Cicli, Array e Programmazione su Sequenze

Luca Tesei
Università di Camerino
luca.tesei at unicam.it

Cicli

- Fino ad ora il nostro codice Java all'interno dei metodi si è limitato ad eseguire istruzioni in sequenza eventualmente prendendo vari rami del comando condizionale `if-else`
- Ciò che ci farà raggiungere l'effettiva potenza di calcolo tipica di tutti i linguaggi di programmazione è l'introduzione dell'iterazione
- L'iterazione altera la normale sequenza di esecuzione delle istruzioni permettendo di ripetere più volte uno stesso pezzo di codice

Iterazione e ricorsione

- Dal punto di vista della teoria della computazione, affinché un linguaggio di programmazione imperativo (C, Pascal, codice nei metodi del Java, ecc.) raggiunga la potenza di calcolo tipica è necessario che abbia almeno un costrutto di iterazione o, alternativamente, una meccanismo di ricorsione
- In Java e in quasi tutti i linguaggi moderni abbiamo sia costrutti iterativi (cicli) sia meccanismi per la ricorsione (si possono scrivere metodi ricorsivi)

Ciclo while

- Il ciclo `while` è uno dei costrutti iterativi del Java.
- Vedremo che il ciclo `do` e il ciclo `for` possono essere espressi in termini del ciclo `while`
- Il ciclo while ha questa sintassi:
  `while (BoolExpr) Com`
- `BoolExpr` può essere una qualsiasi espressione che ha un valore booleano (esattamente come quella dell'`if`). Viene chiamata *guardia* del ciclo

Esempio

- Consideriamo un investimento che parte con un capitale iniziale di 10.000 euro
- Ogni anno, al 31/12, vengono aggiunti al capitale gli interessi maturati in quell'anno, cioè il 5% del capitale presente all'1/1

Ciclo while

- `Com` può essere un qualsiasi comando Java (tra cui il while stesso)
- Viene chiamato *corpo* del ciclo
- Possiamo inserire una sequenza di comandi come corpo del `while` allo stesso modo di come abbiamo fatto per il ramo true o false dell'`if`: inserendo un blocco di istruzioni fra parentesi graffe
- Se il `while` contiene solo una istruzione nel corpo è bene non inserire le parentesi, ma usare l'indentazione

Esempio

La crescita del capitale è rappresentata nella seguente tabella:

Anno	Saldo
0	10.000,00
1	10.500,00
2	11.025,00
3	11.576,25
4	12.155,06
5	12.762,82

Ciclo while - Semantica

L'esecuzione del ciclo `while (E) C` avviene in questo modo:

1. Viene valutata l'espressione booleana `E`.
   (a) Se il valore di `E` è **falso** allora si esce dal ciclo continuando l'esecuzione del codice che segue il `while`
   (b) Se il valore di `E` è **vero** allora si esegue una volta il corpo `C` dopodiché si ritorna al punto 1.

Esercizio

Implementare un classe Investimento che rappresenta un certo investimento con il suo capitale iniziale e il suo tasso di interesse:

```
/** Costruisce un Investimento con un saldo iniziale e un
      tasso di interesse fisso
      @param aBalance saldo iniziale
      @parem aRate il tasso di interesse annuale fisso */
  public Investimento(double aBalance, double aRate){
    // TO DO
  }
```


Esempio

- Vogliamo scrivere un frammento di codice che ci dica quanti anni sono necessari affinché il capitale arrivi *almeno* a 20.000 euro.
- Dobbiamo fare iterare la capitalizzazione degli interessi fino a quando non troviamo che il capitale ha raggiunto o superato i 20.000 euro
- Per questo tipo di problema la soluzione tipica è quella di usare un ciclo, nel nostro caso il `while`

Esercizio

```
/** Continua ad accumulare interessi finch&eacute; il
      saldo non raggiunge un valore desiderato
      @param targetBalance il saldo desiderato */
  public void waitForBalance(double targetBalance) {
    // TO DO
  }
```


Esempio

```
/** @param balance Investimento iniziale
  * @param targetBalance Cifra a cui si vuole arrivare
  * @return Il numero di anni necessari per arrivare al
  *         targetBalance
  */
public int anniRichiesti(double balance, double rate,
                                 double targetBalance){
  int years = 0;
  while (balance < targetBalance) {
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
  }
  return years;}
```

Terminazione: errori tipici

Esempio: calcolare l'ammontare del capitale dopo 20 anni

```
...
int years = 0;
while (years < 20) {
  // manca years++; !!!
  double interest = balance * rate / 100
  balance = balance + interest;
}
...
```


Esercizio

```
/** Restituisce il saldo attuale dell'Investimento
      @return il saldo attuale */
  public double getBalance() {
    // TO DO
  }
  /** Restituisce il numero di anni per i quali
      l'investimento ha accumulato interessi
      @return il numero di anni trascorsi dall'inizio
              dell'investimento */
  public int getYears() {
    // TO DO
  }
```


Terminazione: errori tipici

Esempio: calcolare l'ammontare del capitale dopo 20 anni

```
...
int years = 20;
while (years > 0) {
  years++; // ERRORE!!!
  // doveva essere years--;
  double interest = balance * rate / 100
  balance = balance + interest;
}
...
```


Terminazione

- L'introduzione dei costrutti iterativi dà potenza piena al linguaggio di programmazione, ma introduce un problema significativo
- Attraverso i cicli si possono scrivere programmi che non terminano mai
- Ciò può avvenire per volontà del programmatore: ad esempio se l'applicazione si suppone che debba girare continuamente
- Più spesso invece succede a causa di un errore logico del programmatore. In questo caso il comportamento infinito è indesiderato

Cicli: scelta della guardia

- Nel progetto di un ciclo una parte fondamentale è la scelta della guardia e dei valori di inizializzazione
- Spesso ci sono operatori $<,>,\leq,\geq$
- Errore tipico: errore per scarto di uno
- Cioè usando $<$ ($>$) invece di $\leq$ ($\geq$), o viceversa, si esegue il corpo del ciclo una volta di più o una di meno del necessario

Cicli: scelta della guardia

- Il conteggio degli anni, nella variabile `years`, deve cominciare da 0 o da 1?
- La guardia giusta è `balance < initialBalance * 2` oppure `balance <= initialBalance * 2`?
- Tecnica semplice di risoluzione: fare un caso di uso con numeri semplici e poi generalizzare
- Cercare eventualmente di individuare un invariante

Cicli: scelta della guardia

Esempio: in quanti anni un certo capitale iniziale raddoppia:

```
int years = 0;
// capitale iniziale
double balance = initialBalance;
while (balance < initialBalance * 2) {
  years++;
  double interest = balance * rate / 100
  balance = balance + interest;
}
```


Cicli: invariante

- L'invariante di un ciclo è una condizione che rimane vera al momento della valutazione della guardia durante tutto il ciclo
- Nell'esempio precedente l'invariante è "La variabile `balance` contiene il capitale **dopo** `years` anni"
- E' valido alla prima valutazione della guardia: `balance == initialBalance` è il valore del capitale **dopo** `zero` anni
- Rimane valido alle successive valutazioni della guardia, compresa l'ultima

Ciclo do

Il ciclo do-while ha questa sintassi:

```
do Com while (BoolExpr);
```

Semantica:

1. Viene eseguito il comando (o il blocco) `Com`
2. Viene valutata l'espressione `BoolExpr`
   (a) Se il valore è `false` allora il ciclo esce e l'esecuzione passa all'istruzione successiva
   (b) Se il valore è `true` allora si riparte dal punto 1.

Ciclo do: esempio

Ottenere in input un valore intero con il vincolo che non sia negativo.

```
...
double value;
do {
  String input =
    JOptionPane.showInputDialog(
      ''Inserisci un numero positivo'');
  value = Double.parseDouble(input);
} while (value <= 0);
...
```


Ciclo do: motivazioni di utilizzo

- Si usa quando si è sicuri che bisogna fare almeno una iterazione del corpo all'inizio del ciclo
- Si può esprimere come ciclo while

```
do Com while (BoolExpr);
```

è equivalente a

```
Com
while (BoolExpr) Com;
```


Ciclo do: vantaggio

Risolvendo con il while, senza dover ripetere il corpo, dobbiamo inserire altri elementi.

```
// indica se e' stato inserito un valore
// corretto
boolean done = false;
while(!done) {
  String input =
    JOptionPane.showInputDialog(
      ''Inserisci un numero positivo'');
  value = Double.parseDouble(input);
  if (value > 0) done = true;
}
```

Ciclo for: esempio

Determina il valore di un investimento dopo n anni. Uso tipico.

```
for (int i = 1; i <= n; i++) {
  double interest =
        balance * rate / 100;
  balance += interest;
}
```

Osservazione: il comando `i++` dovrebbe essere seguito normalmente da `;` In questo caso non va messo perché la parentesi chiude il contesto

Ciclo for

Il ciclo `for` è un'abbreviazione di un ciclo `while` che viene spesso usato in una certa forma. La sintassi del `for` è la seguente:

```
for (Com_or_Decl; BoolExp; Com) Com;
```

dove

```
Com_or_Decl ::= Com | Decl
```


Il ciclo e mezzo

- Può accadere che si debba impostare un ciclo in cui l'informazione riguardante il proseguimento dell'iterazione o l'uscita viene determinata nel mezzo dell'esecuzione del corpo
- E' macchinoso, in questi casi, riportare questa informazione al momento della valutazione della guardia tralasciando di fare il "mezzo ciclo" restante
- Il Java ci viene in aiuto mettendo a disposizione due comandi particolari

Ciclo for: semantica

Esprimiamo il significato del for in termini di ciclo while.

Il ciclo `for (C1; E; C2) C`
è equivalente a

```
{
  C1
  while (E) {
    C
    C2
  }
}
```

break e continue

- I comandi `break` e `continue` possono evitare la scrittura di codice troppo macchinoso all'interno del while
- L'esecuzione di `break` provoca l'immediata interruzione dell'esecuzione del corpo e l'usicita dal ciclo corrente (quello più interno rispetto al quale si trova il `break`)
- L'esecuzione di `continue` provoca l'immediata interruzione dell'esecuzione del corpo del ciclo corrente e la rivalutazione della guardia

Il ciclo e mezzo: esempio

- Supponiamo di prendere in input da un `InputStreamReader input` una serie di caratteri e di voler elaborare solo quelli che sono lettere maiuscole
- Quando il flusso di caratteri termina (viene restituito -1) bisogna uscire dal ciclo
- Nel mezzo del corpo del ciclo, dopo aver letto il carattere, sappiamo se il flusso è finito, se bisogna elaborarlo oppure se bisogna passare al carattere successivo

Esempio con break e continue

```
int x;
char c;
while (true) {
  x = input.read();
  if (x == -1){
    break; // Esce dal ciclo
  c = (char) x;
  if (!Character.isLetter(c) ||
      !Character.isUpper(c))
      continue; // Salta l'elaborazione
  // ELABORAZIONE
  }
}
```


Il ciclo e mezzo: esempio

```
int x;
char c;
boolean fineFlusso = false;
while (!fineFlusso) {
  x = input.read();
  if (x == -1){
    fineFlusso = true;
  } else {
    c = (char) x;
    if (Character.isLetter(c) &&
        Character.isUpper(c)) {
        // Elaborazione
    }
  } }
```

Array

- Gli array sono strutture dati molto semplici, esistono in quasi tutti i linguaggi di programmazione imperativi
- Un array è un raggruppamento di un certo numero N di variabili dello stesso tipo
- Tali variabili, chiamate elementi, sono indicizzate da quella di indice 0 a quella di indice $N-1$
- Un array rappresenta in maniera naturale una sequenza finita, di lunghezza massima predeterminata, di variabili dello stesso tipo

Strutture Dati

- Una struttura dati è un raggruppamento di dati semplici
- Lo scopo di una struttura dati è quello di organizzare i dati in maniera da favorire:
  - uno o più algoritmi che operano sui dati stessi, in modo da rendere tali algoritmi più semplici e/o efficienti
  - la rappresentazione della realtà nel modello fatto dall'applicazione

Array

- Un array ha un nome, ad esempio a, un tipo degli elementi, ad esempio `int`, e una dimensione, ad esempio 3
- In Java, per dichiarare un array di questo tipo, si scrive:
  `int[] a = new int[3];`
  oppure, equivalentemente,
  `int a[] = new int[3];`
- `a[i]` denota la variabile di tipo int con indice `i` $\in \{0,1,2\}$ in a

Strutture Dati

Esempi di strutture dati che già conosciamo:

- l'insieme delle variabili istanza di un oggetto, esse sono concettualmente raggruppate e incapsulate (corrispondono al record del Pascal o alle strutture del C)
- liste
- alberi
- pile
- ...

Array - Lunghezza

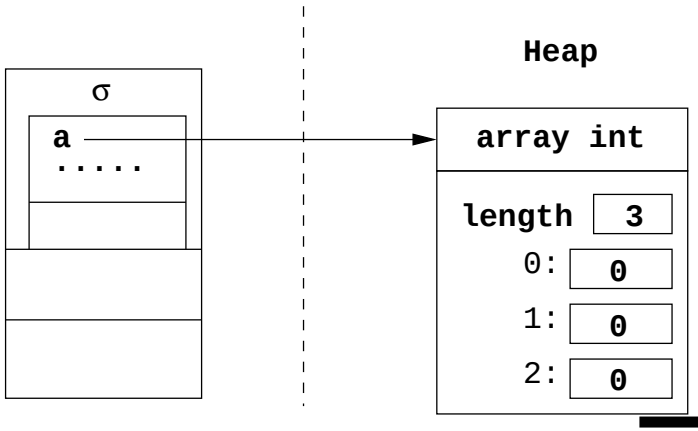
- In ogni caso, una volta che si è creato un array di una certa lunghezza, non è possibile “allungarlo”
- Quello che si può fare è creare un array nuovo più lungo, copiare i valori del vecchio array su quello nuovo e buttare via (cancellare il riferimento) al vecchio array
- In Java esistono classi che fanno tutto questo automaticamente (`ArrayList<E>`). Le vedremo approfonditamente.

Array - Implementazione

- La macchina astratta Java implementa un array come un oggetto
- E’ una differenza fondamentale rispetto all’implementazione classica delle macchine astratte del C, del Pascal, del Fortran o altri
- Il fatto che l’array sia allocato nello heap permette di specificare quanto deve essere lungo (la sua dimensione) anche a tempo di esecuzione

Array - Effetto della dichiarazione

La dichiarazione `int[] a = new int[3];` ha il seguente effetto nella memoria della macchina astratta Java:



Array - Implementazione

- In C, Pascal, Fortran, ecc bisogna scrivere nel programma la costante numerica che rappresenta la lunghezza dell’array che si dichiara (fatti salvi i casi in cui si allocano zone di memoria e si trattano come array)
- In Java invece possiamo dichiarare un array anche così:
  `int a[] = new int[n];`
  dove n è una variabile intera il cui valore viene preso in input, ad esempio

Array - Uso

- Le variabili di un array si comportano esattamente come se fossero variabili istanza pubbliche di un oggetto
- Però per “raggiungerle” si usa la notazione `a[i]` invece che `a.nome`. Ad esempio:

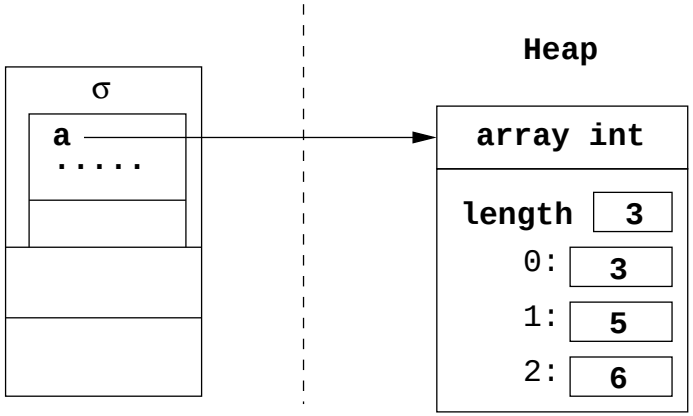
```
...
int i=0;
a[i]=3;                                    (1)
a[i+1]=5;                                  (2)
a[a.length - 1]= (i + 2) * a.length;       (3)
...
```


Array - Implementazione

- L’oggetto che rappresenta l’array ha un campo pubblico `length`, dichiarato `final` e quindi non modificabile, che contiene la dimensione (lunghezza) dell’array
- Quindi `a.length` è un valore intero che indica quanto è lungo l’array a
- Bisogna sempre ricordare però che l’ultimo indice dell’array non è `a.length`, ma `a.length - 1`, poiché il primo indice è `0`

Array - Uso

Porta nel seguente stato:



Array - Valodi di default

- All’interno dell’oggetto che rappresenta l’array sono presenti le `a.length` variabili che contengono gli elementi
- Esse vengono inizializzate allo stesso modo delle variabili istanza degli oggetti normali
- Quindi, in questo caso in cui il tipo è `int`, il valore di default che prendono tutti gli elementi è `0`
- Se, ad esempio, il tipo delle variabili fosse invece un tipo riferimento ad una classe allora i valori sarebbero inizializzati a `null`

Array di riferimenti

Supponiamo di avere una classe:

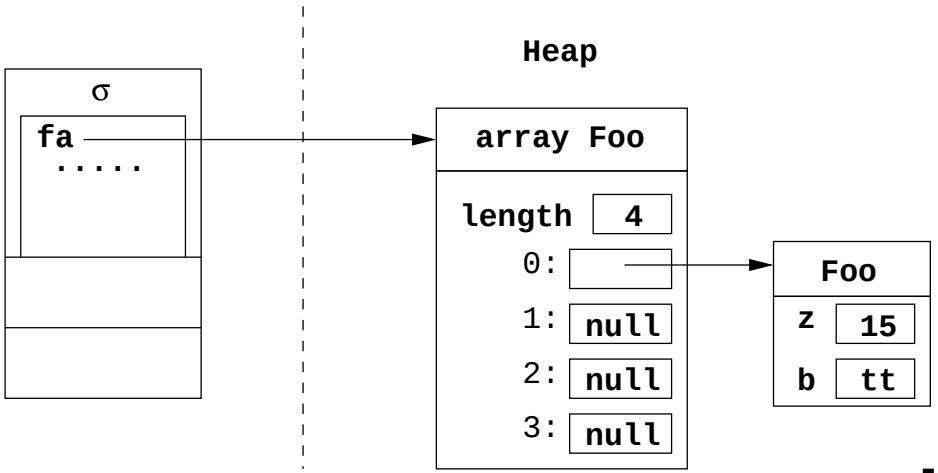
```
public class Foo {
   public int z;
   public boolean b;
   public void condInc(int y) {
      if (this.b) this.z = this.z + y;
   }
}
```

Allora la dichiarazione:

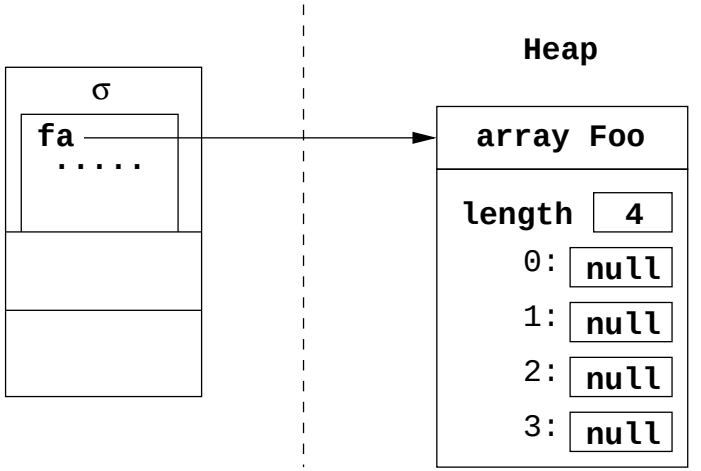
```
Foo[] fa = new Foo[4];
```

ha il seguente effetto...

Array di riferimenti



Array di riferimenti



Array di riferimenti

Vediamo l'effetto delle seguenti istruzioni:

```
...
fa[0] = new Foo();
fa[0].z = 15;
fa[0].b = true;
...
```

Array di riferimenti

Vediamo l'effetto delle seguenti istruzioni:

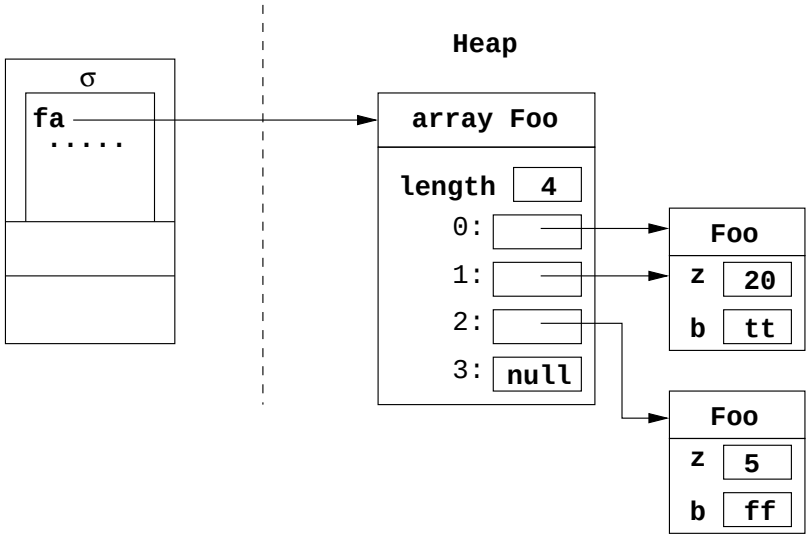
```
...
fa[2].z = 5;
fa[1].condInc(fa[2].z);
...
```


Array di riferimenti

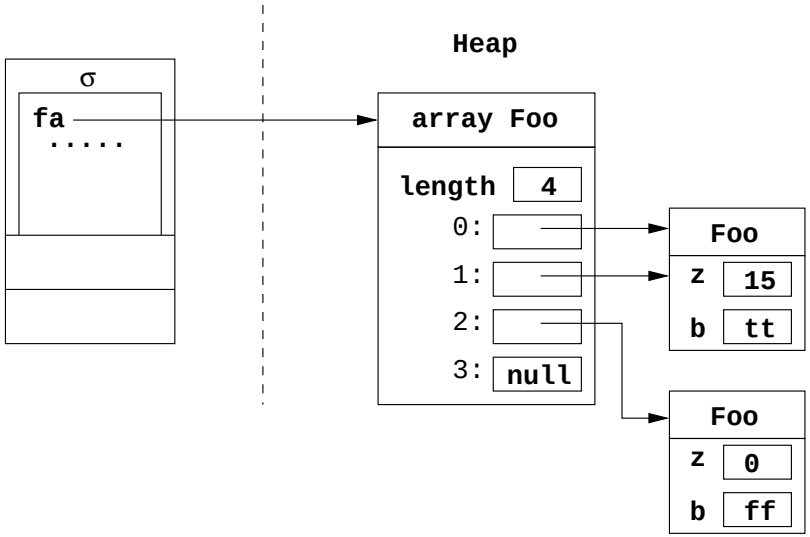
Vediamo l'effetto delle seguenti istruzioni:

```
...
fa[1] = fa[0];
fa[2] = new Foo();
...
```


Array di riferimenti



Array di riferimenti



Array bidimensionali

- In Java è possibile dichiarare array a due dimensioni
- Un array a due dimensioni può essere immaginato come una tabella (matrice) in cui ci sono N righe ed M colonne
- Le righe sono indicizzate da 0 a $N-1$
- Le colonne sono indicizzate da 0 a $M-1$
- Ogni casella della matrice contiene un elemento del tipo dichiarato per l'array

Array bidimensionali

Ad esempio la dichiarazione:
`int[][] m = new int[3][4];`
crea una matrice 3×4 di interi

0	0	0	0
0	0	0	0
0	0	0	0

- `m.length` dà il numero di righe della matrice
- `m[i].length` dà il numero di colonne della matrice se `i` è un indice di riga valido

Array bidimensionali

Ogni elemento della matrice è individuato per mezzo di due indici, il primo indica la riga e il secondo la colonna.
Ad esempio,
`m[1][2] = 3;`

	0	*1*	*2*	*3*
0	0	0	0	0
1	0	0	3	0
2	0	0	0	0

Ciclo for e array

Spesso il ciclo for è quello più naturale per agire sugli array.

Inizializza un array di interi in cui gli elementi contengono le proprie posizioni.

```
int[] a = new int[10];
for (int i = 0; i < a.length; i++)
   a[i] = i;
```

Esempio: ordinamento di un array

Ordinamento dell'array di interi a

```
int i; // indice esterno
int j; // indice interno
int buf; // variabile di appoggio
for (i = 0; i < a.length -1; i++)
   for (j = i + 1; j < a.length; j++)
      if (a[i] > a[j])
        { // scambio
          buf = a[i];
          a[i] = a[j];
          a[j] = buf;
        }
```


Ciclo for e array

- Sposta tutti gli elementi di un array di una posizione verso sinistra.
- Il primo elemento viene perduto
- nella posizione finale dell'array "spostato" viene posto a zero

```
int i; // Dichiaro i fuori
       // dal blocco del for
for (i = 0; i < a.length - 1; i++)
   a[i] = a[i+1];
a[i] = 0; // i è ancora definita
```


Ciclo for e matrici

Esempio: azzeramento di una matrice di interi.

```
for (int i = 0; i < m.length; i++)
  for (int j = 0; j < m[0].length; j++)
    m[i][j] = 0;
```


Esempio: ordinamento di un array

- Algoritmo *bubble-sort*
- Ogni posizione `i`, a partire da 0, viene confrontata con tutte le posizioni successive
- Se l'elemento attualmente in posizione `i` è maggiore di quello in posizione successiva, allora vengono scambiati di posto
- Tutto questo non accade per l'ultima posizione dell'array, dove alla fine si troverà automaticamente un elemento massimo
- Uso di due cicli for annidati con indici `i` e `j`

Programmazione su sequenze

- Vediamo alcuni schemi di programmazione di operazioni tipiche su sequenze di dati
- Come struttura dati per rappresentare le sequenze useremo gli array
- Comunque gli schemi di programmazione sono validi per qualunque struttura dati che rappresenta una sequenza: lista, iterazione su elementi di un insieme, ecc.
- Cambierà solamente la sintassi con cui indicizzare e reperire gli elementi della sequenza

Esercizi

- Implementare i metodi della classe Scacchiera del Tris
- Scrivere un programma che prende in input un numero n e stampa un triangolo del tipo:

```
[]
[][]
[][][]
[][][][]
[][][][][]
[][][][][][]
```

di n righe.

Schemi di programmazione

- Calcolo di aggregati: media, minimo, massimo, somma, ecc.
- Ricerca del primo elemento della sequenza che soddisfa una certa proprietà (Ricerca Lineare Incerta)
- Verifica che tutti gli elementi della sequenza soddisfano una certa proprietà
- Varianti in cui si considerano solo certe porzioni della sequenza

Esercizi

- Modificare la classe CombinationLock in modo da creare cassaforti con combinazioni formate da un numero arbitrario n di lettere
- Scrivere una classe Forzatore che, ricevuta in ingresso una cassaforte così potenziata, tenta di trovare la combinazione con la forza bruta
- Versione soft: il forzatore sa di quante lettere è formata la combinazione
- Versione hard: il forzatore ignora di quante lettere sia formata la combinazione

Esempio: elemento massimo

Trovare il valore massimo presente in una sequenza di interi (array a) e la posizione del primo elemento che ha questo valore nella sequenza

```
// Accumulatore inizializzato al primo elemento
int massimo = a[0];
int posizioneMax = 0;
for (int i = 1; i < a.length; i++)
  if (a[i] > massimo) {
    massimo = a[i];
    posizioneMax = i;
  }
System.out.println("Valore massimo: " + massimo);
System.out.println(
  "Posizione primo elemento massimo: " + posizioneMax);
```


Calcolo di aggregati

- Si deve scorrere tutta la sequenza: ciclo for
- Si definiscono una o più variabili "accumulatori", inizializzate opportunamente
- Ad ogni iterazione del ciclo, a seconda del valore dell'elemento della sequenza che si sta analizzando, viene aggiornato l'accumulatore
- Alla fine del for si determina il valore cercato a partire dal valore dell'accumulatore

Ricerca Lineare Incerta

- Si usa per cercare, in una sequenza, la posizione del primo elemento che soddisfa una certa proprietà
- La proprietà $P(v)$ è definita sui valori contenuti negli elementi della sequenza (es. array a)
- La sequenza potrebbe non contenere nessun elemento che soddisfa la proprietà
- Il risultato è la posizione dell'elemento, se trovato. Altrimenti una variabile booleana indica che l'elemento non è stato trovato

Esempio: somma e media

Somma e media degli elementi di una sequenza di interi (array a)

```
int somma = 0; // Accumulatore
for (int i = 0; i < a.length; i++)
  somma = somma + a[i];
System.out.println("Somma: " + somma);
double media =
        (double) somma / a.length;
System.out.println("Media: " + media);
```

RLI: esempio

- Supponiamo di dover scrivere un metodo che prende in input un array di interi a e un numero n
- Si vuole ricercare in a, se c'è, il primo elemento maggiore di n
- Se lo si trova il metodo deve rispondere con la posizione dell'elemento trovato
- Se non lo si trova il metodo deve rispondere con -1

RLI: schema

Schema di RLI su una sequenza (array a)

```
int i = 0;
boolean trovato = false;
while (i < a.length && !trovato)
   if (P(a[i]))
      trovato = true;
   else
      i = i + 1;
(1)
```


RLI: esempio

```
public int m1(int[] a, int n){
  int i = 0;
  boolean trovato = false;
  while (i < a.length && !trovato)
    if (a[i] > n)
      trovato = true;
    else i = i + 1;
  if (trovato)
    return i;
  else return -1;
}
```


RLI: schema

Nel punto `(1)` del programma:

- Se la variabile booleana `trovato` vale `true` allora il primo elemento dell'array che soddisfa P si trova nella posizione contenuta nella variabile `i`
- Se la variabile `trovato` vale `false` allora si ha che nessun elemento dell'array soddisfa P

Verifica: esempio

- Supponiamo di dover scrivere un metodo che prende in input un array di interi a e un numero intero b
- Il metodo deve rispondere `true` se tutti gli elementi di a sono divisibili per b, altrimenti `false`
- Trasformiamo il problema e cerchiamo quindi, se c'è, il primo elemento di a che **non** è divisibile per b.
- In base al valore di `trovato` il metodo ritornerà il valore booleano appropriato

Verifica su tutti gli elementi

- Si vuole sapere se tutti gli elementi di una sequenza soddisfano una certa proprietà $P(v)$
- La soluzione più semplice ed efficiente è quella di fare una trasformazione logica del problema
- Si risolve il problema negato: si cerca cioè un elemento della sequenza che **non** soddisfa P

Verifica: esempio

```
public boolean m2(int[] a, int b){
  int i = 0;
  boolean trovato = false;
  while (i < a.length && !trovato)
    if (a[i] % b != 0)
      trovato = true;
    else i++;
  if (trovato)
    return false;
  else return true;
}
```


Verifica su tutti gli elementi

- Si imposta quindi lo schema della RLI per trovare il primo elemento della sequenza che soddisfa $!P(v)$
- All'uscita dal ciclo:
  - Se `trovato` è `true` allora la risposta al problema iniziale è NO
  - Se `trovato` è `false` allora la risposta al problema iniziale è SI: tutti gli elementi della sequenza soddisfano la proprietà P

Varianti: esempio

```
public boolean
     porzioneOrdinata(int[] a,
             int start, int stop){
  int i = start;
  boolean trovato = false;
  while (i < stop && !trovato)
    if (a[i]>a[i+1])
      trovato = true;
    else i++;
 // Invece di if (trovato) return false;
 //           else return true;
  return !trovato; }
```


Varianti

- In alcuni casi il problema posto richiede che si cambino gli estremi di ricerca/verifica nella sequenza
- Ad esempio: controllare se una porzione di un array di interi a sia composta da elementi in ordine crescente
- `start` e `stop` sono due parametri di input che indicano la posizione di partenza e di arrivo della porzione di array

ArrayList invece che array

- Abbiamo visto che gli array sono una buona struttura dati per rappresentare sequenze di elementi di uno stesso tipo
- Tuttavia hanno il problema che la loro lunghezza massima deve essere specificata al momento della creazione
- Se abbiamo bisogno di altro spazio dobbiamo ricreare un array più lungo e copiare la prima parte del vecchio array in quello nuovo
- In Java esiste già una classe che fa questo automaticamente

Varianti

- Trasformiamo il problema di verifica nella ricerca del primo elemento che rompe l'ordinamento crescente
- Facciamo partire la ricerca con `i = start`
- Interrompiamo la ricerca se `i < stop` poiché l'ultimo elemento della porzione non influenza la verifica richiesta
- Se l'elemento che rompe l'ordinamento viene trovato rispondiamo `true`, altrimenti `false`

Inserimento

- L'inserimento in una sequenza è un problema classico
- L'inserimento in fondo non comporta in genere problemi di implementazione, anche nel caso di array (se l'array non è pieno)
- L'inserimento in una posizione interna alla sequenza comporta lo spostamento a destra degli elementi interessati
- Quest'ultima operazione richiede un'implementazione adeguata nel caso di un array

ArrayList<E>

- La classe `ArrayList<E>` del pacchetto `java.util` permette di creare e gestire una sequenza di oggetti di tipo `E`
- Il numero di oggetti nella sequenza non è predeterminato: la classe alloca lo spazio necessario al crescere o decrescere degli elementi inseriti
- Per creare un `ArrayList<Foo>` di oggetti di una classe `Foo`:

```
ArrayList<Foo> myList = new ArrayList<Foo>();
```


Inserimento

- C'è un'altra questione importante da tenere presente in caso di inserimento di un elemento in una sequenza, sia in fondo che nel mezzo
- Bisogna stabilire se la sequenza in questione ammette l'inserimento di elementi già presenti, cioè se ammette la presenza di elementi duplicati

ArrayList<E>

- La lista viene creata vuota
- Per aggiungere un oggetto in fondo alla lista corrente:

```
Foo f = new Foo();
myList.add(f);
```

- Per aggiungere un oggetto in posizione `i`:

```
f = new Foo();
myList.add(i,f);
```

- L'oggetto attualmente in posizione `i` e quelli seguenti vengono spostati verso destra

ArrayList<E>: esempio

- Scriviamo un metodo che ottiene in input un oggetto `ArrayList<Foo>` e si occupa di chiamare il metodo `m()` su ogni oggetto contenuto nella lista
- Si applica lo stesso schema del calcolo di aggregati:

```
public void applyM(ArrayList<Foo> list){
  for (int i = 0; i < list.size(); i++)
    list.get(i).m();
}
```


Inserimento

- Nel caso in cui non si vogliano duplicati si deve dapprima fare una RLI per determinare se l'elemento da inserire è già presente nella sequenza
- Solo nel caso in cui la risposta sia negativa si deve poi procedere all'inserimento vero e proprio

ArrayList<E>

- Tutti gli schemi di programmi che abbiamo visto utilizzavano gli array come struttura dati per rappresentare una sequenza
- Ovviamente gli stessi schemi valgono, inalterati, anche nel caso in cui la sequenza è rappresentata con un `ArrayList<E>`
- Basta semplicemente utilizzare la sintassi appropriata

ArrayList<E>

- Per ottenere il numero attuale di elementi della lista:
  `int n = myList.size();`
- Per ottenere l'oggetto in posizione `i`:
  `Foo fooObj = myList.get(i);`
- Per assegnare un nuovo elemento alla posizione `i`:
  `myList.set(i,f);`
- Altre funzionalità della classe possono essere trovate sulle API

ArrayList<E>

- Ad esempio, per ottenere il valore dell'elemento in posizione `i` non si utilizza `a[i]`, ma la chiamata di metodo `a.get(i)`
- Oppure, per cambiare il contenuto della posizione `i`, non si utilizza l'assegnamento `a[i] = nuovoElemento`, ma si chiama il metodo `a.set(i, nuovoElemento)`
- Per ottenere la lunghezza il metodo `size()` invece che la variabile istanza `length`, ecc.

Cancellazione

- Affrontiamo ora il problema della cancellazione di un elemento da una sequenza
- A prima vista è un problema banale, ma ci sono alcuni accorgimenti da prendere per evitare errori logici difficili da individuare
- **Innanzitutto**: se l'elemento in posizione `i` di una sequenza deve essere cancellato allora tutti gli elementi dall'`i+1`-esimo alla fine devono essere spostati di una posizione verso sinistra

Cancellazione

- **Inoltre**: se si sta scorrendo la sequenza e si cancella un certo elemento in posizione `i`, allora il ciclo di scorrimento deve evitare di incrementare l'indice `i` in quella iterazione
- Questo perché la cancellazione avrà fatto in modo da spostare l'elemento e che precedentemente era in `i+1` nella posizione `i`
- Se il ciclo incrementa l'indice `i` l'elemento e non verrà preso in considerazione nello scorrimento

Cancellazione e array

- Gestire la cancellazione con gli array risulta abbastanza insidioso per i motivi visti sopra
- In genere se una sequenza è soggetta ad aggiunte e cancellazioni dinamiche, e quindi ha una lunghezza variabile non predeterminabile, è meglio non usare un array per rappresentare la sequenza
- E' più adatta una struttura dati dinamica come l'`ArrayList<E>`

Iteratori

- Per una sequenza rappresentata con un `ArrayList<E>` possiamo utilizzare un `Iterator<E>` o un `ListIterator<E>`
- Queste due sono interfaces, non classi. Dal nostro attuale punto di vista in cui ci limitiamo a **usare** oggetti di questo tipo non c'è differenza
- Su un oggetto di tipo `Iterator<E>` (o `ListIterator<E>`) possiamo chiamare tutti i metodi documentati nelle relative API (package `java.util`)

Esercizio

- Implementare la cancellazione di un elemento e lo scorrimento con cancellazione su una sequenza rappresentata con un array
- Fare attenzione ai problemi di cui sopra
- Tener presente che durante lo scorrimento l'indice deve avanzare solo se l'ultimo elemento controllato non è stato cancellato

Iteratori

- Un iteratore è un oggetto che permette di scorrere (visitare) **tutti** gli elementi una collezione (nel nostro caso una lista) **una e una sola** volta
- Durante lo scorrimento è possibile cancellare gli elementi visti, senza incappare nei possibili errori di scorrimento visti sopra
- Nella sua versione più semplice mette a disposizione tre metodi

Cancellazione e `ArrayList<E>`

- La classe `ArrayList<E>` ha un metodo `remove(int index)` che implementa la cancellazione in modo corretto: gli elementi a destra del cancellato si spostano tutti di una posizione verso sinistra
- Per effettuare senza errori lo scorrimento con cancellazione (detto anche filtraggio) la classe mette a dispozione un metodo `iterator()` che restituisce un **iteratore** sulla sequenza

Iteratori

1. `boolean hasNext()` indica se esiste un prossimo elemento da visitare nello scorrimento. Se dà false allora tutti gli elementi sono stati visitati
2. `E next()` restituisce il prossimo elemento da visitare, se presente
3. `void remove()` cancella dalla collezione (nel nostro caso lista) l'ultimo elemento che è stato visitato (cioè ottenuto con il metodo `next()`) senza saltare elementi nella visita/scorrimento

Iteratori: esempio

- Supponiamo di avere in input una lista di oggetti di una classe `Foo` rappresentata con un ArrayList
- Supponiamo che nella classe `Foo` sia definito un metodo predicativo `isP()`
- Vogliamo scorrere la lista e "filtrare" gli elementi che danno risposta positiva a `isP()`

Iteratori: esempio

- In altre parole vogliamo cancellare dalla lista tutti gli elementi che rispondono `true` alla chiamata di `isP()`
- Alla fine restituiamo la lista filtrata

Iteratori: esempio

```
public ArrayList<Foo>
        filterP(ArrayList<Foo> list){
  Foo f;
  Iterator<Foo> it = list.iterator();
  while(it.hasNext()){
    f = it.next();
    if (f.isP())
      it.remove();
  }
// La lista è stata modificata durante
// lo scorrimento con l'iteratore
  return list; }
```

Iteratori e insiemi

- Gli oggetti della classe `java.util.HashSet<E>` rappresentano un insieme di oggetti della classe `E`
- Un insieme si differenzia da una lista nel fatto che gli elementi non sono indicizzati e non ci sono elementi uguali ripetuti
- Per utilizzare la classe `HashSet<E>` è obbligatorio ridefinire in maniera appropriata il metodo `equals` e il metodo `hashcode` per gli oggetti della classe `E`

Iteratori e insiemi

- Una volta fatto questo per ricercare un oggetto nell'insieme si può utilizzare il metodo contains
- Per scorrere tutti gli elementi di un insieme possiamo ottenere un iteratore con il metodo `iterator()` della classe `HashSet<E>`
- L'iterazione avverrà seguendo un ordine arbitrario
- Consultare le API

Esercizio

- Modellare con le opporturne classi un ufficio anagrafe
- L'ufficio deve gestire un database di persone di un comune
- I dati sono immagazzinati in liste in memoria (eventualmente prevedere delle classi per salvare su disco le liste attuali e ricaricarle da disco)
- L'ufficio deve mettere a disposizione tutti i classici servizi di: inserimento, ricerca, aggiornamento dati e cancellazione

Aggiornamento

- Un altro problema classico sulle sequenze è quello dell'aggiornamento
- Si tratta di ricercare un certo elemento e poi cambiarlo
- Si implementa, ovviamente, dapprima con una RLI per individuare, se c'è, la posizione da cambiare
- Poi si utilizza l'assegnamento o il metodo `set` (nel caso di ArrayList) per sostituire l'elemento con un altro diverso o con lo stesso elemento aggiornato

Esercizio

- Modellare con le opportune classi gli automi a stati finiti
- Un oggetto automa deve permettere di ottenere tutti i cammini etichettati su una stringa w data e deve avere un metodo che risponde `true` o `false` a seconda che w sia accettata o non accettata
- Un oggetto automa deve restituire un oggetto automa deterministico equivalente a se stesso costruito con l'algoritmo di costruzione dei sottoinsiemi