable: `true/false`. Se accede a las propiedades mediante bucle `for...in` o `Object.keys`

Writable: `true/false`. Se puede modificar.

Configurable: `true/false`. Se puede borrar mediante `delete`. Se puede modificar el valor de `enumerable` y `writable`

Vamos a programar de una vez

Vamos a sondear el campo del aprendizaje automático haciendo un ejercicio de programación genética. Vamos a escribir un programa que genera copias cubistas de cuadros componiendo triángulos semi transparentes.

Vamos a crear nuestro **Cubista Automático**.

Cubista automático

Para implementar a nuestro cubista, necesitamos establecer primero un diccionario de conceptos:

Individuo: un conjunto de N triángulos que, al pintarlos, generan una "versión" del cuadro.

ADN: la estructura de datos con la que representamos los N triángulos de un individuo

Cromosoma: cada uno de los triángulos que componen el ADN

Fitness: el parecido de un individuo al cuadro original

Cubista automático

El funcionamiento general del cubista es muy sencillo:

1. Guardamos el individuo que mejor fitness haya tenido hasta ahora
2. Introducimos alguna mutación aleatoria en su ADN
3. Comparamos el fitness del hijo con el del padre
4. Volvemos a (1)

Cubista automático

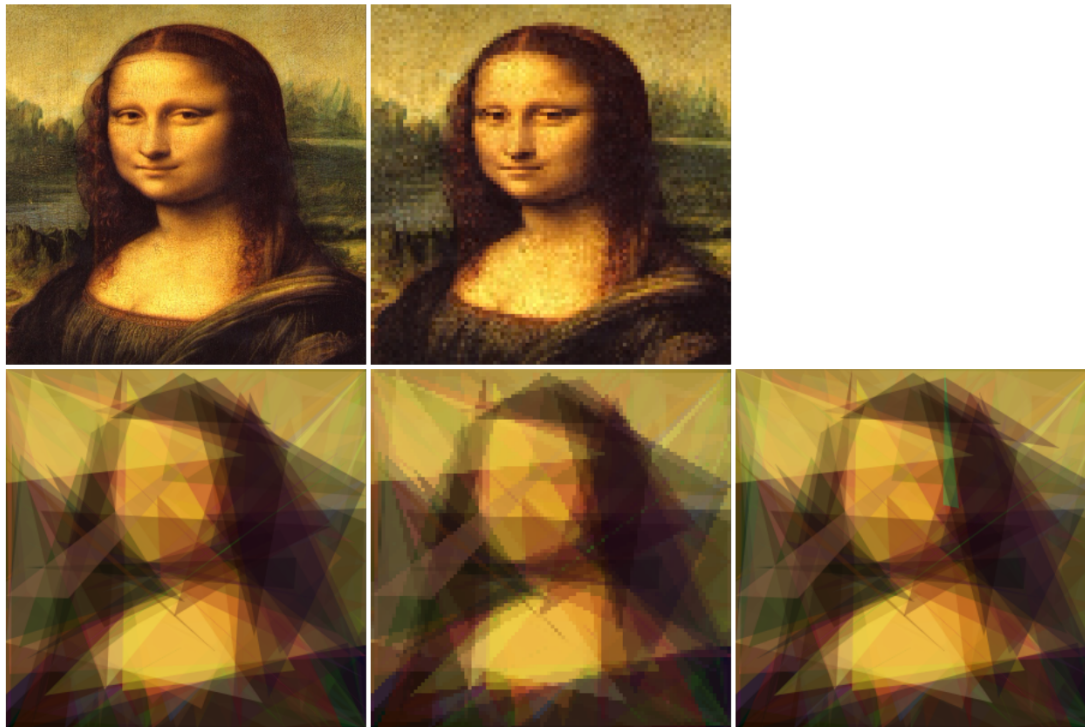
Vuestro trabajo consiste en implementar los métodos:

getReflmageUrl(): devuelve la url del cuadro a copiar

generateRandomIndividual(w, h): genera el primer individuo inicial, con ADN aleatorio pero todos los triángulos empiezan en NEGRO y ALPHA: 0.3

mutate(individual, w, h): que se encarga de generar la descendencia de individual a base de introducir alguna mutación en su ADN

Cubista automático



Cubista automático

Cada mutación debería:

- seleccionar algunos cromosomas aleatoriamente

- tal vez modificar la posición de uno o más de sus vértices

- tal vez modificar algún componente de su color

- tal vez modificar su transparencia

- devolver un nuevo individuo completo

Detalles de implementación

```
{
  dna: {
    0: {
      color: { r: 23, g: 48, b: 100, a: 0.3 },
      sides: 3,
      1: { x: 10, y: 19 },
      2: { x: 23, y: 89 },
      3: { x: 87, y: 45 }
    },
    1: {
      /* ... */
    },
    /* ... */
  }
}
```

Detalles de implementación

Parámetros de configuración, definidos como constantes en el fichero:

POLY_SIDES: Cuántos vértices tiene cada cromosoma (para generar polígonos más complejos)

CHROMOSOMES: Cuántos cromosomas (polígonos) por cada individuo

CHROMOSOME_MUTATION_DELTA: Regula la magnitud de la mutación de posición

CHROMOSOME_COLOR_DELTA: Regula la magnitud de la mutación de color

MUTATION_PROBABILITY: Probabilidad de mutación de cada cromosoma

Detalles de implementación

Teneis unas cuantas utilidades a vuestra disposición en `utils.js`:

`rnd(n)`: Numero entero aleatorio entre 0 y N (N no incluido)

`rndVariation(n, delta)`: Devuelve $n + [\text{rnd}(-\text{delta}), \text{rnd}(\text{delta})]$. Sesgado hacia modificaciones pequeñas.

`sample(array)`: devuelve un elemento aleatorio de array

`maybelog(...)`: `console.log` con muy baja probabilidad de ejecutarse

Detalles de implementación

Notas generales:

Deberiais ver cierto parecido alrededor de la iteración 2k o 3k

Si hacéis mutaciones demasiado bruscas no va a funcionar

Si hacéis mutaciones demasiado sutiles va a tardar muchísimo (y no va a funcionar)

Lo ideal es hacer mutaciones que la mayor parte de las veces sean sutiles, pero de vez en cuando sean más bruscas

Detalles de implementación

Notas generales (2):

Si la probabilidad de mutación es demasiado alta, se perderá demasiada información genética en cada generación y los genes "buenos" no llegaran a la descendencia

Si la probabilidad de mutación es demasiado baja tardaremos mucho en ver resultados (pero funcionara eventualmente)

Si va demasiado lento en vuestro ordenador, cambiad la constante `COMP_GRID_SIZE` en `cubist.js` a un número más bajo (no menos de 35)