Statement ed Espressioni

STATEMENT ED ESPRESSIONI

Quello che vedremo potrà sembrarvi troppo teorico. Ma è fondamentale per rispolverare le eventuali misconcezioni che fisiologicamente potreste aver accumulato negli anni di uso.

STATEMENT ED ESPRESSIONI

- Il codice JavaScript che possiamo scrivere è sempre uno *Statement* oppure una *Espressione*.

- Strumento utile per comprendere la struttura del codice 👉 https://astexplorer.net ci permette di analizzare "ai raggi X" il codice JavaScript (e non solo). La AST (Abstract Syntax Tree) è la rappresentazione di come un inteprete JavaScript vede il codice.

STATEMENT ED ESPRESSIONI

- Le **Espressioni** producono valori.
- Le righe seguenti contengono delle **Espressioni**

```
myvar
1 + 1
myvar + 1
myfunction(1, "str")
```

Anche alcune keyword introducono delle espressioni, ad esempio `new`, `delete` o `void`.

> TRUCCO SEMPLICE
> Il frammento di codice può essere passato come argmento ad una funzione? Se sì allora è una **Espressione**.

STATEMENT ED ESPRESSIONI

- Gli **Statement** sono istruzioni. Un programma è costituito da una sequenza di **Statement**
- Gli Statement sono separati da punti e virgola

> NOTA
> Esiste la ASI (Automatic Semicolon Insertion) che agisce negli a capo in altre situazioni particolari e permette di omettere i punti e virgola. E' uno stile di codice amato da molti, ma ci sono dei casi che causano confusione per cui è buona pratica inserirli sempre.

Esempio

```
if (a) {}; let b = c;
```

STATEMENT ED ESPRESSIONI

Esempi di statement

- Blocco

```
{}
```

> NOTA
> Potreste essere abituati a pensare che in JavaScript non esistano i "blocchi" come in C, ma non è più così. La sintassi sopra mostrata definisce un blocco vuoto.
>
> NOTA 2
> Se siete portati a pensare che sia un "oggetto" state attenti. Vederemo più avanti che dipende da dove le graffe sono collocate.

STATEMENT ED ESPRESSIONI

Esempi di statement

- Statement vuoto

```
/* QUI */;
```

- `if` e `else`

```
if (a) {}
else {}
```

- `try`, `catch`, `switch`, `for`, `while` ... si comportano come siamo abiutati dagli altri linguaggi di programmazione della famiglia ALGOL (C, C++, Java)

`let` e `const` (e `var`)

`let` e `const` (e `var`)

- Con `let` dichiariamo una o più variabili il cui riferimento può essere modificato con successive assegnazioni. Si dice essere un "riferimento mutabile".
- Con `const` dichiariamo una o più variabili il cui riferimento sarà costante e in nessun modo potremo assegnare nuovi valori. Sono "riferimenti immutabili".

> In Java: `const` è uguale alle variabili con il modificatore `final`.
>
> Non ha alcuna somiglianza con le costanti di C!
>
> Non implica immutabilità degli oggetti e valori. Lo vedremo nella seconda giornata.

`let` e `const` (e `var`)

Esempi

```
let a = 1;

a; // 1

a = 2;

a; // 2
```

```
const a = 1;

a; // 1

a = 2; // TypeError!
```

`let` e `const` (e `var`)

- DOMANDA: Cosa succede in questo caso?

```
let a = 1;

let a = 2;

a; // ??
```

  - Il valore `1`
  - Il valore `2`
  - Errore runtime
  - Errore compile time

`let` E `const` (E `var`)

- DOMANDA: Cosa succede in questo caso?

```
let a = 1;

let a = 2;

a; // ??
```

  - Il valore `1`
  - Il valore `2`
  - Errore runtime
  - Errore compile time

> RISPOSTA
> SyntaxError (compile time), con let puoi ri-assegnare ma non ri-definire!

LET E CONST (E VAR) 

- DOMANDA: Cosa succede in questo caso?

```
const a;
```

  - Viene inizializzata la variabile `a` ad `undefined`
  - Errore runtime
  - Errore compile time

LET E CONST (E VAR)

- DOMANDA: Cosa succede in questo caso?

```
const a;
```

  - Viene inizializzata la variabile `a` ad `undefined`
  - Errore runtime
  - Errore compile time

> RISPOSTA
> Errore a compile time! non ha senso una variabile costante senza valore!

Variables Resolution

VARIABLES RESOLUTION

Algoritmo di risoluzione di una variabile

1. Cerco nello scope locale
    1. come variabile (`var`, `let` o `const`)
    2. come funzione con quel nome (`function XX`)
    3. altrimenti vado allo scope più in alto
2. Cerco nello scope di funzione
    1. come parametro della funzione
    2. come nome della funzione
    3. altrimenti vado nello scope più in alto
3. Se sono sullo scope globale, allora cerco
    1. come nome di proprietà dell'oggetto globale (aka `window` sul web e `global` in Node.js)

VARIABLES RESOLUTION

```
let a = 1;

{
    let a = 2;
}

console.log(a);
```

```
let a = 1;
{
    let a = 2;
    console.log(a);
}
```

VARIABLES RESOLUTION

```
let a = 1;

{
    let a = 2;
}

console.log(a); // 1
```

```
let a = 1;
{
    let a = 2;
    console.log(a); // 2
}
```

VARIABLES RESOLUTION

```
let a = 1;
function myfunction(a) {
    console.log(a);
}
myfunction(2);
```

```
let a = 1;
function myfunction() {
    console.log(a);
}
myfunction();
```

```
let a = 1;
function a() {
    console.log(a);
}
a();
```

VARIABLES RESOLUTION

```
let a = 1;
function myfunction(a) {
    console.log(a);
}
myfunction(2); // 2
```

```
let a = 1;
function myfunction() {
    console.log(a);
}
myfunction(); // 1
```

```
let a = 1;
function a() {
    console.log(a);
}
a(); // [object Function]
```

VARIABLES RESOLUTION

```
const a = 1;
{
    let a = 2;
    console.log(a);
}
```

```
const a = 1;
{
    const a = 2;
    console.log(a);
}
```

VARIABLES RESOLUTION

```
const a = 1;
{
    let a = 2;
    console.log(a); // 2
}
```

```
const a = 1;
{
    const a = 2;
    console.log(a); // 2
}
```

Piccolo inserto su assegnazione di variabili

```
let a = 1;
let b = a;

b; // 1
b = 2;

a; // 1
```

```
let c = {};
let d = c;

d; // {}

c.a = 1
d.a = 2;

d.a; // 2
c.a; // ???? risposta: 2!!
```

Datatype

DATATYPE

- In JavaScript tutto è un oggetto. (O almeno ci appare come tale).
- Tutti i valori (al netto di `null` e `undefined`) possono essere trattati come oggetti: `x.y`.

DATATYPE

I valori **primitivi** sono 7:

- `String`
- `Number` e `BigInt`
- `Boolean`
- `Symbol`
- `Null` e `Undefined`

- Questi valori primtivi sono immutabili, ma possono essere manipolati come oggetti. Ad esempio `"aa".toUpperCase() === "AA"`.

DATATYPE

L'*Espressione* `typeof` riceve come argomento una *Espressione* oppure una pura *Reference* e restituisce una stringa che ne identifica il datatype.

```
typeof "test"           // "string"
typeof true             // "boolean"
typeof 12               // "number"
typeof 120n             // "bigint"
typeof Symbol("hi")     // "symbol"

typeof undefined        // "undefined"
typeof iDoNotExist      // "undefined"

console.log(iDoNotExist) // ReferenceError!

typeof (() => {})       // "function"
typeof function () {}   // "function"

typeof null             // "object" !!!!
typeof { a: 1, b : 2 } // "object"
```

DATATYPE

- DOMANDA

```
typeof typeof typeof 1 // ?
```

  - `"null"`
  - `"undefined"`
  - `undefined`
  - `"number"`
  - `"string"`
  - Errore runtime
  - Errore compile time

DATATYPE

- DOMANDA

```
typeof typeof typeof 1 // ?
```

  - `"null"`
  - `"undefined"`
  - `undefined`
  - `"number"`
  - `"string"`
  - Errore runtime
  - Errore compile time

> RISPOSTA
>
> `"string"`