

PARADIGMI DI PROGRAMMAZIONE

A.A. 2021/2022

Laurea triennale in Informatica

4: Istruzioni

ISTRUZIONI ED ESPRESSIONI

Coerentemente con la sua filosofia OOP, il codice in Java è contenuto all'interno di blocchi delimitati da parentesi graffe `{ }` ed è composto da sequenze di istruzioni (*statements*) separate dal carattere `;`.

Un blocco di codice può contenerne altri.

Come già avviene nei compilatori di altri linguaggi, non c'è nessuna garanzia che l'ordine di esecuzione delle istruzioni sia lo stesso del codice:

il compilatore ha grande libertà nel riorganizzare, riscrivere e in generale modificare il sorgente iniziale usando un insieme di trasformazioni che conservano la semantica esterna del blocco di codice.

DICHIARAZIONI

Una istruzione può essere una dichiarazione di un nome e del suo tipo.

Può essere dichiarata una variabile, o una classe, che sono dette *locali* rispetto al blocco che le contiene.

Una variabile viene dichiarata indicando tipo, nome ed eventualmente un valore con cui inizializzarla.

Usando la parola chiave `var`, si può lasciare al compilatore la determinazione del tipo della variabile. Indicando una lista vuota di parametri di tipo `<>` anche questa sarà dedotta (se possibile) dal compilatore.

```
int i = 1;  
var b = java.util.List.of(1, 2);  
Foo a = new Foo("bar");  
List<> l = java.util.List.of("A", "B");  
int[] r = new int[] { 1, 2, 3 };
```



```
class Global {  
  
    class AnotherLocal {  
        void bar() {  
            class Local {}  
            Local l = new Local();  
        }  
    }  
  
    {  
        class Local {}  
        Local a = new Local();  
    }  
}
```

ESPRESSIONI

Una istruzione può consistere in una *espressione*, cioè una sintassi che produce un valore, anche se questo non viene usato.

Le espressioni seguono abbastanza da vicino le regole di precedenza, struttura e semantica della maggior parte dei linguaggi di programmazione.

VALORI LETTERALI

Le espressioni possono contenere valori letterali per i principali tipi primitivi.

Tipo	Esempi
Interi	12, 45L, 0Xf, 077, 0b1111_0000
Decimali	0.0, 3.14f, 1.6e5, -0.5d, 1.0e-9d
Booleani	true, false
Caratteri	'a', 'b', '€', '\uffff'
Stringhe	"abcdef", "知識", "الود"

Non è possibile indicare una costante di larghezza byte o short senza un operatore di cast.

```
byte b = (byte)0xff;  
short s = (short)12800;
```

Le costanti di tipo `String` (e la stessa classe, del resto) hanno un trattamento speciale da parte del compilatore:

- sono oggetti `String`, senza bisogno dell'operatore `new`
- possono essere su più righe (da Java 13)
- come tutti gli oggetti `String`, sono immutabili

Una costante stringa può (ora) essere scritta su più righe (viene detto *text block*):

```
String square = """  
    SATOR  
    AREPO  
    TENET  
    OPERA  
    ROTAS""";
```


La parola chiave `null` indica il valore nullo, ovvero il riferimento che non punta a nessun oggetto.

Il valore nullo è l'unico elemento del tipo nullo, che non ha nome, non si può esprimere, e può essere convertito sempre in ogni altro oggetto.

ASSEGNAMENTO

L'assegnamento = è un operatore,
quindi una assegnazione è una espressione,
e quindi una istruzione.

Il valore dell'espressione è lo stesso assegnato alla
variabile. Il suo tipo dipende dal tipo della variabile
assegnata.

```
int x;  
x = (int)4.6;  
  
int k = 1;  
int[] a = { 1 };  
k += (k = 4) * (k + 2);  
a[0] += (a[0] = 4) * (a[0] + 2);
```

CHIAMATA DI UN METODO

L'esecuzione della chiamata di un metodo, se quest'ultimo ritorna un valore, è una espressione come le altre.

Se non ritorna un valore, è una istruzione a sè stante.

```
System.out.println("k==" + k + " and a[0]==" + a[0]);  
  
int time = System.currentTimeMillis();  
  
System.setProperty("KEY", "VALUE");
```

CREAZIONE DI UN OGGETTO

La creazione di un oggetto è, per alcuni versi, la chiamata di un metodo che ritorna il nuovo oggetto. E' quindi una espressione.

```
StringBuffer buf = new StringBuffer("text");  
  
String up = new StringBuffer("more text").toString()  
    .toUpperCase();
```

E' possibile istanziare direttamente una interfaccia, fornendo l'implementazione al momento della creazione:

```
Comparator< > reverse = new Comparator< String >() {  
  
    @Override  
    public int compare(String o1, String o2) {  
        return -o1.compareTo(o2);  
    }  
  
};
```

OPERATORI

Java supporta tutti i più comuni operatori aritmetici e logici, che solitamente si comportano senza sorprese.

L'operatore `+` è usato anche per la concatenazione di stringhe. L'accesso agli array si opera con le parentesi quadre `[]`.

OPERATORE TERNARIO

L'operatore

`< cond > ? < val1 > : < val2 >`

consente di assegnare uno di due valori, a seconda di una condizione. Solo l'espressione corrispondente al valore selezionato viene valutata.

THIS E SUPER

Le parole chiave `this` e `super` hanno un significato particolare.

`this` permette di indicare l'istanza corrente durante l'esecuzione di un metodo. Può essere utile per risolvere ambiguità di denominazione o per rendere più esplicito il significato di una espressione.

```
class Foo {  
    public final int idx;  
    public final String title;  
  
    public Foo(int idx, String title) {  
        this.idx = idx;  
        this.title = title;  
    }  
}
```

super indica l'oggetto padre nella gerarchia di ereditarietà. Permette (per esempio) di controllare il passaggio degli argomenti al costruttore della classe padre all'interno del costruttore della classe figlio.

```
class A {  
    public final int a;  
    public A(int a) { this.a = a; }  
}
```

```
class B extends A {  
    public final int b;  
    public B(int a, int b) {  
        super(a);  
        this.b = b;  
    }  
}
```

LAMBDA EXPRESSION

Una delle maggiori innovazioni di Java 8 è stata la sintassi della **lambda expression**.

Unita alla inclusione del linguaggio della pratica delle interfacce SAM e funzionali, ha reso alcuni casi d'uso molto comuni decisamente più semplici da scrivere.

La sintassi della **lambda expression** è la seguente:

```
( < lista parametri > ) -> istruzione
```

Il compilatore individua il tipo che è atteso nell'espressione in cui la lambda si trova; il risultato è una istanza di un oggetto che implementa tale tipo, con il comportamento dato dall'istruzione.

() -> 42

() -> { return 42; }

() -> { System.gc(); }

(int x) -> { return x+1; }

x -> x+1

(int x, int y) -> x+y

(x, y) -> x+y

Attenzione: la **lambda expression** non rende Java un linguaggio funzionale.

La **lambda expression** non ha un tipo proprio; è solo una sintassi breve per un caso d'uso molto comune. Il compilatore sostituisce il codice necessario per ottenere lo stesso risultato.


```
import java.util.function.Function;

Function< Integer, String > f = x -> "%d".formatted(x);

Function< > g = new Function< Integer, String > {
    String apply(Integer x) {
        return "%d".formatted(x)
    }
}
```

La combinazione di **lambda expression**, inferenza del tipo delle espressioni, `var` e `diamond operator` permettono di scrivere molti casi d'uso in modo assai più conciso e comprensibile.

Si può arrivare ad uno stile molto vicino ad alcuni linguaggi funzionali; questa possibilità è stata colta dovunque possibile nella libreria standard e da parte di alcune librerie di cui parleremo.

CONDIZIONALI

In Java i costrutti condizionali sono istruzioni, non espressioni.

Questo significa che eseguono blocchi di codice separati a seconda del valore della condizione.

IF-ELSE

L'istruzione

```
if ( < cond > ) < statement >  
    else < statement >;
```

ha la semplice struttura mutuata dal C.

Anche qui la condizione deve essere un'espressione booleana. Le due istruzioni non ritornano valore.

SWITCH-CASE

La selezione fra più valori è anch'essa mutuata dal C, con qualche estensione.

L'espressione di selezione può essere:

- un cosiddetto *tipo scalare*: byte, char, short, int, long
- un corrispettivo *boxed*: Byte, Character, Short, Integer, Long
- una costante stringa
- un valore di una Enumerazione

```
enum Days { LUN, MAR, MER, GIO, VEN, SAT, DOM }  
Days day = ...;
```

```
switch (day) {  
    case LUN:  
        System.out.println("Inizio settimana");  
        break;  
    case SAT, DOM:  
        System.out.println("Weekend!");  
        break;  
    default:  
        System.out.println("Nel mezzo...");  
        break;  
}
```



```
String day = ...;

switch (day) {
    case "Lun":
        System.out.println("Inizio settimana");
        break;
    case "Sab": case "Dom":
        System.out.println("Weekend!");
        break;
    default:
        System.out.println("Nel mezzo...");
        break;
}
```

ESPRESSIONE SWITCH

A partire da Java 14 è parte integrante del linguaggio la sintassi di switch *come espressione*, cioè in grado di ritornare un valore. La forma esteriore è molto simile, ma ci sono alcune fondamentali differenze.

Sono disponibili due sintassi:

```
enum Days { LUN, MAR, MER, GIO, VEN, SAT, DOM }  
Days day = ...;
```

```
String weekPart= switch (day) {  
    case LUN:  
        System.out.println("Inizio");  
        yield "Inizio settimana";  
    case SAT, DOM: {  
        System.out.println("Fine");  
        yield "Weekend!";  
    }  
    default:  
        yield "Nel mezzo...";  
}
```

La parola chiave `yield` è stata scelta per assonanza con altri linguaggi e per distinguersi meglio dall'uso come istruzione.

`yield` può essere preceduto da alcune istruzioni, o incluso in un blocco, o isolato.

Non c'è *fall-through*. L'elenco dei casi deve essere *esaustivo*.

```
enum Days { LUN, MAR, MER, GIO, VEN, SAT, DOM }  
Days day = ...;
```

```
String weekPart= switch (day) {  
    case LUN      -> "Inizio settimana";  
    case SAT, DOM -> {  
        System.out.println("Fine");  
        yield "Weekend!";  
    }  
    default      -> "Nel mezzo...";  
}
```

ITERAZIONI

Anche le istruzioni di iterazione sono decisamente ispirate alla lezione del C, con qualche peculiare modifica.

WHILE

L'istruzione `while` si comporta come da aspettative: finché la condizione valutata è vera, l'istruzione seguente viene eseguita.

```
int i = 0, sum = 0;  
while ( i < 100 )  
    sum += i++;
```


DO

Analogamente, l'istruzione do: l'istruzione viene eseguita, e ripetuta finché la condizione valutata è vera.

```
public static String toHexString(int i) {  
    StringBuffer buf = new StringBuffer(8);  
    do {  
        buf.append(Character.forDigit(i & 0xF, 16));  
        i >>>= 4;  
    } while (i != 0);  
    return buf.reverse().toString();  
}
```

FOR

Anche l'istruzione `for` mantiene la forma originaria: una espressione di inizializzazione, un test ed una di modifica, ed una istruzione da eseguire.

```
int[] vals = new int[] { 5, 4, 3, 2, 1 };  
for ( int i = 0; i < vals.length; i++ )  
    System.out.println(vals[i]);
```

L'istruzione `for` può anche essere usata per un ciclo su di un oggetto che implementa `Iterable` o un array:

```
int[] vals = new int[] { 5, 4, 3, 2, 1 };  
for ( int i: vals ) System.out.println(i);  
  
for ( String s: List.of("foo", "bar", "baz") )  
    System.out.println(s);
```

BREAK - CONTINUE

La parola chiave `break` permette di interrompere immediatamente il ciclo di iterazione più interno in corso, qualsiasi sia il suo tipo.

La parola chiave `continue` permette di interrompere l'esecuzione dell'iterazione corrente, per proseguire immediatamente con la successiva (se applicabile).

ETICHETTE

Ogni istruzione può essere preceduta da una etichetta; le istruzioni `break` e `continue` possono indirizzare una etichetta per riportare l'esecuzione alla istruzione indicata.

Se vi trovate ad usare questa sintassi, fermatevi a ripensare la vostra implementazione.

RETURN

L'istruzione `return` conclude la chiamata al metodo attuale, ritornando il controllo al codice chiamante.

Se il metodo ritorna un valore, è obbligatorio indicare il valore da ritornare; ogni percorso di codice all'interno del metodo **deve** terminare in una istruzione `return`.

Se il metodo è `void`, **non può** essere indicato un valore, e l'istruzione è opzionale.

ECCEZIONI

TRY-CATCH

Se un metodo dichiara la possibilità di lanciare un certo tipo di eccezione, il codice chiamante è obbligato dal compilatore a dichiarare la stessa eccezione oppure a gestirla all'interno di un blocco `try-catch`.

```
class FooException extends Exception {}
```

```
class Foo {  
    void a() throws FooException, BarException { return; }  
    void b() throws BarException {  
        try {  
            a();  
        } catch (FooException e) {  
            e.printStackTrace();  
        } finally {  
            System.out.println("Always");  
        }  
    }  
}
```

Il blocco di codice introdotto dall'istruzione `try` viene eseguito; se viene lanciata una eccezione, il primo blocco `catch` adatto viene eseguito; infine, il blocco `finally` viene eseguito in qualsiasi caso.

TRY-WITH-RESOURCES

Un'altra forma dell'istruzione `try`, detta `try-with-resources` permette di dichiarare delle variabili: queste devono implementare l'interfaccia `AutoClosable`, e verranno automaticamente , e con certezza, "chiuse".

In questa forma le clausole `catch` e `finally` sono entrambe opzionali.

```
static String readFirstLine(String path)
    throws IOException {
    try (BufferedReader br =
        new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

THROW

Per lanciare eccezioni destinate ad essere catturate da una istruzione `try`, è disponibile l'istruzione `throw`. Richiede un oggetto discendente da `Exception` che viene lanciato come se fosse avvenuto un errore.

```
class FooException extends Exception {}

class Foo {
    void a() throws FooException, BarException {
        throw new FooException();
    }
    void b() throws BarException {
        try { a();
        } catch (FooException e) { e.printStackTrace();
        } finally {
            System.out.println("Always");
        }
    }
}
```


ALTRE ISTRUZIONI

L'istruzione vuota, costituita da un solo `;`, è valida, anche se alcuni IDE la segnalano con un avviso.

Analogamente per un blocco vuoto `{ }`; nel caso sia il blocco di un `catch`, l'avviso è più invadente.

La parola chiave `assert` consente di verificare delle condizioni al momento dell'esecuzione del programma. Se l'espressione da verificare ritorna `false`, viene lanciato un errore:

```
assert !importantList.isEmpty();
```

La parola chiave `synchronized` davanti ad un blocco o ad un metodo cambia il suo comportamento rispetto alla concorrenza. Ne parleremo diffusamente a tempo debito.