FUNCIONES

FUNCIONES

ARGUMENTOS

OTROS

FUNCIONES

FUNCIÓN COMO CONCEPTO

- Duplicación de Código
- Código más legible y limpio
- Comienza el concepto de test unitario
- Ahorramos tiempo de picar Código
- Permite definir un conjunto de acciones con una abstracción

FUNCIONES

PRINCIPIOS DE PROGRAMACIÓN

SOLID

- **Single-Responsibility-Principle** (SRP): única responsabilidad
- **Open-Closed Principle** (OCP): Cerrado a la modificación, abierto a la extension
- **Liskov Substitution Principle** (LSP): Si una clase padre y una clase hija comparten una función con ligeras modificaciones, no deberían cambiar sus comportamientos generals (output, input, abstraction, argumentos)
- **Interface Segregation Principle** (ISP): Las funciones expuestas de una clase deben tener su razón de ser. No utilizaremos herencia si no se comparten la mayoría de los métodos
- **Dependency Inversion Principle (DIP):** Las clases padre no pueden depender de alguna propiedad/característica de una clase hija

FUNCIONES

PRINCIPIOS DE PROGRAMACIÓN

DRY

- DON'T REPEAT YOURSELF

KISS

- KEEP IT SMALL AND SIMPLE

FUNCIONES

DECLARACIÓN DE FUNCIONES

- Keyword “function”
- Identificador de la función (verbo!)
- Paréntesis con los argumentos de entrada
- Llaves para definir el Código de la función

```
// Print the name and value of each property of o.  Return undefined.
function printprops(o) {
    for(let p in o) {
        console.log(`${p}: ${o[p]}\n`);
    }
}
```

FUNCIONES

EXPRESIONES DE FUNCIÓN

- Las funciones pueden estar definidas dentro de una variable o argumento

```
// This function expression defines a function that squares its argu
// Note that we assign it to a variable
const square = function(x) { return x*x; };

// Function expressions can include names, which is useful for recurs
const f = function fact(x) { if (x <= 1) return 1; else return x*fact

// Function expressions can also be used as arguments to other functi
[3,2,1].sort(function(a,b) { return a-b; });

// Function expressions are sometimes defined and immediately invoked
let tensquared = (function(x) {return x*x;}(10));
```

FUNCIONES

FUNCIONES FLECHA

- Otra manera de definir las funciones

```
const sum = (x, y) => { return x + y; };
```

FUNCIONES

FUNCIONES ANIDADAS

- Se pueden definir funciones dentro de funciones (not recommended)

```
> function suerte(a, b)
... {function yepe(c, d){a++;b++; console.log(a)}; yepe(1, 1); console.log(a);}
undefined
> suerte(1, 1)
2
2
```

FUNCIONES

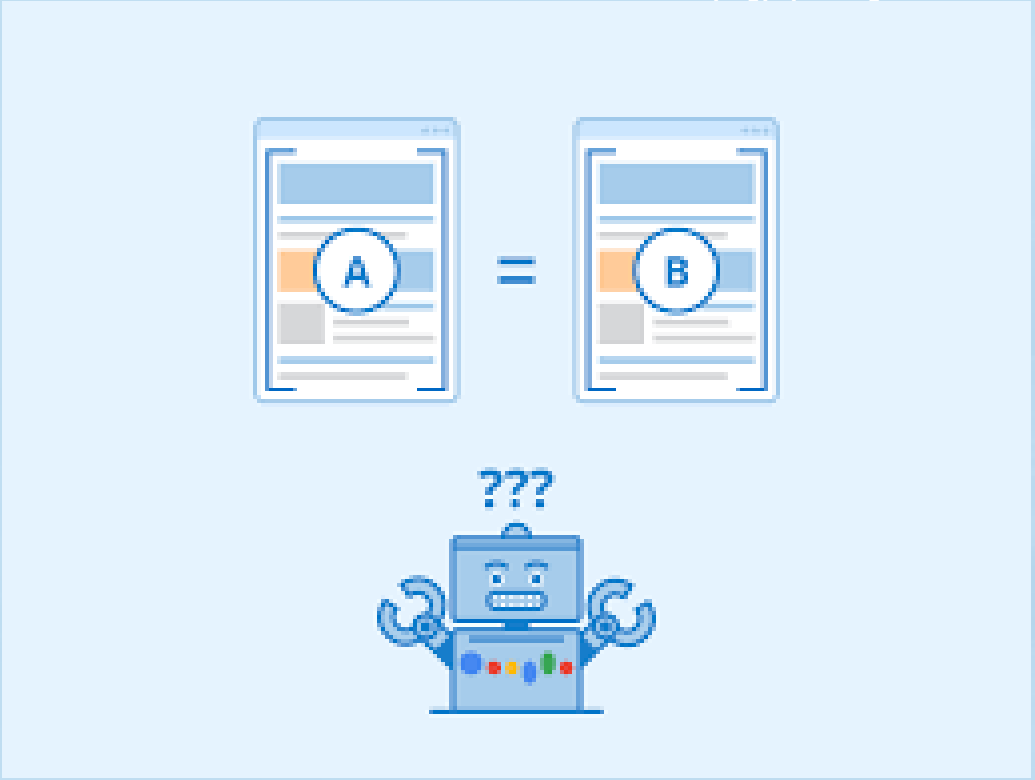
FUNCIONES RECURSIVAS

- Funciones que se llaman a sí mismas (OJO!)

```
> function recursive_yepe(number_yepes)
... {
... console.log(number_yepes);
... number_yepes++;
... if (number_yepes>10){return 1}
... recursive_yepe(number_yepes);
... }
undefined
> recursive_yepe(2)
2
3
4
5
6
7
8
9
10
```

ARGUMENTOS

Las funciones pueden tener unos parámetros de entrada: los argumentos.



ARGUMENTOS

ARGUMENTOS OPCIONALES

Prueba esto en tu ordenador

```
// Append the names of the enumerable properties of object o to the
// array a, and return a.  If a is omitted, create and return a new a
function getPropertyNames(o, a = []) {
    for(let property in o) a.push(property);
    return a;
}
```

ARGUMENTOS

ARGUMENTOS INFINITOS

Prueba esto en tu ordenador

```
function max(first=-Infinity, ...rest) {
    let maxValue = first; // Start by assuming the first arg is bigge
    // Then loop through the rest of the arguments, looking for bigge
    for(let n of rest) {
        if (n > maxValue) {
            maxValue = n;
        }
    }
    // Return the biggest
    return maxValue;
}

max(1, 10, 100, 2, 3, 1000, 4, 5, 6)  // => 1000
```

ARGUMENTOS

DESESTRUCTURACIÓN DE PARÁMETROS

```
function vectorAdd(v1, v2) {
    return [v1[0] + v2[0], v1[1] + v2[1]];
}
vectorAdd([1,2], [3,4])  // => [4,6]
```

```
function vectorAdd([x1,y1], [x2,y2]) { // Unpack 2 arguments into 4 p
    return [x1 + x2, y1 + y2];
}
vectorAdd([1,2], [3,4])  // => [4,6]
```

ARGUMENTOS

TIPO DE LOS ARGUMENTOS

- JavaScript no especifica el tipo
- Atención con no dejar claro el input de las funciones
- Documentación puede resolver el problema

```
> function que_haces(a, b){ return a+b };
undefined
> que_haces(1, 2)
3
> que_haces("suerte", "vaya")
'suertevaya'
> que_haces(1, "y ahora que")
'1y ahora que'
> que_haces("y ahora que", 2)
'y ahora que2'
>
```

OTROS

FUNCIONES COMO VARIABLES

- Las funciones pueden estar dentro de una variable.
- Ejemplo de mapeo (Ejemplo en Veridas)

```
> function suma(a,b){return a+b};
undefined
> function resta(a,b){return a-b};
undefined
> mapeo = {"suma": suma, "resta": resta}
{ suma: [Function: suma], resta: [Function: resta] }
> mapeo["suma"](4, 5)
9
```

OTROS

ALCANCE O “SCOPE”

- El alcance de las variables suele estar al mismo nivel de dónde se define y en niveles inferiores.
- ¿Qué responde este Código?

```
let scope = "global scope";           // A global variable
function checkscope() {
    let scope = "local scope";        // A local variable
    function f() { return scope; }    // Return the value in scope her
    return f;
}
let s = checkscope()();               // What does this return?
```

BIBLIOGRAFÍA

1. JavaScript: The Definitive Guide, 7th Edition
2. Eloquent JavaScript, 3th edition, Marijn Haverbeke
3. Clean Code: A Handbook of Agile Software Craftsmanship (Robert C. Martin Series)
4. https://www.guru99.com/v-model-software-testing.html, 20/03/2022
5. Modern c++ Programming with Test-Driven Development, Jeff Langr
6. Refactoring: Improve the design of existing Code, Martin Fowler
7. Game programming patterns, Robert Nystrom
8. https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898