Programmazione ad Oggetti

Capitolo 0: Sintassi del linguaggio Java

A.A. 2012/2013

laura.bacci@unipv.it

Elementi lessicali del linguaggio

- PROGRAMMA: sequenza di caratteri contenuta in un file (codice sorgente)
- I gruppi di caratteri (token o simboli) che costituiscono gli elementi lessicali del linguaggio Java sono:
  - Parole chiave
  - Caratteri di punteggiatura
  - Operatori
  - Nomi di variabili
  - Costanti letterali
- Nel testo ci sono anche
  - Spazi bianchi
  - Commenti
- Il compilatore esamina gli elementi elencati al primo punto tralasciando spazi e commenti

I commenti

- Ci sono tre tipi di commenti:
  - `//commento` (vengono ignorati i caratteri da // sino alla fine della linea)
  - `/* commento*/` (vengono ignorati tutti i caratteri compresi tra /* e il successivo */)
  - `/** commento*/` (vengono ignorati tutti i caratteri compresi tra /** e il successivo */)
- L'ultimo tipo rappresenta i "commenti di documentazione" e vengono elaborati da un tool che permette di generare automaticamente la documentazione (Javadoc)

Identificatori e insieme di caratteri

- Gli identificatori sono utilizzati per i nomi di entità dichiarate
  - Variabili
  - Costanti
  - Etichette
- Devono iniziare con una lettera
- Possono proseguire con lettere, cifre, simboli, connettivi di punteggiatura (_) mescolati in vario modo
- Possono avere lunghezza qualsiasi
- Sono scritti in UNICODE
- UNICODE: insieme di caratteri a 16-bit. I primi 256 coincidono con ISO-Latin1. I primi 128 sono invece corrispondenti alla codifica ASCII a 7 bit.
- Per codificare caratteri UNICODE si usa la sequenza di escape \uxxxx dove x è una cifra esadecimale (0-F).

Parole chiave

- Non possono essere utilizzate in qualità di identificatori
- * riservate ma non utilizzate

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extend	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const *	for	new	switch	
continue	goto*	pakage	syncronized	

`null true false` non compaiono perché sono costanti letterali

Tipi

- Due categorie di tipi
  - Tipi primitivi
  - Tipi riferimento
- I tipi riferimento sono costituiti dai tipi delle classi, interfacce ed array.
- Ogni tipo è dotato di *costanti letterali (literals o senza nome)*, le quali costituiscono il modo in cui possono essere scritti i valori costanti per il tipo specifico.

Tipi di dati: primitivi (1)

A differenza di altri linguaggi (Smalltalk), Java non è un linguaggio **OO puro**: infatti comprende dati che non sono oggetti. In Java ci sono 9 tipi di dati predefiniti (**primitivi**):

Tipi Primitivi	Dim (bit)	Min Val	Max Val	Tipo "Wrapper"
boolean	--	--	--	Boolean
char	16	Unicode 0	Unicode 2E16-1	Char
byte	8	-128	127	Byte
short	16	-2^{15}=-32768	$2^{15}-1$=32767	Short
int	32	-2^{31}	$2^{31}-1$	Integer
long	64	-2^{63}	$2^{63}-1$	Long
float	32	-3.4E38	3.4E38	Float
double	64	-1.797E308	-1.797E308	Double
void	--	--	--	Void

La ragione per avere tipi primitivi è che creare un oggetto con il costrutto **new** – in particolare una semplice variabile – non è molto efficiente in quanto **new** dispone gli oggetti che crea nell'heap. Per questi tipi Java segue un approccio tipico del C e del C++: invece di creare la variabile con il costruttore **new**, viene creata una variabile "automatica" che non è una referenza ad un oggetto. La variabile contiene il valore ed è posta nello stack così che il suo utilizzo risulta molto più efficiente. Java definisce la dimensione di ogni tipo primitivo. Le dimensioni non cambiano al cambiare dell'architettura della macchina (come nella maggior parte dei linguaggi): questa invarianza è una delle ragioni della grande portabilità dei programmi Java. In Java tutti i tipi numerici sono con segno.

Tipi di dati: primitivi (2)

La dimensione del tipo **boolean** non è esplicitamente definita; la specifica è che esso sia in grado di contenere i valori letterali **true** o **false**.
Esiste inoltre la classe "wrapper" (involucro) relativa a ciascun tipo primitivo: in tal modo se si vuole creare un oggetto (collocato nell'heap) per rappresentare un tipo primitivo, si istanzia un oggetto della classe "wrapper" relativa al tipo primitivo che si deve rappresentare.
Esempio:

```
char c = 'x';
Character C = new Character(c);
```

O più brevemente:

```
Character C = new Character('x');
```

La ragione per fare quest'operazione verrà chiarita in seguito.

Costanti letterali

- Tipo intero
  - Stringhe costituite da cifre ottali, decimali o esadecimali
    - Ottale: con cifra 0 iniziale (ad es. 011)
    - Esadecimale: con cifre iniziali 0x (ad es. 0xFF)
    - Decimale: ogni altro insieme di cifre
  - Esempio: 29 035 0x1D 0X1d rappresentano lo stesso valore
  - Le costanti che terminano per L o l sono considerate `long` (29L)
  - Le costanti intere senza suffisso sono considerate `int`

Costanti letterali

- Tipo virgola mobile
  - Espresse come numeri decimali seguiti dal punto (opzionale) e dall'esponente (opzionale)
  - Esempio: 18. 1.8e1 .18E2 rappresentano lo stesso valore
  - Sono di tipo double a meno che venga specificato il suffisso F o f (18.0f)
  - Due zeri: positivo 0.0 e negativo –0.0

Costanti letterali numeriche: sommario e esempi

Costanti numeriche

- Interi:
  - Decimale: 31
  - Specifica in ottale: prefisso **0** → 037 (=31 decimale)
  - Specifica in esadecimale: prefisso **0x** → 0x1F (=31 decimale)
- Numeri in virgola mobile: **3.1415, 35.78e2, …**
- Le costanti intere sono di tipo **int**
- Le costanti in virgola mobile sono di tipo **double,** ma non **float**

Suggerimento: usare solo **int** e **double**

Esempi

```
short x = 256;          // Assegnazione corretta
short x = 3565567;      // possible loss of precision found: int required: short
long x = 44444442556;   // integer number too large: 44444442556
long x = 444444442556l; // Assegnazione corretta; "l" finale indica costante long
float y =  3.1;         //possible loss of precision found: double required: float
float y =  3.1f;         // Assegnazione corretta; “f" finale indica costante float
float y =  (float) 3.1; // Assegnazione corretta; l'operatore di cast converte la
                                        costante in float
```


Costanti letterali

- Tipo carattere
  - Appaiono all’interno di apici singoli come ‘a’
  - Ogni carattere UNICODE può essere una costante letterale tipo carattere.
- Caratteri speciali e sequenze di escape.
  - \n newline
  - \t tab
  - \b backspace
  - \r return
  - \f form feed
  - \\ lo stesso backslash
  - \’ apice singolo
  - \” apice doppio

Costanti letterali

- Tipo booleano
  - Sono solo `true` e `false`
- Tipo riferimento
  - Solo `null` e rappresenta un oggetto non ancora creato o non valido che non fa parte di nessuna classe pur potendo essere assegnato a qualsiasi variabile riferimento
- Tipo classe
  - Per ogni tipo è previsto un oggetto Class associato cui ci si può riferire in modo diretto facendo seguire al nome del tipo il suffisso "`.class`"
  - Esempi.
    - String.class
    - java.lang.String.class
    - boolean.class

Costanti letterali

- Tipo stringa
  - Appaiono all'interno di doppi apici come "ciao"
  - Possono contenere qualsiasi carattere (tranne newline a meno di usare la sequenza di escape \n)
  - Fanno riferimento a oggetti di tipo String

Variabili

- Possono essere di:
  - tipo primitivo
  - tipo riferimento
- Si usano per:
  - ***Attributi*** (membri di classi o interfacce)
  - ***Variabili locali*** (ovunque all'interno di blocchi di codice)
  - ***Parametri*** (i parametri dichiarati per metodi, costruttori e blocchi catch)

Variabili

- Dichiarazione di attributi e variabili locali:
  - <modificatori> <tipo> <identificatore>
- Dichiarazione e inizializzazione di attributi e variabili locali:
  - <modificatori> <tipo><identificatore>=<inizializzazione>
    - <modificatori> è opzionale
    - <identificatore> può essere una lista di identificatori
- I modificatori possono essere:
  - Modificatori di accesso
  - Modificatori di proprietà
  - L'unico <modificatore> permesso per una variabile locale è <final>

Variabili

- Le variabili attributo hanno valori di inizializzazione di default:
  - Per i tipi primitivi:

Primitive type	Default
boolean	false
char	'\u0000' (null)
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0.0f
double	0.0d

  - Per i tipi riferimento è "`null`"

- ***Le variabili locali non hanno inizializzazione di default: prima di usarle bisogna inizializzarle***

Variabili

- Le variabili parametro si usano come:
  - Parametri di metodi
  - Parametri di costruttori
  - Parametri nei blocchi catch
- Dichiarazione di parametri
  - <modificatore> <tipo> <identificatore>
    - <modificatore> è opzionale e può essere solo <final>
- Non hanno inizializzazione esplicita, ma sono inizializzate implicitamente al valore dell'argomento passato all'invocazione del metodo, costruttore,..

Le variabili final o "costanti"

- Sono le variabili precedute dal modificatore <final>
- Il valore di una variabile final viene assegnato una sola volta e da quel momento in poi resterà sempre lo stesso (immutabile). In questo senso possono essere considerate delle "costanti"
- Prima di essere usate devono essere inizializzate

Variabili e Costanti: sommario ed esempi

I nomi di variabili sono **identificatori.**
Una dichiarazione di variabile può avere una delle seguenti forme:

1.<tipo> <nome>;
2.<tipo> <nome1>,<nome2>,...;
3.<tipo> <nome> = <espressione>;

Esempi:

```
int  cont;
int  i, j, k;
boolean  b=3>4;
boolean  b = false;
char  c = 's';
```

Si può associare a un identificatore un valore che non può essere cambiato nel programma utilizzando la seguente sintassi:

static final ***<tipo> <nome> = <valore>;***

Esempi:

```
static final double e = 2.71828;
static final int codice = 12;
static final char delimiter ='|';
```

Operatori

Un ***operatore*** è un simbolo che indica al linguaggio di eseguire un'operazione su uno o più operandi.

Operando è l'elemento su cui agisce l'operatore (in genere gli operandi sono espressioni).

Gli operatori si dividono in varie categorie:

- Operatore di assegnamento
- Operatori aritmetici (unari e binari)
- Operatori relazionali
- Operatori logici
- Operatore ? (unico caso di operatore ternario)

Operatore di assegnamento

Sintassi :
<variabile> = <espressione>;
I tipi devono essere compatibili.

```
int y;
double x;
y = 3 + 4 * 2;
 /* Assegnazione corretta */
x = y;
 /* Errore di compilazione */
y = x;
```

Java è un linguaggio con un controllo stretto dei tipi

```
int x, y, z ;
/* Pone a 0 le tre variabili */
x = y = z = 0;
/* Nota: associatività da destra a sinistra */
```

Un comando di assegnamento è anche un'espressione: restituisce il valore assegnato.

Operatori aritmetici (1)

Operatori aritmetici in ordine decrescente di priorità. Associano a sinistra.
Tutti gli operatori, se applicati a interi, restituiscono un valore **int** o **long**, mai **byte** o **short**.

Negazione	-	unario
Incremento	++	unario
Decremento	--	unario
Moltiplicazione	*	binario
Divisione	/	binario
Modulo	%	binario
Addizione	+	binario
Sottrazione	-	binario

Operatori aritmetici (2)

Divisione:

```
double n;
n = 16.0 / 5.0; // n = 3.2
n = 16 / 5.0;   // n = 3.2
n = 16.0 / 5;   // n = 3.2
n = 16 / 5;     // n = 3 (Divisione intera)
```

Modulo (= resto della divisione intera):

```
dividendo % divisore
int resto = 16 % 5 // resto = 1
```

Overflow e underflow

- Non dipendono dalla macchina
- Alcune conversioni sono automatiche (come da **int** a **long**), altre no
- L'Overflow viene gestito a livello del linguaggio con il meccanismo delle **eccezioni**

Operatori aritmetici (3)

Incremento e decremento:
++ e –- sono operatori unari che rispettivamente aggiungono o tolgono 1 all'operando a cui sono applicati.
Possono essere usati sia in notazione *prefissa* che *postfissa* ed in entrambi i casi l'effetto è l'incremento della valore dell'operando a cui sono applicati.
Attenzione:

```
int x,y,n=5;
x = ++n; /* prima incrementa n e poi assegna n a x -> x=6*/
y = n++; /* prima assegna n a a y e poi incrementa n -> y=5*/
```

In ogni contesto in cui ci sia solo un incremento la notazione prefissa e postfissa sono del tutto equivalenti.

Operatori di assegnamento compatti

Si possono creare degli operatori di assegnamento compatti utilizzando i 5 operatori matematici binari secondo la seguente sintassi :

<variabile> <operatore>= <espressione>;
equivalente a
<variabile> = <espressione><operatore><espressione>;

Alcuni esempi di operatori di assegnamento compatti e le loro espressioni equivalenti:

x*=y	x = x * y
y -= z + 1	y = y – z + 1
a /= b	a = a / b
x+= y / z	x = x + y / z
y %= 3	y = y % 3

Operatori sui bit

Operatore	Significato	Esempi

>>	"shift" aritmetico a livello di bit - destra	`Int x = 45;`
		`int y = x >> 2; //y = 11`
		`x = -45;`
		`y = x >> 2;  // y = -12`

<<	shift - sinistra	

>>>	**shift logico – destra** (senza segno)	`x = 45;`
		`y = x >>> 2; //y = 11`
		`x = -45;`
		`y = x >>> 2;  // y = 1073741812`

~	**negazione logica a livello di bit**	`~74  //  74 = 0100 1010` `// -75 = 1011 0101`
&	**AND a livello di bit**	x = 13 & 8 // 13 = 0000 1101, 8 = 0000 1000 // x = 8 (0000 1000)
|	**OR a livello di bit**	x = 13 | 2 // 13 = 0000 1101, 2 = 0000 0010 // x = 15 (0000 1111)
^	**XOR a livello di bit**	x = 13^10 // 13 = 0000 1101, 10 = 0000 1010 // x = 7 (0000 0111)

Caratteri

- In C e C++, il tipo di dati **char** usa 8 bit, e usa il codice ASCII (American Standard Code for Information Interchange) per la codifica dei caratteri.
- ASCII usa solo 7 bit (2^7=128 caratteri in tutto). I rimanenti 128 valori rappresentabili con 8 bit sono usati per rappresentare altri caratteri (come lettere con accenti), *con codifiche diverse in sistemi/linguaggi diversi!!*
- In Java, il tipo di dati **char** usa 16 bit, e usa il codice UNICODE, che permette di rappresentare 2^{16}=65535 caratteri
- **char** usa lo stesso numero di byte (2) di **short**, però non usa il segno (***unsigned***)
- Le costanti di tipo carattere si scrivono come **'0'** oppure **\u0000** (zero in UNICODE)

Booleani

- In molti linguaggi (come C e C++) non esiste un tipo di dati **booleano:** si usano interi con opportune convenzioni
- In Java esiste il tipo primitivo boolean
- Sono di tipo boolean:
  - le costanti true e false (gli unici valori)
  - il risultato di espressioni con operatori booleani
  - il risultato di qualunque confronto tra valori

Operatori Booleani (1)

Operatori Booleani in Java in ordine decrescente di precedenza

Tabella di Verità		NOT	AND	XOR	OR
x	y	!x	x & y x && y	x ^ y	x | y x || y
False	False	True	False	False	False
True	False	False	False	True	True
False	True	True	False	True	True
True	True	False	True	False	True

Operatori Booleani (2)

Differenze tra & e && (AND), e tra | e || (OR):

&& e || sono operatori *lazy* (*pigri*) o *short-cut,* ossia valutano il secondo argomento solo se è necessario.

`c && d`	d viene valutato solo se c vale true
`c || d`	d viene valutato solo se c vale false
`(a!=0) & (b/a) >100`	b/a viene valutato sempre, anche se a==0, causando in questo caso un errore
`(a!=0) && (b/a) >100`	Se a==0 , b/a non viene valutato e la condizione vale false

Operatori di Confronto

>	>=	<	<=	==	!=

Per l'uguaglianza si usa '==', e non '=' (assegnamento) Applicati alle espressioni numeriche (int, double, ...) hanno il significato ovvio (attenzione alla verifica di uguaglianza tra due valori double che fallisce a causa delle approssimazioni).
Gli operatori == e != possono essere applicati a qualunque tipo di dato

```
int x = 10;
int y = 20;
boolean b = x < y;          //b true
boolean b1 = x >= y == b;   //b1 false
double r = Math.sqrt(2);
boolean b2 = 2.0 == r * r;  //b2 false
```

Gli operatori di confronto hanno precedenza su quelli logici:

```
int a, x;
boolean b = false;
a = 0;
x = 1;
boolean c = a != 0 & x > 0 == b  //c false
/* l'espressione viene letta come (a != 0) & ((x > 0) == b).
   Notare che  ((a != 0) & (x > 0)) == b vale true */
```

Operazioni di confronto

In molti programmi o algoritmi, l'esecuzione di uno o più passi può dipendere da certe condizioni. Per controllare l'esecuzione di questi passi si può usare l'istruzione condizionale **if-else**. Vediamone alcuni esempi:

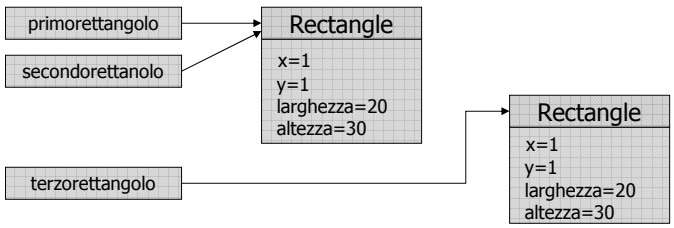
```
if (x == 0) System.out.println("x vale zero");

if (x > y) System.out.println("x e' maggiore di y");
   else System.out.println("y e' maggiore di o uguale a x");

if (x > y) System.out.println("x e' maggiore di y");
   else  if (x < y) System.out.println("y e' maggiore di x");
         else    System.out.println("y e y sono uguali");

if ((anno % 4 == 0 && anno % 100 != 0 ) || anno%400 == 0 )
   System.out.println ("Anno bisestile!");
else
   System.out.println ("Anno non bisestile!");
```


Confronto tra oggetti

Si possono confrontare gli oggetti di una classe con **"=="** oppure con il metodo **equals**.

- (obj1 == obj2) vale **true** solo se **obj1** e **obj2** sono (puntatori al) lo stesso oggetto
- obj1.equals(obj2) *normalmente* vale **true** se i due oggetti hanno "la stessa struttura"

```
Rectangle primorettangolo = new Rectangle(1,1,20,30);
Rectangle secondorettangolo = primorettangolo;
Rectangle terzorettangolo = new Rectangle(1,1,20,30);
boolean b = primorettangolo == secondorettangolo;      // b =true
boolean b1 = primorettangolo == terzorettangolo;       // b1=false
boolean b2 = primorettangolo .equals(terzorettangolo); // b2=true
```



Istruzioni e strutture di controllo

- Come in tutti i linguaggi di programmazione procedurali (e a differenza dei linguaggi dichiarativi) in Java il programmatore deve descrivere in modo dettagliato i passi dell'algoritmo progettato per la risoluzione di un problema. Per controllare il flusso di esecuzione, Java fornisce:
  - **istruzioni (comandi) semplici**
    - assegnamento
    - chiamata di metodo
  - **blocchi di istruzioni**
  - **istruzioni condizionali (decisionali, di scelta)**
    - if-else
    - switch
  - **istruzioni iterative (cicli)**
    - while
    - for
    - do-while
  - **istruzioni di salto**
    - break
    - continue
    - return
    - throw

Istruzioni semplici

- Due tipi principali:
  - Istruzioni di espressione
    - Istruzione contenenti operazioni di assegnamento
    - Istruzioni contenenti forme prefisse e postfisse degli operatori di auto-incremento/decremento (++, --)
    - Invocazioni di metodi
    - Espressioni di creazione di oggetti attraverso **new**
  - Istruzioni di dichiarazione
    - Dichiarazioni di variabile
    - Dichiarazioni di classi locali
  - NOTA: Tutte le istruzioni terminano con **;**

Blocchi di istruzioni

Un insieme di istruzioni e dichiarazioni racchiuso tra parentesi graffe viene detto **blocco di istruzioni** (enunciati, comandi).

In Java, di norma, un'istruzione può essere un'istruzione semplice, un'istruzione composta (iterativa, condizionale) oppure un blocco di istruzioni.

- I blocchi possono essere annidati: all'interno di un blocco può esserci un'altro blocco.
- Le variabili dichiarate in un blocco sono visibili solo all'interno del blocco stesso (variabili locali).
- I loro identificatori non possono essere uguali a identificatori di variabili esterne al blocco.

```
int e = 2;
  {
    int l = 4;
    int e = 5; // non valido: già definito fuori dal blocco
    System.out.println("l+e =" + (l+e));
  }
  System.out.println("l =" + l) // non valido: la variabile non è
                                  definita fuori dal blocco
```


Sintassi e semantica dell' *if* e *if-else*

if (*<condizione>* **)**
  <istruzione-1>

oppure

if (*<condizione>* **)**
  <istruzione-1>
else
  <istruzione-2>

Dove:

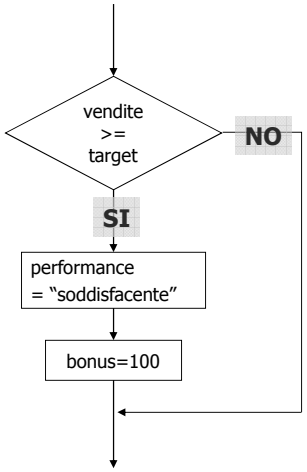
- <*condizione*> è un'espressione di tipo booleano
- <*istruzione-1*> e <*istruzione-2*> possono essere:
  - istruzioni semplici
  - istruzione composte
  - blocchi di istruzioni

L'istruzione **if** valuta la *<condizione>*

- Se la valutazione ritorna il valore **true** esegue *<istruzione-1>.*
- Se la valutazione ritorna il valore **false** esegue *<istruzione-2>* se presente, altrimenti passa al punto successivo.

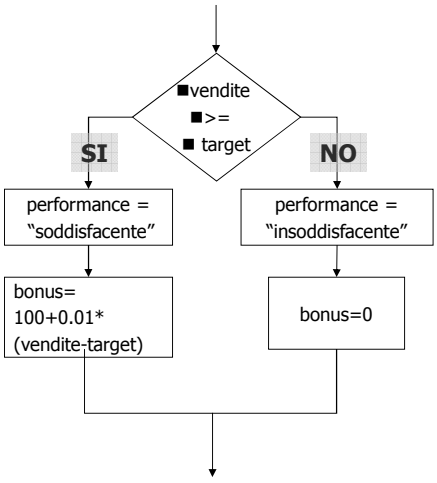
- L'esecuzione passa all'istruzione successiva al comando **if**.
- Le istruzioni **if** possono essere annidate: in questo caso, per evitare ambiguità, ogni **else** si riferisce al primo **if** che lo precede.

Esempi: *if* e *if-else*



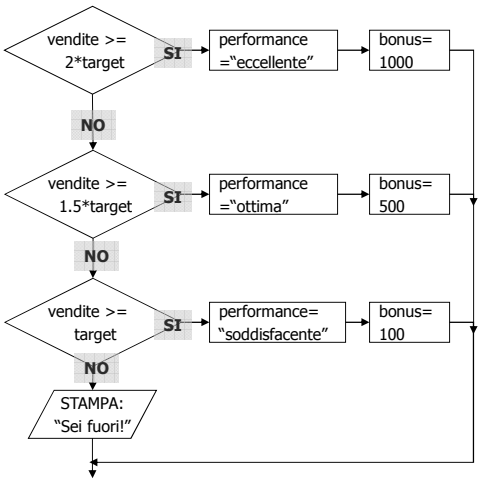

```
if (vendite >= target)
{
   performance="soddisfacente";
   bonus=100;
}
```


Esempi: *if* e *if-else*




```
if (vendite >= target)
{
   performance="soddisfacente";
   bonus=100+
     0.01*(vendite-target);
}
else
{
   performance="insoddisfacente";
   bonus=0;
}
```

Esempi: *if* e *if-else*




```
if (vendite >= 2*target)
{
   performance="eccellente";
   bonus=1000;
}
else if (vendite >= 1.5*target)
{
   performance="ottima";
   bonus=500;
}
else if (vendite >= *target)
{
   performance="soddisfacente";
   bonus=100;
}
else
{
   System.out.println(
      "Sei fuori!");
}
```


Sintassi e semantica dello *switch*

```
switch ( <selettore> )
   {
   case <valore1> :
      <blocco istruzioni1>
   case <valore2> :
      <blocco istruzioni2>
   …
   default:
      <blocco istruzioni N>
   }
```

- <*selettore*> è un'espressione di tipo char, byte, short, o int
- <*valore-i*> è un'espressione costante dello stesso tipo del <*selettore*>.
- Ogni <*valore-i*> deve essere unico.
- <*istruzioni-i*> possono essere:
  - sequenze di istruzioni e dichiarazioni di variabili
  - blocchi di istruzioni
- **default** può essere presente una sola volta
- Quello che segue la parte (<*selettore*>) deve essere un blocco, che può essere vuoto o iniziare con **case** o **default**.

- L'istruzione **switch** valuta il *<selettore>* ed esegue il blocco sequenzialmente
- Se incontra dopo una etichetta **case** un *<valore-i>* uguale al risultato della valutazione del selettore, esegue tutte le *<istruzioni-i>*,… *<istruzioni-N>* successive **.**
- Se non incontra un *<valore-i>* uguale alla risultato della valutazione ed è presente l'etichetta **default** esegue le *<istruzioni-N>* corrispondenti.
- Se non incontra un *<valore-i>* uguale alla valutazione e non è presente l'etichetta **default** passa al punto successivo.
- L'esecuzione passa all'istruzione successiva allo **switch**, dopo il blocco.
- Anche le istruzioni **switch** possono essere annidate: *<istruzioni-i>* può contenere uno **switch**.

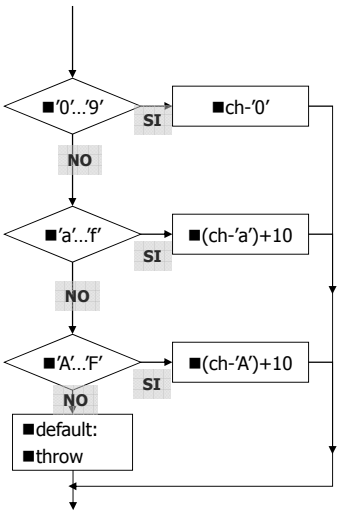
"Uscire" dallo switch: break

- Per evitare che vengano valutate tutte le *<istruzioni-(i+1)>... <istruzioni-N>* successive ad un case soddisfatto si utilizza l'istruzione **break**.
- Il **break** permette di trasferire l'esecuzione all'istruzione successiva al blocco dello **switch**; viene posto successivamente alle *<istruzioni-i>.*
- L'istruzione **switch** si utilizza quindi (solitamente) nella forma:

```
switch ( <selettore> )
        {
          case <valore1> : <istruzioni1> break;
          case <valore2> : <istruzioni2> break;
          ...
          default: <istruzioniN> break;
        }
```

- Nota: Si può uscire anche con un return o un throw

Esempi: *switch*




```
switch(ch)
{
   case '0': case '1': case '2':
   case '3': case '4': case '5':
   case '6': case '7': case '8':
   case '9':
   return (ch-'0');

   case 'a': case 'b': case 'c':
   case 'd': case 'e': case 'f':
   return (ch-'a')+10;

   case 'A': case 'B': case 'C':
   case 'D': case 'E': case 'F':
   return (ch-'A')+10;

   default:
   throw …;
     // carattere non
        esadecimale
}
```

Espressioni condizionali

Sono una eredità di C e C++

- Sintassi: <*condizione*> **?** <*espressione1*> **:** <*espressione2*>
- Semantica: E' una espressione, non un comando.
  - se la <*condizione*> vale true, il risultato è il valore di <*espressione1*>
  - se la <*condizione*> vale false, il risultato è il valore di <*espressione2*>

Esempio:

```
int x = 0;
int y = (x != 0) ? 5/x : 5;

(x > 3)? x++ : x--;     // non valido: l'espressione non è
                        assegnata a nessuna variabile
```


Istruzioni iterative

- Spesso un algoritmo per la risoluzione di un problema richiede di eseguire più volte una sequenza di passi elementari (si pensi ad esempio al calcolo del fattoriale di un numero fornito in input dall'utente).
- Le istruzioni iterative (ripetizioni) consentono di ripetere una sequenza di istruzioni in base a certe condizioni o per un numero definito di volte.
- In particolare Java, fornisce le seguenti istruzioni composte:
  - **while**
  - **for**
  - **do … while**
- L'istruzione **while** viene usata quando uno o più comandi devono essere ripetuti ciclicamente fino a quando una certa condizione non risulta falsa.
- L'istruzione **for** viene usata, di norma, quando uno o più comandi devono essere ripetuti un numero prefissato di volte.
- L'istruzione **do…while** viene usata al posto del **while** quando i comandi devono essere eseguiti almeno una volta.

Sintassi e semantica del *while*

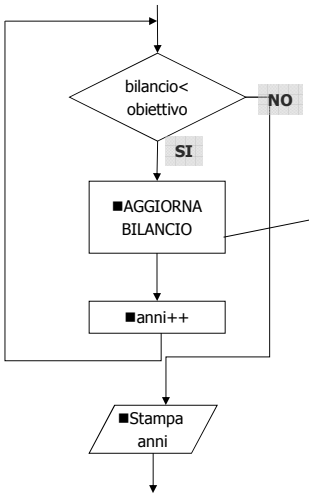
while (*<condizione>* **)**
<corpo>

dove:

- *<condizione>* è un'espressione di tipo booleano, chiamata anche la guardia
- *<corpo>* può essere:
  - un' istruzione semplice
  - un'istruzione composta
  - un blocco di istruzioni

- L'istruzione **while** valuta inizialmente l'espressione booleana **<condizione>**.
  - Se la valutazione ritorna il valore true viene eseguito il **<corpo>**, al termine del quale si riesegue tutto il comando **while**.
  - Se la valutazione ritorna il valore false l'esecuzione passa all'istruzione successiva al **while**.
- Normalmente, la guardia del **while** controlla il valore di una o più variabili, chiamate variabili di controllo del ciclo. Le variabili di controllo vanno inizializzate opportunamente prima del **while**, e devono essere aggiornate nel corpo per garantire la terminazione del ciclo.
- Le istruzioni **while** possono essere annidate.

Esempi: while




```
while (bilancio<obiettivo)
{
   bilancio+=pagamento;
   double interesse=
          bilancio*rateo/100;
   bilancio+=interesse;
   anni++;
}
```

Sintassi e semantica del *for*

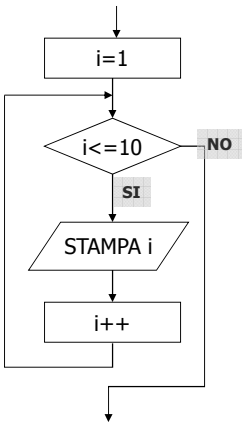
for (*<inizializzazione>***;***<condizione>***;***<aggiornamento>* **)**
<corpo>

dove:

- *<inizializzazione>* è di solito un comando di assegnamento, che assegna un valore iniziale alla variabile di controllo. La variabile può essere dichiarata prima del **for**, o in *<inizializzazione>.*
- *<condizione>* è un'espressione di tipo boolean
- *<aggiornamento>* è di solito un comando che aggiorna la variabile di controllo (ad esempio, la incrementa di 1).
- *<corpo>* può essere:
  - un'istruzione semplice
  - un'istruzione composta
  - un blocco di istruzioni
- Sia *<inizializzazione>* che *<aggiornamento>* possono essere, in generale, liste di istruzioni separate da virgole

- L'istruzione **for** esegue inizialmente i comandi *<inizializzazione>,* dove solitamente sono inizializzate le variabili di controllo.
- Successivamente valuta l'espressione booleana *<condizione>* che di solito contiene condizioni sulle variabili di ciclo.
- Se *<condizione>* ha valore true viene eseguito il comando <corpo>. Quindi si eseguono i comandi *<aggiornamento>* che di solito modificano le variabili di controllo; si torna poi al passo precedente.
- Se *<condizione>* ritorna il valore false, l'esecuzione passa all'istruzione successiva al **for**.

Esempi: *for*




```
for (int i=1;i<=10;i++)
   System.out.println(i);
```

Relazione tra *while* e *for* (1)

- Una istruzione for può essere vista come una istruzione while generalizzata:
  - **while (<condizione>) <istruzione>**
    e' del tutto equivalente a
  - **for (; <condizione> ;) <istruzione>**

- Analogamente,
  - **while (true) <istruzione>**
    e' equivalente a
  - **for (; ;) <istruzione>**

Relazione tra *while* e *for* (2)

- Simmetricamente, un comando **for** può sempre essere sostituito con un **while.**

L'istruzione:

```
for (int i= 1; i<= 20; i++)
   { double interesse = saldo * tasso / 100;
     saldo = saldo + interesse;
   }
```

è equivalente a:

```
int i= 1;
while (i =< 20)
     { double interesse = saldo * tasso /
100;
       saldo = saldo + interesse;
       i++;
     }
```

Relazione tra *while* e *for* (3)

- Poiché **for** e **while** sono equivalenti, la scelta tra i due è essenzialmente una questione di stile di programmazione. E' buona prassi scegliere:
  - **for**
    - Se il numero di volte che bisogna ripetere il <corpo> e' conosciuto nel momento in cui inizia l'esecuzione del comando.
  - **while**
    - In tutti gli altri casi, ad esempio quando il numero di volte che bisogna ripetere il <corpo> dipende da un valore letto all'interno del <corpo>

Sintassi e semantica del *do…while*

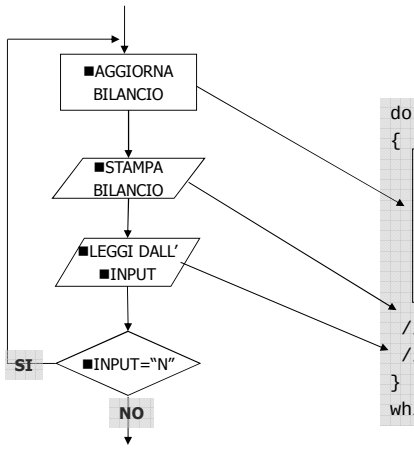
```
do
    <corpo>
while <condizione> ;
```

dove:

- *<condizione>* è un'espressione di tipo booleano
- *<corpo>* **deve sempre essere un blocco, non puo' essere una istruzione semplice o composta**. Quindi le parentesi graffe non si possono omettere, anche se c'e' solo una istruzione.

- L'istruzione **do...while** esegue inizialmente le istruzioni del *<corpo>.*
- Successivamente valuta la *<condizione>.*
  - Se la valutazione ritorna il valore true si torna al passo iniziale.
  - Se la valutazione ritorna il valore false l'esecuzione passa all'istruzione successiva al comando **do...while**.
- Le variabili di controllo vanno inizializzate prima del **do...while**, e aggiornate nelle istruzioni per permettere la terminazione del ciclo.

Esempi: do…while




```
do
{
   bilancio+=pagamento;
   double interesse=
          bilancio*rateo/100;
   bilancio+=interesse;
   anni++;
  // stampa bilancio corrente;
  // pone domanda e prende input;
}
while (input.equals("N"));
```