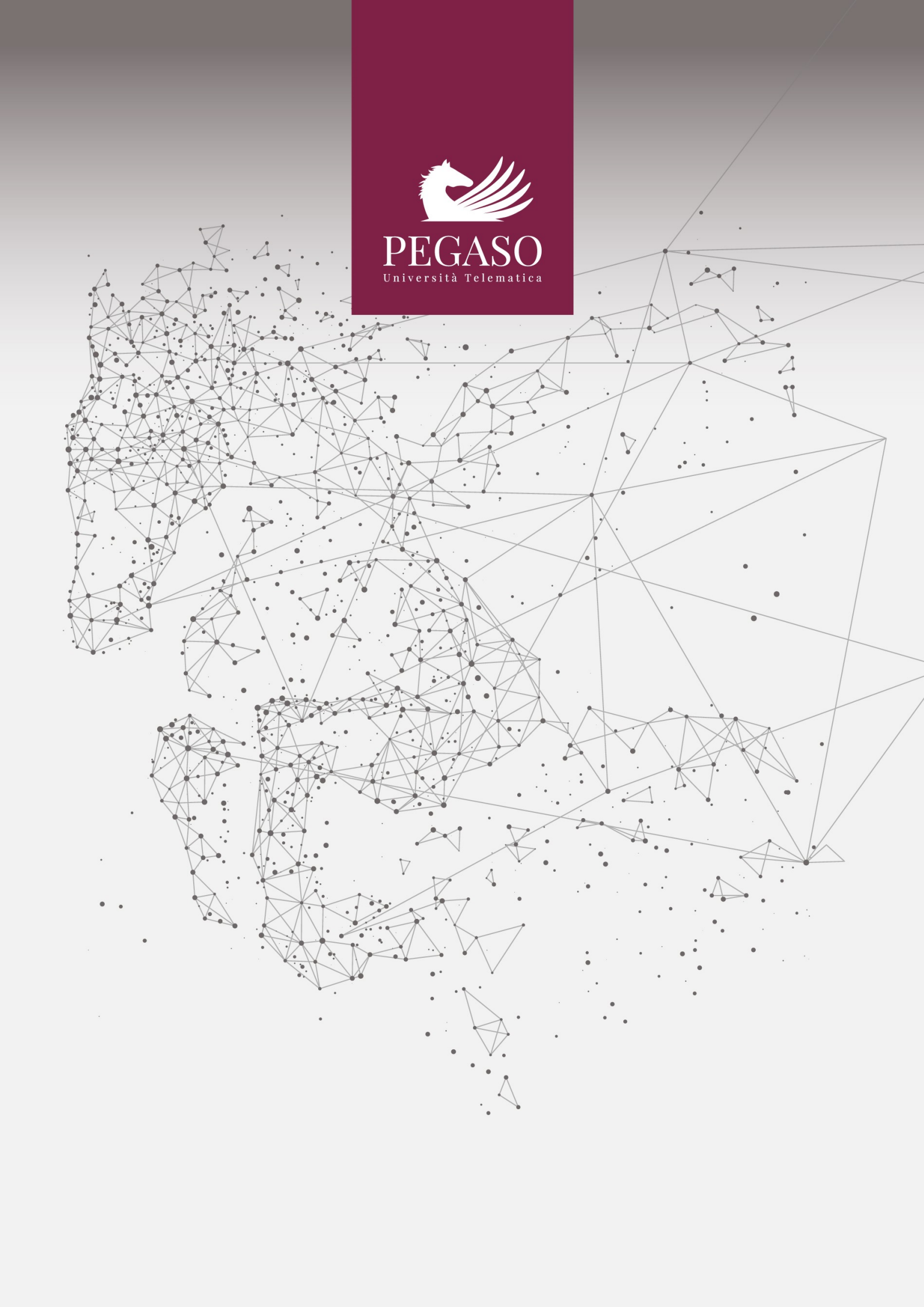
PEGASO
Università Telematica

Indice

1. Tipi primitivi

Java definisce 8 tipi di dati primitivi, ovvero tipi di dati semplici che differenzieremo dai tipi di dati complessi riconducibili agli oggetti. I tipi di dati primitivi possono essere classificati in 4 categorie:

- tipi interi;
- tipi floating point;
- tipo testuale;
- tipo logico-booleano.

I tipi di dati interi sono quattro: byte, short, int e long e vengono memorizzati tramite il metodo del complemento a due.

Differiscono per la lunghezza dei bit e quindi per l'intervallo dei numeri rappresentanti. Infatti un byte può immagazzinare un intero. Un byte può rappresentare un intero utilizzando un byte (8bit) e quindi da -128 a 127, uno short usa 16 bit, in int usa 32 bit e il long usa 64 bit. In generale, l'intervallo dei numeri rappresentati va da -2(n-1) a 2n. In Java non esistono tipi senza segno (unsigned).

Come mostrato negli esempi di Fig. 2, per immagazzinare un intero in Java è possibile usare quattro tipi di sistemi numerici: decimale, binaria, ottale e esadecimale. Per distinguere queste notazioni tra loro e dalla notazione decimale si usa anteporre 0b per la notazione binaria, 0 per la notazione ottale, 0x per la notazione esadecimale.

```
byte b=10;            //notazione decimale: b vale 10
short s = 022;        // notazione ottale: s vale 18
long l = 0x12aCd;     // notazione esadecimale: l vale 74493
int i= 1000000000;    //notazione decimale: i vale 1000000000
int n = 0b10100001010001011010000101000101
                      //notazione binaria: n vale -1589272251
```

Fig. 2: *Uso dei tipi di dati interi*

I tipi di dati floating point servono per memorizzare numeri decimali. Java definisce due tipi a virgola mobile: float (a 32 bit) e double (64 bit).

Java utilizza per i valore floating point (a virgola mobile) lo standard di decodifica IEEE-751.

Oltre alla notazione classica double d = 0.0126; si può usare in alternativa la notazione esponenziale: double d = 1.26E-2;

Il tipo di dato char permette di immagazzinare il singolo carattere. Si tratta di un tipo di dato non particolarmente usato perché nella maggior parte dei casi è necessario usare stringhe che consentono di

immagazzinare sequenze di caratteri. Il tipo primitivo char è immagazzinato in 16 bit e può definire ben 65536 caratteri diversi. La codifica Unicode si occupa di standardizzare tutti caratteri.

Il char memorizza un qualsiasi carattere che si trova sulla tastiera oppure un valore Unicode in esadecimale che identifica univocamente un determinato carattere anteponendo il prefisso \u. Inoltre il char può immagazzinare caratteri di escape speciali, ossia sequenze di caratteri che corrispondono alla pressione su un tasto della tastiera che non è direttamente rappresentato da un carattere o un simbolo, ma che provoca particolari comportamenti nella stampa. Fra questi ad esempio c'è \n che equivale a line feed. In tutti i casi occorre comprendere tra apici singoli il valore da assegnare.

Il tipo di dato boolean utilizza solo un bit per memorizzare un valore e gli unici valori che può immagazzinare sono true e false. Il tipo boolean serve per effettuare scelte che condizionano l'esecuzione del programma.

In Fig. 3 è mostrata una tabella riepilogativa sui tipi di dati primitivi.

Nome tipo	Tipo di valore	Memoria usata	Valori consentiti
`byte`	Intero	1 byte	da -128 a 127
`short`	Intero	2 byte	da -32.768 a 32.767
`int`	Intero	4 byte	da -2.147.483.648 a 2.147.483.647
`long`	Intero	8 byte	da -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
`float`	Numero in virgola mobile (precisione singola)	4 byte	da $-3.40282347 \times 10^{38}$ a $3.40282347 \times 10^{38}$
`double`	Numero in virgola mobile (precisione doppia)	8 byte	da $-1.7*10^{308}$ a $1.7*10^{308}$
`char`	Carattere singolo	2 byte	Tutti i caratteri della codifica Unicode (65.535 caratteri)
`boolean`	Valore di verità	1 bit	`true` o `false`

Fig. 3: *Tabella riepilogativa sui tipi di dati primitivi*

In ogni linguaggio di programmazione esistono costrutti che permettono all'utente di controllare e gestire la sequenza di esecuzione delle istruzioni a runtime. Essenzialmente possiamo dividere questi costrutti in due categorie principali:

- Condizioni (o strutture di controllo decisionali): permettono di effettuare una scelta tra l'esecuzione di istruzioni diverse a seconda che sia verificata una specificata condizione, la quale coincide con il risultato di un'operazione booleana

- if, switch, operatore ternario

- Cicli (strutture di controllo iterative): permettono di decidere il numero di esecuzioni di determinate istruzioni.
  - while, for, do-while, foreach (ciclo for migliorato) Tali costrutti possono essere annidati.

2. Strutture di controllo: la selezione

Quando in un programma spesso c'è bisogno di scegliere tra operazioni diverse, si usa l'istruzione condizionale if-else che consente di prendere semplici decisioni basate su valori immagazzinati. In fase di runtime la JVM testa un'espressione booleana e, a seconda che essa risulti vera o falsa, esegue un certo blocco di istruzioni oppure no. Un'espressione booleana è un'espressione che come risultato può restituire solo valori di tipo boolean, vale a dire true o false.

La sintassi generale è la seguente:

```
if (espressione booleana) {
istruzione_1;
….
istruzione_k;
} else {
istruzione_k+1;
…
istruzione_n;
}
```

Se l'espressione booleana si verifica (ovvero vale true) si esegue il blocco di istruzioni da 1 a k (potrebbe anche essere una sola istruzione), se non si verifica (ovvero vale false) allora si esegue il blocco di istruzione da k+1 a n (anche in questo caso potrebbe anche essere una sola istruzione). Il ramo else potrebbe anche non esserci e quindi indicare di eseguire le istruzioni da 1 a k in caso di true e basta.

I due comandi alternativi sono detti anche rami con else che può essere opzionale come già detto.

Qui abbiamo indicato il caso generico di più istruzioni per ogni ramo che sono racchiusi fra parentesi graffe {...} denominati blocchi di codice. Ovviamente potrebbe anche essere una sola istruzione. In tal caso le parentesi graffe potrebbero anche non essere messe, ma si consiglia comunque di usarle anche per circondare un'unica istruzione. Infatti, questa pratica aggiunge leggibilità e favorisce l'evoluzione del codice.

Nell'esempio in alto di Fig. 1 l'istruzione di stampa viene eseguita se e solo se la variabile numeroLati ha valore 3. In quel caso l'espressione booleana (numeroLati==3) vale true e quindi sarebbe eseguita l'istruzione che segue l'espressione. Se invece l'espressione risultasse false, sarebbe eseguita direttamente la prima eventuale istruzione che segue l'istruzione di stampa. Possiamo estendere la potenzialità del costrutto if mediante parola chiave else come mostrato nell'esempio in basso. La seconda stampa viene eseguita se l'espressione risulta false.

```
…
if (numeroLati == 3)
        System.out.println ("Questo è un triangolo");
// il flusso di esecuzione riprende da qui in ogni caso
System.out.println («Qual è la prossima figura geometrica?");

…
if (numeroLati == 3)
        System.out.println ("Questo è un triangolo");
else
        System.out.println ("Questo non è un triangolo");
…
```

Fig. 2: *Esempi di costrutti if*

Un'espressione booleana è un'espressione che come risultato può restituire solo valori di tipo boolean, vale a dire true o false. Essa si avvale di operatori di confronto e, se necessario, di operatori logici. Sono espressioni booleane le seguenti:

- x==10 (la variabile x `e uguale a 10);
- x!=10 (la variabile x `e diversa da 10);
- (x/2)<=10 (l'espressione x/2 `e minore o uguale a 10);
- (x/2)>=(2+y) (l'espressione x/2 `e maggiore o uguale all'espressione. 2+y);
- (trovato==true) (la variabile boolean è verificata);
- ((x/2)<=10) || ((z/3)>=(2+y)) (verificata se una delle due espressioni risulta vera).

I comandi if-else possono essere annidati. In Fig. 2 sono mostrati due esempi diversi. Come si può vedere in generale non servono le parentesi perché if-else è un unico commando. Nel farlo però il codice diventa sempre più complesso generando situazioni equivoche e bisogna fare attenzione.

```
…
if (saldo >= 0)
        if (saldo > 0)
                System.out.println (" Saldo positivo !");
        else
                System.out.println (" Saldo zero !");
…
if (saldo >=0)  {
        if (saldo >0)
                System.out.println (" Saldo positivo !");
} else
        System . out . println (" Saldo zero o negativo");
```

Fig. 3: *Esempi di costrutti if annidati*

Il primo frammento di codice mostra un if che annida un costrutto if-else (else fa riferimento al secondo if), il secondo mostra un costrutto if-else che annida un costrutto if (else fa riferimento al primo if). In questo secondo caso sono necessarie le parentesi per far riferire l'else al primo if, altrimenti si riferirebbe al secondo.

ATTENZIONE: L'uso attento di spazi e "a capo" nel programma (indentazione) semplifica la lettura e la comprensione del codice.

Si consiglia sempre di usare un blocco di codice (mediante l'uso di parentesi graffe) per circondare anche un'unica istruzione. Questa pratica oltre ad aggiungere leggibilità consente anche di aggiungere istruzioni in un secondo momento, come spesso succede.

Un caso particolare di if-else annidati sono gli if-else concatenati: il ramo else è costituito da un altro if. L'espressione booleana del secondo if considera un caso alternativo a quello considerato dalla condizione del primo if (o degli if precedenti). Da notare l'indentazione: di solito si scrive else if nella stessa riga.

Il funzionamento è abbastanza intuitivo. Se espressione-booleana1 fosse true verrebbero eseguite le isruzione del primo blocco di codice (istruzione_1;….istruzione_k;) e non sarebbero eseguite le istruzioni presenti negli altri blocchi. Infatti, non venendo eseguito il primo blocco dopo il primo else non sono eseguiti neanche gli if che ci sono dopo. Quindi le altre condizioni booleane saranno ignorate e il flusso di esecuzione del codice si sposterà fuori dall'intero costrutto (dopo l'ultima parentesi graffa che definisce il blocco di codice della clausola else).

Se invece espressione-booleana1 fosse false allora si valuterebbe espressione-booleana2 e nel caso fosse true si eseguirebbero le relative istruzioni (istruzione_k+1; …istruzione_j;) ignorando le altre linee di codice. Continuando la valutazione, se anche espressione-booleana2 fosse false si passerebbe a valutare espressione-booleana3 e se valesse true si eseguirebbero le relative istruzioni (istruzione_j+1;…istruzione_h) ignorando il resto del codice. Infine se tutte le espressioni booleane risultassero non verificate (ovvero valessero false) allora verrebbero eseguiti tutte le istruzioni contenuti nel blocco di codice relativo alla clausola else (istruzione_k+1; … istruzione_n). In Fig. 4 un if concatenato.

```
…
if (numeroLati == 3)
      System.out.println ("Questo è un triangolo");
else if (numeroLati == 4)
      System.out.println ("Questo è un quadrilatero");
else if (numeroLati == 5)
      System.out.println ("Questo è un pentagono");
else if (numeroLati == 6)
      System.out.println ("Questo è un esagono");
else
      System.out.println ("Troppi lati! ");
…
```

Fig. 4: *Esempio di if concatenati*

Esiste un operatore che talvolta può sostituire un costrutto if. Si tratta del cosiddetto operatore ternario (detto anche operatore condizionale) che può regolare il flusso di esecuzione come una condizione. La sintassi è la seguente:

var = (espressione booleana)? valore1 : valore2;

dove:

- var: variabile a cui viene assegnato il valore;
- espressione booleana: è condizione di test che viene valutata e restituisce un valore booleano, ovvero vero o falso;
- valore1: se espressione booleana viene valutato come 'true', allora value1 viene assegnato a var;
- valore2: se espressione booleana viene valutato come 'false', value2 viene assegnato a var.

Questo operatore è chiamato come operatore ternario poiché l'operatore ternario utilizza 3 operandi, il primo è un'espressione booleana che restituisce vero o falso, il secondo è il risultato quando

l'espressione booleana restituisce vero e il terzo è il risultato quando l'espressione booleana restituisce falso. Inoltre come tutti gli operatori ritorna un valore.

Un requisito indispensabile è che il tipo della variabile e quello restituito da espr1 e espr2 siano compatibili. È escluso il tipo void.

Inoltre il valore ritornato deve per forza essere catturato:

- con una variabile;
- passando il suo risultato ad un metodo come parametro di input;
- restituendo il suo valore in output tramite un comando return di un metodo;
- in un'espressione switch.

Non è possibile ritornare un valore senza usarlo pena un messaggio di errore di compilazione.

L'operatore ternario non può essere considerato un sostituto del costrutto if, ma è molto comodo in alcune situazioni.

```
…
String resultString = (5> 1)? 'PASS': 'FAIL';

String resultValue = (x >= y)? 'X è maggiore o forse uguale a
y': 'x è minore di y';
```

Fig. 5: *Esempio di operatore ternario*

In Fig. 5 un esempio: l'operatore ternario valuta la condizione di test (5> 1), se restituisce true quindi assegna il valore1, ovvero 'PASS' e assegna 'FAIL' se restituisce false. Poiché (5> 1) è vero, resultString il valore viene assegnato come 'PASS'.

In alcuni casi un comando if può diventare un po' lungo da scrivere.

Consideriamo il frammento di codice in Fig. 6 che sulla base del valore della variabile giorno (definita come intero) stampa la stringa del giorno della settimana corrispondente. Esempio: un programma che legge un numero e lo trasforma in un giorno.

```
…
if (giorno ==1) System.out.println (" Lunedi ");
else if ( giorno == 2) System . out . println (" Martedi ");
else if ( giorno == 3) System . out . println (" Mercoledi ");
else if ( giorno == 4) System . out . println (" Giovedi ");
else if ( giorno == 5) System . out . println (" Venerdi ");
else if ( giorno == 6) System . out . println (" Sabato ");
else if ( giorno == 7) System . out . println (" Domenica ");
else System.out.println (" Numero errato ");
…
```

Fig. 6: *Esempio di if concatenati con giorni della settimana*

In questi casi, di if concatenati andando a confrontare il risultato di una espressione di tipo int o char con un numero di letterali alternativi (ad es. 1,2,3,...) l'if può essere sostituito da uno switch che consente di eseguire determinate istruzioni piuttosto che altre in base al valore che sarà passato durante l'esecuzione al costrutto.

Il costrutto switch si presenta quindi in alcune situazioni come alternativo al costrutto if: Consente di eseguire determinate istruzioni piuttosto che altre, in base al valore passato durante l'esecuzione del costrutto.

Di seguito presentiamo la sintassi:

```
switch (variabile-di-test) {
        case valore_1:
                istruzione_1;
        break;    //istruzione break opzionale
        case valore_2:
                istruzione_2;
                …
                istruzione_k;
        break;
        case valore_3:
        case valore_4:  { //blocco di codice opzionale
                istruzione_k+1;
                …
                istruzione_j;
                }
        break;
        default: { //clausola default opzionale
        istruzione_j+1;
        …
        istruzione_n;
        }
}
```

La parola chiave switch definisce il costrutto stesso ed è delimitato da un blocco di codice. Tra parentesi tonde dichiara una variabile o un'espressione (ovvero un'istruzione che ritorna un valore come

potrebbe essere un metodo) che svolge lo stesso ruolo che svolgeva l'espressione booleana nel costrutto if. La differenza è che con l'espressione booleana dell'if si potevano definire espressioni come per esempio a<10 mentre per lo switch si tratto solo di un valore e non di un'espressione booleana.

Tale valore dovrebbe coincidere con uno dei valori specificati nella varie clausole case.

La parola chiave case viene seguita da un valore univoco costante che lo identifica (detta label, etichetta) e dopo i : vengono specificate le istruzioni da seguire se il valore di test coincide con il valore del case. In linea di massima il valore può essere un tipo intero primitivo (byte, short, char o int mentre long non è ammesso oppure classi Java che possono sostituire tali tipi primitivi, sono dette classi wrapper) o una classe di tipo String (in questo caso bisogna stare attenti nei controlli di uguaglianza in quanto esse sono case sensitive e possono contenere spazi e quindi sarà più facile introdurre errori).

Non è possibile dichiarare due case con la stessa label.

È possibile non dichiarare un blocco di codice da eseguire per un'etichetta andando ad accorpare più case per lo stesso blocco di codice (vedi case valore_3 e case valore_4).

La parola chiave default svolge il ruolo che l'else svolge nel costrutto if. E' l'equivalente di una clausola case che definisce l'istruzione da eseguire se il valore del parametro di test dello switch non coincide ad alcuna label dei case definiti. Le clausole case e default sono opzionali. La parola chiave break

provoca l'immediata uscita dal costrutto, ed è anch'essa opzionale.

Vengono quindi eseguite le istruzioni che seguono la parola chiave case che definisce lo stesso valore che assume variabile-di-test.

Se il valore dei test non coincide con nessuno dei valori dei case dichiarati allora vengono eseguite le istruzioni che seguono la clausola default (se essa è presente, visto che è opzionale). Si consiglia comunque di usare la clausola default anche quando sembra non essercene bisogno. Infatti a priori non sappiamo se il programma evolverà ampliando i possibili valori che può assumere la variabile-di-test e quindi i possibili case. La clausola default potrebbe servire per gestire i nuovi case, con un comportamento standard, sia per scoprire che il costrutto switch deve essere modificato per accomodare i nuovi case.

È anche consigliabile mantenere un ordine logico dei vari case (anche se l'ordine è in teoria indifferente) per non incorrere in dimenticanze e peggiorare la leggibilità.

A seconda del valore intero che assume la variabile di test vengono eseguite determinate espressioni. La variabile di test deve essere o di tipo stringa o di un tipo di dato compatibile con un intero, ovvero un byte, uno short, un char oppure direttamente un int (ma non un long).

Inoltre valore_1…valore_n devono essere espressioni costanti e diverse tra loro. Si noti che la parole chiave break provoca l'immediata uscita dal costrutto.

Il comando break fa saltare l'esecuzione alla parentesi graffa chiusa E.vita, ad esempio, che dopo aver eseguito il caso 3 venga eseguito anche il 4. Tra l'etichetta case N: e il comando break ci può essere più di un comando senza bisogno di parentesi graffe aggiuntive, ma è buona norma metterle.

Se dopo aver eseguito tutte le istruzioni che seguono un'istruzione di tipo case non è presente un'istruzione break, verranno eseguiti tutti gli statement (istruzioni) che seguono gli altri case, sino a quando non si arriverà ad un break (questa tecnica è nota come fall through).

Lo switch può essere usato per rappresentare quindi un modo alternativo all'if concatenato. In Fig. 7 l'esempio di Fig. 6 rappresentato con uno switch.

```
…
switch ( giorno ) {
case 1: System.out.println (" Lunedi "); break;
case 2: System.out.println (" Martedi "); break;
case 3: System.out.println (" Mercoledi "); break;
case 4: System.out.println (" Giovedi "); break;
case 5: System.out.println (" Venerdi "); break;
case 6: System.out.println (" Sabato "); break;
case 7: System.out.println (" Domenica "); break;
default: System.out.println (" Numero errato ");
}
…
```

Fig. 7: *Esempio di switch con giorni della settimana*

Il comando switch consente di accorpare casi in maniera abbastanza semplice come mostrato nell'esempio di Fig. 8 sempre per la stampa dei giorni della settimana.

```
…
switch ( giorno ) {
case 1:
case 2:
case 3:
case 4:
case 5: System . out . println (" Giorno lavorativo "); break;
case 6: System . out . println (" Giorno prefestivo "); break;
case 7: System . out . println (" Giorno festivo "); break;
default: System . out . println (" Numero errato ");
}
…
```

Fig. 8: *Esempio di switch con accorpamento di case*

Se dopo aver eseguito tutte le istruzioni che seguono un'istruzione di tipo case non è presente un'istruzione break, verranno eseguiti tutti gli statement (istruzioni) che seguono gli altri case, sino a quando non si arriverà ad un break (questa tecnica è nota come fallthrough).

È sempre consigliabile usare la clausola default anche quando sembra non essercene bisogno. Infatti a priori non sappiamo se lo switch si evolverà ampliando il numero di case. La clausola default potrebbe servire sia per gestire i nuovi case, con un comportamento standard sia per scoprire che il costrutto switch deve essere modificato per accomodare i nuovi case. Di fatto la clausola default nel costrutto switch è l'equivalente della clausola else nel costrutto if

È consigliabile mantenere un ordine logico dei vari case. Infatti, anche se è possibile ordinare le varie clausole in modo casuale senza alterare il funzionamento del costrutto (ad esempio spostando la clausola default in cima alla lista) con un ordine logico si riduce il rischio di incorrere in dimenticanze e si migliora la leggibilità.

È consigliabile usare il break anche per la clausola default, pure se è posizionata come ultima clausola di un costrutto switch. Infatti, è possibile che in un aggiornamento del codice futuro (magari seguito da una persona diversa) si possa aggiungere un nuovo case, dopo la clausola default. Senza un break si eseguirebbero in sequenza le istruzioni del default seguite, non volute, da quelle del nuovo case.

È consigliabile delimitare il pezzo di codice di ciascun case con delle parentesi graffe, sebbene non strettamente necessario. In tal modo le variabili locali all'interno di un case rimangano tali, altrimenti esse sarebbero visibili all'interno di tutti i case dello stesso costrutto dichiarati successivamente. Limitare la visibilità delle variabili il più possibile laddove sono dichiarate e usate è buona norme e usare un blocco di codice (ovvero le istruzioni racchiuse da {} sebbene talvolta non strettamente necessario aiuta in tale direzione).

Abbiamo visto il costrutto switch nella sua definizione essenziale così come pensato all'inizio, tra l'altro come costrutto derivato da C e C++. Nelle varie versioni di Java tale costrutto è stato rivisto per essere usato come espressione (ovvero un'istruzione che ritorna un valore) e introducendo una sintassi meno verbosa andando anche ad evitare problemi in caso di un break dimenticato (dalla versione 13). Oggi switch è un costrutto nettamente più utile e potente rispetto alla versione originale, ma l'essenza di come funziona è quella descritta.

3. Strutture di controllo: iterazione

I cicli servono per poter iterare (ripetere) più volte delle operazioni, ad esempio, per eseguire una somma su un numero arbitrario di addendi forniti dall'utente. Il comando while consente di ripetere una istruzione (o un gruppo di istruzioni compresi in blocco di codice ovvero racchiuse in parentesi graffe) tante volte fintanto che una condizione specificata è vera. Ogni ciclo di esecuzione viene detto iterazione.

La sintassi è la seguente:

```
[inizializzazione;]
while (espressione_booleana) {
istruzione_1
…
istruzione_n;
[aggiornamento iterazione;]
}
```

Semantica del comando:

1. L'espressione booleana viene valutata
2. Se l'espressione booleana è vera si eseguono le istruzioni e si ricomincia dal punto 1
3. Se la condizione è falsa si salta il corpo e si procede con l'istruzione successiva al while Un semplice esempio viene mostrato in Fig. 9.

```
// ciclo while
…
int i=1;
while (i<=10) {
System.out.println(i);
i++;
}
….
```

Fig. 9: *Esempio con ciclo while*

In primo luogo viene dichiarata ed inizializzata ad 1 una variabile intera i. Poi inizia il ciclo in cui è esaminato il valore booleano dell'espressione tra parentesi. Siccome i è uguale a 1, è minore di 10 e la condizione è verificata. Quindi viene eseguito il blocco di codice nel quale prima sarà stampato il valore di i (ovvero 1) e poi sarà incrementata la variabile stessa di un'unità. A questo punto, terminato il blocco di codice, verrà nuovamente testato il valore dell'espressione booleana con i che è diventato pari a 2. La condizione è di nuovo verificata e quindi di nuovo verrà eseguito il blocco di codice con anche aggiornamento del valore di i. Questo si ripeterà fino a quando i verrà aggiornato a 11 e quindi nella successiva valutazione dell'espressione booleana, la condizione non si verifica e quindi il programma eseguirà l'istruzione successiva al while.

Si noti che all'interno del blocco di codice eseguito ci sono delle istruzioni che possono influire sul verificarsi o meno della condizione al controllo successivo. In questo caso si ha un indice di iterazione i che viene inizializzato prima e poi incrementato dentro il ciclo così che ad un certo punto la condizione non viene più verificata (i diventa maggiore di 10).

Quando scriviamo un ciclo while, in generale potremmo non sapere quante iterazioni esso farà: dipende dal tasso di aggiornamento della variabile che viene valutata come parte dell'espressione booleana (decremento quantità di oggetti dalla cui esistenza dipendono le operazioni previste nel ciclo) oppure dal verificarsi di qualche condizione esterna di qualche tipo (arrivo di un messaggio su una coda).

In alcuni casi, invece, il numero di iterazioni è noto a priori Esempio: in SommaNumeri si chiede all'utente quanti numeri voglia sommare, e il numero inserito dall'utente diventa esattamente il numero di iterazioni del ciclo.

Quando il numero di iterazioni è noto a priori, in alternativa al while possiamo usare il comando for con la seguente sintassi:

```
for (inizializzazione; espressione booleana; aggiornamento) {
istruzione_1;
istruzione_2;
…
istruzione_n;
}
```

dove:

- inizializzazione è un comando eseguito all'inizio del ciclo
- espressione booleana per valutare se effettuare l'esecuzione delle istruzioni o uscire dal ciclo
- aggiornamento è un comando eseguito ad ogni iterazione Le istruzioni possono essere n oppure anche una sola.

La sintassi è più compatta rispetto a quella del ciclo while. Le parentesi tonde di un ciclo for contengono parti separate da punti e virgola. La prima parte è dedicata alla inizializzazione delle variabili locali al ciclo che smetterà di esistere al termine del ciclo. Nella seconda parte invece, bisogna specificare la condizione che deve essere verificata affinché sia eseguita l'interazione. Infine la terza parte è dedicata ad istruzioni (solitamente costituite dall'aggiornamento della variabile inizializzata nella prima parte) che saranno eseguite automaticamente dopo ogni iterazione.

```
// ciclo for
…
for (int n=1; n<=10; n++) {
System.out.println(n);
}
….

for (int i=0, j=10; i<5 || j>5; i++, j--) {
System.out.println(i);
System.out.println(j);
}
…
```

Fig. 10: *Esempi di uso del ciclo for*

Nell'esempio in Fig. 10 dichiariamo una variabile locale n che smetterà di esistere al termine del ciclo. Nella seconda parte invece, è specificata la condizione che ovvero che n deve essere minore o uguale di 10. Infine la terza parte riguarda le istruzioni che saranno eseguite automaticamente dopo ogni iterazione, in questo caso l'incremento di n che avverrà dopo ogni iterazione.

In questo esempio si stamperanno i numeri da 1 a 10 perché il primo aggiornamento avverrà solo dopo la prima iterazione e si esce con n=11 dopo aver stampato 10 e incrementato la variabile di una unità. In sostanza vengono stampati tutti i numeri fino a 10 compreso.

È buona norma non modificare i nel corpo del for.

Si possono anche dichiarare più variabili all'interno del ciclo for, più aggiornamenti e, sfruttando operatori condizionali, anche più condizioni. Vedi il secondo esempio.

Come si vede le dichiarazioni vanno separate da virgole e hanno il vincolo di dover essere tutte dello stesso tipo (in questo caso int). Anche gli aggiornamenti vanno separati da virgole ma in questo caso

non ci sono vincoli. L'espressione booleana è in questo caso legata ad una condizione logica sul valore di entrambe.

I nomi i,j,k... sono usati comunemente per le variabili-contatore dei cicli for.

Il ciclo for è probabilmente il ciclo più utilizzato vista la sua grande versatilità. Inoltre è l'unico ciclo che permette di dichiarare una variabile contatore con visibilità interna al ciclo stesso e quindi si può riutilizzare in un for successivo.

Quando si esegue un ciclo sfruttando un indice (ad esempio su un array i cui elementi sono caratterizzati da un indice) il ciclo for è indubbiamente il più adatto. Per esempio riprendendo l'array alfabeto potremmo stampare tutti i suoi elementi in modo compatto e al tempo stesso chiaro, come mostrato nell'esempio di Fig. 11:

```
// ciclo for sugli array
…
char alfabeto [] =
{'a','b','c','d','e','f','g','h','i','l','m','n','o'
'p','q','r','s','t','u','v','z'};
for (int i=0; i<21; i++) {
System.out.println (alfabeto[i]);
}
….
```

Fig. 12: *Esempio di uso del ciclo for sugli array*

Il ciclo while è utilizzato soprattutto quando non si può calcolare a priori il numero di iterazioni da effettuare e non si ha un indice da aggiornare ad ogni iterazione. Ad esempio nei cicli infiniti dove la sintassi è molto semplice.

```
// uso ottimale ciclo while
…
while (true) { … }
for (;true;){…}  // equivalente a for(;;){…}
….
public class Interruttore {
public static final int ACCESO=0;
public static final int SPENTO=1;
public int posizione;

public void accendiLampadina () {
       while (interruttore.posizione == ACCESO) {
       //codice omesso
       }
}
}
```

Fig. 12: *Esempi di uso del ciclo while*

Più in generale il ciclo while è adatto a dichiarare espressioni booleane con una logica basata sullo stato (state based logic). In particolare si dice che lo stato di un oggetto è definito dal valore delle sue variabili di istanza. Un oggetto quindi a runtime può assumere vari stati a seconda del cambiamento del valore delle sue variabili di istanza. Il ciclo while ben si adatta a gestire gli stati di un oggetto. Come ad esempio possiamo considerare il ciclo while interno al metodo accendiLampadina della classe Interruttore che segue mostrato in Fig.13 dove ACCESO/SPENTO sono in sostanza due costanti che valgono per ogni istanza della classe. posizione invece è la situazione corrente dell'interruttore ovvero se è chiuso (ACCESO) oppure aperto (SPENTO). Verrà modificato in qualche parte del programma in cui la classe è usata. Nel ciclo while si mette del codice che verrà eseguito finché l'interruttore è in posizione di ACCESO per tenere la lampadina illuminata.

Il ciclo do-while è il ciclo meno utilizzato. È però fondamentale nei casi in cui si desideri che le istruzioni del blocco di codice siano eseguite almeno una volta. La sintassi è la seguente:

```
[inizializzazione;]
do {
istruzione_1
…
istruzione_n;
[aggiornamento iterazione;]
} while (espressione_booleana);
```

Viene prima eseguito il blocco di codice e poi viene valutata l'espressione booleana (condizione di terminazione) che si trova a destra della parola chiave while. Se l'espressione booleana è verificata viene rieseguito il blocco di codice, altrimenti il ciclo termina.

Per esempio consideriamo questo semplice frammento di codice:

```
int i=10;
do {
System.out.println(i);
} while (i<10);
```

L'output è semplicemente singolo con la stampa di 10 a video. La prima iterazione è eseguita comunque anche se la condizione poi si rivela falsa.

Dalla versione 5 di Java è stata aggiunta una nuova versione del ciclo for con sintassi molto semplice e compatta:

```
for (variabile_temporanea : oggetto_iterabile) {
istruzione_1;
istruzione_2;
…
istruzione_n;
}
```

Dove oggetto_iterabile è l'array sui cui elementi si vuole iterare, variabile_termporanea dichiara una variabile alla quale, durante l'esecuzione del ciclo, sarà assegnato il valore dell'i-esimo elemento dell'oggetto_iterabile all'i-esima iterazione. Quindi variabile_temporanea rappresenta all'interno del blocco di codice di questo ciclo un elemento dell'oggetto_iterabile. Solitamente le istruzioni del ciclo utilizzano tale

variabile. A questo punto non è applicabile un'espressione booleana per la quale il ciclo deve terminare; questo è già indicativo del fatto che questo for viene usato soprattutto quando si vuole iterare su tutti gli elemento di un oggetto iterabile. In Fig. 13 è mostrato un esempio che stampa a video tutti gli elementi dell'array:

```
// ciclo foreach
…
int[] arr = {1,2,3,4,5,6,7,8,9};
for (int tmp : arr) {
system.out.println (tmp);
}
//stampa a video tutti gli elementi dell'array.
```

Fig. 13: *Esempio di uso del ciclo for migliorato*

Questa versione di for ha diversi limiti rispetto al ciclo for tradizionale. Per esempio, non è possibile eseguire cicli all'indietro e non è nemmeno possibile farlo su più oggetti contemporaneamente. Infine, non è possibile accedere all'indice dell'array dell'elemento corrente, ma per ovviare a questo si può dichiarare un contatore all'esterno del ciclo e incrementarlo all'interno. In casi come questo sarebbe opportuno utilizzare un semplice ciclo while.

4. Strutture di controllo: ramificazione

Nel linguaggio Java, esistono alcune istruzioni speciali che permettono di alterare il normale flusso di esecuzione del programma. Queste istruzioni, dette di ramificazione o branching, consentono di interrompere anticipatamente cicli, saltare iterazioni o terminare l'esecuzione di un metodo. Le tre principali parole chiave utilizzate a questo scopo sono break, continue e return. A esse si aggiunge una forma particolare di break, nota come break con etichetta, utile per gestire strutture di controllo annidate.

L'istruzione break è una delle più comuni e serve a interrompere l'esecuzione di un ciclo prima che la sua condizione booleana venga nuovamente valutata. Quando un break viene eseguito all'interno di un ciclo for, while o do/while, oppure all'interno di uno switch, il controllo esce immediatamente dal blocco corrispondente e prosegue con la prima istruzione successiva al ciclo o allo switch. Questo tipo di comportamento può risultare molto utile nei casi in cui si voglia interrompere un ciclo a seguito di una certa condizione riscontrata durante l'esecuzione, senza attendere il naturale termine del ciclo stesso. Sebbene non sia sempre indispensabile, l'uso del break può contribuire a rendere il codice più efficiente e leggibile, specialmente quando il controllo sull'uscita dal ciclo non è facilmente esprimibile nella condizione iniziale.

Esiste anche una variante dell'istruzione break, detta break con etichetta, che estende ulteriormente le possibilità di controllo del flusso. In presenza di cicli annidati, può accadere di voler uscire non solo dal ciclo più interno, ma da un intero gruppo di strutture iterate. In questo caso, si può definire un'etichetta (un identificatore seguito da due punti) prima del ciclo esterno e poi usare break seguito dal nome dell'etichetta. Quando viene eseguito, break con etichetta causa l'uscita immediata da tutte le strutture iterate racchiuse sotto quell'etichetta, e il controllo riprende dalla prima istruzione successiva all'intero blocco etichettato. Questa tecnica può semplificare in modo significativo situazioni complesse in cui l'uscita condizionale da più livelli di ciclo sarebbe difficile da gestire in altro modo, anche se il suo uso dovrebbe essere dosato con cautela per non compromettere la chiarezza del codice.

L'istruzione continue, invece, non interrompe completamente il ciclo, ma forza il salto immediato alla fine dell'iterazione corrente, facendo riprendere il controllo dalla valutazione della condizione booleana che regola il ciclo stesso. In altre parole, tutte le istruzioni rimanenti all'interno del ciclo vengono ignorate per quell'iterazione, e il ciclo prosegue con la successiva. Questo risulta particolarmente utile quando si vuole saltare specifici casi all'interno di un ciclo in base a certe condizioni, evitando di doverli racchiudere in ulteriori blocchi condizionali. Anche continue può essere utilizzato con etichette, permettendo di saltare a

una specifica iterazione di un ciclo esterno, ma questa forma è meno frequente rispetto a break con etichetta.

Infine, l'istruzione return ha un ruolo più generale, in quanto consente di interrompere l'esecuzione di un intero metodo, restituendo il controllo al metodo chiamante. In Java, return può essere utilizzata in due forme principali. La forma return valore; viene impiegata quando il metodo è dichiarato con un tipo di ritorno diverso da void, e il valore restituito deve essere compatibile con quanto specificato nella dichiarazione del metodo. La forma return;, invece, può essere usata nei metodi dichiarati come void, semplicemente per terminare l'esecuzione in anticipo senza restituire alcun valore. L'uso di return non è limitato alla fine del metodo: può apparire in qualsiasi punto del suo corpo, spesso all'interno di condizioni, per indicare che una certa logica deve portare alla terminazione immediata del metodo.

Tutte queste istruzioni, se usate correttamente, permettono di migliorare la leggibilità, l'efficienza e la chiarezza del codice, fornendo un controllo più preciso sul comportamento del programma. Tuttavia, è bene ricordare che un uso eccessivo o disordinato di queste istruzioni, in particolare di break e continue con etichette, può ridurre la manutenibilità del codice e complicarne la comprensione, specialmente in sistemi di grandi dimensioni. Per questo motivo, si raccomanda di utilizzarle con criterio, privilegiando soluzioni strutturate e leggibili ogni volta che possibile.

Bibliografia

- Il nuovo Java, Claudio De Sio Cesari, Hoepli Informatica