# PROGRAMMAZIONE


Docenti

**Luca Padovani**

**Diletta Cacciagrano**

Studente

**Giorgio Della Roscia**

Università degli studi di Camerino

Informatica per la Comunicazione Digitale (L-31)

Scuola di Scienze e Tecnologie


A.A. 2022/2023

# Indice

# Figure

# Tabelle

# Codici

# Approfondimenti

# Sezione 1

# NOZIONI

*«La programmazione è l'attività di progettazione e realizzazione di algoritmi per mezzo di linguaggi appositi.»*

## 1.1 Algoritmi

Si ha un insieme finito di istruzioni a cui vanno imposte alcune condizioni:

a. assenza di ambiguità nell'interpretazione;

b. restituzione di un risultato in tempo finito;

c. risoluzione di tutti i problemi di una classe.

## 1.2 Compilazione

Per far comprendere linguaggi di alto livello all'elaboratore occorre un componente intermedio.

### 1.2.1 Traduttori

Mutano il codice scritto dal programmatore in modo tale che sia comprensibile dal computer.

#### 1.2.1.1 Assemblatori

Il codice in Assembly di partenza viene convertito, o assemblato, in linguaggui macchina.

#### 1.2.1.2 Compilatori

Il programma scritto con un linguaggio ad alto livello viene tradotto in uno equivalente ma di più basso livello eseguibile.

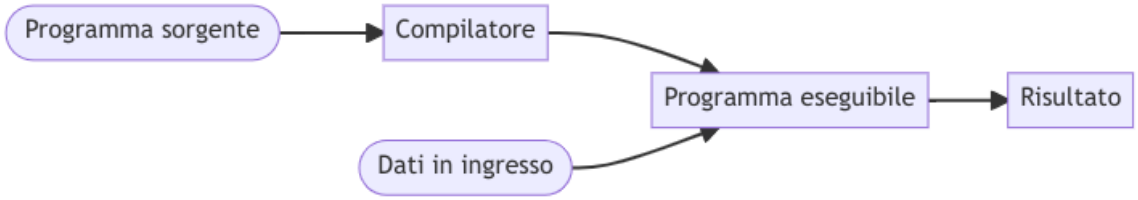



Figura 1.1: Schema collocazione compilatore

### 1.2.2 Interpreti

Viene direttamente eseguito il programma tramite un'interpretazione, appunto, del linguaggio direttamente ad alto livello.

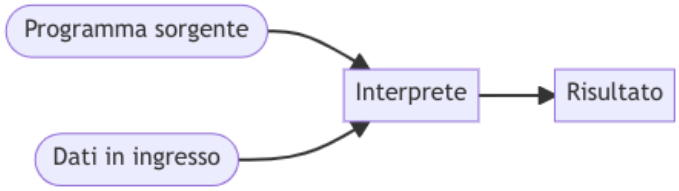



Figura 1.2: Schema collocazione interprete

Il linguaggio Python sfrutta un interprete; ciò semplifica la struttura del codice ma ne rallenta notevolmente i tempi d'esecuzione.

**Nota bene** (compilazione ed interpretazione Java)**.** In Java l'approccio utilizzato mescola entrambi i metodi descritti precedentemente:

1. il programma viene scritto e salvato con estensione .java;
2. esiste un compilatore, javac, che effettua la traduzione in linguaggio macchina e produce file .class, detto bitecode;
3. questi ultimi vengono raccolti dalla JVM (*Java Virtual Machine*) e successivamente eseguiti dall'interprete.

## 1.3 Sistemi di numerazione

Si ha una coppia $(N,\ v)$ in cui:

- $N$ è un insieme di simboli detti numerali;
- $v\ :\ N \to \mathbb{Z}$ è una funzione che associa ad ogni numerale il numero intero che lo rappresenta.

**Esempio** (numerazione romana e araba)**.** Le due numerazioni più rinomate sono:

- romana

  $N = \{X,\ XIII,\ IV,\ \ldots\}$ $\qquad$ $v(X) = 10,\ v(XIII) = 13,\ v(IV) = 4,\ \ldots$

- araba

  $N = \{0,\ 21,\ 072,\ \ldots\}$ $\qquad$ $v(0) = 0,\ v(21) = 21,\ v(072) = 72,\ \ldots$

### 1.3.1 Sistemi di numerazione posizionali

Si ha una base moltiplicatrice che va a modificare i valori delle cifre in base alla loro posizione.

**Esempio** (sistemi di numerazione posizionali)**.** Tre dei sistemi di numerazione posizionali più diffusi sono:

- binario

  $C = \{0, 1\}$ $\qquad v(10111_2) = 2^4 \cdot 1 + 2^2 \cdot 1 + 2^1 \cdot 1 + 2^0 \cdot 1 + = 23$

- decimale

  $C = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ $\qquad v(316_{10}) = 10^2 \cdot 3 + 10^1 \cdot 1 + 10^0 \cdot 6 = 316$

- esadecimale

  $C = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ $\qquad v(AC_{16}) = 16^1 \cdot 10 + 16^0 \cdot 12 = 172$

#### 1.3.1.1 Cambi di base

La base solita, decima, utilizzata nel nostro sistema può essere variata in modo tale che l'intera numerazione si adegui ad essa.

$$10^n + \ldots + 10^2 + 10^1 + 10^0 = k$$

$$2^n + \ldots + 2^2 + 2^1 + 2^0 = k$$

$$16^n + \ldots + 16^2 + 16^1 + 16^0 = k$$

**Esercizio** (cambio di base numerica)**.** Effettuare il cambio base al numero 32 da decimale a binaria: $32_{10} \rightarrow x_2$.

$32/2 = 16$ resto 0 $\qquad$ $4/2 = 2$ resto 0
$16/2 = 8$ resto 0 $\qquad$ $2/2 = 1$ resto 0
$8/2 = 4$ resto 0 $\qquad$ $1/2 = 0$ resto 1

Leggendo i resti ottenuti in ordine di calcolo si ottiene un valore binario: 100000.

##### 1.3.1.1.1 Grandezze informatiche

I calcolatori utilizzano sempre la base binaria. Di fatti l'entità più semplice è il bit e può assumere solamente due valori: zero ed uno. Raggruppando una sequenza di otto cifre binarie si ottiene un byte.

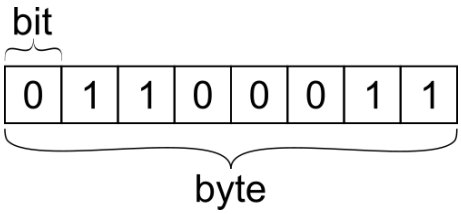



Figura 1.3: Bit e Byte

Ogni architettura mantiene le informazioni in dei pacchetti dimensionati in base al numero massimo di bit che possono viaggiare in parallelo fra la cpu e la memoria principale sul bus.

### 1.3.2 Rappresentazione dei numeri naturali

Con $n$ bit si possono rappresentare $2^n - 1$ numeri naturali. Il complemento di un intero è dato dall'inversione del suo equivalente binario.

> **Esempio** (numero complementare)**.** Scambiando ogni cifra del numero binario si ottiene un secondo valore, complementare al primo.
>
> $$11 \rightarrow \overline{11} \qquad 1011 \rightarrow \overline{1011} = 0100 = 4$$
>
> Possiamo vedere che il complementare del numero 11 è 4.

#### 1.3.2.1 Caratteri

In ogni elaboratore vengono associati dei numeri interi e senza segno ai vari caratteri inseribili da tastiera. Esistono varie codifiche: ASCII, UTF-8, UTF-16 ed ulteriori.

Tabella 1.1: Codifica ASCII lettere minuscole

| char | byte | char | byte |
|---|---|---|---|
| a | 01100001 | n | 01101110 |
| b | 01100010 | o | 01101111 |
| c | 01100011 | p | 01110000 |
| d | 01100100 | q | 01110001 |
| e | 01100101 | r | 01110010 |
| f | 01100110 | s | 01110011 |
| g | 01100111 | t | 01110100 |
| h | 01101000 | u | 01110101 |
| i | 01101001 | v | 01110110 |
| j | 01101010 | w | 01110111 |
| k | 01101011 | x | 01111000 |
| l | 01101100 | y | 01111001 |
| m | 01101101 | z | 01111010 |

### 1.3.3 Rappresentazione di interi con modulo e segno

Per indicare se un valore intero è negativo o positivo ci sono varie possibilità, fra cui:

- assegnazione di un bit al segno, si viene a creare un'ambiguità di rappresentazione dello zero;
- tecnica del complemento a due.

#### 1.3.3.1 Complemento a due

In questo caso si procede seguendo tali passaggi:

1. conversione in base binaria;
2. inversione dei bit (complemento a uno);
3. incrementazione singola.

Da notare è che gli ultimi due passaggi vengono effettuati se, e solo se, il numero di partenza era minore di zero.

**Esempio** (complemento a due intero positivo)**.** Rappresentare il numero 42 in complemento a due con un byte.

1. $42_{10} = \underline{0}0101010_2$

Essendo il numero analizzato positivo ci si ferma al primo punto.

**Esempio** (complemento a due intero negativo)**.** Rappresentare il numero $-42$ in complemento a due con un byte.

1. $-42_{10} = 00101010_2$
2. $00101010 \rightarrow 11010101$
3. $11010101 + 1 = \underline{1}1010110$

Il bit più significativo del numero ha peso per quanto riguarda il segno:

$$0 \rightarrow \text{positivo} \qquad\qquad 1 \rightarrow \text{negativo}$$

#### 1.3.3.2 Numeri con virgola

La rappresentazione dei numeri non interi viene effettuata sfruttando la virgola mobile, floating point:

$$(-1)^s \cdot 10^a \cdot m$$

Dove $s$ è il segno, $a$ l'esponente ed $m$ la mantissa.

## 1.4 Architettura

Ogni elaboratore è basato su del software appoggiato a dell'hardware. La prima rappresentazione riconosciuta è quella di Von Neumann: essa dispone un blocco di comunicazione fra processore e memoria.

### 1.4.1 Memorie

Questi contenitori di dati hanno due parametri che viaggiano in direzioni opposte: capienza e velocità.

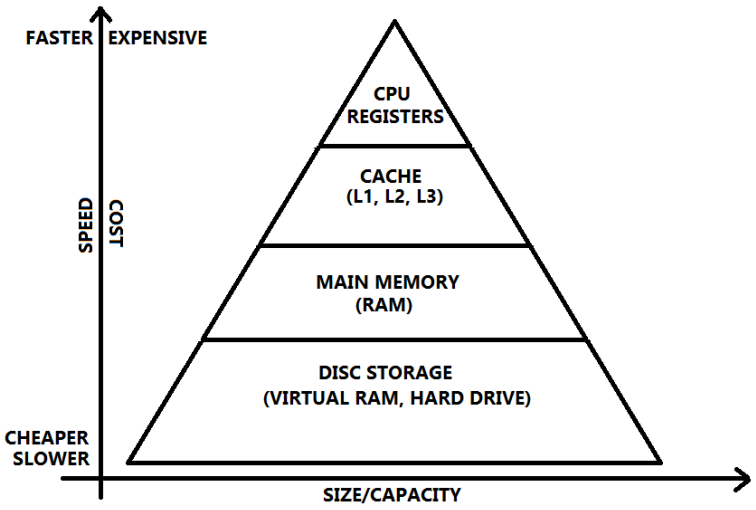



Figura 1.4: Gerarchia delle memorie

#### 1.4.1.1 Stack

La pila dei frame contiene il valore delle variabili locali, l'indirizzo dell'heap dove sono allocati gli array ed ulteriori informazioni necessarie all'esecuzione di metodi.

Tabella 1.2: Struttura frame

| method | | |
|---|---|---|
| program counter | | |
| type | id | value |

#### 1.4.1.2 Heap

Vengono allocati gli oggetti dinamicamente, gli array in modo contiguo ed anche i dati di tipo stringa.

# Sezione 2

# CODICE

Ora si analizzano aspetti prettamente tecnici dediti a scrivere programmi, specificamente in Java ma anche, più generalmente, di tecniche comuni a tutti i linguaggi diffusi.

## 2.1 Esecuzione

Per eseguire un file sorgente Java occorre seguire vari passaggi da terminale.

```
1 javac file.java
2 java file
```

Codice 2.1: Esecuzione da terminale

Questi vanno digitati nello specifico path della workspace.

## 2.2 Sintassi

Occorre rispettare una semantica di linguaggio per far sì che un programma scritto venga eseguito correttamente.

### 2.2.1 Commenti

Sono delle note ignorate dal compilatore ma dedite a facilitare la lettura del codice ai programmatori.

```
1 // commento su una linea
2 /*
3     commento
4     multilinea
5 */
```

Codice 2.2: Commenti

### 2.2.2 Variabili

Sono dei contenitori di memoria in cui alloggiare un determinato dato.

```
1 tipo nome = valore;
```

Codice 2.3: Dichiarazione variabili

L'identificatore può contenere codice alfanumerico ed underscore ma non può iniziare con lettere o esser composto esclusivamente da quest'ultimo.

Tabella 2.1: Tipi di dati

| NOME | BIT | DESCRIZIONE | NOME | BIT | DESCRIZIONE |
|---|---|---|---|---|---|
| byte | 8 | interi piccolissimi | int | 32 | interi classici |
| boolean | 8 | valori di verità | float | 32 | una cifra decimale |
| short | 16 | interi piccoli | double | 64 | due cifre decimali |
| char | 16 | codifica UTF-16 | long | 64 | interi grandi |

Oggetti dello stesso tipo utilizzano la stessa codifica e lo stesso spazio di memoria. La dicitura dei nomi delle variabili è standardizzata secondo due convenzioni: camelCase e snake_case.

#### 2.2.2.1 Assegnamenti

La dicitura è la seguente:

```
1 nome = valore;
```

Codice 2.4: Assegnamento

Questa operazione è distruttiva, ossia rimuove ogni traccia del vecchio dato contenuto nella variabile. In Java è possibile effettuare anche un tipo di assegnamento molto comodo:

```
1 a = b = c = 1;
```

Codice 2.5: Assegnamento multiplo

#### 2.2.2.2 Operatori

Per riprendere elementi matematici si usufruisce della seguente dicitura.

Tabella 2.2: Operatori logici

| NOTAZIONE | SIGNIFICATO |
|---|---|
| $x + y$ | addizione |
| $x - y$ | sottrazione |
| $x * y$ | moltiplicazione |
| $x/y$ | divisione |
| $x\%y$ | modulo |

#### 2.2.2.2.1 Incrementi e decrementi

Quando si vogliono tenere dei contatori o semplicemente aggiungere, o togliere, una singola unità ad una variabile si hanno diversi operatori a disposizione.

Tabella 2.3: Incrementi e decrementi

| OPERATORE | CLASSICO | AZIONE |
|---|---|---|
| $++x$ | $x = x + 1$ | preincrementa, restituisce $x + 1$ |
| $--x$ | $x = x - 1$ | predecrementa, restituisce $x - 1$ |
| $x++$ | $x = x + 1$ | postincrementa, restituisce $x$ |
| $x--$ | $x = x - 1$ | postdecrementa, restituisce $x$ |

#### 2.2.2.2.2 Espressioni logiche

Si basano sul sistema booleano in cui esistono solamente due risposte: `true` o `false`.

Tabella 2.4: Relazioni logiche

| OPERATORE | RELAZIONE |
|---|---|
| == | uguale |
| ! = | diverso |
| < | minore |
| > | maggiore |
| <= | minore o uguale |
| >= | maggiore o uguale |
| \|\| | or |
| && | and |
| ! | not |

Se presi singolarmente i simboli di and ed or stanno ad indicare una comparazione binaria bit a bit. In delle condizioni viene esclusa la logica cortocircuitaria.

#### 2.2.2.3 Widening

Il valore viene convertito, promosso, implicitamente.

$$\text{double} \rightarrow \text{float} \rightarrow \text{long} \rightarrow \text{int} \rightarrow \text{short} \rightarrow \text{byte}$$

Per convertire valori in altri tipi dall'attuale si adopera tale dicitura:

```
1 double a = 1.75;
2 int b = (int) a;
```

Codice 2.6: Casting di tipo

In questo caso viene effettuato un troncamento della parte decimale e non un arrotondamento.

#### 2.2.2.3.1 Narrowing

Si ha l'opposto del widening. Ciò può causare una perdita di informazione.

$$\text{byte} \rightarrow \text{short} \rightarrow \text{int} \rightarrow \text{long} \rightarrow \text{float} \rightarrow \text{double}$$

Nelle operazioni è necessario specificarne l'uso, altrimenti viene generato un errore.

### 2.2.3 Costanti

Dei dati fissi ed immutabili durante l'esecuzione del programma:

```
final tipo nome = valore;
```

Codice 2.7: Dichiarazione di una costante

La parola chiave `final` comunica al compilatore che la variabile non può esser modificata fino al termine dell'esecuzione.

### 2.2.4 Array

Una collezione finita di elementi del medesimo tipo possono essere stipati in un unico contenitore.

```
int[] a = new int[5];
```

Codice 2.8: Inizializzazione array

Si crea un array vuoto composto da cinque elementi. Questi vengono salvati contiguamente nell'heap.

```
int[] arr = {1, 2, 3, 4, 5};
```

Codice 2.9: Dichiarazione array

Si forma un array di tanti elementi quanti quelli mostrati. La dimensione di un array è statica, non modificabile in corso di esecuzione.

```
int[] a = {1, 3, 5};
int n = a.length;
```

Codice 2.10: Lunghezza di un array

Nella variabile $n$ verrà inserito un dato, intero, pari alla dimensione dell'array. Gli elementi vengono indicizzati da 0 fino a $n-1$.

Tabella 2.5: Indicizzazione elementi array

| POSIZIONE | INDICE |
|---|---|
| prima | 0 |
| seconda | 1 |
| ⋮ | ⋮ |
| ultima | $n-1$ |

#### 2.2.4.1 Confronto fra array

Per comparare due array si può utilizzare la condizione di eguaglianza.

```
int[] a = {1, 2, 3, 4}, b = {1, 2, 3, 4};
System.out.println("a == a ? " + (a == a)); //true
System.out.println("a != a ? " + (a != a)); //false
System.out.println("a == b ? " + (a == b)); //false
System.out.println("a != b ? " + (a != b)); //true
```

Codice 2.11: Paragone riferimenti di due array

Per comparare il contenuto occorre procedere secondo un'altra maniera.

```
int[] a = {1, 2, 3, 4};
int[] b = {1, 3, 5, 7};
boolean uguali = true;
if (a.length == b. length)
    for (int x: a.length)
        if (a[x] != b[x]) uguali = false;
else uguali = false;
```

Codice 2.12: Paragone contenuto di due array

#### 2.2.4.2 Array come argomenti di metodi

Qualora si volesse passare un array come argomento di un metodo occorre indicare un riferimento alla regione di memoria dove è salvato.

```
void stampa(int[] a) {
    for (int x: a) System.out.print(a[x] + ", ");
}
int a[] = {2, 3, 4};
stampa(a);
```

Codice 2.13: Tipo array come argomento

Per prendere in ingresso ad un metodo un array di dimensione non specificata a priori esiste una soluzione:

```
void metodo(tipo ... array) { ... }
```

Codice 2.14: Array indefinito come argomento

#### 2.2.4.3 Matrici

La condizione per cui tutti gli elementi dell'array sono altri array di dimensione eguale fra loro viene definita matrice.

```
int[][] m1 = new int[3][4];
int[][] m2 = {
    {2, 4, 6, 8},
    {3, 6, 9, 12},
    {4, 8, 12, 16}
};
```

Codice 2.15: Dichiarazioni di matrici

## 2.3 Programmazione iterativa

Ogni programma viene scritto per risolvere un problema, dispone dunque di uno scheletro alla base. Nell'immagine si nota un cosiddetto diagramma di flusso, o a blocchi.

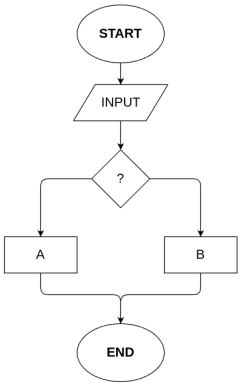



Figura 2.1: Flowchart

Tabella 2.6: Blocchi dei flowchart

| BLOCCO | DESCRIZIONE |
|---|---|
| freccia | flusso di esecuzione |
| ovale | inizio e fine |
| parallelogramma | I/O |
| rombo | condizioni |
| rettangolo | operazioni |

### 2.3.1 Metodi

Esiste codice di utilizzo molto frequente che viene standardizzato e salvato in librerie accessibili, questi sono i metodi. Possono essere invocati ed acquisire degli argomenti fornendo infine un risultato.

```
tipo identificatore(lista di argomenti) {
    blocco di comandi;
    return (tipo) var;
}
```

Codice 2.16: Definizione di un metodo

Il tipo del metodo può essere anche nullo, `void`. In tal modo il metodo non deve restituire nulla.

#### 2.3.1.1 Gestione I/O

Tramite i metodi si comunica con l'utente per fargli operare il programma. Per prendere dati in ingresso si ha:

```
System.out.print("inserire un numero intero: ");
return new Scanner(System.in).nextInt();
```

Codice 2.17: Ricezione input

Mentre per stampare su terminale si utilizza il comando

```
System.out.print("stringa");
```

Codice 2.18: Stampa a video

Se si vuole stampare su una nuova linea:

```
System.out.println("stringa");
```

Codice 2.19: Stampa a video a capo

#### 2.3.1.2 Ricorsione

Tecnica di programmazione utile qualora si necessiti il richiamo di metodi direttamente da dentro il loro blocco relativo. Sempre necessaria è l'imposizione di una condizione di uscita all'interno di quest'ultimo.

> **Esempio** (fattoriale tramite ricorsione)**.** Il fattoriale è definibile come la somma del prodotto di un numero per sé stesso decrementato ripetutamente fino ad uno.
>
> $$n! = n \times (n-1) \times (n-2) \times \ldots \times 2$$

Dato che si va a richiamare lo stesso metodo più volte verranno creati numerosi stack che potrebbero portare ad un overflow.

### 2.3.2 Condizionali

Quando ci si deve interfacciare ad una scelta si pone:

```
if (condizione) operazioneA;
else operazioneB;
```

Codice 2.20: `if` - `else`

Ciò si può estendere con molteplici domande in questo modo:

```
if (condizione1) operazioneA;
else if (condizione2) operazioneB;
else operazioneC;
```

Codice 2.21: `if` - `else if` - `else`

Il tutto può essere ristretto, andando a perdere un po' di leggibilità seguendo la sintassi:

```
(condizione) ? (operazioneA) : (operazioneB);
```

Codice 2.22: `if` compatto

#### 2.3.2.1 Switch case

Se occorrono numerosi blocchi condizionali consecutivi si ricorre ad un'alternativa più leggibile:

```
switch (espressione) {
    case "expvalue": operazione; break;
    case "expvalue": operazione; break;
    case "expvalue": operazione; break;
    default: operazione; break;
}
```

Codice 2.23: `switch case`

Nel caso in cui si ometta il `break`; il programma eseguirà anche il caso successivo. Il `default` è necessario a provvedere un'alternativa se non si rientra in alcun `case`.

### 2.3.3 Ciclici

Nei casi in cui occorre ripetere una determinata istruzione per un certo numero di volte si sfruttano alcuni comandi appositi interscambiabili.

#### 2.3.3.1 While

Questo comando permette di verificare una condizione iniziale e successivamente, se ritenuta vera, entrare dentro al blocco di istruzioni collegato. Viene definito secondo:

```
while (condizione) {
    operazione;
}
```

Codice 2.24: `while`

#### 2.3.3.2 Do while

Dispone dello stesso funzionamento del while ma viene eseguito almeno una volta poiché il controllo è posto in coda al ciclo.

```
do {
    operazioni;
} while (condizione);
```

Codice 2.25: `do while`

Come si nota dall'esempio è necessario il punto e virgola dopo la condizione.

#### 2.3.3.3 For

Ciclo utilizzato quando si conosce a priori il numero di iterazioni da compiere. Contiene tre parametri: un'inizializzazione, una condizione ed un aggiornamento.

```
for (tipo nome = valore; condizione; valoreaggiornato) {
    operazione;
}
```

Codice 2.26: `for`

Il ciclo interessato recupera innanzitutto una variabile riferimento, effettua il controllo, poi esegue il blocco ed infine aggiorna la variabile; ricontrolla nuovamente la condizione, e così fino al termine.

#### 2.3.3.3.1 For each

Un'estensione di tale struttura è stata ideata per esaminare una lista.

```
int[] arr = {1, 2, 3, 4, 5};
int sum = 0;
for (int x: arr) {
    sum += x;
    return sum;
}
```

Codice 2.27: `foreach`

### 2.3.4 Ricerca

Esistono numerose tecniche con cui è possibile trovare un dato elemento in una collezione. Si ha una doppia tendenza, inversa, riguardo due fattori: semplicità ed efficienza.

#### 2.3.4.1 Ricerca sequenziale

```
int ricercaSequenziale(int[] a, int x) {
    for (int i = 0; i < a.length; ++i)
        if (a[i] == x) return i;
    System.out.println("non trovato");
    return -1;
}
```

Codice 2.28: Ricerca sequenziale, o ingenua

Nel caso peggiore, in cui l'elemento non è presente, si hanno $n$ confronti nulli.

#### 2.3.4.2 Ricerca ordinata

```
int ricercaOrdinata(int[] a, int x) {
    for (int i = 0; i < a.length; ++i)
        if (a[i] == x) return i;
        else if (a[i] > x) break;
    return -1;
}
```

Codice 2.29: Ricerca ordinata

Nel caso peggiore, in cui l'elemento è posizionato ultimo, si hanno $n + 1$ confronti.

#### 2.3.4.3 Ricerca binaria

Si divide in due la lista di elementi su cui viene effettuata la ricerca e scartata la metà in cui la condizione risulta impossibile. Viene ripetuto tale processo fino a pescare il dato voluto.

```
int ricercaBinaria(int[] a, int x) {
    int i = 0, j = a.length - 1;
    while (i <= j) {
        int k = ((i + j) / 2);
        if (x == a[k]) return k;
        else if (x < a[k]) j = k - 1;
        else i = k + 1;
    }
    return -1;
}
```

Codice 2.30: Ricerca binaria

Nel caso peggiore, in cui l'elemento è assente, si hanno $2 \log_2 n$ confronti. Per casi dove si ha una dimensione grande risulta più efficiente. Questa tecnica è indubbiamente efficace ma ha come unico requisito una lista ordinata.

### 2.3.5 Ordinamento

Vi sono molteplici approcci per ordinare una lista di elementi. Anche in questo caso la tendenza è doppia e inversa: semplicità ed efficienza.

#### 2.3.5.1 Ordinamento per selezione

Per ogni posizione si scambia l'elemento attivo con quello con valore minimo trovato nell'intervallo da esso al limite.

```
void ordinamentoSelezione(int[] a) {
    for (int i = 0; i < a.length; ++i) {
        int m = i;
        for (int j = i + 1; j < a.length; ++j)
            if (a[j] < a[m]) m = j;
        int t = a[i];
        a[i] = a[m];
        a[m] = t;
    }
}
```

Codice 2.31: Ordinamento per selezione

Questo tipo di ordinamento ha natura quadratica. Dunque si eseguono sempre $\frac{n(n-1)}{2}$ confronti.

#### 2.3.5.2 Ordinamento per fusione

In questo caso si procede seguendo tre passaggi:

1. divisione della sequenza in due sotto-sequenze di egual dimensione;
2. ordinamento delle due sotto-sequenze tramite ricorsione;
3. fusione delle due sequenze ottenute in una nuova sequenza ordinata.

```
int[] fusione(int[] a, int[] b) {
    int[] c = new int[a.length + b.length];
    int i = 0, j = 0, k = 0;
    while (i < a.length && j < b.length)
        if (a[i] <= b[j]) c[k++] = a[i++];
        else c[k++] = b[j++];
    while (i < a.length) c[k++] = a[i++];
    while (j < b.length) c[k++] = b[j++];
    return c;
}
int[] ordinamentoFusione(int[] a, int i, int j) {
    if (i > j) return new int[0];
    else if (i == j) return new int[] {a[i]};
    int k = (i + j) / 2;
    return fusione(ordinamentoFusione(a, i, k), ordinamentoFusione(a, k + 1, j));
}
```

Codice 2.32: Ordinamento per fusione

Risulta certamente più intricato ma è indubbiamente efficace.

## 2.4 OOP

Qualora si necessiti di una nuova tipologia di dato, con determinate caratteristiche, è possibile crearla. Tale direzionamento nella programmazione è definito orientato agli oggetti.

### 2.4.1 Classi

Una famiglia omogenea di oggetti è chiamata classe, ciò indica che dispone di:

- stessa rappresentazione dello stato;
- supporto delle stesse operazioni.

Gli oggetti appartenenti ad una certa classe C vengono detti istanze di C.

```
1 public class C { ... }
```

Codice 2.33: Definizione di una classe

#### 2.4.1.1 Oggetti

Sono delle strutture con loro etichetta, informazioni e comportamenti.

```
1 var obj = new Object();
```

Codice 2.34: Istanza di un oggetto

#### 2.4.1.2 Campi delle classi

Lo stato di una classe è composto da variabili di istanza, o campi.

```
1 public static void main(String[] args) {
2     private int a;
3     private int b;
4 }
```

Codice 2.35: Variabili di istanza di classi

> **Principio** (incapsulamento)**.** I campi vengono definiti `private`, ossia vengono resi inaccessibili dall'esterno della classe, ma predispongono di operazioni pubbliche per accedervi.

#### 2.4.1.3 Metodi

Ogni classe dispone di metodi. Questi servono per far interfacciare le variabili di istanza all'ambiente pubblico.

```
1 public method(int a) { this.a = a; }
```

Codice 2.36: Utilizzo di variabili della classe

##### 2.4.1.3.1 Costruttore

Ogni qualvolta che si crea un'istanza della classe viene eseguito il metodo costruttore. È inoltre possibile creare molteplici metodi costruttori che differiscono fra loro per il numero, o il tipo, di argomenti passati.

```
1 public class C {
2     public C() { ... }
3 }
```

Codice 2.37: Metodo costruttore

Si tratta di un metodo speciale senza tipo di ritorno e facilmente riconoscibile poiché prende il nome dalla classe.

#### 2.4.1.3.2 Getters

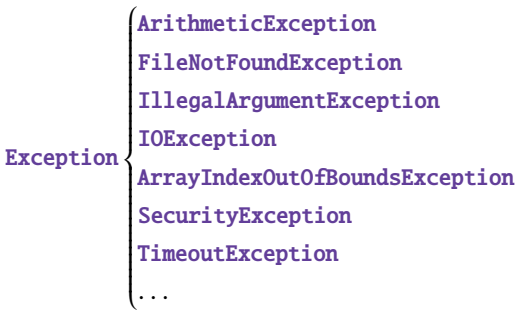
Consentono la lettura del valore di determinati campi della classe.

```
1 public int getA() { return a; }
2 public int getB() { return b; }
```

Codice 2.38: Metodi getters

### 2.4.2 Eccezioni

Per gestire le anomalie generate dal programma nei casi in cui avvengano degli errori si possono inviare delle eccezioni:




```
1 try {
2     ...
3 } catch Exception(e) {
4     throw new IllegalArgumentException("...");
5 }
```

Codice 2.39: Lancio eccezione

Come intuibile dalla parola chiave `new` ogni eccezione è un oggetto di classi presenti nella libraria standard del linguaggio.

#### 2.4.2.1 Try catch

Per tentare l'esecuzione di un blocco di codice ma tenendo conto di eventuali possibili errori si utilizza tale comando:

```
1 public static void main(String[] args) {
2     try { ... }
3     catch (Exception e) { throw new CustomException("..."); }
4     finally {
5         // il try-catch termina e passa per questo blocco
6     }
7 }
```

Codice 2.40: `try catch`

### 2.4.3 Ereditarietà

Si ha la necessità di avere una classe padre, in cui mettere le proprietà in comune, da cui far discendere le altre classi figlie.

```
1 public class B extends A { ... }
```

Codice 2.41: Definizione di una classe derivata

Terminologia:

$$A = \begin{cases} \text{classe padre} \\ \text{classe base} \\ \text{superclasse} \end{cases} \qquad B = \begin{cases} \text{classe figlia} \\ \text{classe derivata} \\ \text{sottoclasse} \end{cases}$$

Tale nomenclatura potrebbe fuorviare in quanto la sottoclasse ha più proprietà rispetto alla superclasse. Queste sono legate fra loro da una relazione di sottotipo:

$$B \text{ estende } A \qquad B \leftarrow A \qquad B <: A$$

> **Principio** (sostituzione di Liskov)**.** Date due classi, una figlia dell'altra, allora è possibile passare un'istanza della prima laddove ne è attesa una della seconda.

#### 2.4.3.1 Classe Object

Ogni classe è figlia di una standard contenuta nelle librerie del linguaggio.

```
1 // codice scritto
2 public class A { ... }
3 // interpretazione
4 public class A extends Object { ... }
```

Codice 2.42: Classe Object

#### 2.4.3.2 Overriding

Ogni metodo presente nella classe padre viene esteso per l'utilizzo nella classe figlia. In certi casi è però utile ritoccare il blocco di istruzioni; tale procedimento è detto overriding.

```
1 public class B extends A {
2     public B(int x) {
3         super(x);
4     }
5     public void submethod(int x) {
6         super.supmethod(x);
7     }
8 }
```

Codice 2.43: Utilizzo di metodi della superclasse nella sottoclasse

La definizione di molteplici metodi con lo stesso nome può essere data da due casistiche:

1. **overriding**
   - numero e/o tipo di argomenti uguale
   - raffinamento metodi delle superclassi
   - tipo di ritorno statico rispetto all'originale
2. **overloading**
   - tipo di ritorno variabile
   - accettazione di diversi parametri
   - numero e/o tipo di argomenti diverso

### 2.4.4 Classi astratte

Qualora si abbia una classe padre concettualmente sensata ma incompleta si adopera una tecnica di astrazione.

```
1 public abstract class A { ... }
```

Codice 2.44: Definizione di classi astratte

Così facendo il linguaggio non permette di creare degli oggetti di tale classe, poiché astratti, ma essa fornisce la struttura per le concrete classi derivate.

#### 2.4.4.1 Metodi astratti

Una classe astratta può avere dei metodi: classici o astratti a loro volta.

```
1 public abstract tipo nome();
```

Codice 2.45: Definizione di metodi astratti

Come è facile notare i questi metodi sono privi di corpo in quanto non esseguibili direttamente nella classe astratta in cui sono contenuti.

### 2.4.5 Interfacce

L'ereditarietà impone un'organizzazione gerarchica ad albero, con la presenza di un'unica superclasse. Per ovviare si introducono le interfacce: un nuovo tipo di dato con relative operazioni correlate.

```
1 public static interface I {
2     public tipo method() { ... }
3 }
```

Codice 2.46: Definizione interfaccia

Per utilizzare i metodi presenti nell'interfaccia nelle altri classi occorre implementarla:

```
1 public static class C implements I { ... }
```

Codice 2.47: Implementazione di un'interfaccia

Queste definiscono il comportamento delle classi che le implementano. Se una classe estende un'interfaccia, ma non utilizza tutti i metodi, allora va dichiarata astratta poiché incompleta.

Tabella 2.7: Differenze fra classi e interfacce

| CLASSE | INTERFACCIA |
|---|---|
| estensione singola | implementazioni multiple |
| contiene campi e metodi | contiene metodi |

Se $C$ è una classe che implementa l'interfaccia $I$, allora: $C <: I$.

# Bibliografia

[SW17] Robert Sedgewick and Kevin Wayne.
Introduction to programming in Java: an interdisciplinary approach. 2nd ed. Serie.
Addison-Wesley, 2017. ISBN: 9780321498052.