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";
  }
}
```

Indentation

Quando scriviamo il nostro codice è bene tenere a mente delle regole fondamentali di *stile*. Per il momento voglio solo introdurre l' *indentazione*. Lungo tutto questo documento troverete diversi spezzoni di codice, i cui margini a volte vengono spostati di qualche carattere. Questo è il processo che permette di dividere visivamente la logica che ruota attorno al nostro codice. Ai fini dell'esecuizione questa non causa alcun problema, potremmo scrivere infatti tutto in una sola riga, ma risulterebbe invece illegibile o, quanto meno, di difficile lettura. E' buona norma abituarsi fin da subito ad usare i rietri. Di norma si utilizza una tabulazione ad ogni indentazione (o due spazi).

Comments

I commenti rappresentano tutto il testo non interpretato e che quindi non fa parte del *flow* (flusso) del programma. Questi sono utili al fine di appuntare note per se stessi (se ritorniamo ad esempio su un progetto a distanza di tempo) e per gli altri sviluppatori che mettono mano al nostro codice.

Abbiamo due definizioni: in riga inserendo doppio *slash* `//` e in riga multipla utilizzando `/*` in apertura e `*/` in chiusura.

```
// Questo codice non verrà mai eseguito
/* Neanche
    questo
    qui */
```

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;  // Chiunque conosce globalIsValid
const PI = 3.14            // questa variabile non cambierà mai
let myAge = 27;            // questa invece cambia ogni anni
                           // Questi sono tutti e 3 degli statement
```

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*). Non è possibile usare gli spazi e i nomi delle *keywords*, come `let` `const` `if` `for` `while` `try` `do` ...

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.

Side-effects

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*.

Return a value

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.

Conditional execution

Come in ogni altro linguaggio di programmazione, anche il Javascript possiamo creare delle condizioni. Se la condizione passata come espressione si realizza allorra ritorna un determinato statement, altrimenti un altro.

Lo statement `if` accetta una *expression* (espressione) come condizione e, se questa risulta essere *true* (vera) allora ne restituisce lo statement all'interno del

block (blocco) che costituisce. Quindi la nostra condizione lavorerà su un valore booleano, dato dal risultato dell'espressione. La sintassi è abbastanza intuitiva: `if (expression) { statement };`

```
function takeTheUmbrella() {
  // tutte le operazioni che ci portano a prendere
  // l'ombrello dall'ingresso di casa
}

let itsRaining = true;

if (itsRaining) {        // La variabile booleana 'itsRaining' è true, dunque
  takeTheUmbrella();     // la funzione 'takeTheUmbrella()' verrà eseguita.
}
```


Se lo statement si riassume in un'unica riga, come nel caso di `itsRaining` possiamo omettere le parentesi graffe: `if (itsRaining) takeTheUmbrella();`

Possiamo specificare ancor più le nostre condizioni usando `else` e `else if` come nell'esempio seguente:

```
let itsRaining = true;

if (itsRaining) {        // La variabile booleana 'itsRaining' è true, dunque
  takeTheUmbrella();     // verrà eseguita.
} else if (itsCloudy) {
  dontStayOutTooLong();
} else {
  takeTheSunglasses();
}
```

Loops

Il concetto di *loop* (ciclo) si rifà semplicemente al concetto di ripetizione, eseguire dunque più volte lo stesso statement. Ma fondamentalmente la peculiarità di un loop è la sua capacità di permettere il ritorno ad una posizione precisa del programma, dove una determinata codizione viene vericata.

Mi spiego meglio. Intato è necessario chiamare all'appello il *pattern* (modello) classico che il loop prevede:

1. si definisce una variabile che funge da *counter* (contatore) e che tiene traccia del progresso del loop
2. si crea il loop, come ad esempio `while` e ad esso si da la condizione da verificare
3. all'iterno del blocco while si definiscono gli statement da eseuire, dove l'ultimo deve essere l'update del nostro *counter* (il che può essere un incremento ad esempio)

While

Vediamo subito una implementazione di quanto detto.

```
let jsExperience = 0;

while (jsExperience < 100) {
  console.log('Livello: ', jsExperience, 'Devi studiare ancora un po di Javascript');
  studyAndCoding();          // la funzione è puramente figurativa, non esiste
  jsExperience ++;
}

console.log('Complimenti, adesso puoi passare a React!');
```

Il ciclo sopra definito ci informa sul nostro livello di esperienza in javascript, se questo è inferiore a 100 allora continuerà ad eseguire lo statement che prevede un `console.log()` con l'esperienza attuale, una funzione che sembrerebbe esercitare lo studio e l'update del counter.

Quando alla fine la nostra variabile `jsExperience` verrà incrementata abbastanza da soddisfare la condizione, allora il ciclo si distrugge e si passerà direttamente allo statement successivo, ovvero i complimenti e l'invito a passare a React!

💡 Incrementare il counter con il doppio operatore più `jsExperience++` è analogo a scrivere `jsExperience = jsExperience + 1` o in un'altra forma abbreviata `jsExperience += 1`. Questo approccio è definito come *Updating Bindings Succinctly*.

⚠️ Provate a togliere il counter dal blocco di un ciclo e vedrete cosa succede! Il ciclo senza la sua condizione di update procederà all'infinito.

For

Il ciclo `for` proprio come `while` segue lo stesso pattern, il che prevede un *counter*, una *condition* e un *update*, ognuno dei quali separato da un *semicolon*.

La sintassi: `for (counter; condition; update);`

```
for (let personsQueuing = 16; personsQueuing > 0; personsQueuing --) {
  giveThemATicket();        // la funzione è puramente figurativa, non esiste
}
```

A differenza del `while` il nuovo ciclo sembra essere più immediato nella scrittura, risultando anche più veloce da leggere. La definizione di un counter, il checking della condizione e l'update avvengono al momento della dichiarazione del ciclo stesso. Come leggere il codice sopra? "*Se ci sono persone in coda dai loro un ticket e rimuovile dalla lista di attesa*", oppure, *"finchè il numero di persone è maggiore di 0 dai loro un ticket* `giveThemATicket()` *e passa alla persona successiva"*.

Consiglio di allena la propria capacità di *abstraction* (astrazione), ovvero pensare, e poi scrivere, le soluzioni astraendole da problemi reali. L'utilizzo di una corretta nominazione delle variabili o delle funzioni gioca un ruolo fondamentale. Dedicherò un capitolo proprio all'astrazione.

Break

Giungere alla fine della condizione, quindi il verificarsi del *false,* non è l'unica possibilità per interrompere un ciclo. A volte capita che vogliamo bloccarne l'esecuzione se qualcosa di particolare succede, come il passaggio di una determiata variabile. In questi casi ci viene in aiuto la *keyword* `break` che semplicemente distrugge il ciclo entro la quale viene chiamata.

```
itsTooLate = 0;
for (let personsQueuing = 16; personsQueuing > 0; personsQueuing --) {
  giveThemATicket();     // la funzione è puramente figurativa, non esiste
  itsTooLate++;
  if (itsTooLate >= 13) {
    break;
  }
}
```

Talvolta può capitare che in qualche ufficio pubblico la coda superi l'orario di ricevimento. In questi casi purtroppo i ticket non vengono più distribuiti. Questo è il caso del `for` di sopra che prevede un incremento della variabile `itsTooLate` la quale, al raggiungimento di un generico 13 (che possono essere il numero degli individui o l'orario), interrompe bruscamente il ciclo for.

Continue

Vi è un'ulteriore supporto alla gestione dei cicli. Se vogliamo ad esempio interrompere non l'intero ciclo, ma la condizione attuale, saltare quindi all'iterazione successiva, allora possiamo utilizzare la keyword `continue`.

```
for (let num = 100; num >= 3; num --) {
  if (num % 3 === 0) {
    if (num % 5 === 0) continue;
    console.log(num, 'è divisibile per 3');
  }
}
```

Invito allo sforzo che questo esempio richiede, verrà ripagato! In pratica possiamo leggere così: *"per ogni numero da 100 a 3, iterando ogni numero uno per uno, se il numero è divisibile per 3 ritorna il numero ma se lo stesso numero è divisibile per 5 allora si salta all'iterazione successiva"*.

L'operatore modulo (%) è qui utilizzato per ottenere i numeri divisibili per 3 e 5. Questa è l'espressione `(num % n === 0)`

Switch case

Considerato l'esempio seguente:

```
let dayOfTheWeek = 'Monday';

if (dayOfTheWeek === 'Monday') {
  console.log(dayOfTheWeek, 'is associated with the Moon');
} else if (dayOfTheWeek === 'Tuesday') {
  console.log(dayOfTheWeek, 'is associated with Mars');
} else if (dayOfTheWeek === 'Wednesday') {
  console.log(dayOfTheWeek, 'is associated with Mercury');
} else if (dayOfTheWeek === 'Thursday') {
  console.log(dayOfTheWeek, 'is associated with Jupiter');
} else if (dayOfTheWeek === 'Friday') {
  console.log(dayOfTheWeek, 'is associated with Venus');
} else if (dayOfTheWeek === 'Saturday') {
  console.log(dayOfTheWeek, 'is associated with Saturn');
} else if (dayOfTheWeek === 'Sunday') {
  console.log(dayOfTheWeek, 'is associated with the Sun');
} else {
  console.log('What planet are you really from?');
}
```

In base al valore della variabile `dayOfTheWeek` restituisce il corrispondente pianeta cui è assegnato il giorno della settimana. Con `switch` possiamo definire lo stesso procedendo come l'esempio sotto, il risultato è identico. Da tenere a mente che a `switch` sono necessari 4 elementi:

- una variabile da analizzare
- (*n)* casi da analizzare
- un `break` alla fine di ognuno di essi
- un caso `default` nel caso in cui nessuno degli altri si verifica.

```
let dayOfTheWeek = 'Monday';

switch (dayOfTheWeek) {
  case 'Monday':
    console.log(dayOfTheWeek, 'is associated with the Moon');
    break;
  case 'Tuesday':
    console.log(dayOfTheWeek, 'is associated with Mars');
    break;
  case 'Wednesday':
    console.log(dayOfTheWeek, 'is associated with Mercury');
    break;
  case 'Thursday':
    console.log(dayOfTheWeek, 'is associated with Jupiter');
    break;
  case 'Friday':
    console.log(dayOfTheWeek, 'is associated with Venus');
    break;
  case 'Saturday':
    console.log(dayOfTheWeek, 'is associated with Saturn');
    break;
  case 'Sunday':
    console.log(dayOfTheWeek, 'is associated with the Sun');
    break;
  default :
    console.log('What planet are you really from?');
    break;
}
```

💡 L'utilizzo di `switch` o di `if` dipende da situazione a situazione, talvolta la scelta è tratta anche dal gusto personale, certo è che non c'è alcun migliore o peggiore tra i due.