Javascript Foundations I

Struttura di un programma

La struttura di un programma JavaScript può essere intesa come formata da:

Expression

L'espressione è la parte più piccola e va a identificare un frammento di *statement* (dichiarazione). Essa è fondamentale poiché produce un *value* (valore).

```
18
x + y
true && false
a <= 27
```

Statement

La dichiarazione (*statement*) è una sequenza di codice completa che diciamo essere più indipendente e autosufficiente della *expression* (termine comunque da prendere con le pinze, in quanto in JavaScript anche un'espressione può esserlo). La possiamo identificare subito in quanto termina con il classico *semicolon* (punto e virgola).

💡 Quando uno statement cambia un valore precedentemente definito, come ad esempio la riassegnazione di una variabile, questo è considerato come *side-effect* (effetto collaterale).

```
let x = 18;
console.log(x);
alert(Math.round(x));
if (true) console.log(x * 2);
true && console.log(x * 2);
x = 5;  // Side Effect
```

Program

Il programma vero e proprio, che sia articolato o un semplice snippet, non è altro che l'insieme di tutti gli statement che lo compongono.

```
function checkIfPresent (keyboard) {
  if (keyboard) {
    return "We can write the code if we want";
  } else {
    return "We need something to continue";
  }
}
```

Binding

Il *binding* (vincolo) non è altro che sinonimo di *variable* ed esso è elemento costituente di ogni codice. Più precisamente con binding intendiamo l'atto di legare una variabile a un nome, e con questa possiamo dialogare più facilmente con la memoria (precisamente locazione in memoria) e quindi dichiarare e/o assegnare ad essa un dato valore:

```
var globalIsValid = true;  // Everyone knows that variable in the program
const PI = 3.14            // this will never change
let myAge = 27;            // this change every year
                           // those are all expressions
```

Fondamentalmente abbiamo tre modi di definire una variabile tramite le apposite keywords:

- `var`

  La versione classica, definisce una variabile globale: possiede uno *scope* globale (l'argomento verrà approfondito nella sezione relativa allo *scope*). L'utilizzo di questa keyword ormai è stato superato con l'introduzione di `let` e `const` con ES6.

- `const`

  Definisce una costante, un particolare tipo di variabile immutabile e non riassegnabile. Se provate a riassegnare il valore a una costante JS vi solleverà un errore. Consiglio di utilizzare principalmente questo tipo di binding.

- `let`

  Definisce propriamente una variabile, mutabile e con scope limitato al *block* (con blocco si intende tutto ciò che viene avvolto da una coppia di *curly brackets* (parentesi graffe) entro la quale è definita.

Siamo soliti pensare alle variabili come contenitori d'informazioni, di valori. In realtà le cose non sembrano essere proprio così: una variabile in JavaScript punta al suo valore in memoria (l'argomento verrà approfondito nella sezione relativa all'*environment* (ambiente).

Questo modello di pensare le variabili ci viene in aiuto in quanto due diverse variabili possono **riferirsi** allo stesso valore (quindi non contenere lo stesso valore).

A parte nel caso di `const` possiamo riassegnare il valore a cui puntiamo con una riassegnazione, usando sempre lo stesso *operatore di assegnazione* (=).

Possiamo anche definire più variabili nella stessa espressione: `let width = 1920, height = 1080;`

Consiglio di riflettere qualche secondo sulla scelta del nome di una variabile; inoltre è ormai consuetudine preferire una formattazione tipo *camelCase* (l'argomento verrà approfondito nella sezione *clean code*).

Environment

Ogni volta che andiamo a creare una variabile questa, correlata al valore cui punta, viene automaticamente inserita in uno stack da qualche parte nella memoria del computer (l'argomento merita un approfondimento a parte). Il luogo dove risiedono tutte questa variabili è chiamato *Environment* (ambiente).

A questi vengono aggiunti tutta una serie di elementi forniti dall'ambiente standard di JavaScript, queste sono le *functions* (funzioni). Abbiamo già fatto ricorso a esse poco sopra, quando per esempio abbiamo chiamalo la funzione (più opportuno chiamarlo metodo, dopo vedremo perché) `console.log()`. L'argomento richiede un capito tutto proprio, per il momento voglio introdurlo qui per concludere con il binding.

Una funzione è un programma in miniatura. Ne esistono di ogni tipo, e di ogni tipo le possiamo scrivere. Per ora ci basta sapere che una funzione avvolge ed esegue una determinata sequenza di codice (quindi di statements) servendosi di un blocco codice (una coppia di parentesi graffe).

```
function checkIfPresent (keyboard) {
  if (keyboard) {
    console.log("We can write the code if we want");
  } else {
    console.log("We need something to continue");
  }
}
```

La funzione sopra accetta un valore come parametro, tale valore viene passato come condizione di uno *if statement* (tra poco ne vedremo il funzionamento). Non c'è nulla di tanto diverso dal comune `console.log()` se non che quest'ultimo è un metodo di un oggetto (presto tutto suonerà più familiare, promesso). Se vogliamo richiamare la nostra funzione possiamo procedere così

```
checkIfPresent(true);    // stamperà a video il valore "We can... "
checkIfPresent(false);   // stamperà a video il valore "We need... "
```

Fondamentalmente quello che succede qui è passare l'argomento true o false all'interno della nostra funzione, il quale verrà processato all'interno della funzione e in base a esso ci viene fornito un risultato finale. In base agli argomenti passati la funzione si comporterà nel modo opportuno.

Questo è soltanto il primo modo di usare una funzione e, nel caso di `checkIfPresent()`, non ritorna alcun valore ma soltanto dei *side-effects* (effetti collaterali).

Il valore passato tra parantesi è definito come *parameter* all'atto della definizione della funzione, mentre quando andiamo a richiamarla si dice *argument*.

Le funzioni però non si comportano soltanto come dei programmi in miniatura. Anzi, il loro punto di maggiore forza è dato dalla loro capacità di generare valore, di ritornare un valore, piuttosto che side-effects.

Anche se con le proprie particolarità, una funzione può comportarsi esattamente come una variabile, ovvero può puntare a un valore. Ma qui il punto di forza: non lo fa staticamente come una classica variabile, bensì dopo aver svolto una serie di operazioni.

Per comprendere meglio questo aspetto delle funzioni, bisogna fare un velocissimo accesso alla programmazione funzionale. Una funzione che rispetti questo particolare paradigma deve rispondere a questi tre requisiti:

- devono comportarsi come le funzioni matematiche, per un dato input restituiscono sempre lo stesso risultato (attenzione non valore, ma risultato, per tanto cambiando il parametro si avranno valori diversi);
- all'interno della funzione non dobbiamo interferire in alcun modo con il mondo esterno alla funzione, per esempio non possiamo modificare il valore di alcuna variabile, ne presentare alcun effetto collaterale;
- la funzioni posso essere passate e restituite da altre funzioni;

Questo è uno degli argomenti più impegnativi e a primo approccio tutto ciò suonerà come incomprensibile. Via via però questi concett diverranno sempre più chiari.

Per riassumere e tirare velocemente le somme: se una funzione risponde a queste condizioni, essa può essere definita come *pure function* (funzione pura).

```
function divisionBetween(firstNum, secondNum) {
  return firstNum / secondNum;
}
```

La funzione non modifica alcuna variabile esterna (seconda condizione) e restituisce un'operazione matematica tra gli argomenti passati (prima condizione).

```
divisionBetween(10, 5);  // Ritornerà 2, mentre
divisionBetween(28, 2);  // 14
```

Nel seguente caso, a differenza della funzione `checkIfPresent()`, la funzione si dice *return a value* (ritorna un valore) e, riagganciandoci a quanto detto prima, se una funzione ritorna un valore il suo comportamento sarà simili a quello di una variabile: si comporta come una qualunque espressione. Questo particolare comportamento, tipico di JavaScript, definisce le funzioni come *first-class objects*.

Sebbene questo sia un argomento più avanzato, ci basta sapere che, ancora una volta, se la nostra funzione restituisce una valore allora possiamo passare le nostre funzioni come argomento di un'altra funzione.

Considerate il seguente statement `console.log(Math.round(3.14));`

La funzione (più precisamente metodo, vedremo più avanti perchè) `round()` dell'oggetto `Math` è una funzione che *ritorna* il numero passato come argomento approssimato all'intero più vicino, dunque 3. Se proviamo a lanciare infatti `Math.round(3.14)` vedremo proprio questo. Questo valore è a sua volta passato come argomento di `console.log()` Non poi troppo diverso dallo scrivere direttamente `console.log(3)`. Ecco che abbiamo passato una funzione (e il suo valore di ritorno) come argomento di un'altra funzione (terza condizione).

```
const firstOperation = divisionBetween(30, 2);
const secondOperation = divisionBetween(20, 4);
divisionBetween(firstOperation, secondOperation);

// Oppure protremmo scriverla direttamente cosi:
divisionBetween(divisionBetween(30, 2), divisionBetween(20, 4));
```

Quest'ultima chiamata è effettivamente un po forzata, ma abituarsi a ragionare su esempi come questi può aiutare nell'apprendimento.