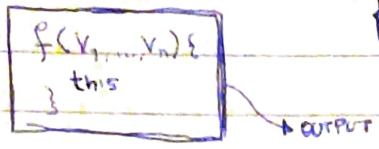
FUNZIONI

valori utilizzabili : { argomenti $(v_1, v_2, \dots, v_n)$
this



f → come proprietà obj: METODO
→ designato per creare obj: COSTRUTTORE
→ funzione

f è un oggetto (e come tale ha il suo costruttore Function)

- Può essere assegnato a una variabile e passato a un'altra funzione
- Posso impostargli proprietà come a qualsiasi oggetto e addirittura chiamare metodi su di esso.
- È una closure.

DEFINIZIONE

- function xxx(){...}
- (v) ⇒ (...)
- In oggetto {area: function(x,y){...}} ≈ {area(x,y){...}}
- Function() costruttore di function
- function* (generatore)
- async function (funzioni asincrone)

NB: Il nome xxx diventa una variabile il cui valore è la funzione stessa.

- Per chiamare una funzione f solo se è definita:

f?.(5,6) ≈ (f!==null && f!==undef) ? f(x):undefined

se f esiste, la chiamo con (5,6)

DEFINIZIONE (strict mode)

function declaration

```
function distance(x,y) {
  return y-x;
}
```

- hoisted
- distance diventa il nome di una variabile il cui valore è la funzione stessa.

function expression

- Il nome della funzione è opzionale.

```
const square = function(x){
    return x*x;
}
```

- Però se voglio lo posso specificare, utile per la ricorsione

```
const f = function fact(x) {
    ... fact(x-1)}
```

- Si può usare come argomento di altre funzioni.

```
[].sort(function(a,b){...})
```

- Può essere definita e subito invocata

```
let t = (function(x){return x*x;}(10))
```

- NO hoisted, let e const non si possono spostare!

arrow function

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

ancora più compatto

```
const f = (x,y) => x+y
```

con un solo parametro

```
(x) => x*x;
x => x*x;
```

senza parametri

```
() => 42;
```

occhio che comunque potrebbero generarsi equivoci

---

function declaration

```
function f(v){return v;}
```

- hoisted
- ha il prototype
- this (invocation context)
  - undefined

function expression

```
const f = function(v){return v}
```

- No hoisted
- ha il prototype
- this (inv. contex)
  - undefined

arrow function

```
const f = (v) => v;
```

- no hoisted
- NON ha prototype
- ~~Eredita il~~ this (i. con) ~~dall'ambiente dove è definita~~
  - dove sono definite

```
function f () {
  this.v = 5
  function g(){...}
}
```

→ se lo definisco come ⇒ vede il valore definito in f.

INVOCAZIONE (Struct mode)

1) FUNCTION INVOCATION

let d = distance(2,4);

Invocation Context

NO strict → Global

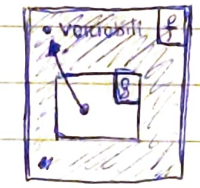
STRICT → undefined

2) METHOD INVOCATION:

o.m()

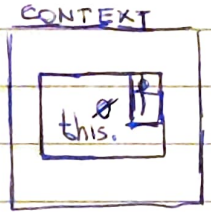
Invocation Context : o

- ∀ funzione g ∨ vede le variabili di f (innestata)



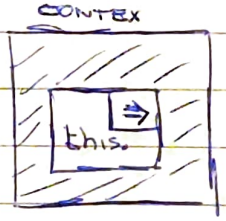
f{ g{} }

- Consideriamo un contesto che NON sia un oggetto



NON - ARROW

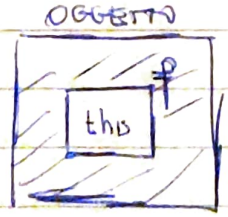
this in f è undefined



ARROW

this in ⇒ è context

- Consideriamo un contesto che SIA un oggetto



NON - ARROW

this in f è l'oggetto

idem

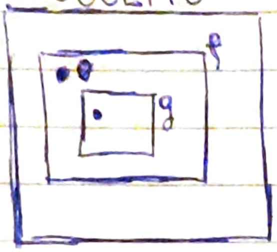
RIASSUNTO

This di una NON-ARROW è undefined || oggetto

This di una ARROW è il contesto in cui è definito

Es.

OGGETTO

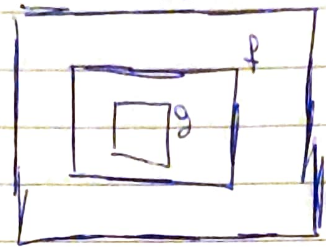


f, g non arrow

- $this_f$ = oggetto
- $this_g$ = undefined (perché il contesto di g è f e f non è un oggetto, quindi è undefined)

- g vede le var_f.

OGGETTO



f non arrow, g arrow

- $this_f$ = oggetto
- $this_g$ = f ($this_g$ = $this_f$ = oggetto)
- g vede le var_f

③ CONSTRUCTOR

O = new Object()
imposta anche il prototype
il [new] è un modo di chiamare la funzione costruttore

④ INDIRECT

La funzione è un oggetto, per cui ha dei metodi che possono essere chiamati

f.call(...)
f.apply(...)

Permettono di utilizzare una funzione in un contesto diverso da quello per cui è stata pensata!

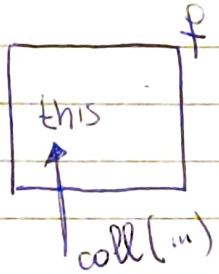
f.call(obj, (...))

↓ imposta il this della funzione

(...) parametri funzione

```
f() {
    ... use this
}
```

f.call(obj)
inietto da fuori il this, di modo che posso cambiargliele di volta in volta



Es:

```
function saluta (saluto) {
    return saluto + this.nome;
}
```

definito in un contesto non oggetto, il this vale undefined quindi significa che questo funzione ha senso solo se chiamata con call.

saluta.call({nome: "Mauro"}, "Ciao") → "Ciao Mauro"
(this: {nome: "Mauro"}; saluto: "Ciao")