

Nello schema i blocchi moltiplicazione e potenza sono collegati bidirezionalmente perché il risultato della moltiplicazione serve al metodo potenza, che però a suo volta utilizza quando richiama il metodo moltiplicazione nel ciclo successivo.

Moltiplicazione

Il metodo, presi due numeri come input ci restituirà come output in cima allo stack il risultato.

Problemi riscontrati durante lo svolgimento

All'inizio la scelta era ricaduta su un metodo **iterativo**, e non ricorsivo (come visto a lezione), per verificare se riuscivo a creare il metodo autonomamente.

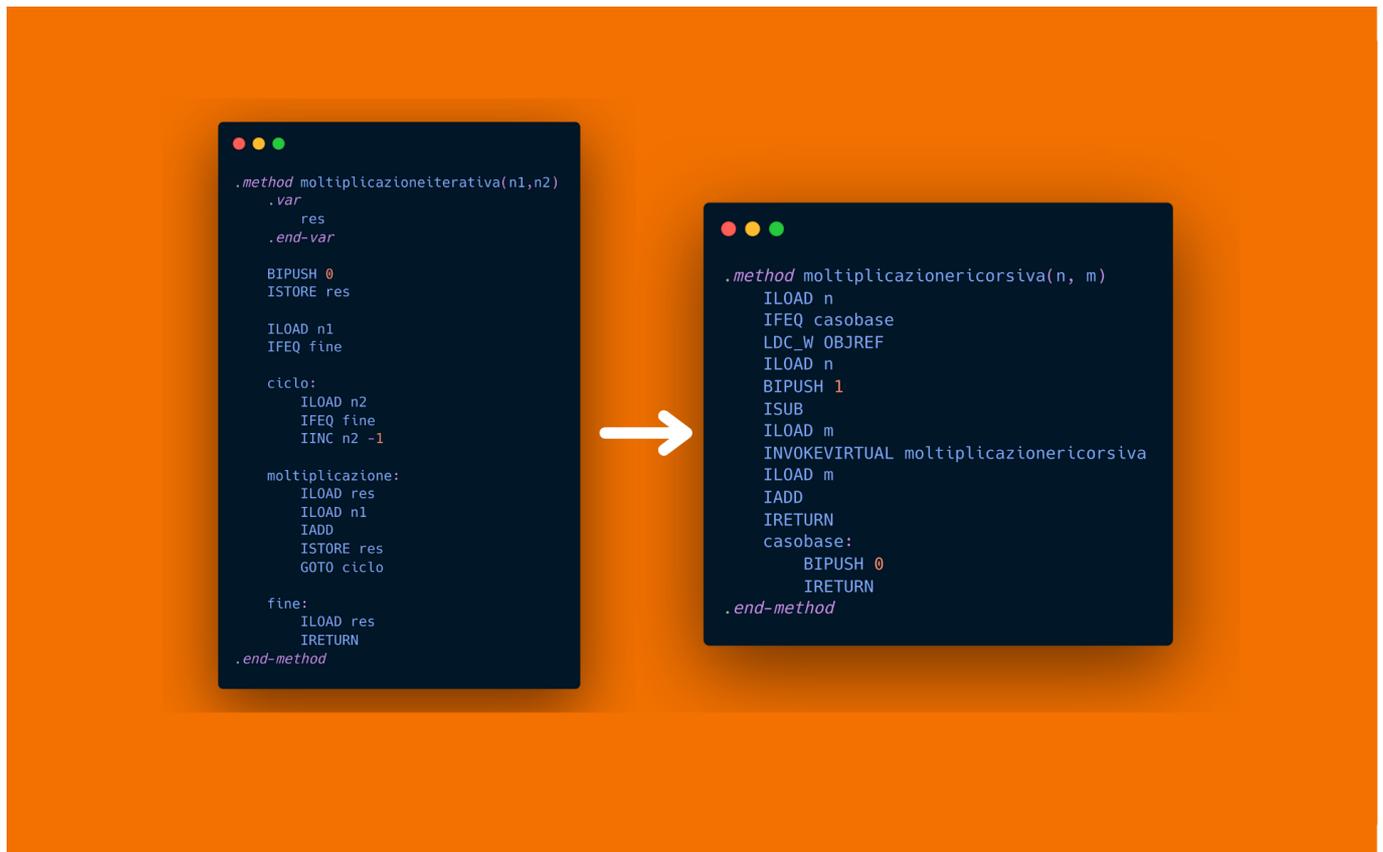
Il metodo era funzionante, però per calcoli leggermente più complessi il tempo di esecuzione era troppo elevato, e considerando che questo metodo veniva chiamato molteplici volte non era una soluzione sostenibile. Per questo motivo ho deciso di cambiare metodo passando ad un codice **ricorsivo**, riuscendo così a dimezzare il tempo di esecuzione. Infatti in questo modo la funzione viene richiamata più volte (occupando più celle nello stack, quindi più Ram) ma l'esecuzione è più veloce.

Il metodo prende due parametri interi, n e m , e restituisce il risultato della moltiplicazione di n e m attraverso una tecnica di ricorsione.

La funzione controlla se n è uguale a zero (label *caso base*) usando l'istruzione `IFEQ`, se la condizione è vera restituisce **immediatamente** 0 usando l'istruzione `BIPUSH` e `IRETURN`.



Se n è diverso da zero, la funzione richiama se stessa passando come argomenti $n - 1$ e m e somma m al risultato ottenuto dalla chiamata ricorsiva, finché $n - 1$ non sarà uguale a zero.



Potenza

La potenza è composta da un loop che utilizza l'esponente come indice, e finché questo indice non va a zero la funzione **richiama il metodo moltiplicazione** e salva il risultato.

In particolare il metodo prende in input due parametri: *base* ed *exp*, rispettivamente il valore della base e dell'esponente.

Il metodo utilizza una variabile *res* per salvare il risultato parziale di ogni iterazione del ciclo. Inizialmente, *res* viene impostato uguale alla base.

Il metodo quindi controlla se l'esponente è uguale a 0 e, in tal caso, restituisce 1 (perché ogni numero elevato a 0 è uguale a 1).

Se l'esponente è diverso da 0, il metodo entra in un ciclo che decrementa l'esponente a ogni iterazione finché non diventa uguale a 0. Ad ogni iterazione il metodo calcola la moltiplicazione tra *res* e la *base* (per questo motivo prima del ciclo abbiamo impostato *res* al valore di *base*) e salva il risultato nella variabile *res*.



Infine, quando l'esponente raggiunge 0, il metodo restituisce il valore di *res* ovvero il risultato della potenza richiesta.

```
.method potenza(base, exp)
  .var
    res
  .end-var

  ILOAD base
  ISTORE res

  ILOAD exp      //Carico l'esponente e controllo se è nullo.
  IFEQ casozero //Se è nullo vado alla label casozero (Esempio: n^0= 1)

  mastrodonardo:
    IINC exp -1 //Decremento l'esponente perchè viene utilizzato come
    ILOAD exp   //contatore per il ciclo finchè non diventa zero.
    IFEQ fine   //Controllo se l'esponente è zero, se lo è vado alla label fine
    LDC_W OBJREF
    ILOAD base
    ILOAD res
    INVOKEVIRTUAL moltiplicazionericorsiva
    ISTORE res

    GOTO mastrodonardo

  casozero:
    BIPUSH 1 //Restituisco 1 perchè n^0= 1
    IRETURN

  fine:
    ILOAD res //Carico il valore di res sullo stack e lo ritorno
    IRETURN

.end-method
```

Stampa dei Valori

Il metodo per la Stampa dei Valori è composto da due parti:

Prima Parte: per la stampa del valore dell'esponente, per ":" e per lo spazio.

Seconda Parte: per la stampa del risultato in Codice Binario composto da 32 bit

Per la prima parte la soluzione è abbastanza semplice, ci servirà caricare sullo stack il valore in esadecimale del **Codice ASCII** del simbolo da stampare, in particolare abbiamo:

- Stampa Esponente: sommiamo 30 in esadecimale al valore dell'esponente per ottenere il **codice ASCII**, questo è possibile perché le cifre nella tabella ASCII sono decodificate dal valore 0x30 al 0x39. Bisogna fare questa operazione perché l'istruzione `OUT` (istruzione per stampare un valore sulla Console) lavora esclusivamente con il Codice ASCII Esempio: $3 + 0x30 = 0x33 = '3'$
- Stampa ":" in questo caso basterà caricare in cima allo stack il valore 0x3A, ovvero il valore dei due punti del Codice ASCII e poi usare l'istruzione `OUT`
- Stampa " " (spazio): anche qui basterà caricare 0x20 sullo stack e poi usare l'istruzione `OUT`



La seconda parte invece è più complessa, perché inizia assegnando il valore 32 a *cont*, poi carica il valore di *n* sullo stack e inizia un ciclo che si ripete fino a quando *cont* non diventa 0.

Per ogni iterazione, il valore di *cont* viene caricato sullo stack e viene eseguito un'istruzione `IFEQ` per verificare se è uguale a 0. Se lo è, il ciclo termina e viene eseguito il codice dopo "fineCiclo". In caso contrario, viene duplicato il valore di *n* sullo stack e viene eseguita l'istruzione `IFLT` per verificare se è minore di 0. Se lo è, viene stampato il carattere "1" (codice ASCII 0x31) sulla console, altrimenti viene stampato il carattere "0" (codice ASCII 0x30). Dopo, passando per la label *dopoIF*, il valore di *n* viene sommato a quello duplicato e viene decrementato il valore di *cont* di 1 ($cont - 1$). Infine, il ciclo viene ripetuto.

Dopo che il ciclo è terminato, viene eseguito il codice della label *fineCiclo*, che rimuove l'ultimo valore dallo stack (che sarebbe il valore duplicato di *n* nell'ultima iterazione), stampa una newline (codice ASCII 0x0A) sulla console e restituisce il valore di *n*.

```
.method stampa(n, exp)
  .var
    cont
  .end-var

  ILOAD exp      //Stampo il valore dell'Esponente caricando il suo valore
  BIPUSH 0x30    //sullo stack e pushando 0x30 così possiamo trasformare il valore
  IADD          //in Carattere ASCII
  OUT

  BIPUSH 0x3A    //Stampo ":"
  OUT

  BIPUSH 0x20    //Stampo " " (spazio)
  OUT

  BIPUSH 32
  ISTORE cont
  ILOAD n

  ciclo:
    ILOAD cont
    IFEQ fineCiclo
    DUP
    IFLT stampaUno
    BIPUSH 0x30
    OUT
    GOTO dopoIF

  stampaUno:
    BIPUSH 0x31
    OUT

  dopoIF:
    DUP
    IADD
    IINC cont -1
    GOTO ciclo

  fineCiclo:
    POP
    BIPUSH 0x0A //Stampo la newline
    OUT
    ILOAD n
    IRETURN

.end-method
```

Conclusione

In conclusione, la scrittura in codice JAS si è rivelata una sfida interessante e impegnativa, in particolare con la funzione *Potenza* e *Stampa*, e la l'evoluzione della Memoria sullo Stack.



Programmare in un linguaggio di basso livello è risultato abbastanza ostico, anche quando si sviluppano programmi non troppo sofisticati, soprattutto per la fase di Debug.

