# mapio – Orali del 2020-07-08

January 20, 2021

Cos'è lo *specificand set*? A cosa serve? In relazione a quale parte del corso ne abbiamo parlato? Perché la specifica deve essere sufficientemente generale? Quale esempio si può fare a riguardo? Richiedere nella specifica che la stampa di un insieme di numeri li restituisca ordinati: non è generale perché non c'è alcuna ragione che un insieme sia ordinato. Che problema può costituire imporre questa specifica? Chi scrive l'implementazione è svantaggiato, in quanto costretto a fare un'implementazione complessa. Quindi, imponendo una caratteristica che non è propria dell'astrazione, taglio le gambe al programmatore.

Perché in Java ci sono tipi apparenti e tipi concreti? Qual è la caratteristica del linguaggio che richiede la presenza di queste due declinazioni? Perché in Java può succedere che ci sia un tipo apparente diverso da un tipo concreto? Ho una forma sintattica per dichiarare il tipo degli oggetti, che non sono obbligato a rispettare durante gli assegnamenti. In Python, ad esempio, ci sono solo tipi concreti.

---

Non è l'implementazione a guidare le scelte dell'astrazione. Per esempio, la classe `Token` non è immutabile grazie all'assenza di metodi mutazionali, ma è immutabile perché non voglio che cambi. Per cui è l'entità astratta ad essere immutabile: devo modellarla in quanto immutabile, e quindi implementarla con le accortezze opportune affinché questo accada.

[ ]:
```java
/**
 * Un token corrisponde ad un'operazione atomica che può assumere un valore
 * numerico in virgola mobile, oppure una stringa.
 * Siccome non ci sono metodi mutabili per il cambiamento dello stato,
 * l'oggetto Token è da considerarsi mutabile.
 */
public class Token {
    private String data;

    // …
}
```

Oltre all'errore osservato prima, sarebbe stato opportuno commentare la scelta di rappresentare il token con una stringa anche qualora questo sia numerico.

[ ]:
```java
public boolean èNumero() {
    try {
        double temp = Double.parseDouble(data)
        return true
```

```
    } catch (NumberFormatException e) {
        return false
    }
}

public Double numero() {
    try {
        return Double.parseDouble(data);
    } catch (NumberFormatException e) {
        System.err.println(e);
        return null;
    }
}
```

La variabile `temp` non è mai usata, quindi non ha senso dichiararla. Sarebbe stato più sensato tentare di invocare la funzione `Double.parseDouble` e basta. Tutto ciò ha ancora meno senso visto che il metodo `numero` non utilizza `èNumero`. In alternativa sarebbe stato meglio tenere `temp` da parte.

[ ]:
```
/**
 * L'oggetto Stack rappresenta un array di token …
 * Siccome ci sono metodi mutazionali, Stack è un oggetto mutabile
 */
public class Stack {
    private ArrayDeque<Token> stack;

    // …

    public void pushStack(Token t) {
        // Il seguente controllo non apparteneva alle specifiche, ma era da
        // intendersi come un "state tranquilli che non ci provo a mettere più
        // di 1024 elementi." Quindi sarebbe stato possibile anche
        // implementare `Stack` con un vettore
        if (lengthStack() > 1024)
            throw new IllegalStateException("La coda è piena e supera i 1024␣
 ↪token.");

        stack.offerFirst(t);
    }
}
```

In che senso `rappresenta un array di token`? Non è un array, ma una `Deque` basata su un array. Sarebbe meglio dire: «Come rappresentazione della pila ho scelto una `Deque`.» Ancora una volta, non è vero che `Stack` è mutabile perché ci sono metodi mutazionali: `Stack` è mutabile perché ho bisogno che lo sia, e come ovvia conseguenza devo definire metodi mutazionali. Il metodo `pushStack` dovrebbe accertarsi che il parametro `t` non sia `null`, perché in luogo di un parametro può essere posto un oggetto di tipo compatibile: `Token t` viene passato per riferimento, quindi potrei chiamare tale metodo con `pushStack(null)`, oppure

[ ]:
```
Token t = null;
pushStack(t);
```

È vero che la rappresentazione di `Token` è diversa da `null`, posto che la classe sia scritta bene, ma ciò non vuol dire che una variabile di tipo `Token` non possa essere `null`. Non controllare se `t` è `null` resta comunque una scelta plausibile, posto che a seguito di una pop dallo stack si controlli se l'oggetto ottenuto è `null` o meno.

Il vantaggio di usare un `InputStream` è che un programma di diversi GiB può essere letto ed elaborato usando una quantità costante di memoria. Non è quindi necessario leggerlo per intero ed elaborarlo solo successivamente. È scomodo che la classe `Parser` inserisca anche i token nella pila, perché diventa più complicato stamparli al contrario come richiesto.

Funzioni totali e parziali: cosa sono? Quando scegliamo di usare le une, e quando le altre? Come si potrebbe scrivere in Java una procedura che prende interi, stringhe, eccetera? Il punto è che tale questione non riguarda i tipi. I generici servono a limitare i tipi che una funzione prende in input, ed in un certo senso a conoscerli. La differenza tra funzioni totali e parziali riguarda i possibili valori per i quali è specificato il comportamento. Classico esempio della radice quadrata, che ha per dominio il sottinsieme dei reali positivi. La specifica delle funzioni parziali contiene sempre una clausola che indica il dominio sulla quale questa può essere invocata per ottenere un valore specificato; se le si passasse qualcosa fuori da quel dominio, restituirebbe un risultato ignoto.

Cosa si intende per *type safety* in Java? Un compilatore non solleva eccezioni, ma emette errori; è durante la fase di esecuzione che possono essere sollevate eccezioni. Quando vengono individuati tali errori, ed in che modo? Solo in fase di compilazione? Ci sono una serie di controlli che possono essere fatti durante la compilazione, e che riguardano tipi elementari e tipi di riferimento: per quest'ultimi, la situazione è più delicata a causa di tipi apparenti e concreti. Cosa si intende per *automatic storage management*? Si possono eliminare gli elementi di un vettore? No, la sua lunghezza è fissa. Java controlla che, se dichiaro un vettore di $ n $ elementi, vengano accedute solo le posizioni da $ 0 $ ad $ n - 1 $. La gestione automatica della memoria fa sì che un oggetto resti sullo heap finché nel codice è presente un riferimento al suo indirizzo. Se così non fosse, il *garbage collector* di Java libera la memoria associata all'oggetto in un momento non specificato dell'esecuzione. Questa caratteristica è assente in C, per esempio.

Cosa sono i tipi concreti ed i tipi apparenti? Chi può avere tipi concreti ed apparenti? A cosa ci riferiamo con queste espressioni? Il compilatore non esegue codice Java, ma lo compila. Ad eseguirlo c'è la macchina virtuale. Il compilatore non alloca memoria, ed è per questo che ci sono tipi concreti e tipi apparenti: il tipo apparente è quello che si determina durante la compilazione a partire da quelli dichiarati per le variabili. Perciò il tipo apparente è esattamente il tipo con cui le variabili sono dichiarate. Si può inoltre inferire con semplici regole nel caso di espressioni che coinvolgono costanti o variabili: per esempio,

[ ]:
```
Integer i = 1;
String s = "Ciao";

i = i + 1;
i = s.length();
i = i + 3.14;
```

`Integer i` ha per tipo apparente `Integer`; anche `i + 1` ed `s.length()` hanno per tipo apparente

`Integer`. `i + 3.14` ha per tipo apparente `Double`, grazie al casting implicito. Quindi, il compilatore determina i tipi delle espressioni e delle variabili in base alla dichiarazione di variabili ed alle segnature dei metodi. Tutto ciò senza bisogno di eseguire alcuna porzioni di codice o di allocare memoria. In un momento diverso, il programma viene posto in esecuzione e può accadere che le espressioni abbiano un tipo diverso da quello che il compilatore ha stabilito. Perché? Grazie al polimorfismo: in Java è infatti possibile assegnare ad una variabile un oggetto dello stesso tipo, oppure che sia un sottotipo di quello della variabile.

[ ]:
```
Number n = new Integer(3);
```

La variabile `n` ha per tipo apparente `Number`, e per tipo concreto `Integer`. Durante la fase di esecuzione, il tipo di un'espressione viene determinato in base ai tipi concreti.

[ ]:
```

```